Unix Programming

UNIT-V

B.Tech(CSE)-V SEM

UNIT-V : UNIX Process

- What is a process.
- Process structure.
- Process identifiers.
- Starting a new process, Waiting for a process.
- Zombie process.
- System call interface for process management- fork, vfork, exit, wait, waitpid, exec system call.

What is a process?

- A **process** is a program in execution in memory or in other words, an instance of a program in memory.
- Any program executed creates a process.
- A program can be a command, a shell script, or any binary executable or any application.
  - **<u>Process attributes:</u>**
    A process has some properties associated to it:
  - **PID** : Process-Id. Every process created in Unix/Linux has an identification number associated to it which is called the process-id.
  - **PPID** : Parent Process Id: Every process has to be created by some other process. The PID of the parent process is called the parent process id(PPID).
  - **TTY**: Terminal to which the process is associated to. Every command is run from a terminal which is associated to the process.
  - **UID**: User Id- The user to whom the process belongs to.
  - **File Descriptors**: File descriptors related to the process: input, output and error file descriptors.

Process Structure:

- The memory allocated to UNIX processes can be divided into three logical parts:
  - `Text Segment` -The text section contains the machine-language code to be executed by the machine on behalf of the process.
  - `Data Segment`-The data section contains a representation of data preset to initial values. It also includes the amount of space to be allocated by the kernel for uninitialized data (known for historic reasons as bss).
  - `Stack Segment` -The stack area usually contains procedure-based, downward growing, data frames.

Process Structure:

- The exact information present in any process structure will vary from one implementation to another, but all process structures minimally include:
  - Process id
  - Parent process id (or pointer to parent's process structure)
  - Pointer to list of children of the process
  - Process priority for scheduling, statistics about CPU usage and last priority.
  - Process state
  - Signal information (signals pending, signal mask, etc.)
  - Machine state
  - Timers

Process Structure:

- Usually the process structure is a very large object containing much more additional information.
- Typical substructures referenced in the process structure may include such things as the:
    - Process's group id
    - User ids associated with the process
    - Memory map for the process (where all segments start, and so on)
    - File descriptors
    - Accounting information
    - Other statistics that are reported such as page faults, etc.
    - Signal actions
    - Pointer to the user structure.

Process Identifiers:

- Every process has a unique process ID, a non-negative integer. As processes terminate, their IDs can be reused.
- Most UNIX systems implement algorithms to delay reuse so that newly created processes are assigned IDs different from those used by processes that terminated recently.
- This prevents a new process from being mistaken for the previous process to have used the same ID.
- There are some special processes, but the details differ from implementation to implementation:
  - Process ID 0: **scheduler process** (often known as the **swapper**), which is part of the kernel and is known as a system process
  - Process ID 1: **init** process, invoked by the kernel at the end of the bootstrap procedure.
    - It is responsible for bringing up a UNIX system after the kernel has been bootstrapped. **init** usually reads the system-dependent initialization files (/etc/rc* files or /etc/inittab and the files in /etc/init.d) and brings the system to a certain state.
    - It never dies.
    - It is a normal user process, not a system process within the kernel.
    - It runs with superuser privileges.

Process Identifiers (contd)

- Each UNIX System implementation has its own set of kernel processes that provide operating system services.
- On some virtual memory implementations of the UNIX System, process ID2 is the **pagedaemon**. This process is responsible for supporting the paging of the virtual memory system.
- In addition to the process ID, there are other identifiers for every process.
- The following function return these identifiers:

```
#include <sys/types.h>
#include <unistd.h>
pid_t getpid(void);         /* Returns: process ID of calling process */
pid_t getppid(void);        /* Returns: parent process ID of calling process */
uid_t getuid(void);         /* Returns: real user ID of calling process */
uid_t geteuid(void);        /* Returns: effective user ID of calling process */
gid_t getgid(void);         /* Returns: real group ID of calling process */
gid_t getegid(void);        /* Returns: effective group ID of calling process */
```

Starting a new process- fork()

- The only way a new process is created by the Unix kernel is when an existing process calls the **fork** function.

```
#include <sys/types.h>
#include <unistd.h>

pid_t fork(void);

            /* Returns: 0 in child, process ID of child in parent, –1 on error */
```

- The new process created by fork is called the **child process**.
- This function is called once but returns twice.
- The only difference in the returns is that the return value in the child is 0, whereas the return value in the parent is the process ID of the new child.

