Operating systems

Processes

Created by
Enrico Fraccaroli
enrico.fraccaroli@univr.it

Table of Contents

Attributes of a process

Recall to the previous lecture

Process

A *process* is an instance of an executing program.

From the Kernel's point of view, a process consists of:

- user-space memory containing program code,
- the variables used by that code, and
- a set of kernel data structures that maintain information about the process's state (e.g. page tables, table of open files, signals to be delivered, process resource usage and limits, . . .)

Attributes of a process

Identifier

Process identifier

The `getpid` system call returns the process ID of the calling process.

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

pid_t getpid(void);
```

The pid_t data type used for the return value of `getpid` is an integer type for the purpose of storing process IDs.
With the exception of a few system processes such as init (process ID 1), there is no fixed relationship between a program and the process ID of the process that is created to run that program.

```
// to see the init process
user@localhost[~]$ ps auxf
```

N.B. The `getpid` system call is always successful!

Real and effective process user-ID (1/3)

The `getuid` and `getgid` system calls return, respectively, the real user ID and real group ID of the calling process. The `geteuid` and `getegid` system calls perform the corresponding tasks for the effective IDs.

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

uid_t getuid(void);  // Real user ID
uid_t geteuid(void); // Effective user ID

gid_t getgid(void);  // Real group ID
gid_t getegid(void); // Effective group ID
```

N.B. They are always successful!

Real and effective process user-ID (2/3)

- **real** user ID and group ID identify the user and group to which the process belongs,
- **effective** user ID and group ID are used to determine the permissions granted to a process when it tries to perform operations.

Here is the content of file `program.c`:

```
#include <unistd.h>
#include <sys/types.h>
int main (int argc, char *argv[]) {
    printf("PID: %d, user-ID: real %d, effective %d\n",
        getpid(), getuid(), geteuid());
    return 0;
}
```

Real and effective process user-ID (3/3)

```
user@localhost[~]$ gcc -o program program.c
user@localhost[~]$ ls -l program
   -r-xr-xr-x 1 Professor Professor 8712 Jan 16 16:27 program

user@localhost[~]$ ./program
   PID: 1234, user-ID: real 1000, effective 1000

user@localhost[~]$ sudo ./program
   PID: 1423, user-ID: real 0, effective 0

user@localhost[~]$ sudo chmod u+s program
user@localhost[~]$ ls -l program
   -r-sr-xr-x 1 root Professor 8712 Jan 16 16:27 program

user@localhost[~]$ ./program
   PID: 4321, user-ID: real 1000, effective 0
```

Keep in mind: if the **S**ticky bit is unset, then the user's permissions are granted to the executable to perform operations. If it is set, then the owner's permissions are granted to executable.

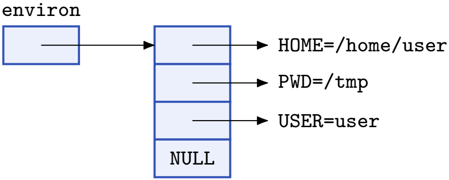
Attributes of a process

Environment

Process environment (1/5)

Each process has an associated array of strings called the environment list, or simply the environment. Each of these strings is a definition of the form `name = value`. When a new process is created, it **inherits** a copy of its parent's environment.

The structure of the environment list is as follows:



Process environment (2/5)

Within a C program, the environment list can be accessed by either:

1. using the global variable `char **environ`. Originally it was used specifically in POSIX systems, now this technique is widely used and supported by many systems.
2. or you can also receive the current environment as third argument of the `main` function. This technique is recognized as standard C, but it is not supported by all the operating systems.

