

Processes Creation : Fork () (video attached)

fork () is a function that creates a new program with new process and new process ID. This results in two processes running. One is a parent process and the other is a child process.

CHILD PROCESS CREATION USING FORK :

What the function fork () does is , it dynamically clones the parent process bit by bit including status of variables. The entire memory, registers and status are all cloned and is identical to the parent process. Now at this time, we have two processes and they are identical. The child process has a process ID too. The execution of the child process starts after the line next to fork () while the parent process also continues execution after the line next to fork () .

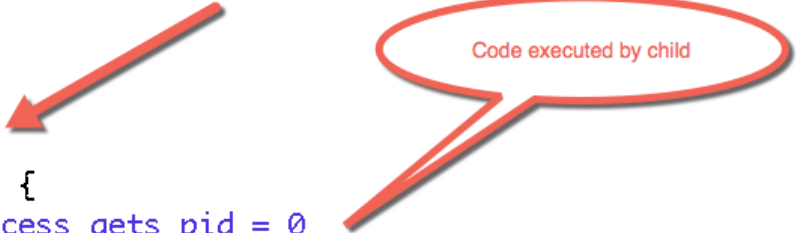
When a fork function call returns creating a child process, **the return value is zero for the child process and non-zero for the parent process.**

Look at this simple code : Figure


```

2 #include <stdio.h>
3 #include <sys/types.h>
4
5 void main( void )
6 {
7     pid_t pid;
8
9     pid = fork();
10    if (pid == 0) {
11        // Child process gets pid = 0
12        printf("I am the child process, my id is: %d ***\n", getpid( ) );
13    }
14
15 }

```



In line 9, fork is called. When the function returns, the child process gets a return value of zero. The parent process gets the process ID of the child as the return value. Now, we have two programs executing at the same program. A very classic case of two processes running same program but executing two different blocks.

[how to force child to execute a code after fork](https://www.youtube.com/watch?v=oiXhV6mwDgA)  [\(https://www.youtube.com/watch?v=oiXhV6mwDgA\)](https://www.youtube.com/watch?v=oiXhV6mwDgA)

Linux operating system : how to force a ch...



[Minimize Video](#)


NOTE: For the child processes, the value returned is zero. For the parent process, the value returned is the **PROCESS ID** of the child.

when you run the above program, the child processes will continue executing the printf statement because the pid is zero. The parent process will not print the printf statement because the pid value is non-zero and is the process ID of the child.

In the next new program, now we have lines of code that is created to the else part.

```
void main( void )
{
    pid_t pid;

    pid = fork();
    if (pid == 0) {
        printf("I am the child process, My id is: %d \n", getpid( ) );
    } else {
        printf("I am the parent process My id is: %d \n", getpid( ) );
        printf("My Child's id is:  %d ***\n", pid );
    }
}
```



```
c-prog>gcc ver1.c
c-prog>./a.out
I am the parent process My id is: 31079
My Child's id is: 31080 ***
I am the child process, My id is: 31080
c-prog>
```

We expanded the program to include a piece of code for the parent to execute. As you can see, the else part of the if statement is executed by the parent. The variable pid value printed by the parent is the child process ID. In the child process, you can print the process ID of the child using getpid ().

The main point is, when a fork is called :

it returns a value zero to the Child Process

and its return the process ID of the child to the parent.

There are two processes that are running now.

All variables , open files, system tables, almost everything is a duplicate in the two processes.

In the above program, the parent process is executing its block first and doesn't wait for the child. Most of the time, the parent process should wait and check on the child process and its statuses. This is done using the function wait.

Why should I wait for my child process ?

There might be a situation when the child would terminate before the parent would call wait. In this situation, the parent process has no way to figure out what happened to the child process. In this situation, the child process would remain a zombie state and resources allocated to the child would remain active in the kernel and won't be released for future processes. The kernel keeps a copy of the state of the zombie child. When the parent calls wait, the kernel responds to the function using the state it has and then releases the resources and child that is terminated will not be in a zombie state.

