Lab 3

Task 1:
Write a program p1.c that forks a child and prints the following (in the parent and child process),
Parent : My process ID is: 12345
Parent : The child process ID is: 15644
and the child process prints
Child : My process ID is: 15644
Child : The parent process ID is: 12345.

soln:

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


int main() {
	pid_t pid = fork(); // fork used to create child process
	if (pid == 0) {
		fprintf(stdout, "Child : My process ID is: %d\n", getpid());
		fprintf(stdout, "Child : The parent process ID is: %d\n", getppid());
		exit(0);
	}
	else {

		fprintf(stdout, "Parent : My process ID is: %d\n", getpid());
		fprintf(stdout, "Parent : The child process ID is: %d\n", pid);
		fflush(stdout); sleep(1);
		exit(0);
	}
}
```

Explaination :
pid_t is a signed integer which will hold process id

The Fork system call is used for creating a new process in Linux, and Unix systems, which is called the ***child process***, which runs concurrently with the process that makes the fork() call (parent process). After a new child process is created, both processes will execute the next instruction following the fork() system call.

The child process uses the same pc(program counter), same CPU registers, and same open files which use in the parent process. It takes no parameters and returns an integer value.

Below are different values returned by fork().

- ***Negative Value***: The creation of a child process was unsuccessful.
- ***Zero***: Returned to the newly created child process.
- ***Positive value***: Returned to parent or caller. The value contains the process ID of the newly created child process.

So if pid==0 condition is true for child process, and pid>0 i.e else part will be executed by parent process.

Task 2: Write another program p2.c that does exactly same as in previous exercise, but the parent process prints it's messages only after the child process has printed its messages and exited. Parent process waits for the child process to exit, and then prints its messages and a child exit message,
Parent : The child with process ID 12345 has terminated.

soln:

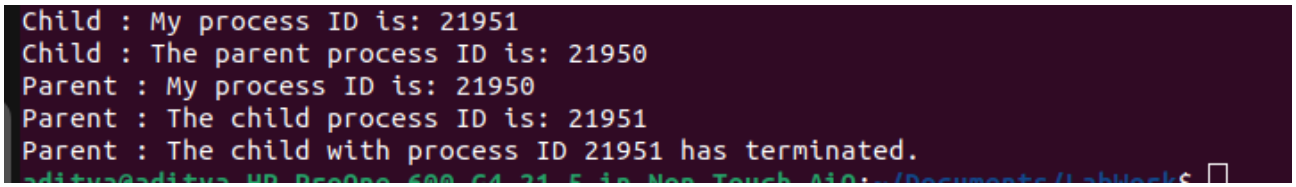
```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <stdlib.h>

int main() {
	pid_t pid = fork();
	if (pid == 0) {
		fprintf(stdout, "Child : My process ID is: %d\n", getpid());
		fprintf(stdout, "Child : The parent process ID is: %d\n", getppid());
		exit(0);
	}
	else if(pid>0){
		pid_t child = wait(NULL);
		fprintf(stdout, "Parent : My process ID is: %d\n", getpid());
		fprintf(stdout, "Parent : The child process ID is: %d\n", child);
		fprintf(stdout, "Parent : The child with process ID %d has terminated.\n", child);
		exit(0);
	}
	else{
		fprintf(stdout, "Child process Creation failed.");
		exit(0);
	}

}
```

Explaination:
A call to wait() blocks the calling process until one of its child processes exits or a signal is received. After child process terminates, parent ***continues*** its execution after wait system call

instruction.
Child process may terminate due to any of these:

- It calls exit();
- It returns (an int) from main
- It receives a signal (from the OS or another process) whose default action is to terminate.

**Child process exits and wait return child process id, then parent process continues after wait call.



• Task 3: A program mycat.c, that reads input from stdin and writes output to stdout. Further, write a program p3.c that executes the binary program mycat (compiled frommycat.c) as a child proces of p3.

soln:

program – p3.c

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



int main() {
        pid_t pid = fork();
        if (pid == 0) {
                char *args[] = {NULL};
                execvp("./mycat", args);
                exit(0);
        }
        else {
                pid_t child = wait(NULL);
                fprintf(stdout, "Parent : The child has terminated.\n");
                exit(0);
        }
}
```

program mycat.c

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

int main()
{
  char buf[512];
  int n;
  for(;;){
     n = read(0, buf, sizeof buf);
          if(n == 0) break;
          if(n < 0){
             fprintf(stdout, "read error\n");
                  exit(-1);
          }

     if(write(1, buf, n) != n)
          {
             fprintf(stdout, "write error\n");
                  exit(-1);
     }

  }
  return 1;
}
```

First compile the mycat.c using : g++ mycat.c -o mycat
then compile and run p3.c

The `execvp` function is part of the `exec` family of functions in C, which is used to replace the current process with a new process. Here we are trying to execute a program named "mycat" using the arguments specified in the `args` array. Currently passing Null. So that it will read from standard input in mycat using read commad. Read return no of characters reads (n).
Write prints on standard output the (n) characters. And again loop until any of if condition satisfies.

`n = read(0, buf, sizeof buf)`: This line uses the `read` function to read data from the standard input (file descriptor 0) into the buffer `buf`. It reads up to `sizeof(buf)` bytes. The number of bytes read is stored in the variable `n`.
The line `write(1, buf, n)` is used to write data from the buffer `buf` to the standard output (file descriptor 1). The parameters of the `write` function are as follows:

1. `1`: The file descriptor indicating the output stream. In this case, `1` refers to the standard output.
2. `buf`: The buffer containing the data you want to write.

3. `n`: The number of bytes you want to write from the buffer.

What happens when input is redirected to p3?

