

Example of fork() in C

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main()
{
    // make two process which run same
    // program after this instruction
    pid_t p = fork();
    if(p<0){
        perror("fork fail");
        exit(1);
    }
    printf("Hello world!, process_id(pid) = %d \n",getpid());
    return 0;
}
```

Output

```
Hello world!, process_id(pid) = 31
Hello world!, process_id(pid) = 32
```

Example 2:

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
void forkexample()
{
    pid_t p;
    p = fork();
    if(p<0)
    {
        perror("fork fail");
        exit(1);
    }
    // child process because return value zero
    else if ( p == 0)
        printf("Hello from Child!\n");

    // parent process because return value non-zero.
    else
        printf("Hello from Parent!\n");
}
int main()
{
    forkexample();
    return 0;
}
```

Output

```
Hello from Parent!
Hello from Child!
```

Note: In the above code, a child process is created. `fork()` returns 0 in the child process and positive integer in the parent process. Here, two outputs are possible because the parent process and child process are running concurrently. So we don't know whether the OS will first give control to the parent process or the child process.

Example of pipe() in C

Example 1:

For the first example, create a new C source file **1_pipe.c** and type in the following lines of codes.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

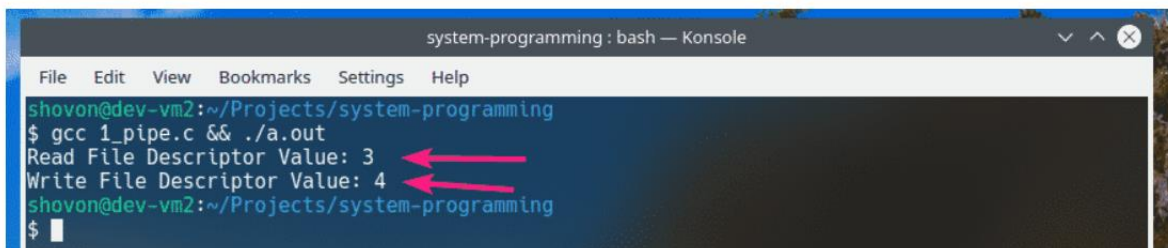
int main(void) {
    int pipefds[2];

    if(pipe(pipefds) == -1) {
        perror("pipe");
        exit(EXIT_FAILURE);
    }

    printf("Read File Descriptor Value: %d\n", pipefds[0]);
    printf("Write File Descriptor Value: %d\n", pipefds[1]);

    return EXIT_SUCCESS;
}
```

If you run the program, you should see the following output. As you can see, the value of the read pipe file descriptor **pipefds[0]** is **3** and write pipe file descriptor **pipefds[1]** is **4**.



```
system-programming : bash — Konsole
File Edit View Bookmarks Settings Help
shovon@dev-vm2:~/Projects/system-programming
$ gcc 1_pipe.c && ./a.out
Read File Descriptor Value: 3
Write File Descriptor Value: 4
shovon@dev-vm2:~/Projects/system-programming
$
```

Example 2:

Create another C source file **2_pipe.c** and type in the following lines of codes.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>

int main(void) {
    int pipefds[2];
    char buffer[5];

    if(pipe(pipefds) == -1) {
        perror("pipe");
        exit(EXIT_FAILURE);
    }

    char *pin = "4128\0";

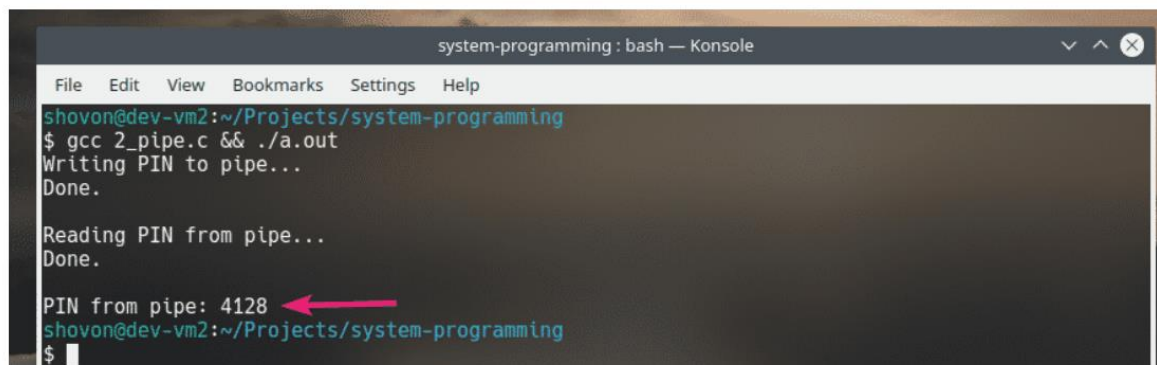
    printf("Writing PIN to pipe...\n");
    write(pipefds[1], pin, 5);
    printf("Done.\n\n");

    printf("Reading PIN from pipe...\n");
    read(pipefds[0], buffer, 5);
    printf("Done.\n\n");

    printf("PIN from pipe: %s\n", buffer);

    return EXIT_SUCCESS;
}
```