Process environment (3/5)

Displaying the process environment, **first** technique:

```
#include <stdio.h>
// Global variable pointing to the enviroment of the process.
extern char **environ;

int main(int argc, char *argv[]) {
    for (char **it = environ; (*it) != NULL; ++it) {
        printf("--> %s\n", *it);
    }
    return 0;
}
```

```
user@localhost[~]$ ./program
--> $HOME=/home/Professor
--> $PWD=/tmp
--> $USER=Professor
```

Process environment (4/5)

Displaying the process environment, **second** technique:

```c
#include <stdio.h>
int main(int argc, char *argv[], char* env[]) {
    for (char **it = env; (*it) != NULL; ++it) {
        printf("--> %s\n", *it);
    }
    return 0;
}
```

```
user@localhost[~]$ ./program
--> $HOME=/home/Professor
--> $PWD=/tmp
--> $USER=Professor
```

Process environment (5/5)

```c
#include <stdlib.h>
// Returns pointer to (value) string, or NULL if no such variable exists
char *getenv(const char *name);
// Returns 0 on success, or -1 on error
int setenv(const char *name, const char *value, int overwrite);
// Returns 0 on success, or -1 on error
int unsetenv(const char *name);
```

- given a variable name, `getenv` returns a pointer to its string value, or `NULL` if no environment variable exists with the specified name.
- `setenv` adds name=value to the environment, unless a variable identified by name already exists and overwrite has the value 0. If overwrite is nonzero, the environment is always changed.
- `unsetenv` removes the variable identified by name from the environment.

Attributes of a process

Working directory

Process working directory (1/3)

A process can retrieve its current working directory using `getcwd`.

```
#include <unistd.h>

// Returns cwdbuf on success, or NULL on error.
char *getcwd(char *cwdbuf, size_t size);
```

On success, `getcwd` returns a pointer to `cwdbuf` as its function result. If the pathname for the current working directory exceeds size bytes, then `getcwd` returns `NULL`.

The caller must allocate the `cwdbuf` buffer to be at least size bytes in length. (Normally, we would size `cwdbuf` using the `PATH_MAX` constant.)

Process working directory (2/3)

The `chdir` system call changes the calling process's current working directory to the relative or absolute pathname specified in pathname.

```c
#include <unistd.h>

// Returns 0 on success, or -1 on error
int chdir(const char *pathname);
```

The `fchdir` system call does the same as `chdir`, except that the directory is specified via a file descriptor previously obtained by opening the directory with `open`.

```c
#define _BSD_SOURCE
#include <unistd.h>

// Returns 0 on success, or -1 on error.
int fchdir(int fd);
```

Process working directory (3/3)

```c
char buf[PATH_MAX];
// Open the current working directory
int fd = open(".", O_RDONLY);
getcwd(buf, PATH_MAX);
printf("1) Current dir:\n\t%s\n", buf);

// Move the process into /tmp
chdir("/tmp");
getcwd(buf, PATH_MAX);
printf("2) Current dir:\n\t%s\n", buf);

// Move the process back into the initial directory
fchdir(fd);
getcwd(buf, PATH_MAX);
printf("3) Current dir:\n\t%s\n", buf);

// Close the file descriptor
close(fd);
```

Here is the output:

```
1) Current dir:
   /home/Professor
2) Current dir:
   /tmp
3) Current dir:
   /home/Professor
```

Attributes of a process

File descriptor table

Process file descriptor table (1/5)

Each process has an associated *file descriptor table*. Each entry represents an input/output resource (e.g. file, pipe, socket) used by the process. The directory /proc/<PID>/fd[1] contains a symbolic link for each entry of the file descriptor table of a process.

A created process has always three file descriptors (`stdin`, `stdout`, `stderr`)

```
user@localhost[~]$ sleep 30 &
[1] 1344

user@localhost[~]$ ls -l /proc/1344/fd
total 0
lrwx------ 1 Professor Professor 0 Jan 18 12:35 0 -> /dev/pts/0
lrwx------ 1 Professor Professor 0 Jan 18 12:35 1 -> /dev/pts/0
lrwx------ 1 Professor Professor 0 Jan 18 12:35 2 -> /dev/pts/0
```

[1] Process information pseudo-file system, it doesn't contain 'real' files but runtime system information.