Starting a new process (Contd)

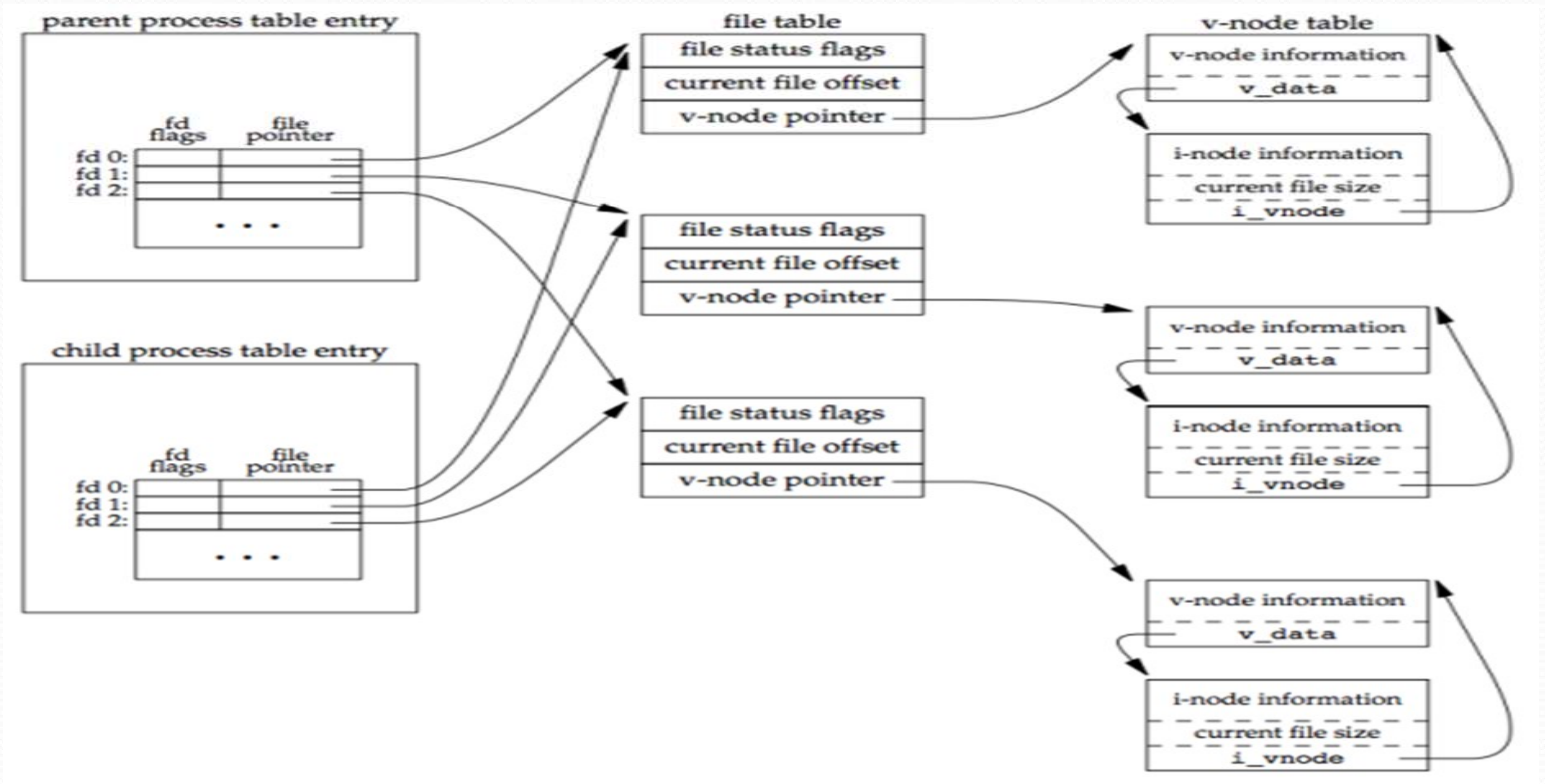
- The reason the child's process ID is returned to the parent is because a process can have more than one child, so there is no function that allows a process to obtain the process IDs of its children.
- The reason **fork** returns 0 to the child because a process can have only a single parent, and the child can always call **getppid** to obtain the process ID of its parent.
- Both the child and the parent continue executing with the instruction that follows the call to **fork**. The child is a copy of the parent. For example, the child gets a copy of the parent's data space, heap, and stack. Note that this is a copy for the child the parent and the child do not share these portions of memory. The parent and the child do share the text segment.
- Copy-on-write (COW) is used on modern implementations: a complete copy of the parent's data, stack and heap is not performed. The shared regions are changed to read-only by the kernel. The kernel makes a copy of that piece of memory only if either process tries to modify these regions.

