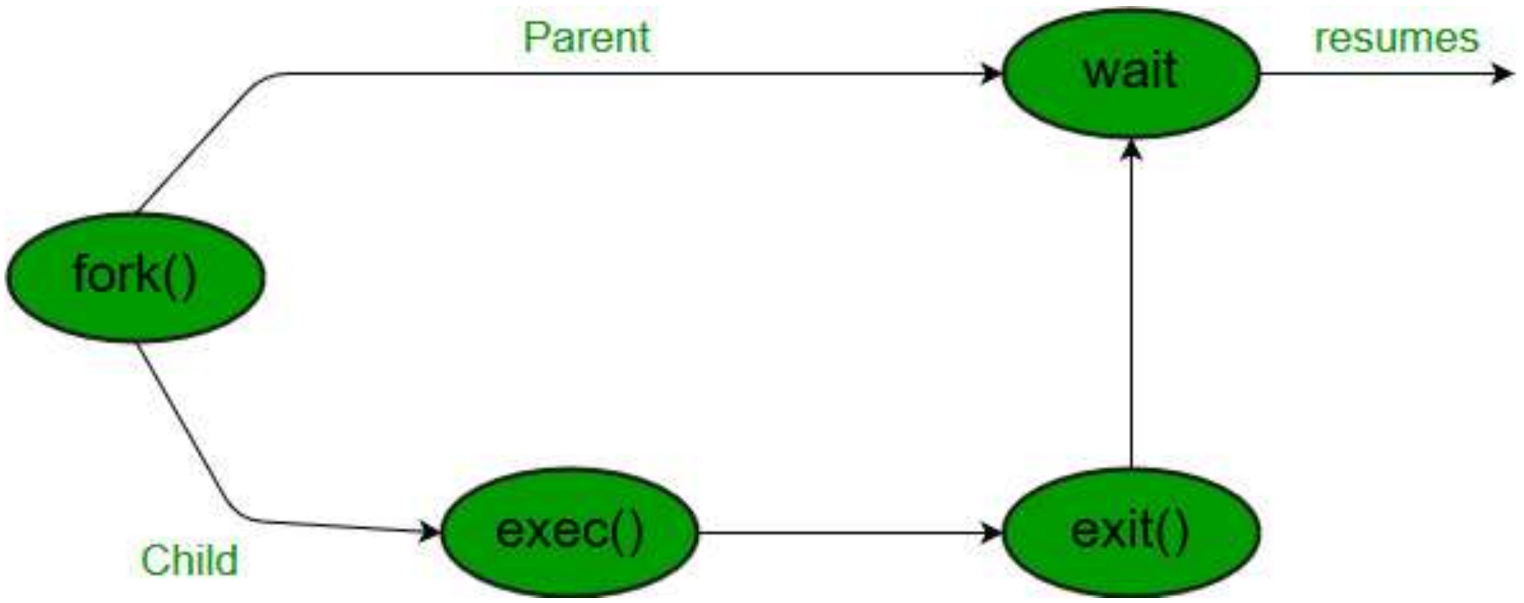
Process Creation
Part 2

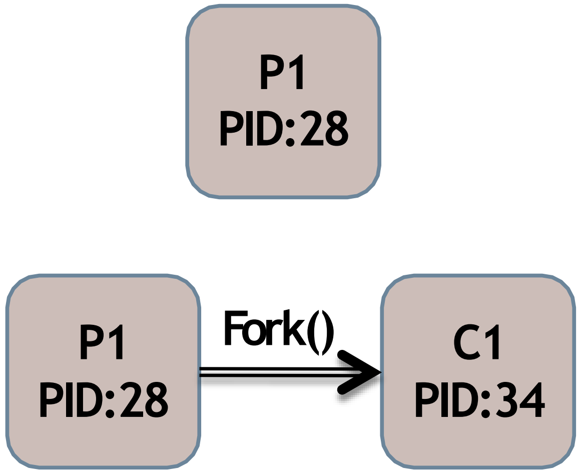
System Calls

- fork()
- exec()
- wait()
- exit()
- getpid(), getppid()
- getpgrp()


Parent
wait
resumes
fork()
Child
exec()
exit()

The "fork()" system call - PID

- pid<0: the creation of a child process was unsuccessful.
- pid==0: the newly created child.
- pid>0: the *process ID* of the child process passes to the parent.



