



Assignment 07 | Operating System

CE-092

Assignment submission for Operating System subject week 7.

nevilparmar24@gmail.com

Dup2 System call - duplicate a file descriptor

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

```
int dup2 ( int oldfd, int newfd) ;
```

dup2() makes newfd be the copy of oldfd.

If the newfd was previously opened, it is silently closed before being rescued.

The steps of closing and reusing the file descriptor *newfd* are performed *atomically*.

Note the two following points :

- If oldfd is not a valid file descriptor, then the call fails, and newfd is not closed.
- If oldfd is a valid file descriptor, and newfd has the same value as oldfd, then dup2() does nothing, and returns newfd.
-

On success, it returns the new file descriptor . On error, -1 is returned.

Exec Functions : replaces the current process image with the new process image

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

```
int execl(const char *pathname, const char *arg, .../* (char *) NULL */);
```

The exec() family of functions replaces the current process image with a new process image.

The initial argument for these functions is the name of a file that is to be executed.

The `const char *arg` and subsequent ellipses can be thought of as `arg0, arg1, ..., argn`. Together they describe a list of one or more pointers to null-terminated strings that represent the argument list available to the executed program. The first argument, by convention, should point to the filename associated with the file being executed. The list of arguments must be terminated by a null pointer, and, since these are variadic functions, this pointer must be cast `(char *) NULL`.

The `exec()` functions return only if an error has occurred. The return value is `-1`, and `errno` is set to indicate the error.

Task 1:

Write a program to implement `ls | sort` functionality using the system calls and functions covered in the lab.

Code:

```
#include<stdio.h>
#include<unistd.h>
#include<sys/types.h>
#include<stdlib.h>
#include<sys/wait.h>

#define readEnd 0
#define writeEnd 1

void report_and_exit(const char* msg) {
    perror(msg);
    exit(EXIT_FAILURE);    /** failure **/
}

int main()
```

```

{
    int pipeFds[2];
    int status = pipe(pipeFds);

    if(status == -1)
    {
        report_and_exit("Pipe Failed ! \n");
    }

    int pid = fork();

    if(pid == -1)
    {
        report_and_exit("Fork Failed ! \n");
    }

    else if (pid > 0)
    {
        /* parent process */

        wait(NULL);
        close(readEnd); // close the standard input
file
        dup2(pipeFds[0], readEnd);
        close(pipeFds[1]);
        execl("/bin/sort", "sort", NULL);
    }

    else
    {
        /* child process */

```

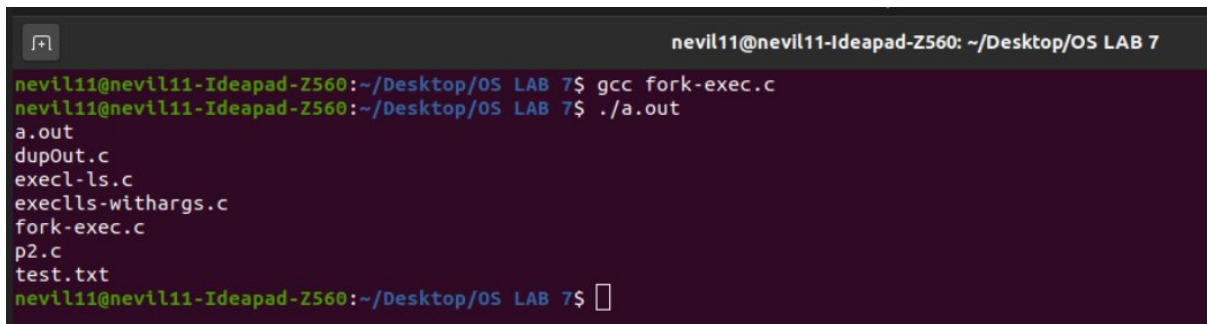
```

        close(writeEnd); // close the standard output
file
        dup2(pipeFds[1], writeEnd); // now pipeFds[1]
(the writing end of the pipe) is the standard output
        close(pipeFds[0]);
        execl("/bin/ls", "ls", NULL);
    }

    return 0;
}