Process file descriptor table (2/5)

Displaying the file descriptor entries of a process

```c
char buf[PATH_MAX];
// Replace %i with PID, and store the resulting string in buf.
snprintf(buf, PATH_MAX, "/proc/%i/fd/", getpid());

DIR *dir = opendir(buf);
struct dirent *dp;
while ((dp = readdir(dir)) != NULL) {
    if ((strcmp(dp->d_name,".") != 0) && (strcmp(dp->d_name,"..") != 0)) {
        printf("\tEntry: %s\n", dp->d_name);
    }
}
closedir(dir);
```

```
user@localhost[~]$ ./program
    Entry: 0 // link to stdin
    Entry: 1 // link to stdout
    Entry: 2 // link to stderr
    Entry: 3 // link to /proc/<PID>/fd directory
```

Process file descriptor table (3/5)

Important

At a new entry of the file descriptor table is always assigned the lowest available index.

Redirecting the standard output stream of a process to a file named myfile

```c
// We close STDOUT which has FD 1. The remaining file descriptors  have
//  index 0 (stdin) and 2 (stderr).
close(STDOUT_FILENO);
// We open a new file, to which will be assigned FD 1 automatically
//  because it is the lowest available index in the table.
int fd = open("myfile", O_TRUNC | O_CREAT | O_WRONLY, S_IRUSR | S_IWUSR);
// Printf uses the FD 1, thus, it will print on the file.
printf("ciao\n");
```

No string will be displayed on terminal, as stdout stream is closed.
However, all string printed by printf will be reported in myfile.

Process file descriptor table (4/5)

The `dup` system call takes an open file descriptor, and returns a new descriptor that refers to the same open file description. The new descriptor is guaranteed to be the lowest unused file descriptor.

```
#include <unistd.h>

// Returns (new) file descriptor on success, or -1 on error.
int dup(int oldfd);
```

Process file descriptor table (5/5)

Redirecting stdout and stderr of a process to a file named myfile

```
// FDT: [0, 1, 2] -> [0, 2]
close(STDOUT_FILENO);
// FDT: [0, 2]      -> [0]
close(STDERR_FILENO);
// FDT: [0]         -> [0, 1]
int fd = open("myfile", O_TRUNC | O_CREAT | O_WRONLY, S_IRUSR | S_IWUSR);
// FDT: [0, 1]      -> [0, 1, 2]
dup(1);
// FDT: [0: STDIN, 1: myfile, 2: myfile]
printf("Have a good ");
fflush(stdout);
fprintf(stderr, "day!\n");
```

```
user@localhost[~]$ cat myfile
Have a good day!
```

Operations with processes

Operations with processes

Termination

Process termination (1/5)

The process calling `_exit()` is always successfully terminated.

```c
#include <unistd.h>

void _exit(int status);
```

The first byte of the `status` argument defines the *termination status* of the process. By convention, the zero value indicates that the process terminated **successfully**, a nonzero status value indicates that the process terminated **unsuccessfully**.

Process termination (2/5)

Programs generally call `exit()` rather than `_exit()`.

```
#include <stdlib.h>     // N.B. provided by C library

void exit(int status);
```

C library defines the macros `EXIT_SUCCESS` (0) and `EXIT_FAILURE` (1)

The following actions are performed by `exit()` method:

- Call exit handlers (see next slides).
- Flush `stdio` stream buffers.
- Call `_exit()`, using the value supplied in `status`.

Process termination (3/5)

An *exit handler* is a function that is registered during the life of a process. It is automatically called during the process termination via *exit()*.

```
#include <stdlib.h>

// Returns 0 on success, or nonzero on error.
int atexit(void (*func)(void));
```

The `atexit()` adds the provided function pointer `func` to a list of functions that are called during the process termination.

`func` has to be defined to take no argument and return no value.

If more *exit handler* are registered, then they are called in reverse order of registration.

