

# UNIX Programming Guide – Part 5

---

## Content Summary

1. Implementing a shell- Version 1
2. fork() system call
  - a. printf() and fork
  - b. Forking Cost
  - c. Fork Bombs
3. Improving the shell – Version 2

## Implementing a shell - Version 1

### Shell

A shell, in its simplest form is a program that takes an input string from the user and executes some program corresponding to the input string. To execute any program, we have to give it environment and arguments

1. The arguments to the program are obtained by parsing the input string
2. Assume the environment variable used is the default

For simplicity, also assume that the shell does not handle piping, background processes, sequential execution or redirection. Hence a valid input to such a shell can consist only of a sequence of words separated by blanks or tabs.

Since we will use the default PATH variable and environment in our program, `execle()`, `execve()`, `execlp()` and `execvp()` calls will not be required. A shell program does not know before-hand (during compile-time) the number of arguments that it needs to pass to the new program. Hence making an `execl()` call to invoke the new program is not possible. The best option in this case would be the `execv()` call.

### *Program to implement a shell – version 1*

```
# include<string.h>
# include<stdio.h>
# include<errno.h>

# define MAXARG 20
# define MAXCMD 100

int make_args(int *argc_ptr, char *argv[], int max) {
    static char cmd[MAXCMD];
    char *cmd_ptr;
```

```

    int i=0;
    fgets(cmd,sizeof(cmd),stdin);
    cmd_ptr=cmd;
    for(i=0;i<max;i++){
        if((argv[i]=strtok(cmd_ptr," \t\n"))==NULL)
            break;
        cmd_ptr=NULL;
    }
    *argc_ptr=i;
    return 1;
}

int main(void){

    char *argv[MAXARG];
    int argc;
    int i=0;

    while(1){

        printf("@ ");
        if(make_args(&argc,argv,MAXARG) && argc > 0){
            execvp(argv[0],argv);
            printf("Execution unsuccessful");
            printf("%s",strerror(errno));
        }
        else
            printf("Error constructing arguments");
    }

    return 0;
}

```

The default environment is passed to the program. As for the arguments, the function `make_args` constructs the arguments by parsing the input string. The function splits the input string based on tab spaces or new line and returns the split strings to the `argv` array. Recall , the first argument to `execvp()` is the filename followed by the argument array. `execvp()` call searches for this executable using the `PATH` variable. Also recall, a return from the `execvp()` call means that the call has been unsuccessful.

The “`errno`” in UNIX offers a useful debugging facility. When any system call returns an error, it sets “`errno`” to a value that corresponds to the error. `strerror(errno)` is a function that prints the description(string) that corresponds to a `errno`.

```
prithvi.bits-pilani.ac.in - PuTTY
varuni@blade6:~/unixstuff/process_stuff/ProcessDemo$ ./a.out
@ ls
a.out      execdemo.c  first_fork.c  fork_loop_changed.c  nice.c      samp
environm.c execl_print.c fork_loop.c   fork_use.c           process_id.c shel
varuni@blade6:~/unixstuff/process_stuff/ProcessDemo$ ./a.out
@ cal 3 2011
      March 2011
Su Mo Tu We Th Fr Sa
      1  2  3  4  5
 6  7  8  9 10 11 12
13 14 15 16 17 18 19
20 21 22 23 24 25 26
27 28 29 30 31

varuni@blade6:~/unixstuff/process_stuff/ProcessDemo$ ./a.out
@ wc execdemo.c
 10  16 146 execdemo.c
varuni@blade6:~/unixstuff/process_stuff/ProcessDemo$ █
```

When this program is executed, it prompts the user to enter a command and executes the command and displays the output. Run the program with invalid commands and user defined executables.

Apart from the limited capabilities, the principal problem with this shell is that it terminates after executing a single command successfully. However, practical shells keep reading user input till the user explicitly chooses to terminate the shell. Note that this disadvantage is because there is no return from a successful `exec()` and hence the loop in the program cannot run more than once. To overcome this disadvantage and build a better shell, we use the `fork()` system call.

## fork() System call

`fork()` call creates a “new” process. The child process’ context will be the same as the parent process. After a `fork()` call, two copies of the same context exist, one belonging to the child and another to the parent. Contrast this to `exec()`, where a single context will exist because of child context overwriting the parent.