So, we need to call wait function in the parent after a fork function. See the note section of the man pages:

"A child that terminates, but has not been waited for becomes a "zombie". The kernel maintains a minimal set of information about the zombie process (PID, termination status, resource usage information) in order to allow the parent to later perform a wait to obtain information about the child. As long as a zombie is not removed from the system via a wait, it will consume a slot in the kernel process table, and if this table fills, it will not be possible to create further processes. If a parent process terminates, then its "zombie" children (if any) are adopted by [init](https://linux.die.net/man/8/init) (<https://linux.die.net/man/8/init>) process, which automatically performs a wait to remove the zombies"

What could we do inside the Child process after a fork ?

Remember, after fork, there are two separate processes running. We could add and execute any code inside these processes. In real life, server side programming involves the parent forking a child to serve a request from a client. The child process serves the request from a client and dies after it is done serving. For example, http request from your browser will be a request to the server. The

server forks a child, the child serves the webpage to the browser and dies. The server goes on to the next request.

simple Algorithm of a web server :

```
do {  
    request_received_from_browser ( ) ;  
    pid = fork ( )  
    if ( pid == 0 ) {    //child process  
        child serves the web page and dies  
    }  
    else { // Parent processes  
        go back to serve more browser requests  
    }  
  
} while ( 1 ) ;
```

Disadvantages with FORK : Creating fork is a very expensive operation: the system has to copy the entire program, it has to create a new process, manage the child and parent relationships, save the status of all variables. If two processes have to communicate (what is known as interprocess communication) , an extensive programming effort has to be put for the two processes to share data or work collaboratively. Most of these efforts have issues with portability too and management of code across system is a nightmare for software developers.

Open File Descriptors:

When a fork() is performed, the child receives duplicates of all of the parent's file descriptors. These duplicates are made in the manner of dup(), which means that corresponding descriptors in the parent and the child refer to the same open file description. These attributes of an open file are

shared between the parent and child. For example, if the child updates the file offset, this change is visible through the corresponding descriptor in the parent.

This is demonstrated in this example:

```
void main(void)
{
    FILE *fp = fopen ( "data.txt", "w");
    pid_t  pid1, pid;
    int    status;

    if ((pid1 = fork()) < 0) {
        printf("Failed to create child process 1\n");
        exit(1);
    }
    else if (pid1 == 0) {
        sleep ( 4 );
        printf ( "  Child Process PID %d \n", getpid() );
        fprintf (fp, " Child created Process PID %d \n", getpid() );
        exit(0);
    }

    printf("Parent created  child PID1 %d \n", pid1);
    fprintf(fp, "Parent created  child PID1 %d \n", pid1);

    pid = wait(&status);
    printf("*** Parent sees child PID %d terminated ***\n", pid);
    fprintf(fp, "*** Parent sees child PID %d terminated ***\n", pid);
    exit(0);
}

[srivatss@athena:105]> gcc fileSharing.c
[srivatss@athena:106]> ./a.out
Parent created  child PID1 27546
  Child Process PID 27546
*** Parent sees child PID 27546 terminated ***
```

The diagram illustrates the sharing of a file descriptor between a parent and a child process. A red arrow points from the `FILE *fp` variable in the parent process to the `fprintf(fp, ...)` call in the child process. A red bracket on the right side of the code, spanning the child's execution block, is labeled "FILE descriptor shared". Another red bracket on the right side of the terminal output, spanning the parent's output lines, indicates the parent's execution flow.