Characteristic of fork()- File Sharing

- One characteristic of fork is that all file descriptors that are open in the parent are duplicated in the child, because the **dup** function had been called for each descriptor.
- The parent and the child **shareafile** table entry for every open descriptor.
- It is important that the parent and the child share the same file offset. Otherwise, this type of interaction would be more difficult to accomplish and would require explicit actions by the parent.
- There are two normal cases for handling the descriptors after a **fork**:
  - The parent waits for the child to complete.
  - Both the parent and the child go their own ways. After the fork, both the parent and child close the descriptors that they don't need, so neither interferes with the other’s open descriptors. This scenario is often found with network servers.

fork()- File Sharing

- For a process that has three different files opened for standard input, standard output, and standard error, on return from fork, we have the arrangement shown below:



fork()- charactreristics

- Besides the open files, other properties of the parent are inherited by the child:
  - Real user ID, real group ID, effective user ID, and effective group ID
  - Supplementary group IDs
  - Process group ID
  - Session ID
  - Controlling terminal
  - The set-user-ID and set-group-ID flags
  - Current working directory
  - Root directory
  - File mode creation mask
  - Signal mask and dispositions
  - The close-on-exec flag for any open file descriptors
  - Environment
  - Attached shared memory segments
  - Memory mappings
  - Resource limits

fork()- Characteristics

- The differences between the parent and child are:
  - The return values from fork are different.
  - The process IDs are different.
  - The two processes have different parent process IDs:
    - the parent process ID of the child is the parent
    - the parent process ID of the parent doesn’t change.
  - The child’s **tms_utime**, **tms_stime**, **tms_cutime**, and **tms_cstime** values are set to 0.
  - File locks set by the parent are not inherited by the child.
  - Pending alarms are cleared for the child.
  - The set of pending signals for the child is set to the empty set

fork()- Characteristics

- **The two main reasons for fork to fail:**
  - If too many processes are already in the system, which usually means that something else is wrong
  - If the total number of processes for this real user ID exceeds the system's limit. (CHILD_MAX specifies the maximum number of simultaneous processes per real user ID.)
- **The two uses for fork:**
  - When a process wants to duplicate itself so that the parent and the child can each execute different sections of code at the same time.
    - This is common for network servers—the parent waits for a service request from a client. When the request arrives, the parent calls fork and lets the child handle the request. The parent goes back to waiting for the next service request to arrive.
  - When a process wants to execute a different program.
    - This is common for shells. In this case, the child does an exec right after it returns from the fork.

vfork() system call:

- The function `vfork` has the same calling sequence and same return values as fork, but the semantics of the two functions differ.
- `vfork` is intended to create a new process when the purpose of the new purpose is to `exec` a new program.
- The `vfork` function creates the new process, just like `fork`, without copying the address space of the parent into the child, as the child won't refer that address space; the child simply calls `exec (or exit)` right after the `vfork`.
- Instead, the child runs in the address space of the parent until it calls either `exec or exit`.

vfork() system call:

- This optimization is more efficient on some implementations of the UNIX System, but leads to undefined results if the child:
  - modifies any data (except the variable used to hold the return value from `vfork`)
  - makes function calls
  - returns without calling `exec` or `exit`
- Another difference between the two functions is that `vfork` guarantees that the child runs first, until the child calls `exec` or `exit`.
- When the child calls either of these functions, the parent resumes.

exit() system call:

- A process can terminate normally in five ways :
  1. Executing a return from the main function. This is equivalent to calling exit.
  2. Calling the `exit` function, which includes the calling of all exit handlers that have been registered by calling `atexit` and closing all standard I/O streams.
     - ISO C does not deal with file descriptors, multiple processes (parents and children), and job control. The definition of this function is incomplete for a UNIX system.
  3. Calling the `_exit` or `_Exit` function.
     - `_Exit`: defined by ISO C to provide a way for a process to terminate without running exit handlers or signal handlers
     - `_exit`: called by exit and handles the UNIX system-specific details; _exit is specified by POSIX.1.
     - Whether standard I/O streams are flushed depends on the implementation.
     - On UNIX systems, `_Exit` and `_exit` are synonymous and do not flush standard I/O streams.

exit() system call:

4. Executing a `return` from the start routine of the last thread in the process.
   - The return value of the thread is not used as the return value of the process. When the last thread returns from its start routine, the process exits with a termination status of 0.
5. Calling the `pthread_exit` function from the last thread in the process.

- The three forms of abnormal termination:
  - **Calling `abort`**. This is a special case of the next item, as it generates the `SIGABRT` signal.
  - When the process receives certain signals. The signal can be generated by:
    - the process itself, e.g. calling the `abort` function
    - some other processes
    - the kernel, e.g. the process references a memory location not within its address space or tries to divide by 0
  - The last thread responds to a cancellation request. By default, cancellation occurs in a deferred manner: one thread requests that another be canceled, and sometime later the target thread terminates.

exit() system call:

- Regardless of how a process terminates, the same code in the kernel is eventually executed.
- This kernel code closes all the open descriptors for the process, releases the memory that it was using, and so on.
- The terminating process is to be able to notify its parent how it terminated by passing an **`exit status`** as the argument to one of the three exit functions.
- In the case of an abnormal termination, the kernel (not the process) generates a termination status to indicate the reason for the abnormal termination.
- In any case, the parent of the process can obtain the termination status from either the wait or the `waitpid` function.

exit() system call:

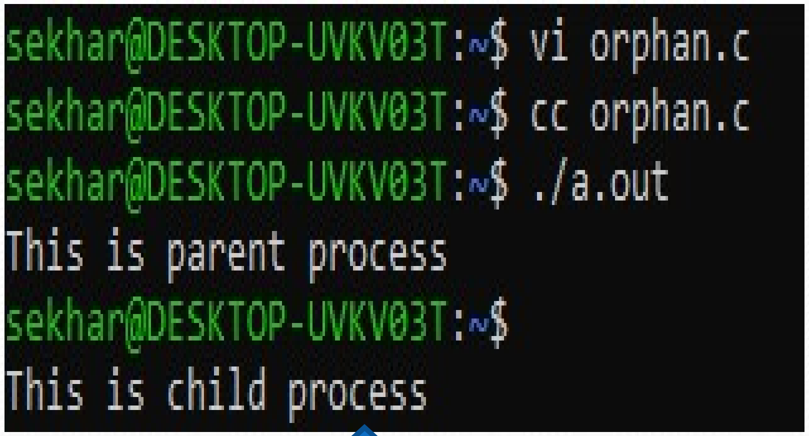
- **Exit status vs. termination status:**
  - **Exit status**: is the argument to one of the three exit functions or the return value from main.
  - **Termination status**: the exit status is converted into a termination status by the kernel when `_exit` is finally called. If the child terminated normally, the parent can obtain the exit status of the child.