Consider a piece of program

```
...
pid_t pid = fork();
printf("PID: %d\n", pid);
...
```

The parent will print:

```
PID: 34
```

And the child will **always** print:

```
PID: 0
```

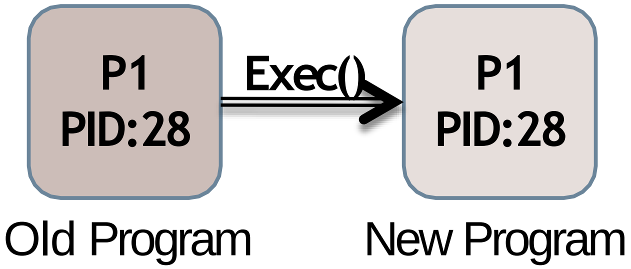
The "exec()" System Call

- The `exec()` call replaces a current process' image with a new one (i.e. loads a new program within current process).
- The new image is either regular executable **binary file** or a **shell script**.
- There's **not** a syscall under the name `exec()`. By `exec()` we usually refer to a family of calls:
  - int execl(char *path, char *arg, ...);
  - int execv(char *path, char *argv[]);
  - int execle(char *path, char *arg, ..., char *envp[]);
  - int execve(char *path, char *argv[], char *envp[]);
  - int execlp(char *file, char *arg, ...);
  - int execvp(char *file, char *argv[]);

Where l=argument list, v=argument vector, e=environmental vector, and p=search path.

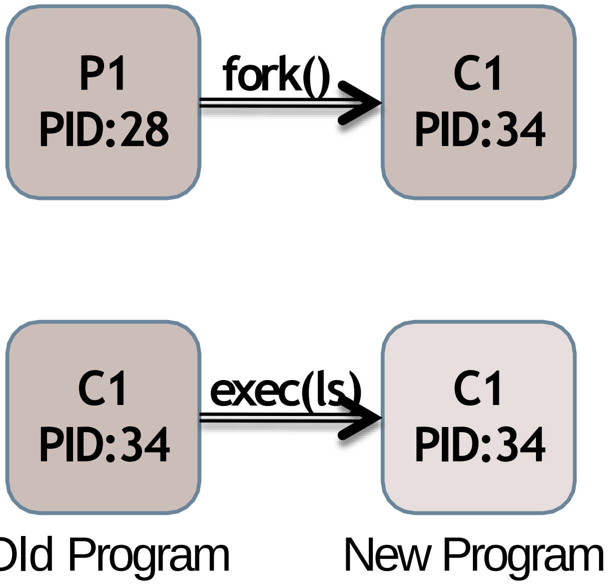
The "exec()" System Call

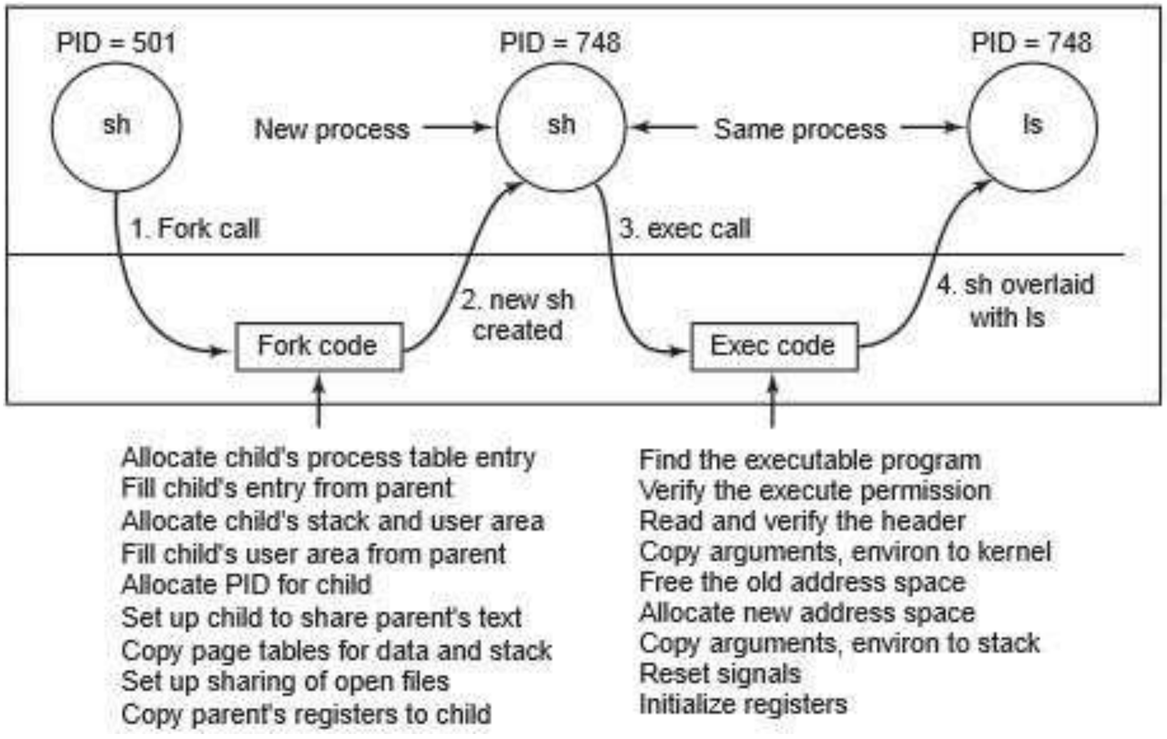
- Upon success, `exec()` never returns to the caller. It replaces the current process image, so it cannot return anything to the program that made the call. If it does return, it means the call failed. Typical reasons are: non-existent file (bad path) or bad permissions.
- As a new process is not created, the process identifier (PID) does not change, but the **machine code**, **data**, **heap**, and **stack** of the process are replaced by those of the new program.
- For more info: `man 3 exec;`