The system wide open descriptor table is demonstrated in this figure:

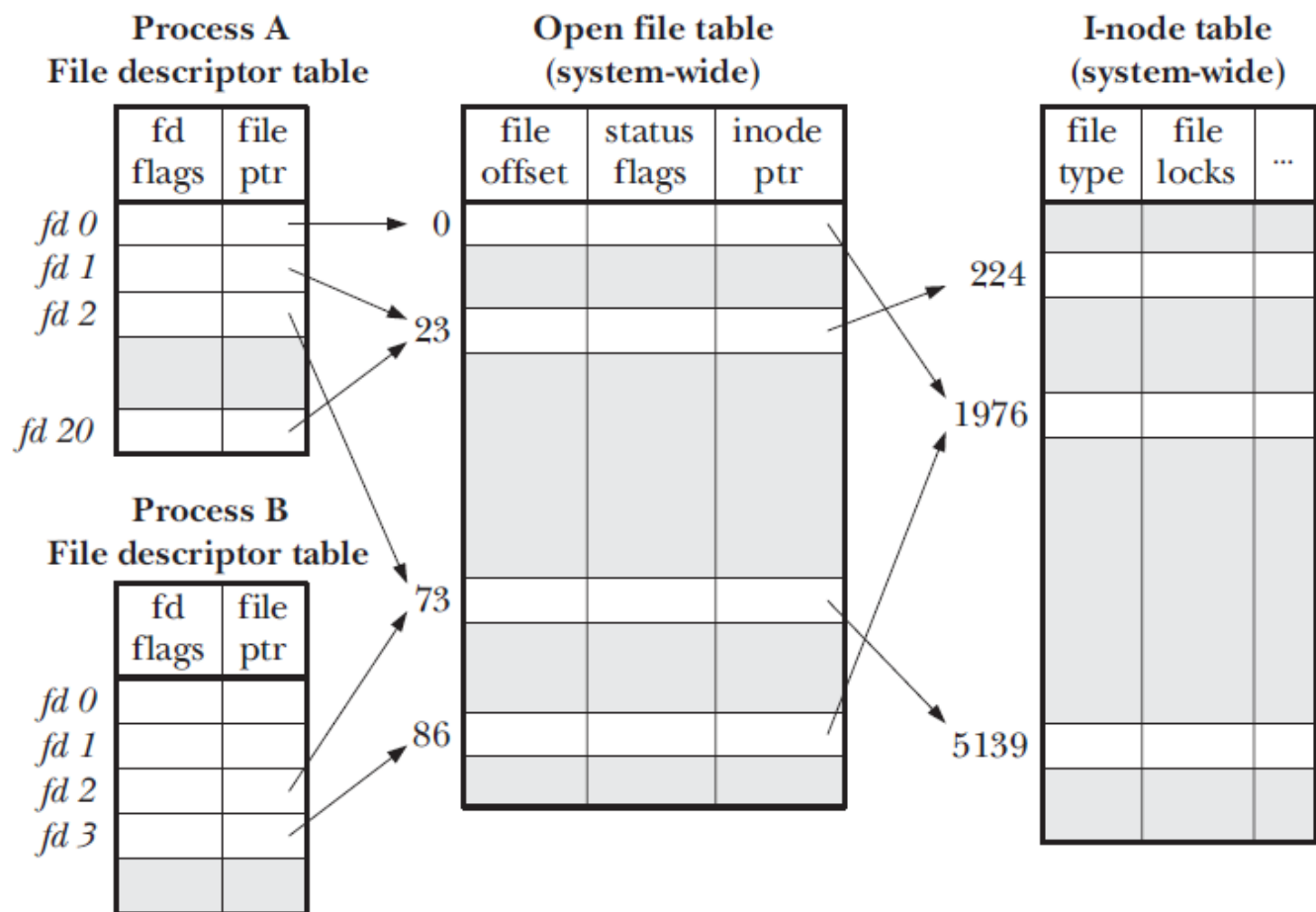
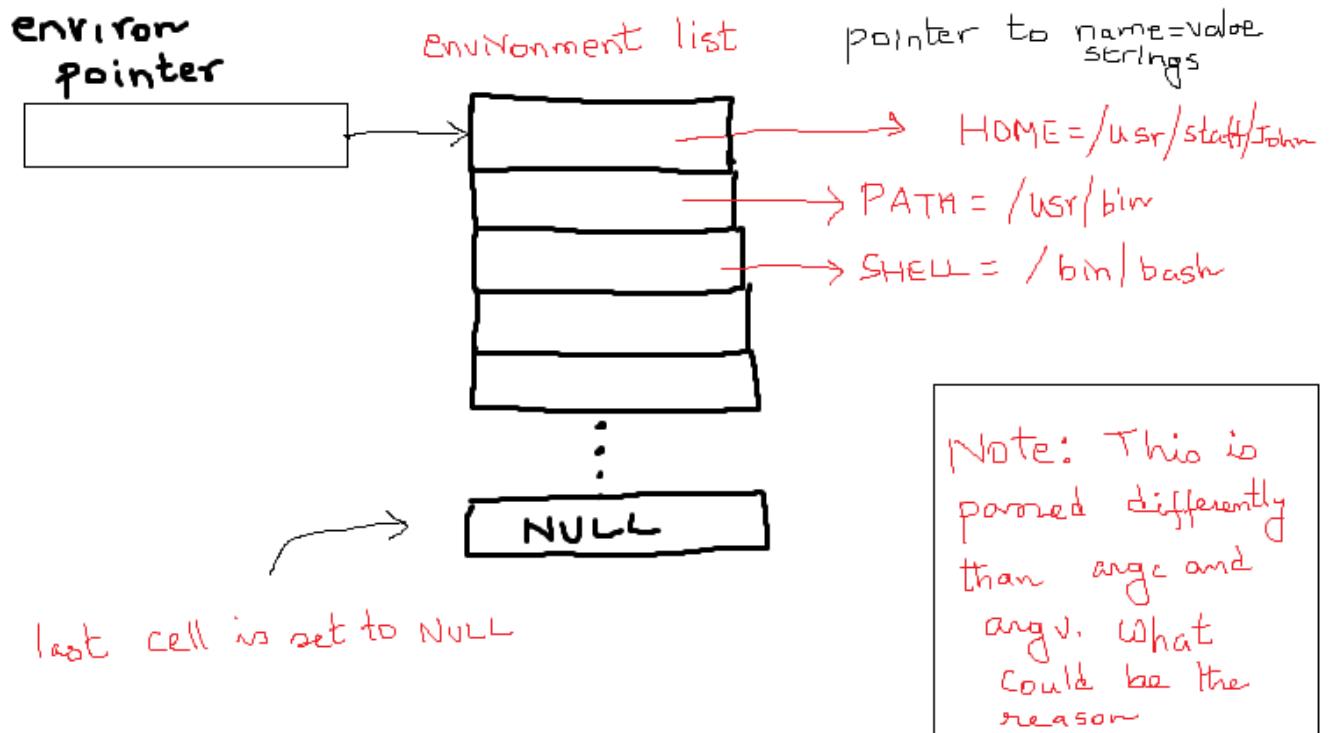