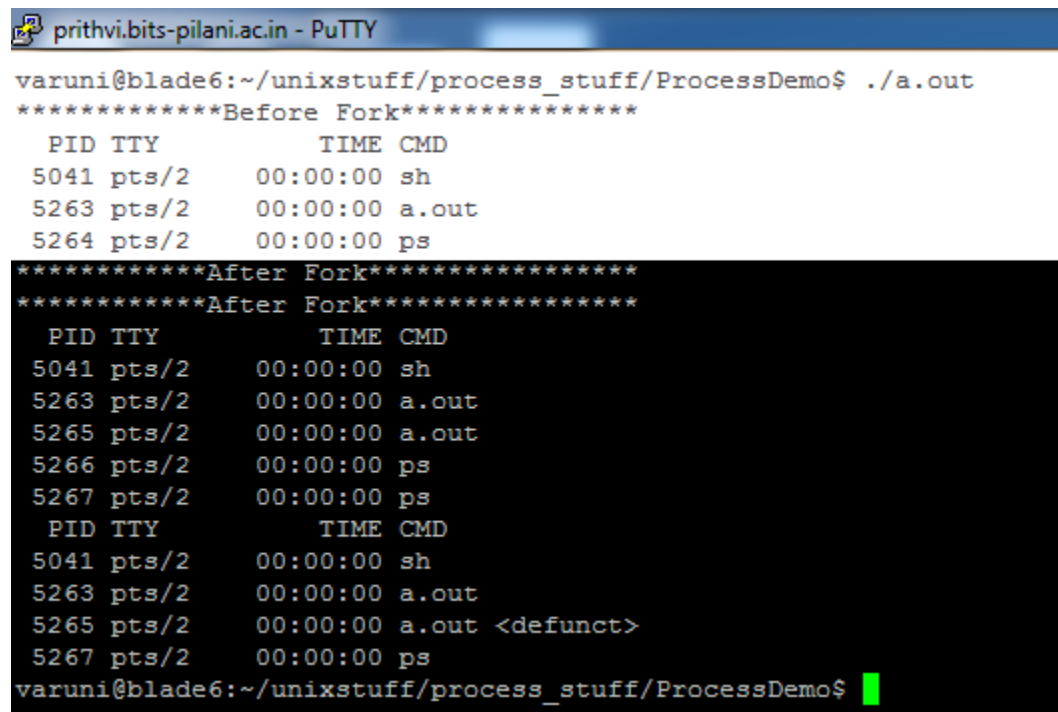
```
# include<unistd.h>
int fork(void);
/*Returns child process-ID and 0 on success and -1 on error */
```

The return value of `fork()` is of particular interest. After `fork()` returns both the child and the parent receive the return. The return value is however different in both cases. The child receives a 0 as return value from `fork()` and the parent receives the process-ID of the child.

### *Program to demonstrate simple `fork()` usage*

```
1# include<stdio.h>
2int main(void){
3    printf("***** Before Fork*****\n");
4    system("ps");
5
6    fork();
7
8    printf("***** After Fork *****\n");
9    system("ps");
10   return 0;
11}
```

Note that the “ps” command lists all the active processes in the system at that point of time. An example output would look like below:



```
prithvi.bits-pilani.ac.in - PuTTY
varuni@blade6:~/unixstuff/process_stuff/ProcessDemo$ ./a.out
*****Before Fork*****
  PID TTY          TIME CMD
 5041 pts/2        00:00:00 sh
 5263 pts/2        00:00:00 a.out
 5264 pts/2        00:00:00 ps
*****After Fork*****
*****After Fork*****
  PID TTY          TIME CMD
 5041 pts/2        00:00:00 sh
 5263 pts/2        00:00:00 a.out
 5265 pts/2        00:00:00 a.out
 5266 pts/2        00:00:00 ps
 5267 pts/2        00:00:00 ps
  PID TTY          TIME CMD
 5041 pts/2        00:00:00 sh
 5263 pts/2        00:00:00 a.out
 5265 pts/2        00:00:00 a.out <defunct>
 5267 pts/2        00:00:00 ps
varuni@blade6:~/unixstuff/process_stuff/ProcessDemo$
```

In the source code, lines 1-6 are executed by only one process - the parent. After the `fork()` call, two processes exist both containing the same source code. Note that the child process starts execution from line 7. Lines 7-11 will be executed twice -

once by the parent and once by the child. In the screenshot above, the white area shows the output before the fork() call, executed by the parent. There is a shell process running with pid -5041 , the process corresponding to the "ps" command - 5264, and also the parent process itself - 5263. The output in the black-area of the screenshot, shows execution by both the parent and child. Note that there is no sequence or order in which the code of the parent and child will be executed. Hence the output looks a little confusing. Make the following modifications to the code and then run the program again.

```
1# include<stdio.h>
2int main(void){
3    int ret;
4    printf("***** Before Fork*****\n");
5    system("ps");
6
7    ret=fork();
8    if(ret==0){
9        printf("***** After Fork *****\n");
10       system("ps");
11    }
12    else if(ret>0)
13        wait();
14    return 0;
15}
```