"fork()" and "exec()" combined

- Often after doing `fork()` we want to load a new program into the child. *E.g.*: a shell





Allocate child's process table entry
Fill child's entry from parent
Allocate child's stack and user area
Fill child's user area from parent
Allocate PID for child
Set up child to share parent's text
Copy page tables for data and stack
Set up sharing of open files
Copy parent's registers to child

Find the executable program
Verify the execute permission
Read and verify the header
Copy arguments, environ to kernel
Free the old address space
Allocate new address space
Copy arguments, environ to stack
Reset signals
Initialize registers

- int execl(const char *path, const char *arg, …, NULL);

execl(“./lab7",“./lab7",NULL);

- int execlp(const char *file, const char *arg, …, NULL);

execlp("echo","Hello World!",NULL);

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

execv(path, (char**)arg);

- int execvp(const char *file, char *const argv[]);

execvp("echo", (char**)arg);

- int execle(const char *path, const char *arg, …, NULL, char * const envp[]);

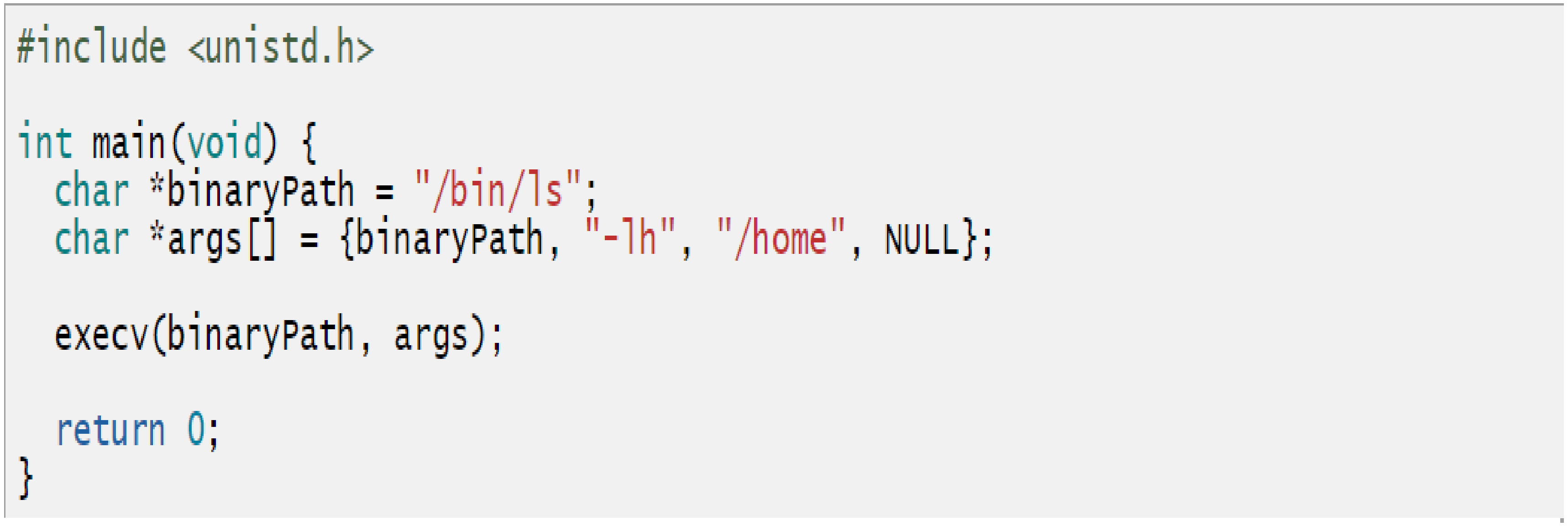
execle(path, path, arg1, NULL, (char**)arg);