Figure 5-2: Relationship between file descriptors, open file descriptions, and i-nodes

environ variable



Sample program is:

```
#include <stdio.h>
#include <stdlib.h>
extern char **environ; // pointer to pointer global variable
                        // copied to our memory address by the OS

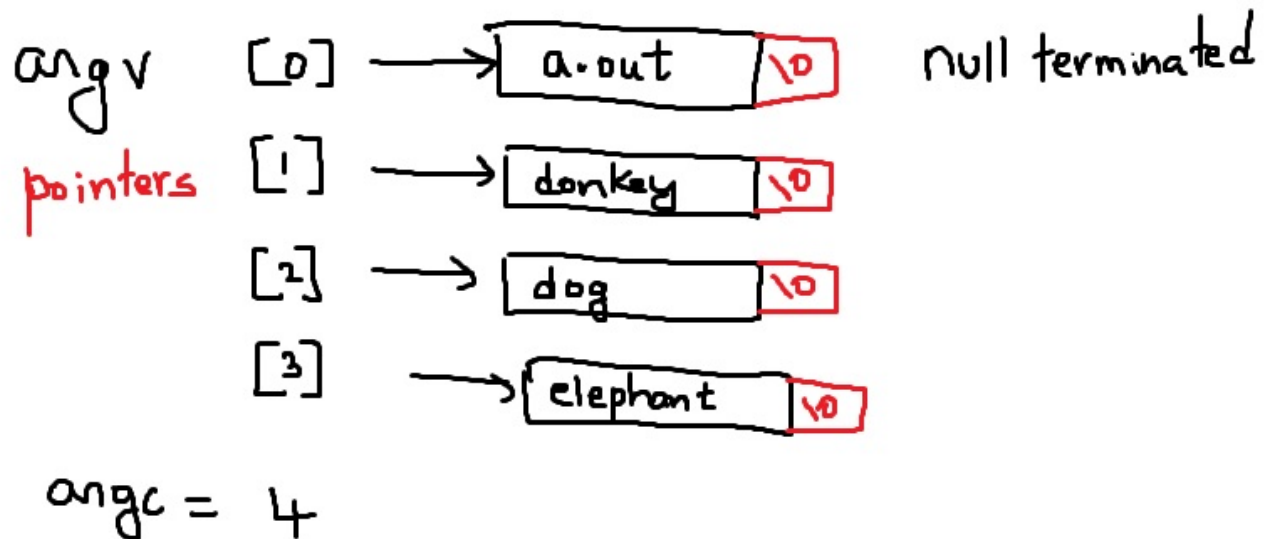
int main( int argc, char *argv [ ] )
{
    int i = 0;

    for ( i = 0 ; environ[i] != NULL ; i++ )
        printf ( "value at %d is %s \n", i, environ[i] );
}
```

environ global variable is checked for not NULL

passing arguments to C program (argc argv)

a.out donkey dog elephant



To print the arguments:

```
for ( i = 1 ; i < argc ; i++ )
```

```
printf ( " The argument are %s \n", argv[i]);
```

Wait functions

Why do we need wait function:

here is the video:

<https://www.youtube.com/watch?v=Tt5qpp3141U> ⇨ <https://www.youtube.com/watch?v=Tt5qpp3141U>



<https://www.youtube.com/watch?v=Tt5qpp3141U>

Sample Program using wait function

Some documentation about wait function in FORK

```
int status ;
```

```
wait ( &status) ;
```

```
waitpid (pid_t pid, &status, options);
```

`wait (&status)` is same as `waitpid (-1 , &status, 0) ;`

`pid = -1` means do wait for any of the children

`option = 0` means do not wait if any child has not exited.

The two system calls are used to wait for state changes in a child of the calling process, and obtain information about the child whose state has changed. A state change is considered to be: the child terminated; the child was stopped by a signal; or the child was resumed by a signal. In the case of a terminated child, performing a wait

allows the system to release the resources associated with the child; if a wait is not performed, then the terminated child remains in a "zombie" state (see NOTES below)

Sample Code using wait (), please note the highlighted calls are macros. You can actually see the expansion of the macro using -E option. The sample code creates a child. The child sleeps for 60 seconds. Parent process waits to check the status of the child.

```
int main( void )
{
    int status;
    pid_t pid;

    pid = fork();

    if (pid == 0) {
        printf(" *** Child process is %d ***\n", getpid( ) );
        sleep ( 60 ) ;
    } else    {
        wait ( &status);
        if ( WIFEXITED ( status) )
            printf ( "Child exited normally %d \n", WEXITSTATUS (status) );
        else if ( WIFSIGNALED ( status) )
            printf ( "Child exited by a Signal #%d \n", WTERMSIG (status) );
        }
}
```

1. Try to run this program in the background. wait for 60 seconds.
2. Again, try to run the program in the background. terminate the client process using the kill command. Kill is the command to send a signal to another process in the same group. There are various signals Linux supports. We will discuss them soon

