

**Information Technology University of the Punjab**  
**SE301T Operating Systems – Spring 2025**  
**Assignment 1 [CLO1]**

**Deadline:** **Solution**

**Total Marks: 60**

**Instructions:**

- i. This is an individual assignment so every student must submit their own solution.
- ii. You will need a Linux-based environment to attempt this assignment (WSL/dual-boot/VMWare etc.).
- iii. All questions require you to write and run code. Use gcc in a Linux shell to compile and run your programs.
- iv. As submission, you will be required to submit a single report in PDF format on Google Classroom. Your report should have all your codes and screenshots of all outputs that show the complete working of your program.
- v. Clearly label the start of all questions in your report. Also, make sure to highlight the important bits of your solution.
- vi. You can take help from the textbook/reference books or the Linux man pages (man fork etc.). Discussion among peers without showing the solution is acceptable, but you must attempt individually. Your submissions will be checked for plagiarism including AI generated content and if found, it will be dealt with according to the anti-plagiarism policy.
- vii. Make sure to submit it by the deadline. Late submissions will not be accepted.

1. [5] Write a program that opens a file (with the `open()` system call) and then calls `fork()` to create a new process. Can both the child and parent access the file descriptor returned by `open()`? What happens when they are writing to the file concurrently, i.e., at the same time? (use the `write()` system call to write to the file)

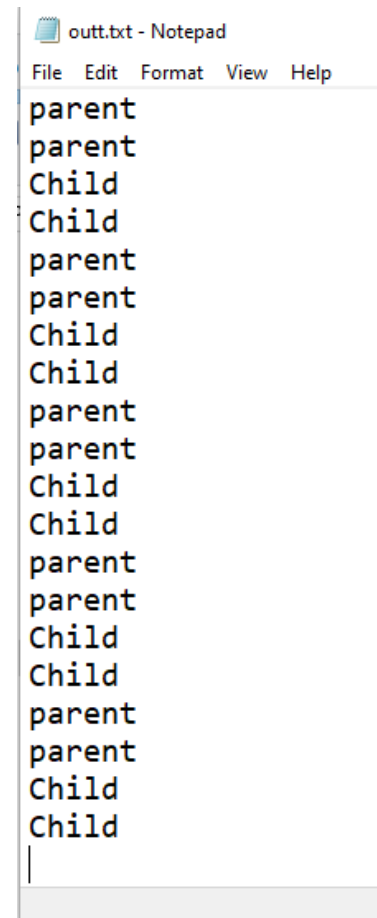
```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/wait.h>

int main() {
    int fd = open("outt.txt", O_CREAT|O_WRONLY|O_TRUNC, 0644);

    int rc = fork();

    if (rc == 0) {
        for (int i = 0; i<10;i++)
            write(fd, "Child\n", 6);
        printf("child has written\n");
    }
    else {
        for (int i = 0; i<10;i++)
            write(fd, "parent\n",7);
        printf("parent has written!\n");
        wait(NULL);
    }

    close(fd);
    return 0;
}
```



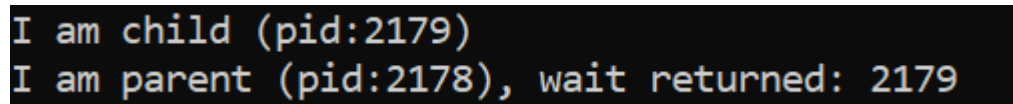
Output file screenshot on the right.

As both are writing concurrently, we can see that both get turns to write to the file randomly.

2. [5] Now write a program that forks and uses `wait()` to wait for the child process to finish in the parent. Simulate work in child by using `sleep()` and print some message in both processes that helps identify them, e.g. "I am parent, pid:12", "I am child, pid:14". Print the return value of `wait()` in the parent. What happens if you use `wait()` in the child?

```
int main(){
    int rc = fork();

    if (rc == 0){
        //wait(NULL);
        sleep(3);
        printf("I am child (pid:%d)\n", getpid());
        exit(0);
    }
    else
    {
        int wc = wait(NULL);
        printf("I am parent (pid:%d), wait returned: %d\n", getpid(), wc);
        exit(0);
    }
}
```



```
I am child (pid:2179)
I am parent (pid:2178), wait returned: 2179
```

Calling `wait()` in child has no effect on the program, as `wait()` only waits for children of the caller, and as the child does not have any children, `wait` returns immediately.

