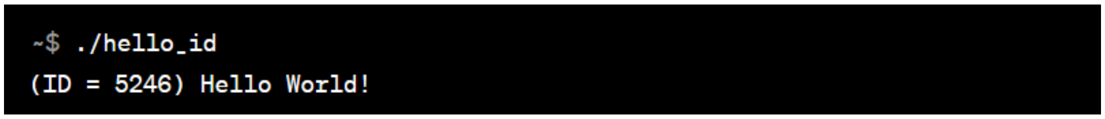
CS 321 – Operating Systems
Computer Science Dep.
Assiut University

Chapter 5 - Sheet

Q1: Write a program in C that prints the process ID alongside “Hello World.” Utilize the `getpid()` function included in `unistd.h` library.

Here is how your program should behave:

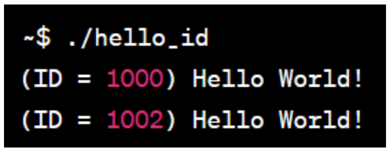


```
~$ ./hello_id
(ID = 5246) Hello World!
```

Note: *“5246” is an arbitrary value returned by $getpid()$, your process ID value will be different.*

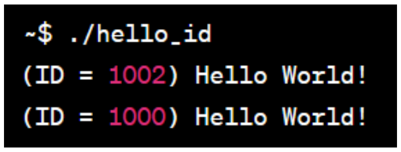
Q2: Develop a C program that utilizes the fork() system call in the first line. Then, on the second line, print the process ID alongside the phrase "Hello World."

The program should behave as follows:



```
~$ ./hello_id
(ID = 1000) Hello World!
(ID = 1002) Hello World!
```

Or

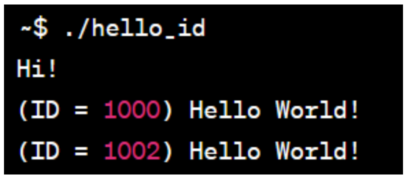


```
~$ ./hello_id
(ID = 1002) Hello World!
(ID = 1000) Hello World!
```

Note: *the output is non deterministic, the order in which the parent and child execute after the $fork()$ call is not guaranteed.*

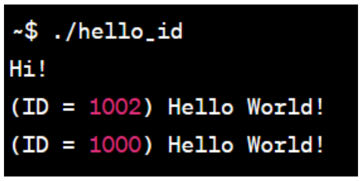
Q3: Develop a C program that prints a "Hi" message on the first line. Then, on the next line, use the fork() system call to create a new process. Finally, on the last line, print the process ID alongside the phrase "Hello World."

The program should behave as follows:



```
~$ ./hello_id
Hi!
(ID = 1000) Hello World!
(ID = 1002) Hello World!
```

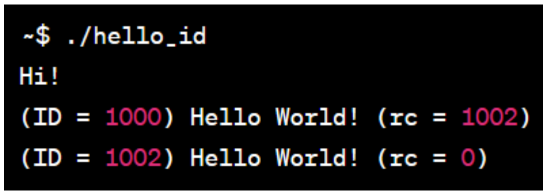
Or



```
~$ ./hello_id
Hi!
(ID = 1002) Hello World!
(ID = 1000) Hello World!
```

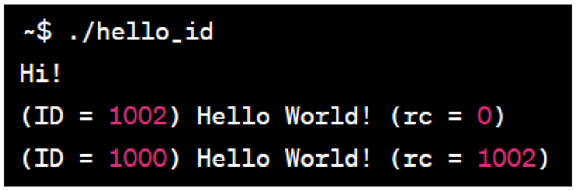
Q4: Create a C program that prints a "Hi" message on the first line. Then, on the next line, use the fork() system call to create a new process and store the return code from fork() in an integer variable called `rc`. Finally, on the last line, print the process ID alongside the phrase "Hello World!" followed by the value of `rc`.

