

## More on Linux Processes

### Objective

- To change the executable image of a process and get it to do a totally different job using `exec()`
- To investigate what parts of a process scope/PCB persists after `fork()` and/or after `exec()` system calls

### Description

A process is a program in execution. It is the basic unit for running applications and performing work in a computer system. An OS creates the data structures it needs to track and manage a process throughout its life cycle (from inception all the way through its termination). To do so, the OS should provide system calls to create a process, change its context so it does something different than its parent, and send the process's status information back to its parent and more.

Linux allows a process to create a new process that is a replica of itself using the `fork()` system call. The parent process, using the `wait()` system call, can wait for its newly created child until the child finishes its work. The child process can perform a new task by changing its context to a new executable image using a version of `exec()`.

- `int execl(const char *path, const char *arg, ...);`
- `int execlp(const char *file, const char *arg, ...);`
- `int execlx(const char *path, const char *arg, ..., char * const envp[]);`
- `int execv(const char *path, char *const argv[]);`
- `int execlp(const char *file, char *const argv[]);`
- `int execlpx(const char *file, char *const argv[], char *const envp[]);`
- `pid_t fork(void);` <http://man7.org/linux/man-pages/man2/fork.2.html>
- `pid_t vfork(void);` <http://man7.org/linux/man-pages/man2/vfork.2.html>
- `pid_t getpid(void);` <http://man7.org/linux/man-pages/man2/getpid.2.html>
- `pid_t getppid(void);`
- `pid_t wait(int *wstatus);` <http://man7.org/linux/man-pages/man2/waitpid.2.html>
- `pid_t waitpid(pid_t pid, int *wstatus, int options);`
- `int open(const char *pathname, int flags);` <https://man7.org/linux/man-pages/man2/open.2.html>
- `int close(int fd);` <https://man7.org/linux/man-pages/man2/close.2.html>
- `int read(int fd, void *buf, size_t count);` <https://man7.org/linux/man-pages/man2/read.2.html>
- `int write(int fd, const void *buf, size_t count);` <https://man7.org/linux/man-pages/man2/write.2.html>

### Submission:

- *One compressed file that contains all C and pdf files*
- *Submit the answers to the P1 in pdf format*
- To get used to system calls please pay attention to the following:
  - o Check the success of every system call you use
  - o Make sure a parent process waits for all its children
  - o When using pipes, make sure to close the side the process will not use

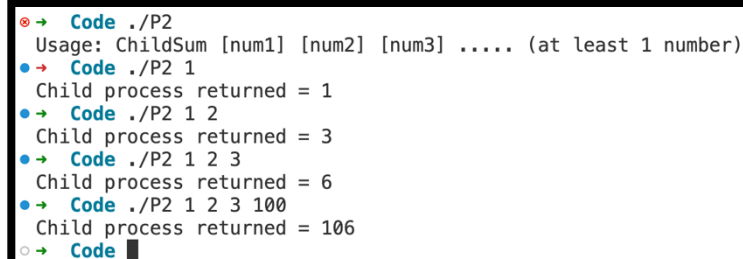
**1. More on the life cycle of a Linux process:**

Run the program `part1.c` multiple times and use the output to justify the answers to the following questions:

- Do newly created child processes, C1 and C2, have their own copies of the different variables declared in the parent process (e.g. `index`, `now`, `value`, `outBuffer`, `outFD`)?
- Is there a fixed order of execution between child processes C1 and C2? And how do you explain that?
- Modify the code so C1 will complete printing its output before C2 starts printing its own output.

**2. Return a value from child process to parent process (lastnameChildCP.c):**

Write a C program that gets a list of numbers through the command line, creates a child process, passes it the list of numbers, and waits for the child to compute the sum of these numbers. The parent process gets the sum result from the child process, prints it, and then terminates (see sample runs below).



```
➤ Code ./P2
Usage: ChildSum [num1] [num2] [num3] ..... (at least 1 number)
➤ Code ./P2 1
Child process returned = 1
➤ Code ./P2 1 2
Child process returned = 3
➤ Code ./P2 1 2 3
Child process returned = 6
➤ Code ./P2 1 2 3 100
Child process returned = 106
➤ Code
```

**3. What PCB information is retained on Exec:**

When an `exec` version is executed, most of the process PCB is retained, including the PID, and the PPID. Depending on the `exec` version, the environment variables could also be retained.

Write 2 programs (`testMyExec.c` and `myExec.c`) to test if the above parts of the process PCB are retained on `exec`:

**a) Test PID and PPID:**

`testMyExec.c`: should print the process PID and PPID then call `execlp` and pass it the executable version of `myExec`.

`myExec.c`: should print its PID and PPID and exit.

Compare the outputs of both print statements and decide if the PID and PPID are retained after `exec`.

**b) Test open file descriptors:**

`testMyExec.c`: should open a file for writing. Write the process PID and PPID to the file then call `execlp` and pass it the executable version of `myExec` and the file descriptor for the file that was open for writing.

`myExec.c`: should write its PID and PPID to the file and exit. Check if the file exists and what data does it have.

**c) Test environment variables:**

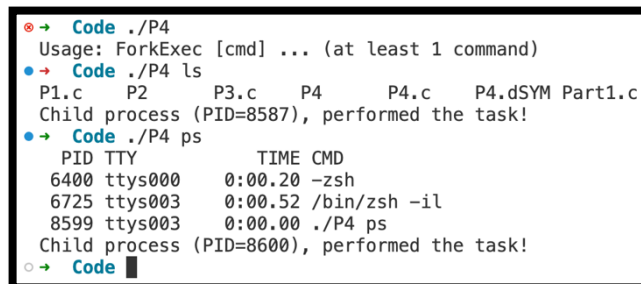
`testMyExec.c`: change `envp[1]` to be your first and last name. Print the environment variables of the original process, then call `execle` and pass it the executable version of `myExec` and `envp`.

`myExec.c`: Print the environment variables of the process after `exec` then exit. Compare the environment variables and see if they are the same.

**4. Combined Fork and Exec:**

Write a function `ForkExec(...)` that combines the `fork()` and `exec()` system calls. The function takes a NULL terminated array of strings, creates a process to run the new executable and returns after the child process terminates. For example, running `ForkExec(temp)` with `temp = {"head", "-n", "3", "file.txt"}` should print the first 3 lines of `file.txt`. Test your code by the running the following `main.c`:

```
int ForkExec(char *temp[])
{
    ...
}
void main(int argc, char *argv[])
{
    if(argc < 2)
    {
        printf("Usage: ForkExec [cmd] ... (at least 1 command)\n");
        return -1;
    }
    int workProcessID = ForkExec(argv);
    printf("Child process (PID=%d), performed the task!\n", workProcessID);
    return 0;
}
```



```

➔ Code ./P4
Usage: ForkExec [cmd] ... (at least 1 command)
➔ Code ./P4 ls
P1.c  P2      P3.c  P4      P4.c  P4.dSYM Part1.c
Child process (PID=8587), performed the task!
➔ Code ./P4 ps
  PID TTY          TIME CMD
 6400 ttys000    0:00.20 -zsh
 6725 ttys003    0:00.52 /bin/zsh -il
 8599 ttys003    0:00.00 ./P4 ps
Child process (PID=8600), performed the task!
➔ Code █

```