Sample code using waitpid Function

```

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

void main( void )
{
    int status;
    pid_t pid;

    pid = fork();
    if (pid == 0) {
        printf("Child process is %d \n", getpid( ) );
        sleep ( 2 );
    } else {
        waitpid (pid, &status, WUNTRACED | WCONTINUED );
        if ( WIFEXITED ( status) )
            printf ( "Child exited normally %d \n", WEXITSTATUS (status) );
        else if ( WIFSIGNALED ( status) )
            printf ( "Child exited by a Signal #%d \n", WTERMSIG (status) );
    }
}

```

Process Creation: System () function call

```
system ( "ls" );
```

This will execute the command ls

```
system ("pwd" );
```

 This will execute the command pwd

How system works : it forks, executes exec functions in the child process. While the child executes the function, the parent waits.

Exec Family of Functions (video attached)

There is a system function called exec (6 of them). exec functions will execute any program you pass it to it. You also need to pass parameters if any.

When you launch the new program mentioned in the exec, the new program will replace the caller in memory, though the process ID remains the same. It is as if you launched a new program.

Here is the document :

[exec-family-functions.docx \(https://csus.instructure.com/courses/94454/files/14915326/download?wrap=1\)](https://csus.instructure.com/courses/94454/files/14915326/download?wrap=1)

Here is the video :

[linux operating system fork execl family of functions execl execlp execl](https://www.youtube.com/watch?v=duYojgFuT3s) 
(<https://www.youtube.com/watch?v=duYojgFuT3s>)



(<https://www.youtube.com/watch?v=duYojgFuT3s>)

EXEC FAMILY OF FUNCTIONS

A parent program can call a exec family of functions to launch a new program. The new program will replace the parent program, but not the process. The new program will get the process ID of the parent program. The segments such as text, initialized data, uninitialized data (bss), and stack of the calling process are overwritten according to the contents of the newly loaded program.

The parent program will not be able to check the status of the exec function on success. Only on error, the parent program can check and take remedial actions. We will discuss six functions, the difference between these functions is just in the format of the parameters passed. We also provided an example for each function.

<p>execl - this function takes the command path, name and optional parameters and NULL parameters.</p> <p>execl ("/bin/ls", "ls", "-l", NULL)</p>	<p>execv – Very identical to execl, except the name and optional parameters and NULL parameters are passed separately in a array</p> <p>char *args [] = { "ls", "-l", NULL };</p> <p>execv ("/bin/ls", args);</p>
<p>execle - In addition to execl, this also takes the environment variables in a NULL terminated string array.</p>	<p>execve - In addition to execv, this also takes the environment variables in a NULL terminated string array.</p>
<p>execlp – Same as execl , except it recognizes the path of the command using the \$PATH variable. Absolute path is optional</p>	<p>execvp - Same as execv , except it recognizes the path of the command using the \$PATH variable. Absolute path is optional</p>

The functions listed on the left side differ from the right side on one thing: function on the right side take optional parameters of the command in the array of strings. Whereas, you have to list them individually for functions on the left hand side, See the highlighted text in row 1. Functions that end in 'e' take the environment variables , and functions that end 'p' recognizes the \$PATH variables for you to omit the absolute path of the command.

EXECL

```
1. int execl ( const char *file, const char *arg0, ..., NULL);
```

file : is the filename of the file that contains the executable image of the new process.

arg0, ..., NULL : is a variable length list of arguments that are passed to the new process image. Each argument is specified as a null-terminated string, and the list must end with a NULL pointer. The first argument, arg0, is required and must contain the name of the executable file for the new process image. If the new process image is a normal SAS/C main program, the list of arguments will be passed to argv as a pointer to an array of strings.