The program should behave as follows:



```
~$ ./hello_id
Hi!
(ID = 1000) Hello World! (rc = 1002)
(ID = 1002) Hello World! (rc = 0)
```

Or



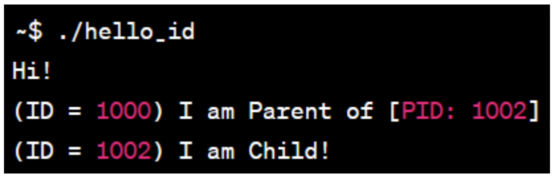
```
~$ ./hello_id
Hi!
(ID = 1002) Hello World! (rc = 0)
(ID = 1000) Hello World! (rc = 1002)
```

CS 321 – Operating Systems
Computer Science Dep.
Assiut University

Q5: Develop a C program that prints a "Hi" message on the first line. Then, on the next line, use the `` `fork()` `` system call to create a new process and store the return code from `` `fork()` `` in an integer variable called `` `rc` ``. Utilize the value of `` `rc` `` to differentiate between the parent (original) process and the child (copy) process.

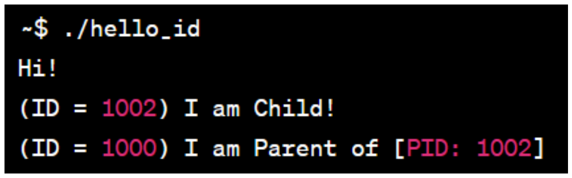
In the parent process, print "I am Parent of [PID: 1002]", where 1002 is the process ID of the child process. In the child process, print "I am Child!" instead of the "Hello World" message.

The program should behave as follows:



```
~$ ./hello_id
Hi!
(ID = 1000) I am Parent of [PID: 1002]
(ID = 1002) I am Child!
```

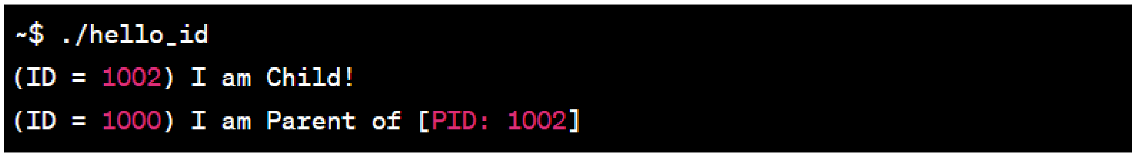
Or



```
~$ ./hello_id
Hi!
(ID = 1002) I am Child!
(ID = 1000) I am Parent of [PID: 1002]
```

Q6: Develop a C program that utilizes the `` `fork()` `` system call to create two processes. Enforce the order of execution by making the parent (original) process wait for the execution of the child (copy) process. Use the `` `wait(NULL)` `` function, which is included in the `` `sys/wait.h` `` library, to achieve this synchronization.

The program should behave as follows:

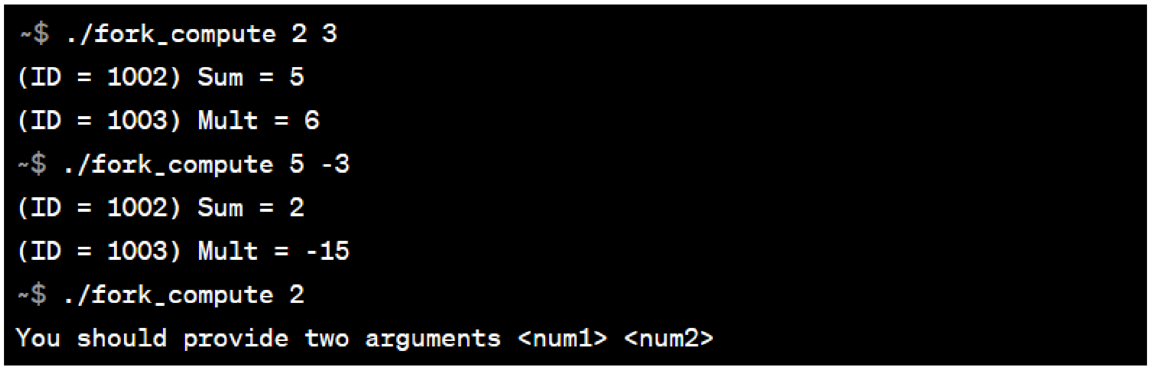


```
~$ ./hello_id
(ID = 1002) I am Child!
(ID = 1000) I am Parent of [PID: 1002]
```