Lines 1-7 would be executed only by the parent. After fork() call both the processes would have instructions corresponding to lines 8-15. A crucial point here is that the value of 'ret' would be 0 in the child and a non-zero process-id in the parent. Hence the if condition in line 8 would be false in the parent process and true in the child process. Lines 9-10 would be executed by only the child. Condition in line 12 would be true only for the parent and hence wait() would be executed only by parent. (wait() call would be covered later in this lab sheet). Lines 14-15 would be executed

by both the parent and child.

```
prithvi.bits-pilani.ac.in - PuTTY
varuni@blade6:~/unixstuff/process_stuff/ProcessDemo$ ./a.out
*****Before Fork*****
  PID TTY          TIME CMD
 5041 pts/2        00:00:00 sh
 5323 pts/2        00:00:00 a.out
 5324 pts/2        00:00:00 ps
*****After Fork*****
  PID TTY          TIME CMD
 5041 pts/2        00:00:00 sh
 5323 pts/2        00:00:00 a.out
 5325 pts/2        00:00:00 a.out
 5326 pts/2        00:00:00 ps
varuni@blade6:~/unixstuff/process_stuff/ProcessDemo$
```

Since `fork()` has no arguments, the only reason for a fork failure would be resource exhaustion. The child process gets exact copies of most of the system-data, user-data and instruction segments. However there remain some attributes that are different for the child process and the parent.

1. Obviously, the process-id
2. The parent may have different threads running different pieces of code. However, the child process can access only the thread that executed the `fork()` call.(More about threads later)
3. The child gets duplicates of the file descriptors open in the parent. File description is shared but the file descriptors are unique. This could be visualized as two pointers pointing to the same address location. When the child changes the file offset, the change is reflected in the parent also. However if one file descriptor is closed, the other still remains.
4. The child's accumulated execution time is reset to zero. This means the child's execution time does not include the execution time of the parent. This is useful for scheduling purposes.

## printf() and fork()

Consider the output of the following program:

```
void forktest(void) {
    int pid;
    printf("start of test\n");
    pid=fork();
    printf("Return at %d",pid);
}
```

```
}
```

Output:

```
start of test
Return at 5412
Return at 0
```

In the above case, the return from the parent got printed first and then the return from the child happened. But the execution order is not guaranteed. Now try to run the same program but with the output redirected to a file (`./a.out > fileout.txt`)

Output:

```
start of test
Return at 5412
start of test
Return at 0
```

This time the statement “start of test” got printed twice. This happens because `printf` had written its output in a buffer and the child inherited the buffer which was unflushed. Just before the program exited both the child and the parent flushed the buffer and hence the statement got printed twice.

When the output was not re-directed to a file, `printf` knew that the output device was a terminal and hence it has to behave more interactively. So it did not buffer its output in the earlier case.

## Forking Cost

`fork()` calls are enormously costly in terms of computing resources. A clone of the parent’s context is to be made which involves copying of the data segment. Usually the instruction segment (code) is not copied because the segment is read-only and can hence be shared between parent and child. The data segment to be copied can be very huge. Recall that most of the time, `fork()` is immediately followed by `exec()` – which will overwrite the existing data and code segments. Hence copying of all the huge data segment is wasted.

A scheme that overcomes this handicap is called the “copy-on-write” scheme. In this scheme, after `fork()` the parent and the child share the same data segment. This will remain so, as long as the data segment is unmodified. When the parent or child modifies a particular page, a copy of only that page is made. So, there would be two copies of modified pages while the unmodified pages would be shared.

Another scheme is provided by `vfork()`. `vfork()` is an obsolete call and has exactly the same syntax as `fork()`. The only difference between `vfork()` and `fork()` is that `vfork()` does not make a copy of the data segment. Data segment will be shared between child and the parent and both will be working\modifying on the same data space. If not carefully controlled, this could lead to serious data corruption. `vfork()` call should usually be avoided (and hence it is obsolete) and used only in situations where it certain that the `fork()` will always be followed by `exec()`.

## Fork bombs

Repeated `fork()` calls, can eventually lead to resource exhaustion and may collapse the system. This is a important concern in a multi-user environment. Repeated `fork()` calls (also known as fork bombs) by a single user would lead to denial of system resources for all other users.