Orphan process:

- A process whose parent process no more exists i.e. either `finished` or `terminated` without waiting for its child process to terminate is called an `orphan process`.
- A process can be orphaned `intentionally` or `unintentionally`.
  - An **`intentionally orphaned process`** runs in the background without any manual support. This is usually done to start an indefinitely running service or to complete a long-running job without user attention.
  - An `unintentionally orphaned process` is created when its parent process crashes or terminates. Unintentional orphan processes can be avoided using the process group mechanism.
- The **`orphan process`** is soon adopted by `init` process, once its parent process dies.

Demonstration of orphan process:

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

int main()
{
        // Create a child process
        int pid = fork();

        if (pid > 0)
        printf("This is parent process\n");

        // Note that pid is 0 in child process
        // and negative if fork() fails
        else if (pid == 0)
        {
            sleep(10);
            printf("\nThis is child process\n");
        }

        return 0;
}
```



```
sekhar@DESKTOP-UVKV03T:~$ vi orphan.c
sekhar@DESKTOP-UVKV03T:~$ cc orphan.c
sekhar@DESKTOP-UVKV03T:~$ ./a.out
This is parent process
sekhar@DESKTOP-UVKV03T:~$
This is child process
```

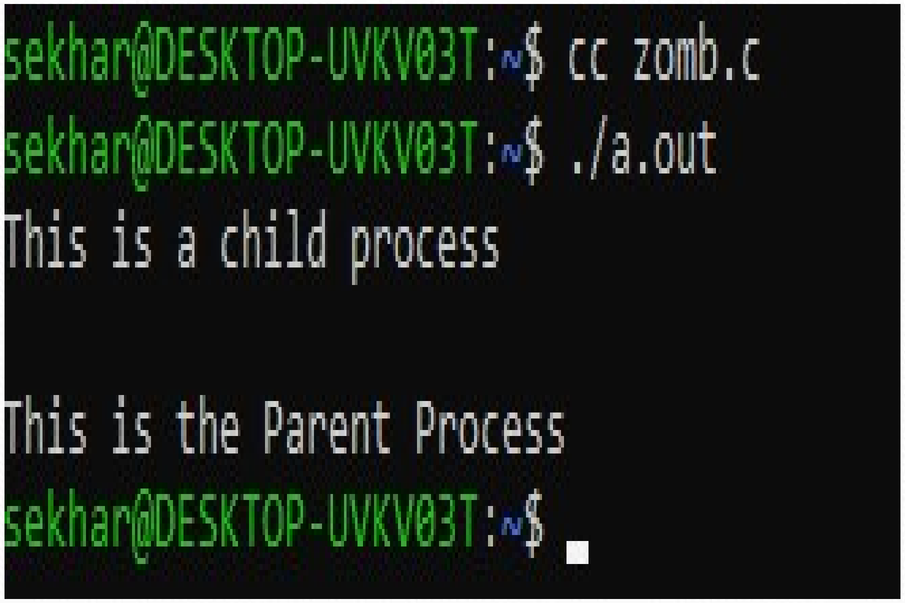
Parent process finishes execution while the child process is running. The child process becomes orphan.

Zombie process:

- A process cannot leave the system until parent process accepts its termination code
- If parent process is dead; `init` *adopts process and* accepts code
- If the parent process is alive but is unwilling to accept the child's termination code (never executes wait()), the child process will remain a **`zombie process`**.
- Zombie processes do not take up system resources:
  - No data, code, stack
  - But use an entry in the system's fixed-size process table

Demonstration of Zombie Process:

```
#include <stdlib.h>
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main()
{

        int pid = fork();

        if (pid > 0)
        {

        sleep(10);
        printf("\nThis is the Parent Process\n");
        }


        else{
        printf("This is a child process\n");
        exit(0);
        }

        return 0;
}
```



```
sekhar@DESKTOP-UVKV03T:~$ cc zomb.c
sekhar@DESKTOP-UVKV03T:~$ ./a.out
This is a child process