Note: *now the output of the program is deterministic.*

Q7: Develop a C program that accepts two integer arguments from the command line. The program should use the `` `fork()` `` system call to create another process. *Each process should perform a distinct operation*: one process should compute the sum of the two numbers, while the other should compute their multiplication. The program should print the process ID and the result of each operation. If the user provides less or more than two numbers, the program should display an error message.

Here are some examples of how the program should behave:



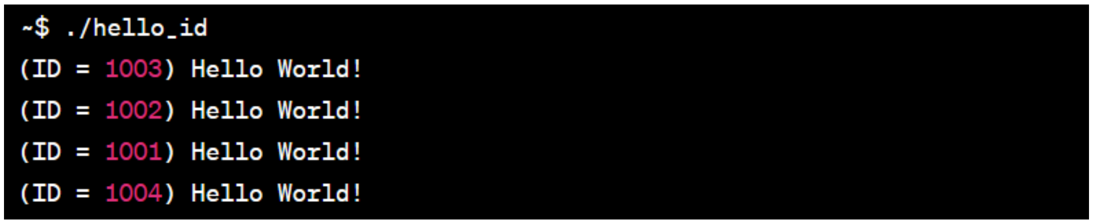
```
~$ ./fork_compute 2 3
(ID = 1002) Sum = 5
(ID = 1003) Mult = 6
~$ ./fork_compute 5 -3
(ID = 1002) Sum = 2
(ID = 1003) Mult = -15
~$ ./fork_compute 2
You should provide two arguments <num1> <num2>
```

Note: *the output of the program is non-deterministic.*

CS 321 – Operating Systems
Computer Science Dep.
Assiut University

Q8: Develop a C program that uses the `fork()` system call twice consecutively to create four processes. Then, in the last line, print the process ID alongside the phrase "Hello World!"

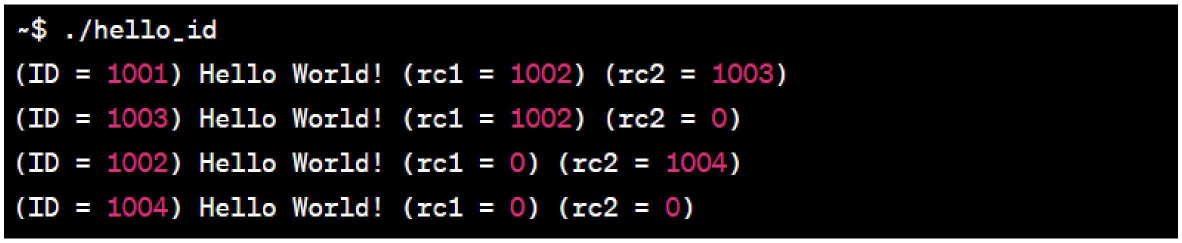
One possible output (out of 24) of the program is:



```
~$ ./hello_id
(ID = 1003) Hello World!
(ID = 1002) Hello World!
(ID = 1001) Hello World!
(ID = 1004) Hello World!
```

Q9: Develop a C program that uses the `fork()` system call twice consecutively to create four processes. Store the return codes of the first and second `fork()` calls in variables `rc1` and `rc2`, respectively. Then, on the last line, print the process ID alongside "Hello World!" and the values of `rc1` and `rc2`.