Once I run the program, the correct output is displayed as you can see.



```
system-programming : bash — Konsole
File Edit View Bookmarks Settings Help
shovon@dev-vm2:~/Projects/system-programming
$ gcc 2_pipe.c && ./a.out
Writing PIN to pipe...
Done.

Reading PIN from pipe...
Done.

PIN from pipe: 4128
shovon@dev-vm2:~/Projects/system-programming
$
```

Example of mmap() in C

Memory allocation (Example1.c)

```
#include <stdio.h>
#include <sys/mman.h>

int main(){
    int N=5;
    int *ptr = mmap ( NULL, N*sizeof(int),
        PROT_READ | PROT_WRITE, MAP_PRIVATE | MAP_ANONYMOUS, 0, 0 );

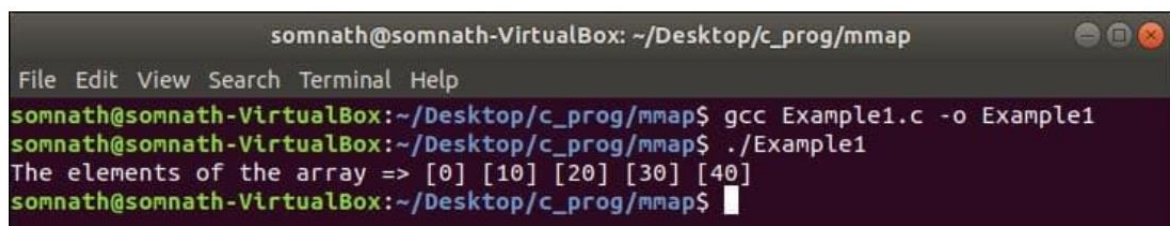
    if(ptr == MAP_FAILED){
        printf("Mapping Failed\n");
        return 1;
    }

    // Fill the elements of the array
    for(int i=0; i<N; i++)
        ptr[i] = i*10;

    // Print the elements of the array
    printf("The elements of the array => ");
    for(int i=0; i<N; i++)
        printf("[%d] ",ptr[i]);

    printf("\n");
    int err = munmap(ptr, 10*sizeof(int));
    if(err != 0){
        printf("UnMapping Failed\n");
        return 1;
    }

    return 0;
}
```



The screenshot shows a terminal window titled 'somnath@somnath-VirtualBox: ~/Desktop/c_prog/mmap'. The terminal contains the following commands and output:

```
somnath@somnath-VirtualBox:~/Desktop/c_prog/mmap$ gcc Example1.c -o Example1
somnath@somnath-VirtualBox:~/Desktop/c_prog/mmap$ ./Example1
The elements of the array => [0] [10] [20] [30] [40]
somnath@somnath-VirtualBox:~/Desktop/c_prog/mmap$
```

In Example1.c we allocate memory using mmap. Here we used PROT_READ | PROT_WRITE protection for reading and writing to the mapped region. We used the MAP_PRIVATE | MAP_ANONYMOUS flag. MAP_PRIVATE is used because the mapping region is not shared with other processes, and MAP_ANONYMOUS is used because here, we have not mapped any file. For the same reason, the *file descriptor* and the *offset* value is set to 0.

Reading file (Example2.c)

```
#include <stdio.h>
#include <sys/mman.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>

int main(int argc, char *argv[]){
    if(argc < 2){
        printf("File path not mentioned\n");
        exit(0);
    }

    const char *filepath = argv[1];
    int fd = open(filepath, O_RDONLY);
    if(fd < 0){
        printf("\n\"%s \" could not open\n",
            filepath);
        exit(1);
    }

    struct stat statbuf;
    int err = fstat(fd, &statbuf);
    if(err < 0){
        printf("\n\"%s \" could not open\n",
            filepath);
        exit(2);
    }

    struct stat statbuf;
    int err = fstat(fd, &statbuf);
    if(err < 0){
        printf("\n\"%s \" could not open\n",
            filepath);
        exit(2);
    }

    char *ptr = mmap(NULL, statbuf.st_size,
        PROT_READ|PROT_WRITE, MAP_SHARED,
        fd, 0);
    if(ptr == MAP_FAILED){
        printf("Mapping Failed\n");
        return 1;
    }
    close(fd);

    ssize_t n = write(1, ptr, statbuf.st_size);
    if(n != statbuf.st_size){
        printf("Write failed");
    }

    err = munmap(ptr, statbuf.st_size);
    if(err != 0){
        printf("UnMapping Failed\n");
        return 1;
    }
    return 0;
}
```

```
somnath@somnath-VirtualBox: ~/Desktop/c_prog/mmap
File Edit View Search Terminal Help
somnath@somnath-VirtualBox:~/Desktop/c_prog/mmap$ cat > file1.txt
Lorem Ipsum is simply dummy text
of the printing and typesetting industry.
^C
somnath@somnath-VirtualBox:~/Desktop/c_prog/mmap$ gcc Example2.c -o Example2
somnath@somnath-VirtualBox:~/Desktop/c_prog/mmap$ ./Example2 /home/somnath/Desktop/c_prog/mmap/file1.txt
Lorem Ipsum is simply dummy text
of the printing and typesetting industry.
somnath@somnath-VirtualBox:~/Desktop/c_prog/mmap$
```

In Example2.c we have mapped the file “file1.txt”. First, we have created the file, then mapped the file with the process. We open the file in O_RDONLY mode because here, we only want to read the file.