This is the Parent Process
sekhar@DESKTOP-UVKV03T:~$
```

Waiting for a Process- wait() & waitpid()

- When a process terminates, either normally or abnormally, the kernel notifies the parent by sending the SIGCHLD signal to the parent.
- Because the termination of a child is an asynchronous event (it can happen at any time while the parent is running).
- This signal is the asynchronous notification from the kernel to the parent.
- The parent can choose to ignore this signal, or it can provide a function that is called when the signal occurs: a signal handler.
- The default action for this signal is to be ignored.

wait() and waitpid():

- A process that calls `wait` or `waitpid` can:
  - Block, if all of its children are still running
  - Return immediately with the termination status of a child, if a child has terminated and is waiting for its termination status to be fetched
  - Return immediately with an error, if it doesn't have any child processes
- If the process is calling `wait` because it received the `SIGCHLD` signal, we expect wait to return immediately. But if we call it at any random point in time, it can block.

wait() and waitpid():

```
#include <sys/wait.h>
pid_t wait(int *statloc);
pid_t waitpid(pid_t pid, int *statloc, int options);

        /* Both return: process ID if OK, 0 (see later), or –1 on error */
```

- The differences between these two functions are:
  - The `wait` function can block the caller until a child process terminates, whereas `waitpid` has an option that prevents it from blocking.
  - The `waitpid` function doesn't wait for the child that terminates first; it has a number of options that control which process it waits for.

wait() and waitpid():

- Macros to examine the termination status returned by wait and waitpid:

Macro	Description
WIFEXITED(status)	True if status was returned for a child that terminated normally.
WIFSIGNALED(status)	True if status was returned for a child that terminated abnormally, by receipt of a signal that it didn't catch.
WIFSTOPPED(status)	True if status was returned for a child that is currently stopped.
WIFCONTINUED(status)	True if status was returned for a child that has been continued after a job control stop (XSI option; waitpid only).

Wait for a specific Process: waitpid()

- `waitpid` function can be used to wait for a specific process.
- The interpretation of the pid argument for `waitpid` depends on its value:
  - *`pid == -1`* → Waits for any child process. In this respect, waitpid is equivalent to wait.
  - *`pid > 0`* → Waits for the child whose process ID equals *pid*.
  - *`pid == 0`* → Waits for any child whose **process group ID** equals that of the calling process.
  - *`pid < -1`* → Waits for any child whose process group ID equals the absolute value of *`pid`*.