```

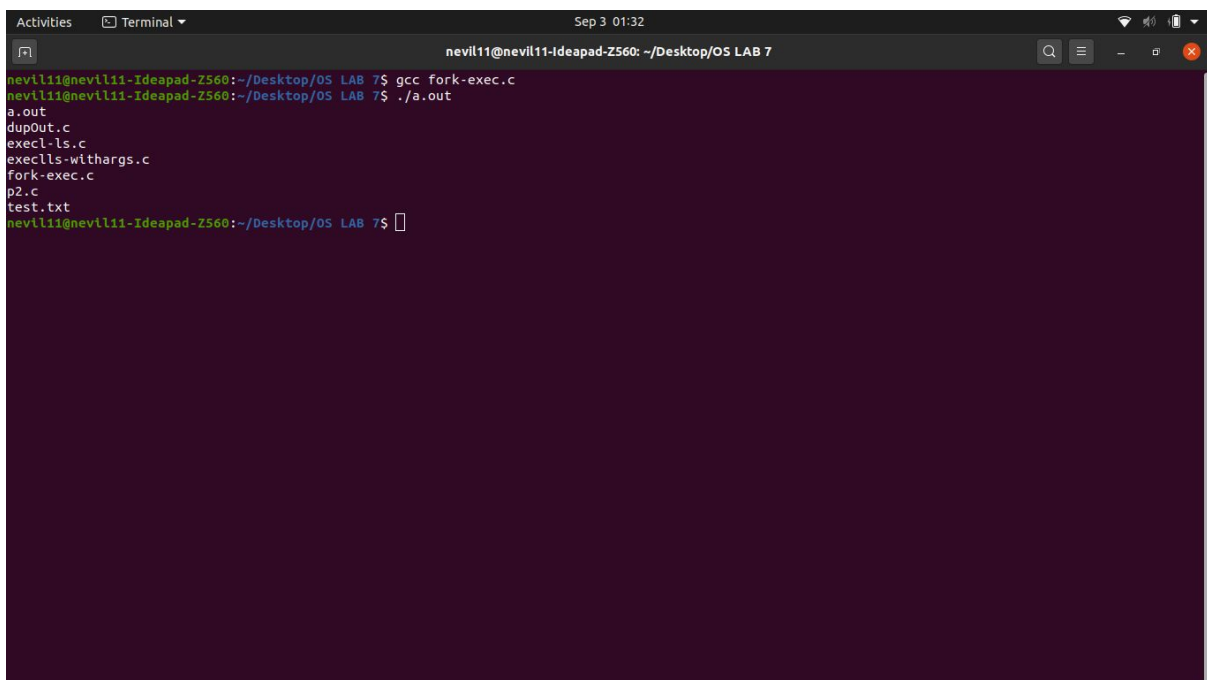
Output:



```

nevil11@nevil11-Ideapad-Z560: ~/Desktop/OS LAB 7
nevil11@nevil11-Ideapad-Z560:~/Desktop/OS LAB 7$ gcc fork-exec.c
nevil11@nevil11-Ideapad-Z560:~/Desktop/OS LAB 7$ ./a.out
a.out
dupOut.c
execl-ls.c
execlls-withargs.c
fork-exec.c
p2.c
test.txt
nevil11@nevil11-Ideapad-Z560:~/Desktop/OS LAB 7$ 

```



```

Activities  Terminal  Sep 3 01:32
nevil11@nevil11-Ideapad-Z560: ~/Desktop/OS LAB 7
nevil11@nevil11-Ideapad-Z560:~/Desktop/OS LAB 7$ gcc fork-exec.c
nevil11@nevil11-Ideapad-Z560:~/Desktop/OS LAB 7$ ./a.out
a.out
dupOut.c
execl-ls.c
execlls-withargs.c
fork-exec.c
p2.c
test.txt
nevil11@nevil11-Ideapad-Z560:~/Desktop/OS LAB 7$ 

```

Task 2:

Write a program to achieve following:

- Child process should open a file with the contents to be sorted, pass the contents to the parent process.
- Parent process should sort the contents of the file and display.

