Unix signals

1. In the code below the **wait** immediately blocks the parent until the child finishes, this means the next get_request() won't be caled until the current request is fully handled, making the server sequential, so there is not point to it. Calling **wait** after the infinite loop will never be reached, so it is pointless.

```
while(1) {
    req = get_request();
    pid = fork();
    if(pid == 0) {
        res = process_request(req);
        send_response(res);
        exit(0);
    }
    wait(0); // all is sequential now :-(
}
```

2. Signals are mechanisms that interrupt a process execution and determine it to run a handler associated with the signal number.
3. Ctrl-C actually sends signal number 2, also known as SIGINT to the process, which causes the default handler to get executed, and stop the process.
4. A process can assign a custom function to be executed when a certian signal is recieved. Let's write a program that refuses to stop when it recives Ctrl-C

```
#include <stdio.h>
#include <signal.h>

void f(int sgn) {
    printf("I refuse to stop!\n");
}

int main(int argc, char** argv) {
    signal(SIGINT, f);
    while(1);
    return 0;
}
```

To kill this process we have to open another terminal and write the bash command

- ps aux | grep ctrl-c.exe
- kill -9 <PID>

5. The **signal** system call does not signal anything, it only registers a function to be called when the signal is received. To send a signal to a process use the system call **kill**.
6. So how do we use signals to solve the concurrent server zombie problem? Whenever a child process stops, the parent receives signal SIGCHLD. We can register a function to SIGCHLD that simply calls wait, and thus the child process, stops being a zombie right away. This is shown in the solution on the left. An even simpler method is to ignore the SIGCHLD signal which causes the system to immediately kill the child process and not turn it into a zombie. This is shown in the solution on the right.

```
#include <stdio.h>
#include <signal.h>

void f(int sgn) {
    wait(0);
}

int main(int argc, char** argv) {
    signal(SIGCHLD, f);
    while(1) {
        req = get_request();
        pid = fork();
        if(pid == 0) {
            res = process_request(req);
            send_response(res);
            exit(0);
        }
    }
    return 0;
}
```

```
#include <stdio.h>
#include <signal.h>

int main(int argc, char** argv) {
    signal(SIGCHLD, SIG_IGN);
    while(1) {
        req = get_request();
        pid = fork();
        if(pid == 0) {
            res = process_request(req);
            send_response(res);
            exit(0);
        }
    }
    return 0;
}
```

Examples

1. Implement a program that upon receiving SIGINT (Ctrl-C) asks the user if he/she is sure the program should stop, and if the answer is yes, stops, otherwise it continues.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>
#include <string.h>

void f(int sgn) {
    char s[32];
    printf("Are you sure you want me to stop [y/N]? ");
    scanf("%s", s);
    if(strcmp(s, "y") == 0) {
        exit(0);
    }
}