Example 1:

```
main ( )  
  
{  
    execl ( "/usr/bin/cal", "cal", "2017", NULL );  
}
```

another example, echo prints SYSTEM PROGRAMMING

```
execl ( "/bin/echo", "echo", "SYSTEM PROGRAMMING", NULL );
```

EXECLE

```
2. int execl ( const char *file,  
               const char *arg0, ..., NULL,  
               char *const envp [ ] );
```

file : is the filename of the program that is to be launched.

arg0, ..., NULL : is a variable length list of arguments that are passed to the new program. Each argument is specified as a null-terminated string, and the list must end with a NULL pointer. The first argument, arg0, is required and must contain the name of the executable file for the new process image. The number of strings in the array is passed to the main() function as argc.

envp is a pointer to an array of pointers to null-terminated character strings. A NULL pointer is used to mark the end of the array. Each character string pointed to by the array is used to pass an environment variable to the new process image. Each string should have the following form:

" var = value "

Example 1:

```
main ( )  
{  
    char *env[ ] =  
        { "SYSTEM PROGRAMMING", NULL };  
  
    execl ( "/bin/echo", "echo", env[0], NULL, env );  
}
```

another example

```
execl ( "/bin/echo", "echo", environ[3], NULL, environ );
```

EXECLP

```
3. int execlp(const char *path, const char *arg0, ..., NULL);
```

path : identifies the location of the new process in the system. If the path argument contains a slash (/), it is assumed that either an absolute or a relative pathname has been specified. If the path argument does not contain a slash, the directories specified by the PATH environment variable are searched in an attempt to locate the file.

arg0, ..., NULL : is a variable length list of arguments that are passed to the function. Each argument is specified as a null-terminated string, and the list must end with a NULL pointer. The first argument, arg0, is required and must contain the name of the executable file.

Example 3:

```
main ( )  
{  
    execlp ( "ls", "ls", "-lF", NULL );  
}
```

another example

```
execlp ( "./echo_1.sh", "./echo_1.sh", "Jack", "Sam", "Pam", NULL );
```

EXECV

```
4. int execv(const char *file, char *const argv[] ) ;
```

file : is the filename of the file that contains the executable image of the new process.

argv : is a pointer to an array of pointers to null-terminated character strings. A NULL pointer is used to mark the end of the array. Each character string pointed to by the array is used to pass an argument to the new process image. The first argument, argv[0], is required

Example:

```
main ( )
{
char *paramList[ ] = { "ls", "-l", NULL} ;
execv ( "/bin/ls", paramList );
}
```

EXECVE

```
5. int execve ( const char *filename,      char *const argv [ ] ,  
               char *const envp [ ] ) ;
```

filename : is the filename of the program to be launched by the function

argv : is a pointer to an array of pointers to null-terminated character strings. A NULL pointer is used to mark the end of the array. Each character string pointed to by the array is used to pass an argument to the new process image. The first argument, argv[0], is required and must contain the name of the executable file for the new process image.

envp : is a pointer to an array of pointers to null-terminated character strings. A NULL pointer is used to mark the end of the array. Each character string pointed to by the array is used to pass an environment variable to the new process image.

Example :

```
main ( )  
{  
    char *paramList[ ] = { "echo", environ[4], NULL };  
    execve ( "/bin/echo", paramList, environ );  
}
```

EXECVP

```
6. int execvp ( const char *path, char *const argv [ ] ) ;
```

path: identifies the location of the new program. If the path argument contains a slash (/), it is assumed that either an absolute or a relative pathname has been specified. If the path argument does not contain a slash, the directories specified by the PATH environment variable are searched in an attempt to locate the file.

argv : is a pointer to an array of pointers to null-terminated character strings. A NULL pointer is used to mark the end of the array. Each character string pointed to by the array is used to pass an argument to the new process image. The first argument, argv[0], is required and must contain the name of the executable file for the new process image.

Example:

```
main ( )  
{  
  char *paramList[ ] = { "ls", "-l", NULL } ;  
  execvp ( "ls", paramList );  
}
```