- int execve(const char *file, char *const argv[], char *const envp[]);

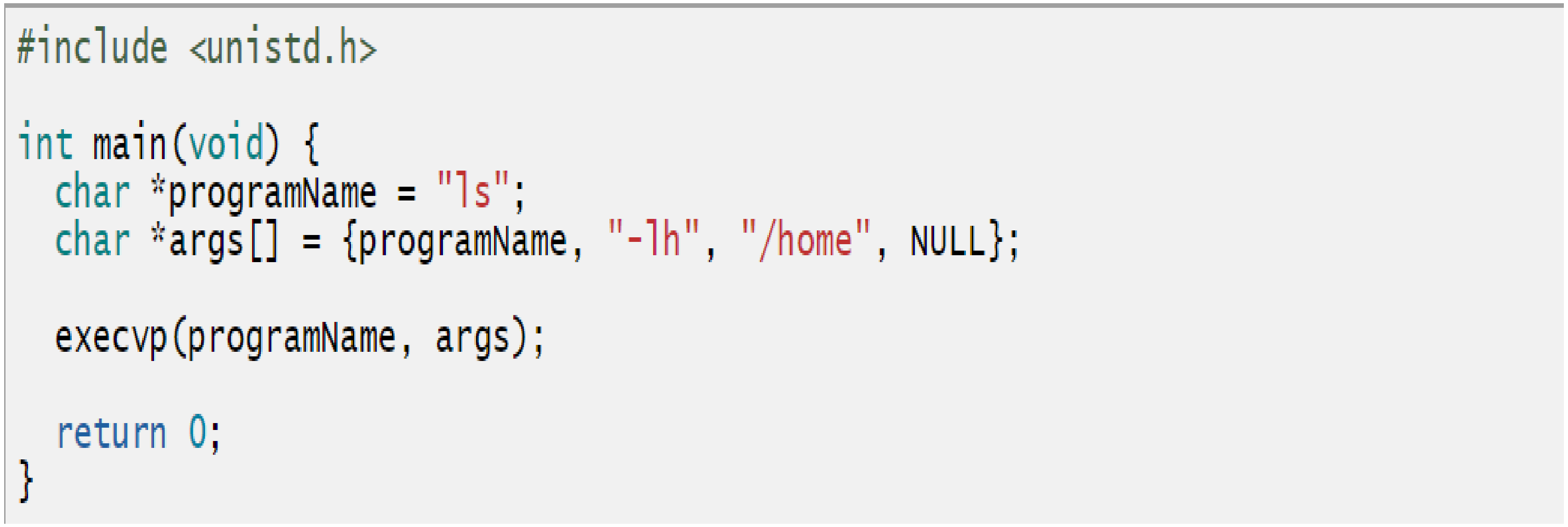
execve(path, (char**)arg, (char**)arg1);

```c
#include <unistd.h>

int main(void) {
  char *binaryPath = "/bin/ls";
  char *arg1 = "-lh";
  char *arg2 = "/home";

  execl(binaryPath, binaryPath, arg1, arg2, NULL);

  return 0;
}
```

```c
#include <unistd.h>

int main(void) {
  char *programName = "ls";
  char *arg1 = "-lh";
  char *arg2 = "/home";

  execlp(programName, programName, arg1, arg2, NULL);

  return 0;
}
```