3. [10] Modify the above program to create two more children of the same parent process. Each child should sleep for some time and then print a message to identify it (e.g. "This is child 1.\n") and exit. Newer/younger child should sleep more. Use `waitpid()` in the parent to wait for the children such as the following output is printed by the complete program:

```
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.
All three children finished working. Parent exiting.
```

```
int main(){
    int rc1 = fork();

    if (rc1 == 0){
        //wait(NULL);
        sleep(3);
        printf("I am first child (pid:%d)\n", getpid());
        exit(0);
    }
    int rc2 = fork();
    if (rc2 == 0){
        sleep(4);
    }
}
```

```

        printf("I am second child (pid:%d)\n", getpid());
        exit(0);
    }

    int rc3 = fork();
    if (rc3 == 0){
        sleep(5);
        printf("I am third child (pid:%d)\n", getpid());
        exit(0);
    }

    printf("This is parent, waiting for first child\n");
    waitpid(rc1,NULL,0);
    printf("This is parent, waiting for second child\n");
    waitpid(rc2,NULL,0);
    printf("This is parent, waiting for third child\n");
    waitpid(rc3,NULL,0);
    printf("All three children finished working. Parent exiting\n");
    exit(0);
}

```

```

This is parent, waiting for first child
I am first child (pid:2181)
This is parent, waiting for second child
I am second child (pid:2182)
This is parent, waiting for third child
I am third child (pid:2183)
All three children finished working. Parent exiting

```

4. [10] Run the following program and explain its output. Identify the lines of code that execute in each child and the parent and also explain what each line does.

1	#include <stdio.h>	22	close(pipefd[1]);
2	#include <stdlib.h>	23	
3	#include <unistd.h>	24	execlp("echo", "echo", "Hello,
4	#include <sys/wait.h>	25	world!", NULL);
5		26	perror("execlp failed");
6	int main() {	27	exit(1);
7	int pipefd[2];	28	}
8	if (pipe(pipefd) == -1) {	29	
9	perror("pipe failed");	30	pid_t child2 = fork();
10	exit(1);	31	if (child2 == -1) {
11	}	32	perror("fork failed");
12		33	exit(1);
13	pid_t child1 = fork();	34	}
14	if (child1 == -1) {	35	
15	perror("fork failed");	36	if (child2 == 0) {
16	exit(1);	37	close(pipefd[1]);
17	}	38	dup2(pipefd[0], STDIN_FILENO);
18		39	close(pipefd[0]);
19	if (child1 == 0) {	40	
20	close(pipefd[0]);	41	execlp("wc", "wc", "-c", NULL);
21	dup2(pipefd[1], STDOUT_FILENO);	42	perror("execlp failed");

```

43         exit(1);
44     }
45
46     close(pipefd[0]);
47     close(pipefd[1]);
48     wait(NULL);
49     wait(NULL);
50
51     return 0;
52 }

```

14

Parent:

Creates a pipe. This pipe is available to all children spawned after its creation.

In lines 45, 46, it closes the pipe read and write ends for itself.

In lines 47 and 48, two calls to wait for the two child processes.

Child 1:

Line 20: Closes read end of the pipe.

Line 21: Duplicates the write end of the pipe to `STDOUT_FILENO`. `dup2` closes the second argument file descriptor and copies to it the first file descriptor. So this implies that anything that uses `stdout` to output something to the terminal inside child here onwards, will actually redirect its output to the child's write end of the pipe.

Line 22: Closes write end of the pipe. `STDOUT_FILENO` now points to the write end of the pipe, so this can be closed now. It may be closed here or later in the child.

Line 24: Calls `exec` to execute the `echo "Hello, world!"` shell command, this uses the `echo` program in `usr/bin`. Normally, `echo` prints its arguments on terminal, so `Hello, world!` would have been printed on the terminal, but as `STDOUT_FILENO` now holds the write end of the pipe, `Hello, world!` would be directed to the pipe and not be printed.

Line 25 and 26: Only execute if `exec` fails. As `exec` replaces the current process's address space with the new program, so these lines should not run in case of `exec` success.

Child2:

Line 36: Closes write end of the pipe.

Line 37: Duplicates the read end of the pipe to `STDIN_FILENO`. `dup2` closes the second argument file descriptor and copies to it the first file descriptor. So this implies that anything that uses `stdin` to input something from the terminal inside child here onwards, would actually take input from the child's read end of the pipe.

Line 38: Closes read end of the pipe. `STDIN_FILENO` now points to the read end of the pipe, so this can be closed now. It could have been closed later in the child; it would not matter.

Line 40: Calls `exec` to execute `"wc -c"` command. This counts the number of bytes (`-c` option) in the argument (not given, so it expects input from the user – `STDIN` on shell). As `STDIN` now holds the child's read end, so `wc` would take input from that read end. As pipe read wait for something to be written at the write end by some other process using the same pipe, so this waits here for child 1 to write `Hello, world!` When it gets written, `wc` counts the number of bytes (14 including the `NULL` terminator) and prints it on `STDOUT` (`wc` can still use `stdout` as `stdout` in Child2 has not been changed).