### *Program to demonstrate repeated forking*

```
# include<stdio.h>

int main(void){
    int cnt=0;
    int pid;

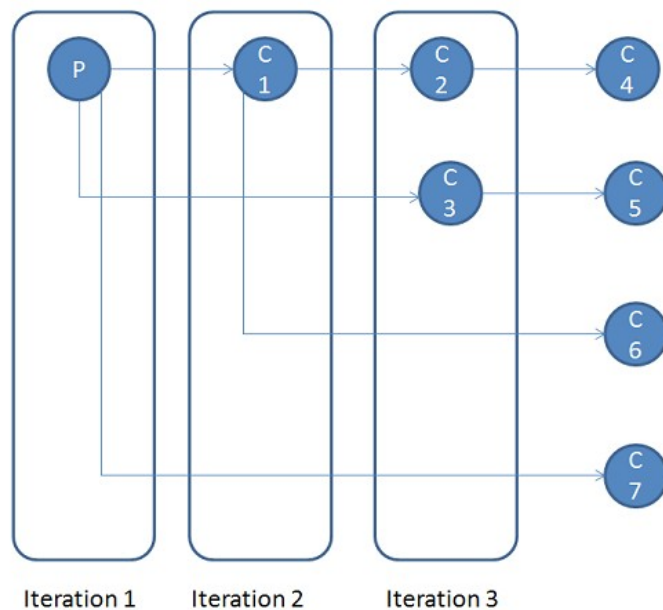
    for(cnt=0;cnt<3;cnt++){
        pid=fork();
        printf("Now in process %d\n",getpid());
    }

    return 0;
}
```

The loop in this program runs for 3 times and calls `fork()` each time. So, how many new processes would be created? 3 is the wrong answer!



The number of children would be : 7. This can be explained with the figure



*In the first iteration, only the parent process exists and it creates a single child, C1.*

*In the second iteration both P and C1 exist and each fork to give C3 and C2 respectively*

*In the last iteration P,C1,C2,C3 exist and each fork to give C7,C6,C4,C5 respectively.*

below:

### *Program to demonstrate controlled forking*

```
# include<stdio.h>
int main(void){

    int cnt=0;
    int pid;

    for(cnt=0;cnt<3;cnt++){
        pid=fork();

        if(pid==0)
            continue;
        else
            break;
    }

    printf("Process %d\n",getpid());
    return 0;
}
```

If the intent is to create n processes in a loop, then a controlled loop as shown in this program should be used.

## Combining fork() and exec()

fork() and exec() are seldom useful as stand-alone system calls. Most of the times, fork() and exec() are used as a combination. fork() is usually called first to create a new process and then exec() is called to load a fresh program into the child process. The following program illustrates how a combination of fork() and exec() calls can improve our basic shell program.

## Implementing a shell - Version 2

```
void execute(int argc, char *argv[]){
    int pid;

    switch(pid=fork()){
        case 0:
            execvp(argv[0], argv);
            printf("Child not successful\n");
            _exit(-2);
        case -1:
            printf("Unable to create child\n");
            break;
        default:
            wait(NULL);
    }
}

int main(void){
    char *argv[MAXARG];
    int argc;
    int i=0;

    while(1) {
        printf("@ ");
        if(make_args(&argc, argv, MAXARG) && argc > 0) {
            execute(argc, argv);
        }
        else
            printf("Error constructing arguments");
    }
    return 0;
}
```

The `make_args` function is the same as in version-1. Instead of making a direct call to `execv()`, here we call another function `execute()`. The `execute` method creates a new child process and runs the user specified command in the child using `execv()`. The parent merely “waits” for the child to complete and then loops again.

Note that the principal disadvantage of the version-1 shell was that it was suicidal! Every successful command caused the shell to terminate. But the version-2 shell overcomes this disadvantage by creating a child process everytime and executing the command in the child. The main program will continuously run and prompt for user-input just like a real shell.

This program contains two new system calls - `wait()` and `_exit()` which will be discussed in the next lab sheet.

## Exercise

1. Fork Bombs can be used to manipulate scheduling in a multi-user environment - Justify this statement
2. Write a program that will create a child process. Have the parent print out its pid and that of its child. Have the child print its pid and that of its parent. Have the processes print informational messages during various phases of their execution as a means of tracing them. A typical printout might contain the following output (not necessarily in this order).
  - Immediately before the fork. Only one process at this point.
  - Immediately after the fork. This statement should get printed twice.
  - Immediately after the fork. This statement should get printed twice.
  - I'm the child. My pid is XXXX. My parent's pid is XXXX.
  - I'm the parent. My pid is XXXX. My child's pid is XXXX.
3. Write a program that will create a process tree structure as shown below. Again, have the processes print informational messages to verify that their parent-child relationship is that as shown. So processes B and C should both report the same parent pid (that of A). Also, processes E, and F should both report the same parent pid (that of C) and D should report its parent as being B.