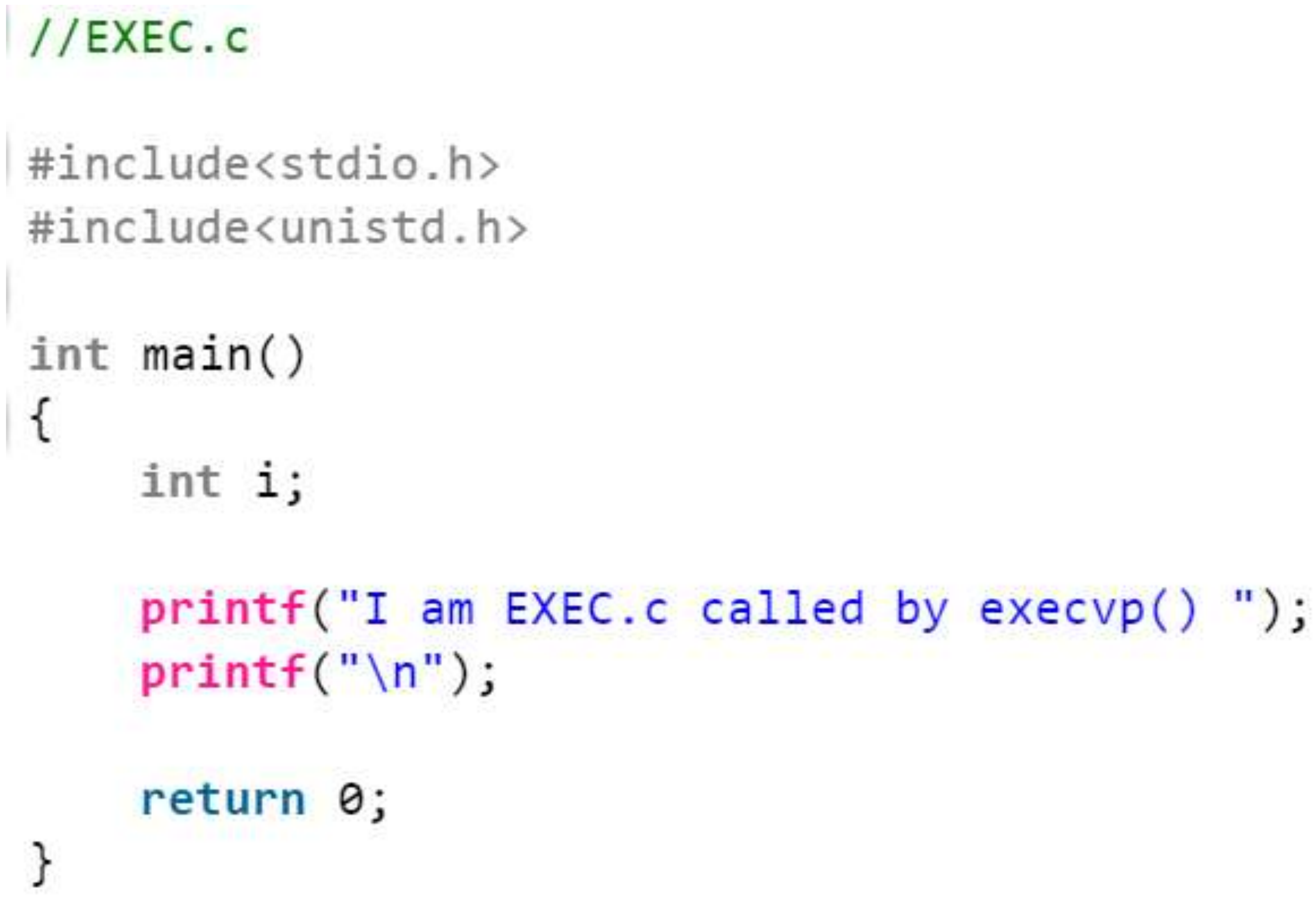
```c
#include <unistd.h>

int main(void) {
  char *binaryPath = "/bin/ls";
  char *args[] = {binaryPath, "-lh", "/home", NULL};

  execv(binaryPath, args);

  return 0;
}
```