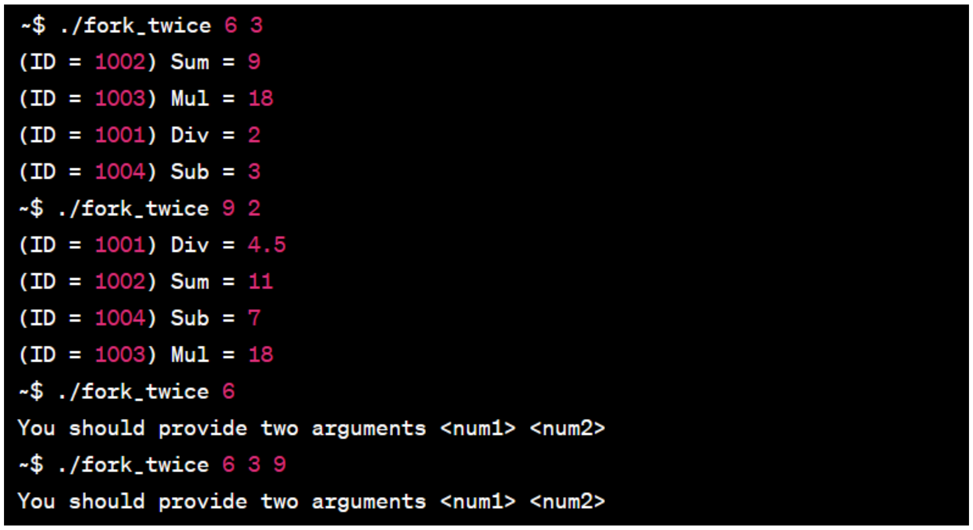
One possible output of the program (out of 24 possible outputs) is:



```
~$ ./hello_id
(ID = 1001) Hello World! (rc1 = 1002) (rc2 = 1003)
(ID = 1003) Hello World! (rc1 = 1002) (rc2 = 0)
(ID = 1002) Hello World! (rc1 = 0) (rc2 = 1004)
(ID = 1004) Hello World! (rc1 = 0) (rc2 = 0)
```

Q10: Develop a C program that exclusively accepts two integer arguments from the command line. The program should generate four processes using only two `fork()` calls and no more than that. Each process should execute a distinct arithmetic operation, namely addition, multiplication, division, or subtraction, on the input integers. To differentiate between the four processes, the program should utilize the values of `rc1` and `rc2`. If the user enters a number of integers other than two, the program should output an error message.

Here are some examples of the expected program behavior:

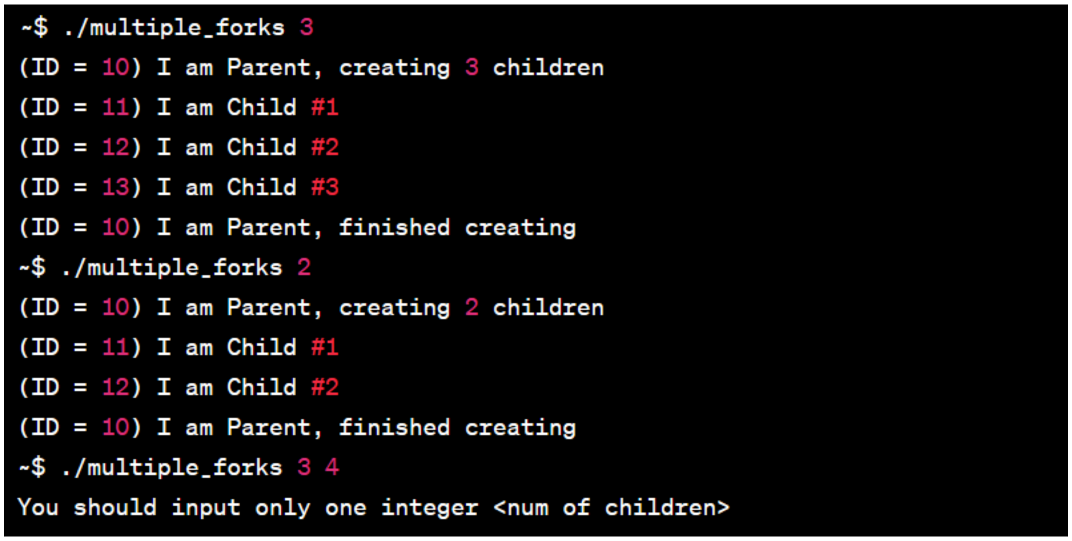


```
~$ ./fork_twice 6 3
(ID = 1002) Sum = 9
(ID = 1003) Mul = 18
(ID = 1001) Div = 2
(ID = 1004) Sub = 3
~$ ./fork_twice 9 2
(ID = 1001) Div = 4.5
(ID = 1002) Sum = 11
(ID = 1004) Sub = 7
(ID = 1003) Mul = 18
~$ ./fork_twice 6
You should provide two arguments <num1> <num2>
~$ ./fork_twice 6 3 9
You should provide two arguments <num1> <num2>
```