```
aditya@aditya-HP-ProOne-600-G4-21-5-in-Non-Touch-AiO:~/Documents/LabWork$ ./a.out <abc.txt
An "agent" is anything that takes actions in the world.
Parent : The child has terminated.
```

In this case input abc file is redircted to p3. Mycat will print the content of file and terminates ,because once it reach end of file it reads 0 characters. So if(n==0) conditions true.

Task 4a: Writing to a file without opening it
Write a program p4a.c which takes a file name as command line argument. Parent opens file and forks a child process. Both processes write to the file, “hello world! I am the parent” and “hello world! I am the child”. Verify that the child can write to the file without opening it. The parent process should wait for the child process to exit, and it should display and child exit message. Refer sample outputs.

soln:

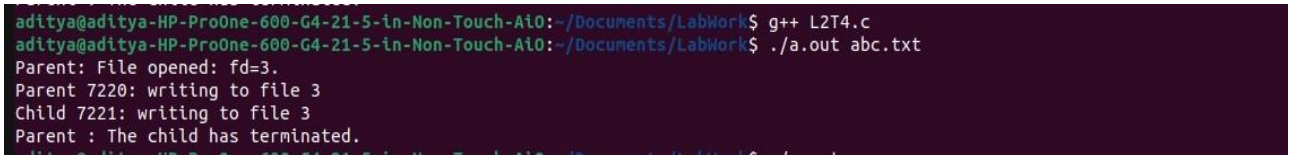
```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <stdlib.h>
#include <fcntl.h>
#include <string.h>

int main(int argc, const char **argv) {
        int fd = open(argv[1], O_RDWR | O_CREAT);
         if (fd == -1) {
      perror("open");
      return 1;
        }
        fprintf(stdout, "Parent: File opened: fd=%d.\n", fd);
        pid_t pid = fork(); // create child process
        if (pid == 0) {
                fprintf(stdout, "Child %d: writing to file %d\n", getpid(), fd);
                const char *hello_child = "hello world! I am the child\n";
                write(fd, hello_child, strlen(hello_child));
                exit(0);
        }
        else {
                fprintf(stdout, "Parent %d: writing to file %d\n", getpid(), fd);
                const char *hello_parent = "hello world! I am the parent\n";
                write(fd, hello_parent, strlen(hello_parent));
                pid_t child = wait(NULL);
                fprintf(stdout, "Parent : The child has terminated.\n");
                exit(0);
```

```
        }
}
```

above code will take file name as command line argument.
argv[1] contain file name,
open will open the file and O_RDWR is flag to open file in read write mode, O_CREAT is flag to create file if not exist.

The `open` function returns a file descriptor (`int fd`) if the operation is successful, which you can use to perform subsequent read and write operations on the opened file. If the `open` function fails, it returns -1, indicating an error.

Using fd parent will write to file and wait until child completes using wait call. Once child write to file and terminates, call return to wait and then parent completes its execution and terminates.



Task 4b: Input file re-direction magic

Write a program p4b.c which takes a file name as a command line argument. The program should print content of the file to stdout from a child process. The child process should execute the mycat program.

Cannot use any library functions like printf, scanf, cin, read, write in the parent of child process.

Hint: close of a file results in it's descriptor to be reused on a subsequent open.

Note: Do not make any modifications in mycat.c

soln:

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

int main(int argc, const char **argv) {
	close(0);
	int fd = open(argv[1], O_RDWR | O_CREAT);
	fprintf(stdout, "Parent: File opened: fd=%d.\n", fd);
	pid_t pid = fork();
	if (pid == 0) {
		char *args[] = {NULL};
		execvp("./mycat", args);
		exit(0);
	}
	else {
		pid_t child = wait(NULL);
		fprintf(stdout, "\nParent : The child has terminated.\n");
		exit(0);
	}
}
```

`close(0)`: It closes the standard input (file descriptor 0) for the parent process.
But again parent is opening the file so same file descriptor 0 will be reused for this file.
`pid_t pid = fork();`: This line forks the process into a parent process and a child process. The child process inherits a copy of the parent's file descriptors, including the `fd` file descriptor. Both the parent and the child will continue executing from this point.
Child process (`pid == 0`): The child process is executed after the fork. It uses the `execvp` function to replace its current image with the "mycat" program, passing no arguments (`args` array is set to `NULL`). This runs the "mycat" program, which will read and write the file content to std output.