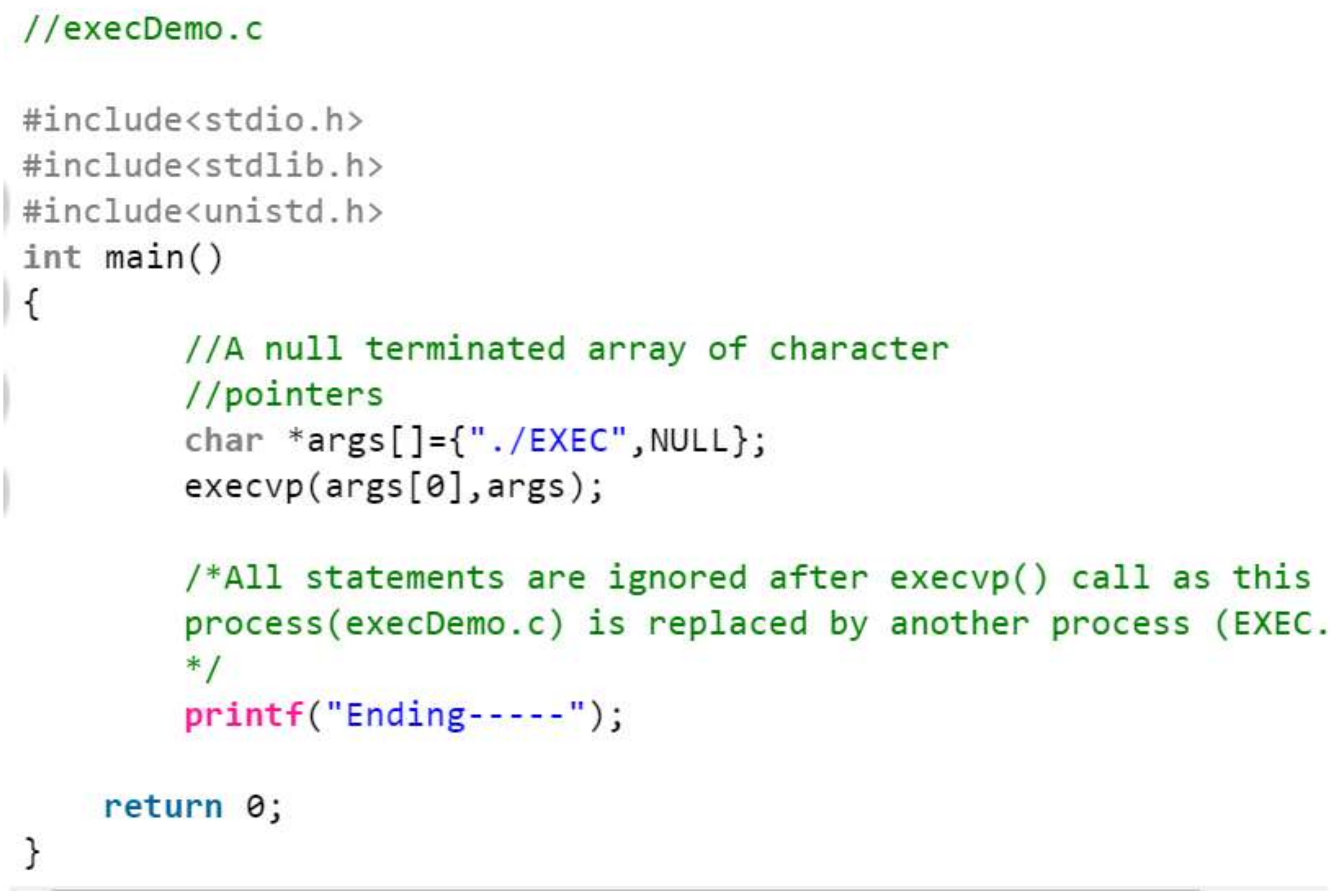
```c
#include <unistd.h>

int main(void) {
  char *programName = "ls";
  char *args[] = {programName, "-lh", "/home", NULL};

  execvp(programName, args);

  return 0;
}
```

```c
#include <unistd.h>

int main(void) {
  char *binaryPath = "/bin/bash";
  char *arg1 = "-c";
  char *arg2 = "echo "Visit $HOSTNAME:$PORT from your browser."";
  char *const env[] = {"HOSTNAME=www.linuxhint.com", "PORT=8080", NULL};

  execle(binaryPath, binaryPath,arg1, arg2, NULL, env);

  return 0;
}
```

```c
#include <unistd.h>

int main(void) {
  char *binaryPath = "/bin/bash";
  char *const args[] = {binaryPath, "-c", "echo "Visit $HOSTNAME:$PORT
    from your browser."", NULL};
  char *const env[] = {"HOSTNAME=www.linuxhint.com", "PORT=8080", NULL};

  execve(binaryPath, args, env);

  return 0;
}
```



```c
//EXEC.c

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

int main()
{
    int i;

    printf("I am EXEC.c called by execvp() ");
    printf("\n");

    return 0;
}
```



```c
//execDemo.c

#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
int main()
{
        //A null terminated array of character
        //pointers
        char *args[]={"./EXEC",NULL};
        execvp(args[0],args);

        /*All statements are ignored after execvp() call as this
        process(execDemo.c) is replaced by another process (EXEC.
        */
        printf("Ending-----");

    return 0;
}
```