int main(int argc, char** argv) {
    signal(SIGINT, f);
    while(1);
    return 0;
}
```

2. Write a program that creates three child processes that just loop forever. When the parent process gets SIGUSR1 it sends SIGUSR2 to the child processes. When the parent process gets SIGUSR2, it sends SIGKILL to the child processes. The child processes should report the signals received. Note in the solution below that the child process does not override the SIGKILL handler, as it will not have any effect.

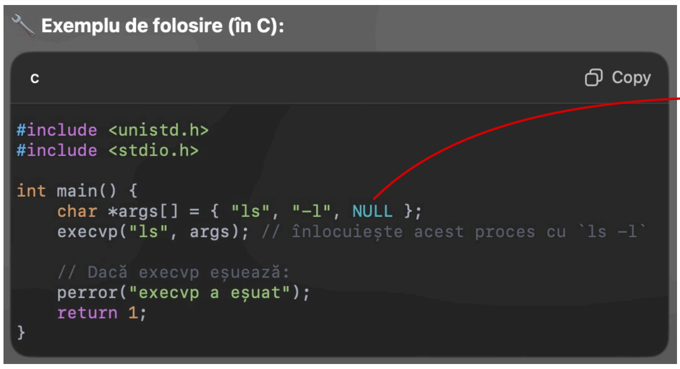
Running other programs using the exec system calls

1. To run another program from an existing process, the UNIX system offers the **exec** system calls. There are six of them, but we only present four of them, and use probably just one or two. The table below shows the four variants differing by whether the program arguments are an array or specified directly as function arguments, and wether the program is given with an absolute path or whether it should be searched for in the **PATH**.

		Search PATH for the program	
		Yes	**No**

Arguments passed as	**Array**	`char* a[] = {"grep",`
		`"-E",`
		`"/an1/gr911/",`
		`"/etc/passwd",`
		`NULL}`
		`execvp("grep", a);`
			`char* a[] = {"/bin/grep",`
			`"-E",`
			`"/an1/gr911/",`
			`"/etc/passwd",`
			`NULL}`
			`execv("/bin/grep", a);`

	List	`execlp("grep",`
		`"grep",`
		`"-E",`
		`"/an1/gr911/",`
		`"/etc/passwd",`
		`NULL);`
			`execl("/bin/grep",`
			`"/bin/grep",`
			`"-E",`
			`"/an1/gr911/",`
			`"/etc/passwd",`
			`NULL);`


🔧 **Exemplu de folosire (în C):**

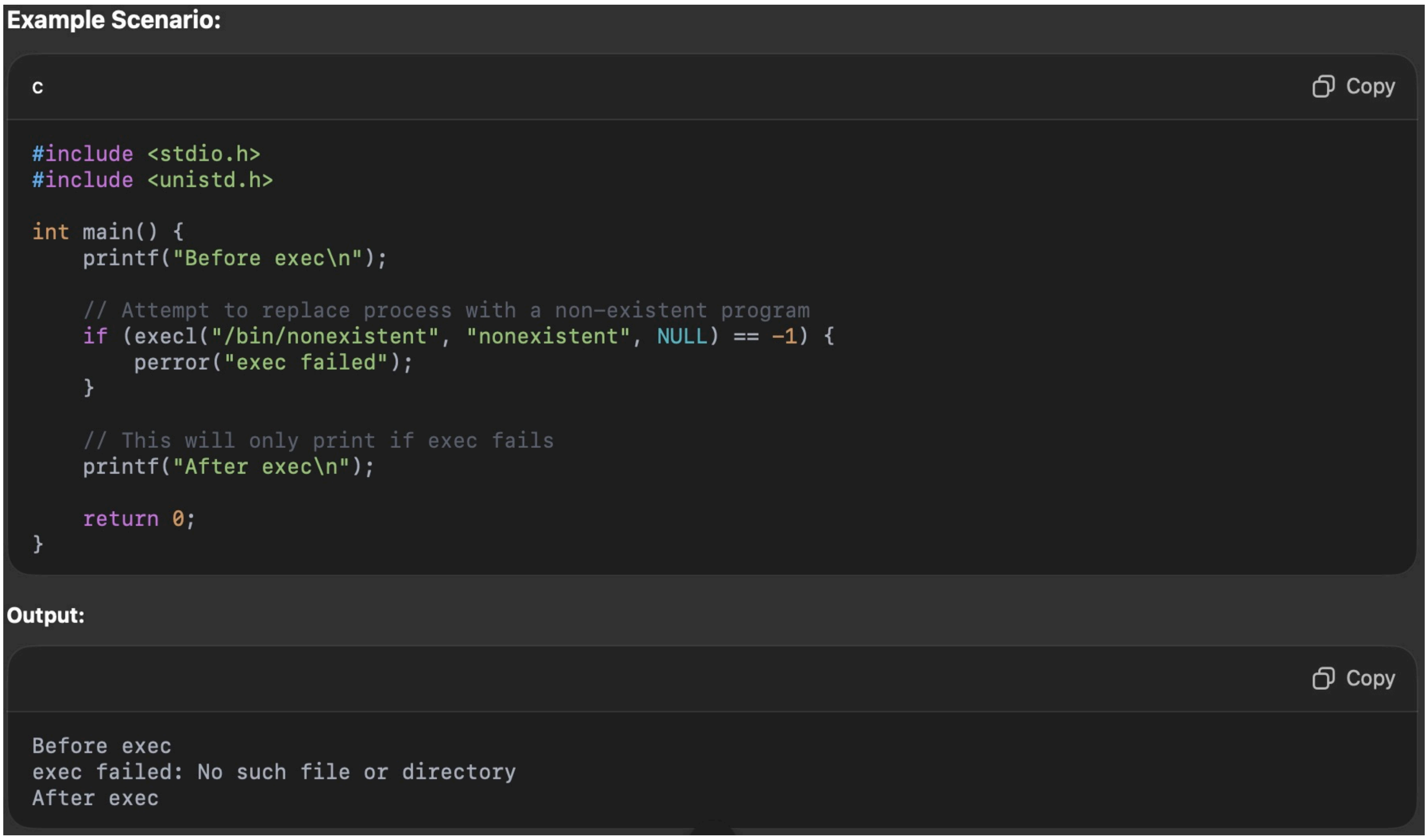
```c
#include <unistd.h>
#include <stdio.h>

int main() {
    char *args[] = { "ls", "-l", NULL };
    execvp("ls", args); // înlocuiește acest proces cu `ls -l`

    // Dacă execvp eșuează:
    perror("execvp a eșuat");
    return 1;
}
```

Nu exista un numar fix de argumente, astfel ca NULL este utilizat pentru sfarsitul listei.

2. What exactly is the **PATH**? A UNIX Shell variable that contains paths where the Shell should look for the program you run, unless you specify an absolute or relative path like ./myprog
3. Unexpected behaviour : exec replace the current process image with a new program image. If the exec call fails the calling process continues to execute with the current process image.



Example Scenario:

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

int main() {
    printf("Before exec\n");

    // Attempt to replace process with a non-existent program
    if (execl("/bin/nonexistent", "nonexistent", NULL) == -1) {
        perror("exec failed");
    }

    // This will only print if exec fails
    printf("After exec\n");

    return 0;
}
```

Output:

```
Before exec
exec failed: No such file or directory
After exec
```

4. As the first argument is always the command name, we need to pass that argument explicitly.
5. The last NULL argument is required in order to mark end of the arguments.
6. If exec system calls overwrite the current process, how can i run another program and still keep my process? Simple, just create a child process and run exec in it.

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

int main(int argc, char** argv) {
    int pid = fork();
    if(pid == 0) {
        execlp("grep", "grep", "-E", "/an1/gr211/", "/etc/passwd", NULL);
        exit(1);
    }
    // do some other work in the parent
    wait(0);
    return 0;
}
```

Examples

Write a program that gets programs as command line arguments, and execute them using fork and exec. In case exec fails or the program it executes exits with an error code, report the relevant details standard error. Notice in the solution below, the usage of **strerror**, **errno**, **wait** and **WEXITSTATUS** in the solution above.

```c
int main(int argc, char** argv) {
  int i, status;
  for(i=1; i<argc; i++) {
    if(fork() == 0) {
      if(execlp(argv[i], argv[i], NULL) == -1) {
        fprintf(stderr, "Failed to execute program \"%s\": %s\n", argv[i], strerror(errno));
        exit(0);
      }
    }
    wait(&status);
    if(WEXITSTATUS(status) != 0) {
      fprintf(stderr, "Program \"%s\" failed with exit code %d\n", argv[i], WEXITSTATUS(status));
    }
  }
  return 0;
}
```