CS 321 – Operating Systems
Computer Science Dep.
Assiut University

Q11: Develop a C program that exclusively accepts a single integer argument from the command line. The program should employ the input value to generate a specified number of child processes, with each child being a copy of a single, parent process. On the first line of output, the parent process should print "I am a parent, creating 5 children..." as an example, where "5" represents the input value obtained from the terminal. Following this, each child process should sequentially print "I am Child #1", "I am Child #2", and so forth, ensuring that the output is deterministic, and the order of child processes is increasing. Once the children's processes are created, the parent should print "I am parent, finished creating" in the last line.

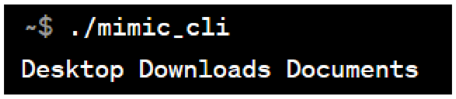
Here are examples of how the program should behave:



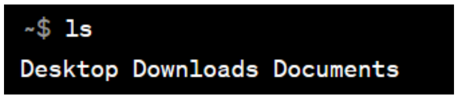
```
~$ ./multiple_forks 3
(ID = 10) I am Parent, creating 3 children
(ID = 11) I am Child #1
(ID = 12) I am Child #2
(ID = 13) I am Child #3
(ID = 10) I am Parent, finished creating
~$ ./multiple_forks 2
(ID = 10) I am Parent, creating 2 children
(ID = 11) I am Child #1
(ID = 12) I am Child #2
(ID = 10) I am Parent, finished creating
~$ ./multiple_forks 3 4
You should input only one integer <num of children>
```

Q12: Develop a C program that replicate the functionality of the "ls" command, but from within the program itself, rather than invoking it from the command line. Utilize the `execvp()` function included in `unistd.h` library.

Here is the expected behavior of the program vs CLI Command:



```
~$ ./mimic_cli
Desktop Downloads Documents
```

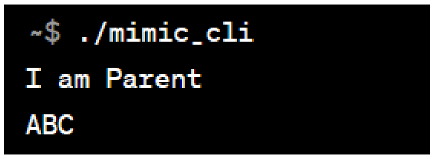


```
~$ ls
Desktop Downloads Documents
```

Note: *ls is a program code part of the OS, you can try programs you've made as well.*

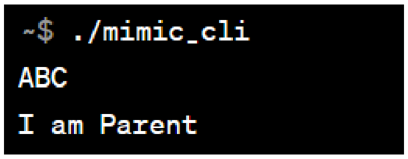
Q13: Develop a C program that utilizes the `fork()` system call to create two processes. In the parent process, print "I am Parent," while in the child process, use the `execvp()` system call to replicate the functionality of the *"echo"* command, but from within the program, with the string "ABC," instead of using it directly from the command line.