Process termination (4/5)

```c
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
void func1() { printf("\tAtexit function 1 called\n"); }
void func2() { printf("\tAtexit function 2 called\n"); }
int main (int argc, char *argv[]) {
    if (atexit(func1) != 0 || atexit(func2) != 0)
        _exit(EXIT_FAILURE);
    exit(EXIT_SUCCESS);
}
```

Here is the output of the program:

```
user@localhost[~]$ ./exit_handlers
    Atexit function 2 called
    Atexit function 1 called
```

Process termination (5/5)

One other way in which a process may terminates is to return from `main()`

- performing an explicit `return n` is equivalent to calling `exit(n)`;
- performing an implicit `return` or falling off the end of `main()` is equivalent to calling `exit(0)` in C99 standard. Otherwise, the behaviour of the process is undefined.

Operations with processes

Creation

Process creation (1/6)

```c
#include <unistd.h>

// In parent: returns process ID of child on success, or -1 on error.
// In created child: always returns 0.
pid_t fork(void);
```

The `fork()` system call creates a new process, the *child*, which is an almost exact duplicate of the calling process, the *parent*.

- After the execution of a `fork()`, two processes exist, and, in each process, execution continues from the point where the `fork()` returns.
- It is indeterminate which of the two processes is next scheduled to use the CPU.
- The *child* receives duplicates of all parent's file descriptors and the attached shared memories (see Filesystem and IPC slides)

Process creation (2/6)

```c
#include <unistd.h>
int main (int argc, char *argv[]) {
    int stack = 111;
    pid_t pid = fork();
    if (pid == -1)
        errExit("fork");
    // -->Both parent and child come here !!!<--
    if (pid == 0)
        stack = stack * 4;
    printf("\t%s stack %d\n", (pid==0) ? "(child )" : "(parent)", stack);
}
```

Process creation (3/6)

Here is the output of the program:

```
user@localhost[~]$ ./example_fork
    (parent) stack 111
    (child ) stack 444

user@localhost[~]$ ./example_fork
    (child ) stack 444
    (parent) stack 111
```

The terminal output shows that:

- the *child* process gets its own copy of the parent's variables;
- the execution of both *parent* and *child* processes continue from the point where the `fork()` returned;

Process creation (4/6)

Each process has a parent, namely the process that created it (see previous slides about `fork()`).

```
#include <unistd.h>

// Always successfully returns PID of caller's parent.
pid_t getppid(void);
```

The ancestor of all processes is the process `init` (PID=1). If a *child* process becomes **orphaned** because its *parent* terminates, then the *child* is “adopted" by the process `init`. The subsequent calls to `getppid()` in the *child* return 1.

Process creation (5/6)

```c
#include <unistd.h>
int main (int argc, char *argv[]) {
    pid_t pid = fork();
    if (pid == -1) {
        errExit("fork");
    }
    if (pid == 0) {
        printf("(child ) PID: %d PPID: %d\n", getpid(), getppid());
    }
    else {
        printf("(parent) PID: %d PPID: %d\n", getpid(), getppid());
    }
    return 0;
}
```

Process creation (6/6)

The execution of the previous example has **three** different scenarios:

1. The *child* is executed after the *parent*, and the *parent* is not terminated

```
(parent) PID: 402 PPID: 350
(child ) PID: 403 PPID: 402
```

2. The *child* is executed before the *parent* process

```
(child ) PID: 403 PPID: 402
(parent) PID: 402 PPID: 350
```

3. The *child* is executed after the termination of the *parent*

```
(parent) PID: 402 PPID: 350
(child ) PID: 403 PPID: 1
```

Question: Whose the process ID 350 may belong to?