Example of signal() in C

Example 01

Within the very first POSIX example of signals, we are going to demonstrate the use of a simple signal() function to create and invoke a signal. Therefore, we have been using the signal.h library of C to utilize the signal functions, the stdio.h library is used for standard output and input functions and the unistd.h library is used to use user-defined structures, i.e., functions. The code contains a simple Handler function named “Handle” with integer type argument “s.” This variable “s” will be working as a signal. It contains a simple printf statement to send a signal “stop” to the user.

The main() function starts by creating a signal using the “signal” function here while using its option “SIGINT” to initialize a signal and the “Handle” function to use this function as a signal handler. The “for” loop starts from “1,” but it has no end, i.e., it’s going to increment but never stops until you do it yourself by exiting the program. The loop contains a printf statement of C to display that we are currently in the Main() function. By utilizing the Ctrl+C shortcut, we can invoke the Handle() function to display a signal message.

On each iteration of the “for” loop, it sleeps for 5 seconds and then gets incremented. Let’s execute our signal C code after properly saving and removing its errors.

```
#include <stdio.h>
#include <signal.h>
#include <unistd.h>

void Handle(int s) {
    printf("\nStop.....\n");
}

int main() {
    signal(SIGINT, Handle);
    for(int i=1;; i++) {
        printf("%d: Main.....\n", i);
        sleep(5); }
    return 0;
}
```

When you run this program with the compiler of “C” and execute it with the “.a/.out” query, you will see an infinite execution of the main() thread.

```
kalsoom@virtualbox:~$ gcc sig.c
kalsoom@virtualbox:~$ ./a.out
1: Main.....
```

When we pressed “Ctrl+C” to invoke the signal handler function, the Handle() function gets executed, and it displayed “Stop” at the moment. After that, the control again goes to the main() method, and it again starts to execute its printf statement using the infinite “for” loop. We pressed the shortcut key “Ctrl+Z” to forcefully stop this program.

```
kalsoom@virtualbox:~$ ./a.out
1: Main.....
2: Main.....
^C
Stop.....
3: Main.....
4: Main.....
^Z
[7]+ Stopped ./a.out
```


Let's say you want to ignore the signal invoked by a user. This can be done using the SIG_IGN parameter of the signal() function without using any handler function. We are using the same code with a slight change in the signal() function. Using this function, when a user invokes a signal using the "Ctrl+C" shortcut, the program will ignore the signal and continue to execute the main() function thread. Let's just quickly save and execute this program to see the output.

```
#include <stdio.h>
#include <signal.h>
#include <unistd.h>
int main() {
    signal(SIGINT, SIG_IGN);
    for(int i=1;; i++) {
        printf("%d: Ignoring the signal.....\n", i);
        sleep(5); }
    return 0;
}
```

After this code execution, the main() function printf() statement continues to execute even after using the "Ctrl+C" shortcut. So, we forcefully stopped the program at the end.

```
kalsoom@virtualbox:~$ gcc sig.c
kalsoom@virtualbox:~$ ./a.out
1: Ignoring the signal.....
^C
2: Ignoring the signal.....
^C
3: Ignoring the signal.....
^Z
[8]+  Stopped                  ./a.out
```

Example 02

If you want to automatically raise a signal for your C program without using the Ctrl+C shortcut key, you can do that as well by utilizing the `raise()` function of the signal library. Therefore, we have updated the code from the first example and added a `Handle()` function once again. The `main()` function starts with the creation of a signal using the `signal()` function by passing it an option “SIGUSR1” with the “Handle” function in the parameters. This SIGUSR1 option is the key to generating an automatic signal. The “for” loop sleeps for 10 seconds after executing the `printf` statement and then calls the `raise()` function with the option SIGUSR1 to generate an automatic signal by calling the “Handle” function. That's it; we are ready to execute this program for now.

```
#include <stdio.h>
#include <signal.h>
#include <unistd.h>
void Handle(int s) {
    printf("\nRaise.....\n");
    sleep(5);
}
int main() {
    signal(SIGUSR1, Handle);
    for(int i=1;; i++) {
        printf("%d: Main.....\n", i);
        sleep(10);
        raise(SIGUSR1); }
    return 0;
}
```

After the successful compilation, the execution takes place. The `main()` thread gets executed, and the program sleeps for 10 seconds while it's in the loop.

```
kalsoom@virtualbox:~$ gcc sig.c
kalsoom@virtualbox:~$ ./a.out
1: Main.....
```

After 10 seconds, the `raise` function has been invoked to generate an automatic signal by calling the `Handle` function. The execution sleeps for the next 5 seconds in the `Handle` function.

```
kalsoom@virtualbox:~$ ./a.out
1: Main.....

Raise .....
```