# Shell Design

In this lab, you will design a small OS shell. It waits for you to enter command from a set of commands that we will specify below. If the command is a built-in command, as indicated below, your process will just execute it. If not, then it must do two things: **fork** a new process, then uses **execve** to make this new child process morph to the new process. The non-built-in commands will be basic Linux commands that you do no need to write, just launch them using the combination for fork and execve. If the user types the word "exit" then the program ends. Otherwise, another iteration will be executed.

The main loop, in pseudo-code will be as follows. Be careful: the following is not a full-fledged C code. So don't assume that copying and pasting a statement will work. You need to write a real program. The following part is just to help you see the big picture. However, the printed statements must be printed as they are.

```
while(1){

  printf("lab1> ");
  fgets()  ← get the input from the user. You can use a different API if you want.
  printf("Parent Process %d\n", getpid());

  if (the string entered by the user is a built-in command)
  {
     Check the string and based on it, do the corresponding tasks as stated below.
     If the built-in command is "exit"  then ends the whole process using exit(0);
   }
   else
   {
       pid = fork();
       if( pid == 0 )  ← the created child process
       {
          printf("Child process %d will execute the command %s\n", ....); ← print process
          ID of the child process and the string entered by the user.

         execve() ← use the string entered by the user to transform the child process to
         the new process.

        if execution of the child process reaches this point, it means execve failed. In that
        case, print "Command Not Found!\n" and exits.
       }
    }
wait till the child process finishes
}
```

Some Important C APIs that may be useful to you and you need to check how to use before implementing this lab. With the exception of fork(), execve(), and wait(), you don't have to use all those APIs but they may be helpful depending on how you design your program. Of course, feel free to use any other APIs you want.

- fork()
- execve()
- wait()
- getpid()
- strcmp()
- strcat()
- strcpy()
- fgets()
- prtintf()

## The Built-in Commands

If the user enters one of the following commands, then do not fork and execve but execute them directly in your current process. You can implement each command as a function, except exit of course, and you call that function when you compare the user input string with the commands below. The commands are case sensitive.

- **printid** : print on the screen **The ID of the current process is *X*** where x is the process ID.
- **exit :** ends the program and exits.
- **greet:** prints "Hello\n" on the screen

## Commands That Require fork and execve

- **/bin/ls** : lists the files in the current directory
- **/bin/pwd** : prints the path of the current working directory
- **/bin/ps** : prints the status of the current processes
- **/bin/date :** prints current date and time.
- **/bin/lscpu**: prints information about the current processes of the machine.

The user will type the command only and not the whole path. That is, the user will type: ls, pwd, etc. You will need to add the /bin/ to the user input and use that as the first argument for execve(), while the second argument is a pointer to an array of string (review the arguments of the main() function in C that contains the process name in element 0, and you can put NULL in the third argument.

Example: Suppose the user inputs: ls
Then the inputs to execve will be: execve("/bin/ls", progname, NULL);
Where progname was declared as: char * argv[]= {"ls", NULL};

## Important Notes:

1. All the labs in this course are to be implemented in C.
2. You may want to include the following header files at the beginning of your program to be able to use string operations, if you want, and the process related APIs.
   ```
   #include <stdlib.h>
   #include <stdio.h>
   #include<unistd.h>
   #include<sys/types.h>
   #include<sys/wait.h>
   ```
3. Name your source file with your NetIT. So, if your NetID is xyz, the file must be named xyz.c
4. You compile your code with: **gcc -Wall -o mykernel -Og xyz.c**
5. The compilation and execution must be done on Courant CIMS machines. You may need to remote login to them. It won't work on your MAC or Windows machines even if you have the nice terminal that looks like Linux.