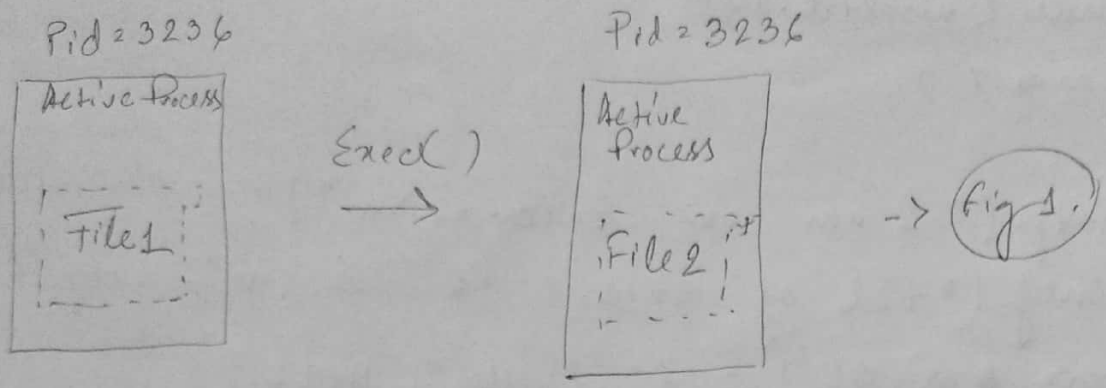
1. Exec() System call.

It is used to execute a file in an active process.



-> Fig 1.

When Exec() is invoked, the previous file is replaced as shown in Fig 1. and new file is executed.

The new program is loaded in the same process space. (Space as in memory).

So, the process id will be same. It won't get changed. Ie, we are not creating any new process.

So, process id will not be changed but the data, stack, heap, code every other related informations will get updated with the other program (loaded)

2. Write the command:

man exec ←|

you can see there multiple types of the same exec() system call. Each of them can have different syntaxes as you can see.

3. Exercise using Exec() System call: ——

we need to prepare two files for this —

1. Calling program
2. Called program.

Scanned with CamScanner

Exec.c :

```
#include <stdio.h>
#include <unistd.h>
int main()
{
    printf("I am in Exec.c \n");
    printf("Pid of Exec.c is %d \n", getpid());
    char *argv[] = {"./hello", NULL};
    execv(argv[0], argv);
    printf("coming back to exec.c"); // This will not be executed if Exec() runs successfully
    return 0;
}
```

[N.B.: By going through the manual page of Execv system call you will get to know that Execv system call.

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

So, the first parameter is nothing but the path & the second parameter is nothing but the array of character pointers.

In our program Exec.c I have passed the parameter as "./hello.c" - is the path of the program. I have written "./" as both my

programs "exec.c" & "hello.c" stays in the same ~~program~~ folders.

- hello.c

```
#include <stdio.h>
#include <unistd.h>
int main()
{
    printf("I am in hello.c \n");
    printf("Pid of hello.c is %d \n", getpid());
    return 0;
}
```

Execution of both the programs:

```
$ gcc -o hello hello.c ↵
$ gcc -o execv exec.c ↵
$ ./execv ↵

I am in exec.c
Pid of exec.c is 1519
I am in hello.c
Pid of hello.c is 1519
```

o/p.

Scanned with CamScanner

open( ) system call :—

```
#include <unistd.h>
#include <stdio.h>
#include <fcntl.h>
int main()
{
    int fd;
    char buffer[80];
    char msg[50] = "hello Students : God Bless All";
    fd = open("sample.txt", O_RDWR);
    printf("fd = %d ", fd);
    if (fd != -1)
    {
    printf("\n sample.txt opened with read write
                                    access \n");
    write (fd, msg, sizeof(msg));
    lseek (fd, 0, SEEK_SET);
    read (fd, buffer, sizeof(msg));
    printf("\n %s was written to my file \n", buffer
    close(fd);
    }
    return 0;
}
```

[N.B: you can go to the command prompt and execute a command as —

$ man open. ↵

where you will able to see that if you there are different syntaxes of open & create.

Scanned with CamScanner

Secondly if my 'Sample.txt' is present in my same directory where program "open.c" is present, then the syntax of open() system call will be like exactly how it is done in the program "open.c".
But if 'Sample.txt' is present in some other folder then we have to mention the path (Total path).

int open (const char *pathname, int flags, mode_t mode);

The flags include three modes:—

O_RDONLY
O_WRONLY
O_RDWR

Now this open() system call returns a file descriptor. In my program I have used 'fd' to store the file descriptor value..

The basic file descriptor values are

0 - standard i/p
1 - " o/p
2 - " Error.

If open system call fails then it returns −1.

- The next is write System Call!

write (int fd, const void *buf, size_t count);
(what you want to write.)

I have used lseek as I want my file to be accessed from first data byte.
Please go & see lseek manual.

Scanned with CamScanner