Operations with processes

Monitoring

Monitoring a child Process (1/12)

The `wait` system call waits for one of the children of the calling process to terminate. (see `waitpid` for *status* input argument).

```
#include <sys/wait.h>

// Returns PID of terminated child, or -1 on error.
pid_t wait(int *status);
```

The following actions are performed by `wait`:

- If calling process have no unwaited-for children, then `wait` returns -1 and `errno` is `ECHILD`.
- If no child has yet terminated, then `wait` blocks the calling process until a child terminates. If a child has already terminated, then `wait` returns immediately.
- If `status` is not `NULL`, information about the terminated child is stored in the integer variable which `status` points to (next slides).

Monitoring a child Process (2/12)

```
for (int i = 1; i <= 3; ++i) {
    // Fork and ignore fork failures.
    if (fork() == 0) {
        printf("Child %d sleeps %d seconds...\n", getpid(), i);
        // Suspends the calling process for i seconds
        sleep(i);
        _exit(0);
    }
}
pid_t child;
while ((child = wait(NULL)) != -1)
    printf("wait() returned child %d\n", child);
if (errno != ECHILD)
    printf("(wait) An unexpected error...\n");
```

Monitoring a child Process (3/12)

Example output:

```
user@localhost[~]$ ./example_wait
child 75 sleeps 1 seconds
child 76 sleeps 2 seconds
child 77 sleeps 3 seconds
wait() returned child 75
wait() returned child 76
wait() returned child 77
```

Question: What happens to a child that terminates before its parent has had the chance to perform a `wait`?

Monitoring a child Process (4/12)

The kernel deals with this situation by turning the terminated child into a *zombie* process. This means that most of the resources held by the child are released back to the system. The only parts of the terminated process still maintained are:

1. its process ID;
2. its termination status;
3. the resource usage statistics.

If the parent process terminates without calling `wait`, then the zombie child process is "adopted" by the process `init`, which will perform a `wait` system call some time later eventually.

Monitoring a child Process (5/12)

The `waitpid` system call suspends execution of the calling process until a child specified by `pid` argument has changed state.

```c
#include <sys/wait.h>

// Returns a PID, 0, or -1 on error.
pid_t waitpid(pid_t pid, int *status, int options);
```

The `status` argument is the same of `wait` (see next slides). The value of `pid` determines what child process we want to wait.

- `pid` $\geq$ 0, wait for the child having PID equals to `pid`.
- `pid` $=$ 0, wait for any child in the same caller's process group[2].
- `pid` $<$ -1, wait for any child in the process group $|$`pid`$|$.
- `pid` $=$ -1, wait for any child.

[2] The processes can be organized in process group and sessions to support shell job control in Linux (topic not included in Operating System program)

Monitoring a child Process (6/12)

```
#include <sys/wait.h>

// Returns a PID, 0, or -1 on error.
pid_t waitpid(pid_t pid, int *status, int options);
```

The `options` argument of the `waitpid` system call is an OR of zero or more of the following constants:

- `WUNTRACED`: return when a child is stopped by a signal or it terminates.
- `WCONTINUED`: return when a child has been resumed by delivery of a `SIGCONT` signal.
- `WNOHANG`: If no child specified by `pid` has yet changed state, then return immediately, instead of blocking (i.e., perform a "poll"). In this case, the return value of `waitpid` is 0.
- 0: then `waitpid` waits only for terminated children.

Monitoring a child Process (7/12)

```c
pid_t pid;
for (int i = 0; i < 3; ++i) {
    pid = fork();
    if (pid == 0) {
        // Code executed by the child process...
        _exit(0);
    }
}
// The parent process only waits for the last created child
waitpid(pid, NULL, 0);
```

Monitoring a child Process (7/12)

```c
pid_t pid = fork();
if (pid == 0) {
    // Code executed by the child process
} else {
    // Waiting for a terminated/stopped | resumed child process.
    waitpid(pid, NULL, WUNTRACED | WCONTINUED);
}
```

Monitoring a child Process (9/12)

The `status` value set by `waitpid`, and `wait`, let us distinguish the following events for a child process:

1. The child process terminated by calling `_exit` (or `exit`)
   - The macro `WIFEXITED` returns true if the child exited normally.
   - The macro `WEXITSTATUS` returns the exit status of the child process.