Here are examples of how the program should behave:



```
~$ ./mimic_cli
I am Parent
ABC
```

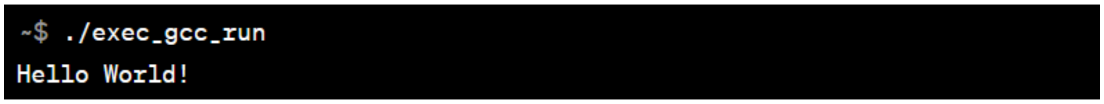
or



```
~$ ./mimic_cli
ABC
I am Parent
```

Q14: Develop a C program that utilizes the `fork()` system call to create two processes. One process will be responsible for compiling a file named "$hello.c$" into an executable file named "hello_execvp," using `execvp()` while the other process will execute the resulting "$hello\_execvp$" program using `execvp()` as well. Please ensure that "$hello.c$" is created before using it inside your code for this question. Caution: note that the program must maintain the order of operations to guarantee determinism; thus, the compilation must precede the execution.

Here is the expected behavior of the program:



```
~$ ./exec_gcc_run
Hello World!
```

CS 321 – Operating Systems
Computer Science Dep.
Assiut University

Learning Objectives:

- **Q1: Understand the concept of a unique process identifier (PID) assigned to each process in RAM, allowing the operating system to differentiate between multiple processes.**

- **Q2: Comprehend the functionality of the `` `fork()` `` system call, which creates a duplicate of the calling process with almost identical address space, resulting in two distinct processes.**

- **Q3: Recognize that the Program Counter (PC) Register is also duplicated during `` `fork()` ``, causing the child process to start its execution as if returning from the `` `fork()` `` system call.**

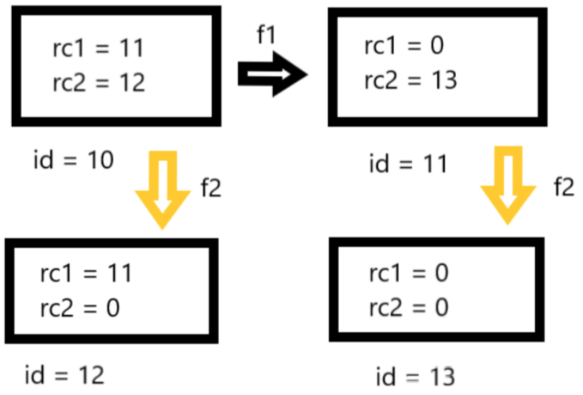
- **Q4: Grasp the nature of the return code (rc) from the `` `fork()` `` system call, where the parent process receives the PID of the child process as the return code, while the child process receives 0 as the return code.**

- **Q5: Utilize the return code (rc) to distinguish between the original (parent) process and the child (copy) process. Also avoid relying on specific PID values or assumptions about the relationship between parent and child PIDs.**
    - **It's not guaranteed that the child process ID will be always greater than the parent process ID, thus you can't use PID to differentiate between the parent and child process.**

- **Q6: Utilize the `` `wait()` `` system call to enforce the order of execution between parent and child processes.**

- **Q7: Recognize that programs using the `` `fork()` ``system call can also process data to accomplish specific tasks.**

- **Q8-Q9: Understand the concept of consecutively using multiple `fork()` system calls, leading to a doubling of the number of processes each time. Demonstrated through a practical example demonstrating the creation of a copy with the same code and data, including the variable $rc1$, and $rc2$.**



- **Q10: a practical example demonstrating how to distinctly differentiate between four processes distinctly. Helping you distinguishing between a broader range of processes.**

- **Q11: a practical example of enforcing the order of execution among a wider variable range of child processes.**

- **Q12: Comprehend the `execvp()` system call, which allows replacing the code inside a process with another code, either included in the operating system (e.g., "ls," "echo") or executable programs written by the user (e.g., "./echo", "./hello").**

- **Q13: Realize the potential of forking a child process and replacing its code with something different from the original process, with further understanding found in the book's section on "5.4 Why? Motivating The API."**

- **Q14: Provide a practical example that combines the use of `fork()`, `execvp()`, and `wait()` system calls. The program will compile and execute an existing file in one step.**