Code:

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <sys/wait.h>
#include <fcntl.h>

#define readEnd 0
#define writeEnd 1

void report_and_exit(const char* msg) {
    perror(msg);
    exit(EXIT_FAILURE);    /** failure **/
}

int main(int argc, char *argv[]) {
    int pipeFds[2];
    char buffer[100];

    if (argc < 2) {
        fprintf(stderr, "usage: progname <filename>");
        exit(EXIT_FAILURE);
    }
```

```

if (pipe(pipeFds) == -1) {
    report_and_exit("PIPE ERROR");
}

int pid = fork();

if(pid == -1){
    report_and_exit("fork failed");
}

else if (pid > 0) {

    /* Parent process */

    wait(NULL);
    close(readEnd); // close the standard input
file
    dup2(pipeFds[0], readEnd);
    close(pipeFds[1]);
    execl("/bin/sort", "sort", NULL);

} else {

    /* Child Process */

    int fd;
    if ((fd = open(argv[1], O_RDONLY)) == -1) {
        report_and_exit("error opening file");
    }

    close(writeEnd); // close the standard output

```

```
file
    dup2(pipeFds[1], writeEnd); // now pipeFds[1]
    (the writing end of the pipe) is the standard output
    close(pipeFds[0]);

    char singleLine[150];

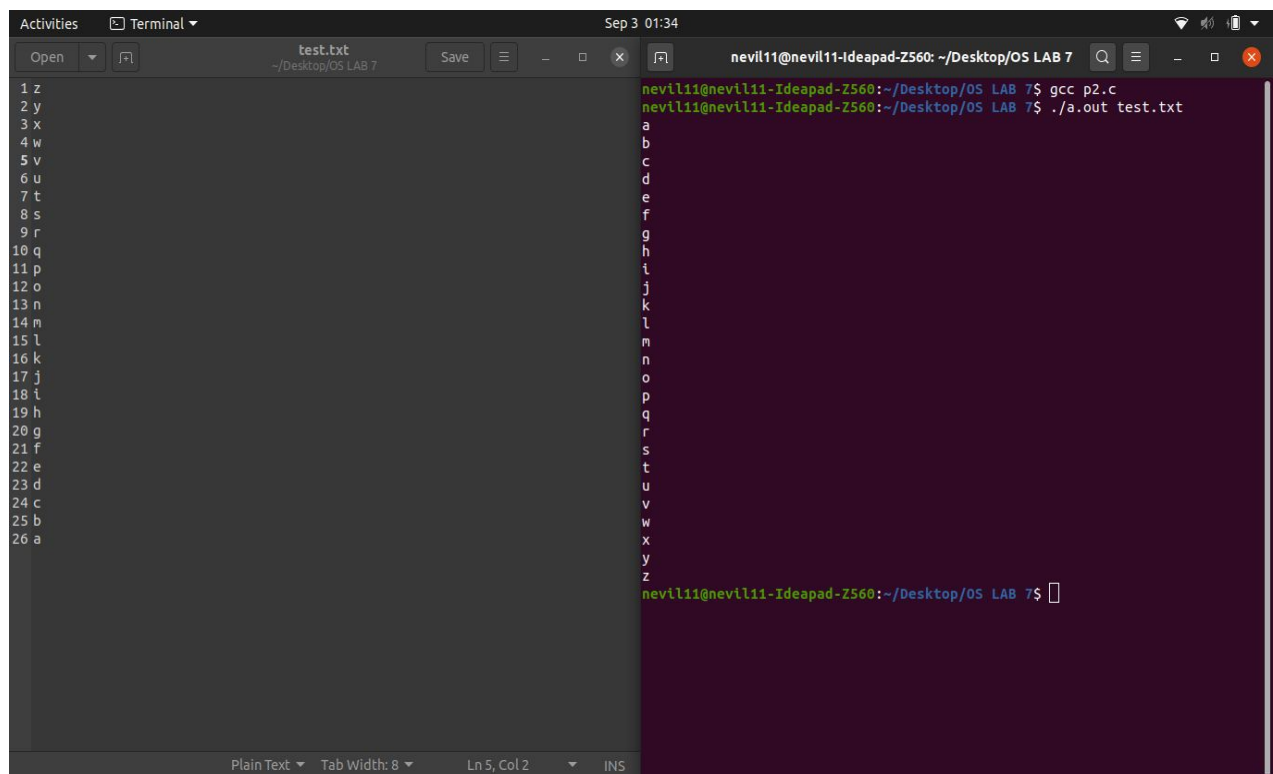
    while (read(fd, singleLine ,
sizeof(singleLine))) {
        write(pipeFds[1], singleLine,
strlen(singleLine));
    }