	15 … 8	7 … 0
Normal Termination	exit status (0-255)	0

```
waitpid(-1, &status, WUNTRACED | WCONTINUED);
if (WIFEXITED(status)) {
    printf("Child exited, status=%d\n", WEXITSTATUS(status));
}
```

Monitoring a child Process (10/12)

2. The child was terminated by the delivery of an unhanded signal.
   - The macro `WIFSIGNALED` returns true if the child was killed by a signal.
   - The macro `WTERMSIG` returns the number of the signal that caused the process to terminate.

	15 … 8	7	6 … 0
Killed by Signal	unused		termination signal

```
waitpid(-1, &status, WUNTRACED | WCONTINUED);
if (WIFSIGNALED(status)) {
    printf("Child killed by signal %d (%s)", WTERMSIG(status), strsignal(WTERMSIG(status)));
}
```

The `strsignal(int sig)` is a method of `string.h` which returns a string describing the signal `sig` (see IPC part 1).

Monitoring a child Process (11/12)

3. The child was stopped by a signal.
   - The macro `WIFSTOPPED` returns true if the child process was stopped by a signal.
   - The macro `WSTOPSIG(status)` returns the number of the signal that stopped the process.

	15 … 8	7 … 0
Stopped by Signal	stopping signal	0x7F

```c
waitpid(-1, &status, WUNTRACED | WCONTINUED);
if (WIFSTOPPED(status)) {
    printf("Child stopped by signal %d (%s)\n", WSTOPSIG(status), strsignal(WSTOPSIG(status)));
}
```

Monitoring a child Process (12/12)

4. The child was resumed by a `SIGCONT` signal.
   - The macro `WIFCONTINUED` returns true if the child was resumed by delivery of `SIGCONT`.

	15	0
Continued by Signal	0xFFFF	

```
waitpid(-1, &status, WUNTRACED | WCONTINUED);
if (WIFCONTINUED(status)) {
    printf("child resumed by a SIGCONT signal\n");
}
```

or

```
waitpid(-1, &status, WCONTINUED);
printf("child resumed by a SIGCONT signal\n");
```

Program execution (exec library functions)

The exec Library Functions (1/2)

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

```
#include <unistd.h>
// None of the following returns on success, all return -1 on error.
int execl (const char *path, const char *arg, ... ); // ... variadic functions
int execlp(const char *path, const char *arg, ... );
int execle(const char *path, const char *arg, ... , char *const envp[]);
int execv (const char *path, char *const argv[]);
int execvp(const char *path, char *const argv[]);
int execve(const char *path, char *const argv[], char *const envp[]);
```

Note: 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 Library Functions (2/2)

function	`path`	arguments (`argv`)	environment (`envp`)
exec**l**	pathname	list	caller's environ
exec**lp**	filename	list	caller's environ
exec**le**	pathname	list	array
exec**v**	pathname	array	caller's environ
exec**vp**	filename	array	caller's environ
exec**ve**	pathname	array	array

- **path**: *pathname* means the absolute path to an executable. While *filename* means the name of an executable, which is sought in the list of directories specified in the `PATH` environment variable.
- **argv**: a NULL-terminated list/array of pointers to string defining the command line argument of the program.
- **envp**: a NULL-terminated array of pointers to string (`name = value`) defining the environment of the program.