Line 41 and 42: Only execute if `exec` fails. As `exec` replaces the current process's address space with the new program, so these lines should not run in case of `exec` success.

5. [30] In this question, you'll measure the costs of a system call and context switch. Measuring the cost of a system call is relatively easy. For example, you could repeatedly call a simple system call (e.g., performing a 0-byte read), and time how long it takes; dividing the time by the number of iterations gives you an estimate of the cost of a system call.

One thing you'll have to take into account is the precision and accuracy of your timer. A typical timer that you can use is `gettimeofday()`; read the man page for details. What you'll see there is that `gettimeofday()` returns the time in microseconds since 1970; however, this does not mean that the timer is precise to the microsecond. Measure back-to-back calls to `gettimeofday()` to learn something about how precise the timer really is; this will tell you how many iterations of your null system-call test you'll have to run in order to get a good measurement result.

```

#define _GNU_SOURCE
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>

```

```

#include <sched.h>
#include <sys/time.h>

int main() {
    struct timeval start, end;
    gettimeofday(&start, NULL);
    int pid = getpid();
    gettimeofday(&end, NULL);
    printf("Total time taken by getpid system call: %ld\n", end.tv_usec -
start.tv_usec);
    exit(0);
}

```

This code measures time of the getpid system call only once.

**Total time taken by getpid system call: 3**

Measuring the cost of a context switch is a little trickier. The lmbench benchmark does so by running two processes on a single CPU, and setting up two UNIX pipes between them; a pipe is just one of many ways processes in a UNIX system can communicate with one another. The first process then issues a write to the first pipe, and waits for a read on the second; upon seeing the first process waiting for something to read from the second pipe, the OS puts the first process in the blocked state, and switches to the other process, which reads from the first pipe and then writes to the second. When the second process tries to read from the first pipe again, it blocks, and thus the back-and-forth cycle of communication continues. By measuring the cost of communicating like this repeatedly, lmbench can make a good estimate of the cost of a context switch. You can try to re-create something similar here, using pipes, or perhaps some other communication mechanism such as UNIX sockets.

One difficulty in measuring context-switch cost arises in systems with more than one CPU; what you need to do on such a system is ensure that your context-switching processes are located on the same processor. Fortunately, most operating systems have calls to bind a process to a particular processor; on Linux, for example, the `sched_setaffinity()` call is what you're looking for. By ensuring both processes are on the same processor, you are making sure to measure the cost of the OS stopping one process and restoring another on the same CPU.

```

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

int main() {
    int fd[2];
    int fd1[2];
    char buffer[100];

    if (pipe(fd) == -1) {
        perror("pipe failed");
        exit(1);
    }
    if (pipe(fd1) == -1) {
        perror("pipe failed");
        exit(1);
    }
}

```

```

pid_t pid = fork();

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

if (pid == 0) { // Child process
    cpu_set_t mask;
    int cpu = 0; // Set affinity to CPU 0
    struct timeval received_time, end;

    // Clear the mask and set CPU 0
    CPU_ZERO(&mask);
    CPU_SET(cpu, &mask);

    sched_setaffinity(0, sizeof(mask), &mask);

    close(fd[0]); // Close read end of pipe 1
    close(fd1[1]); // Close write end of pipe 2
    write(fd[1], "Hello from child!", 17); // Write to pipe 1
    read(fd1[0], &received_time, sizeof(received_time)); // Read from pipe 2
    gettimeofday(&end, NULL);
    printf("Total time: %ld\n", end.tv_usec - received_time.tv_usec);
    close(fd[1]); // Close write end of pipe 1
    close(fd1[0]); // Close read end of pipe 2
}

else { // Parent process

    cpu_set_t mask;
    int cpu = 0; // Set affinity to CPU 0
    struct timeval start;

    // Clear the mask and set CPU 0
    CPU_ZERO(&mask);
    CPU_SET(cpu, &mask);

    sched_setaffinity(0, sizeof(mask), &mask);

    close(fd[1]); // Close write end of pipe 1
    close(fd1[0]); // Close read end of pipe 2
    read(fd[0], buffer, sizeof(buffer)); // Read from pipe 1
    gettimeofday(&start, NULL);
    write(fd1[1], &start, sizeof(start)); // Write to pipe 2
    printf("Parent received: %s\n", buffer);
    close(fd[0]); // Close read end of pipe 1
    close(fd1[1]); // Close write end of pipe 2
}

return 0;
}

```

```
Parent received: Hello from child!  
Total time: 164
```

This code measures time for only one context switch.