- The `waitpid` function returns the process ID of the child that terminated and stores the child's termination status in the memory location pointed to by *`statloc`*.

wait() and waitpid():

- **<u>Errors of `wait` and `waitpid`</u>**
  - With `wait`, the only real error is if the calling process has no children.
  - With `waitpid`, it's possible to get an error if the specified process or process group does not exist or is not a child of the calling process
- **<u>The `waitpid` function provides three features that aren't provided by the `wait` function:</u>**
  - The `waitpid` function lets us wait for one particular process, whereas the wait function returns the status of any terminated child.
  - The `waitpid` function provides a nonblocking version of `wait`. There are times when we want to fetch a child's status, but we don't want to block.
  - The `waitpid` function provides support for job control with the `WUNTRACED` and `WCONTINUED` options.

exec() system call:

- One use of the `fork` function is to create a new process (the child) that then causes another program to be executed by calling one of the `exec` functions.
  - When a process calls one of the `exec` functions, that process is completely replaced by the new program which starts executing at its main function.
  - The process ID does not change across an `exec`, because a new process is not created.
  - `exec` merely replaces the current process (its text, data, heap, and stack segments) with a new program from disk.

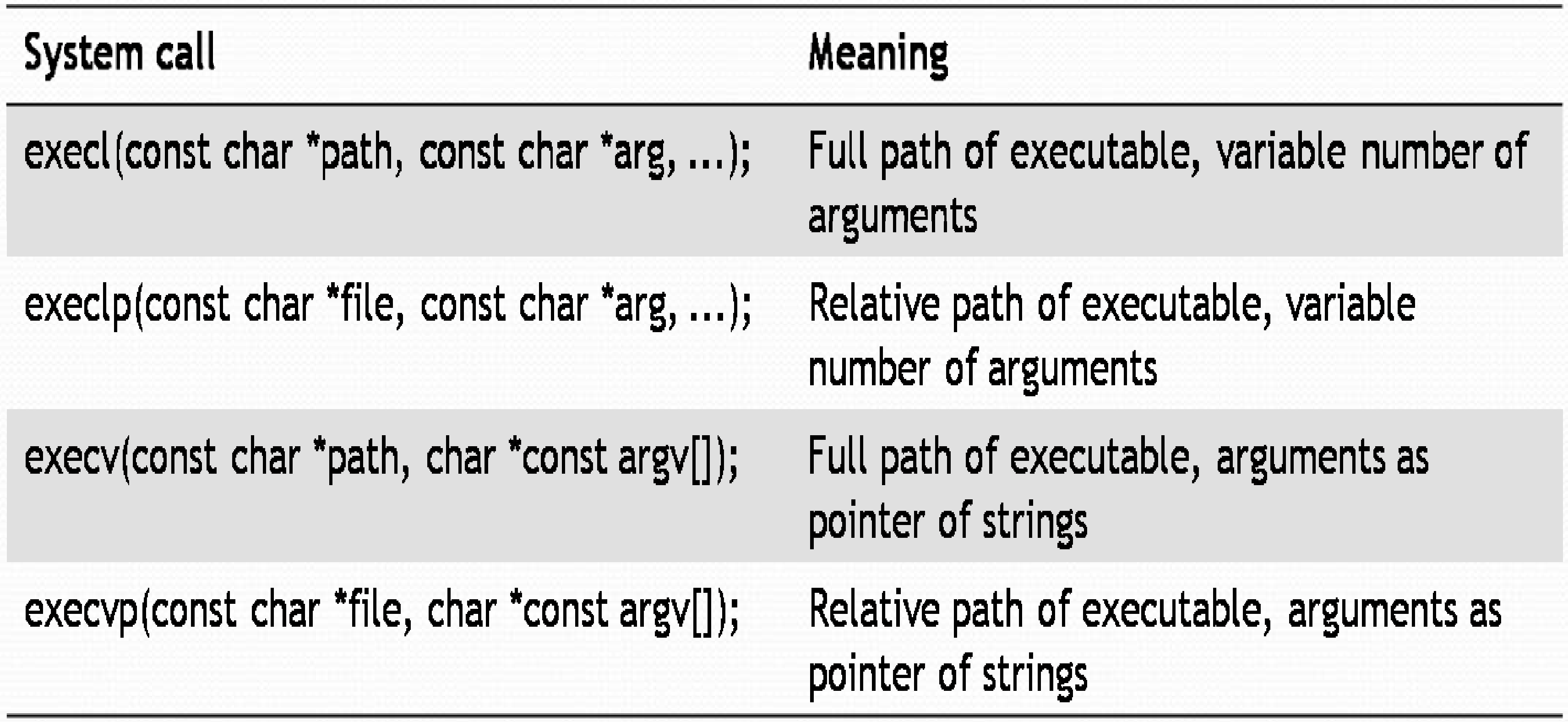
exec() system call:

- There are seven different exec functions:

```
#include <unistd.h>
 int execl(const char *pathname, const char *argo, ... /* (char *)o */ );
int execv(const char *pathname, char *const argv[]);
int execle(const char *pathname, const char *argo, ... /* (char *)o, char
*const envp[] */ );
int execve(const char *pathname, char *const argv[], char *const
envp[]);
 int execlp(const char *filename, const char *argo, ... /* (char *)o */ );
int execvp(const char *filename, char *const argv[]);
 int fexecve(int fd, char *const argv[], char *const envp[]);

                    /* All seven return: –1 on error, no return on success */
```

The first four take a pathname argument, the next two take a filename argument, and the last one takes a file descriptor argument.

exec() family of system calls:



System call	Meaning
execl(const char *path, const char *arg, ...);	Full path of executable, variable number of arguments
execlp(const char *file, const char *arg, ...);	Relative path of executable, variable number of arguments
execv(const char *path, char *const argv[]);	Full path of executable, arguments as pointer of strings
execvp(const char *file, char *const argv[]);	Relative path of executable, arguments as pointer of strings