Example (1/2)

Program: example.c

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
int main(int argc, char *argv[])
{
    printf("PID of example.c = %d\n", getpid());
    char *args[] = {"Hello", "C", "Programming", NULL};
    execv("./hello", args);
    printf("Back to example.c");
    return 0;
}
```

Example (2/2)

Program: hello.c

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
int main(int argc, char *argv[])
{
    printf("We are in hello.c\n");
    printf("PID of hello.c = %d\n", getpid());
    return 0;
}
```

```
user@localhost[~]$ gcc -o example example.c
user@localhost[~]$ gcc -o hello hello.c
user@localhost[~]$ ./example
    PID of example.c = 4733
    We are in Hello.c
    PID of hello.c = 4733
```

execl(...) function

Using the program `printenv` to print the environment variable HOME.

```
#include <unistd.h>
#include <stdio.h>
int main (int argc, char *argv[]) {
    execl("/usr/bin/printenv", "printenv", "HOME", (char *)NULL);
    perror("Execl");
}
```

`path` (pathname)	:"/usr/bin/printenv"
`argv` (list)	:"printenv", "HOME", (char *)NULL
`envp` (-)	:caller's environment

```
user@localhost[~]$ ./example_exec
/home/user
```

execlp(...) function

Using the program `printenv` to print the environment variable HOME.

```
#include <unistd.h>
#include <stdio.h>
int main (int argc, char *argv[]) {
    execlp("printenv", "printenv", "HOME", (char *)NULL);
    perror("Execp");
}
```

`path` (filename)	:“printenv”
`argv` (list)	:“printenv", “HOME", (char *)NULL
`envp` (-)	:caller’s environment

```
user@localhost[~]$ ./example_exec
/home/user
```

execle(...) function

Using the program `printenv` to print the environment variable HOME.

```c
#include <unistd.h>
#include <stdio.h>
int main (int argc, char *argv[]) {
    char *env[] = {"HOME=/home/pippo", (char *)NULL};
    execle("/usr/bin/printenv", "printenv", "HOME", (char *)NULL, env);
    perror("Execle");
}
```

`path` (pathname)	:“/usr/bin/printenv”
`argv` (list)	:“printenv", “HOME”, (char *)NULL
`envp` (array)	:“HOME=/home/pippo”, (char *)NULL

```
user@localhost[~]$ ./example_exec
/home/pippo
```

execv(...) function

Using the program `printenv` to print the environment variable HOME.

```c
#include <unistd.h>
#include <stdio.h>
int main (int argc, char *argv[]) {
    char *arg[] = {"printenv", "HOME", (char *)NULL};
    execv("/usr/bin/printenv", arg);
    perror("Execv");
}
```

`path` (pathname)	:	"/usr/bin/printenv"
`argv` (array)	:	"printenv", "HOME", (char *)NULL
`envp` (-)	:	caller's environment

```
user@localhost[~]$ ./example_exec
/home/user
```

execvp(...) function

Using the program `printenv` to print the environment variable HOME.

```c
#include <unistd.h>
#include <stdio.h>
int main (int argc, char *argv[]) {
    char *arg[] = {"printenv", "HOME", (char *)NULL};
    execvp("printenv", arg);
    perror("Execvp");
}
```

`path` (filename)	:	“printenv”
`argv` (array)	:	“printenv", “HOME”, (char *)NULL
`envp` (-)	:	caller’s environment

```
user@localhost[~]$ ./example_exec
/home/user
```

execve(...) function

Using the program `printenv` to print the environment variable HOME.

```
#include <unistd.h>
#include <stdio.h>
int main (int argc, char *argv[]) {
    char *arg[] = {"printenv", "HOME", (char *)NULL};
    char *env[] = {"HOME=/home/pippo", (char *)NULL};
    execve("/usr/bin/printenv", arg, env);
    perror("Execve");
}
```

path (pathname)	:	“/usr/bin/printenv”
argv (array)	:	“printenv", “HOME”, (char *)NULL
envp (array)	:	“HOME=/home/pippo", (char *)NULL

```
user@localhost[~]$ ./example_exec
/home/pippo
```

Final remarks on the exec library functions

What you should always keep in mind when you use an `exec` function:

- The program input parameter points to an executable;
- List and array are always terminated with a NULL pointer

```
(char *)NULL;
```

- By convention, the first item of argv is the name of the program;
- All exec functions return no result on success.