    close(fd);

}

return 0;
}
```

Output:



```
1 z
2 y
3 x
4 w
5 v
6 u
7 t
8 s
9 r
10 q
11 p
12 o
13 n
14 m
15 l
16 k
17 j
18 i
19 h
20 g
21 f
22 e
23 d
24 c
25 b
26 a

nevil11@nevil11-Ideapad-Z560: ~/Desktop/OS LAB 7
nevil11@nevil11-Ideapad-Z560:~/Desktop/OS LAB 7$ gcc p2.c
nevil11@nevil11-Ideapad-Z560:~/Desktop/OS LAB 7$ ./a.out test.txt
a
b
c
d
e
f
g
h
i
j
k
l
m
n
o
p
q
r
s
t
u
v
w
x
y
z
nevil11@nevil11-Ideapad-Z560:~/Desktop/OS LAB 7$
```

Execv Function

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

```
int execv(const char *path, char *const argv[]);
```

Between execl and execv there is no difference other than the format of the arguments. They both end up calling the same underlying system call `execve()`.

Example Program :

Create hello.c file and compile it using `gcc hello.c -o hello`

```
#include<stdio.h>

int main(int argc, char *argv[]) {
    printf("Filename: %s\n", argv[0]);
    printf("%s %s\n", argv[1], argv[2]);
    return 0;
}
```


Make execv.c file which runs the executable hello file as a newly created process and make a note that the statements written after the execv call are never executed unless some error occurs executing execv function .

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

int main() {

    char *temp[] = {NULL, "I am ", "a execv
process", NULL};

    temp[0]="hello";
    execv("hello",temp);

    // This line will never get executed as the current
process is replace by the newly created execv proces
    printf("error");
    return 0;

}
```

Output :

```
nevil11@nevil11-Ideapad-Z560: ~/Desktop/OS LAB 7
nevil11@nevil11-Ideapad-Z560:~/Desktop/OS LAB 7$ gcc hello.c -o hello
nevil11@nevil11-Ideapad-Z560:~/Desktop/OS LAB 7$ gcc execv.c
nevil11@nevil11-Ideapad-Z560:~/Desktop/OS LAB 7$ ./a.out
Filename: hello
I am  a execv process
nevil11@nevil11-Ideapad-Z560:~/Desktop/OS LAB 7$
```

```
Activities Terminal Sep 4 00:18
nevil11@nevil11-Ideapad-Z560: ~/Desktop/OS LAB 7
nevil11@nevil11-Ideapad-Z560:~/Desktop/OS LAB 7$ gcc hello.c -o hello
nevil11@nevil11-Ideapad-Z560:~/Desktop/OS LAB 7$ gcc execv.c
nevil11@nevil11-Ideapad-Z560:~/Desktop/OS LAB 7$ ./a.out
Filename: hello
I am a execv process
nevil11@nevil11-Ideapad-Z560:~/Desktop/OS LAB 7$
```

Nevil Parmar
CE-092
<https://nevilparmar.me>