

```
#include <stdlib.h>
#include <string.h>
#include <ctype.h>

#define MAXPAROLA 30
#define MAXRIGA 80

int main(int argc, char *argv[])
{
    int freq[MAXPAROLA]; /* vettore di contatori
delle frequenze delle lunghezze delle parole */
    char riga[MAXRIGA];
    int i, inizio, lunghezza;
    FILE * f;

    for(i=0; i<MAXPAROLA; i++)
        freq[i]=0;

    if(argc != 2)
    {
        fprintf(stderr, "ERRORE, serve un parametro con il nome del file\n");
        exit(1);
    }
    f = fopen(argv[1], "r");
    if(f==NULL)
    {
        fprintf(stderr, "ERRORE, impossibile aprire il file %s\n", argv[1]);
        exit(1);
    }

    while( fgets( riga, MAXRIGA, f ) != NULL )
```

Processes

Process creation

Stefano Quer and Pietro Laface

Dipartimento di Automatica e Informatica

Politecnico di Torino

fork and exec system calls

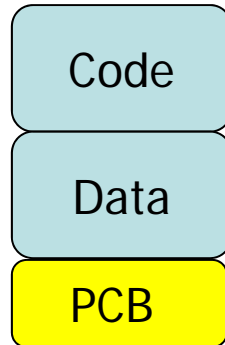
- ❖ System call **fork** creates a new process duplicating the calling process, then
 - Parent and child execute **different code sections**
Example: a network server duplicates itself at each client request, and the child serves the request while the parent waits for a new client request
 - Parent and child execute **different code**
 - Example: a command interpreter (shell)
 - Uses the **exec** system call

exec system call

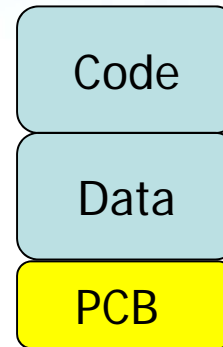
- ❖ System call **exec substitutes** the process code with the executable code of another program
- ❖ The new program begins its execution as usual (from main)
- ❖ In particular **exec**
 - Does not create a new process
 - Substitutes the calling process image with the image of another program.
 - The process PID does not change
 - fork → duplicates an **existent process**
 - exec → executes a **new program**

Address space

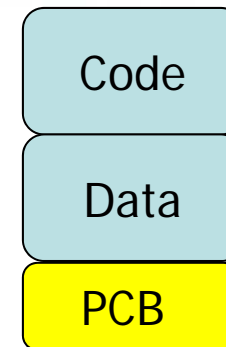
Process



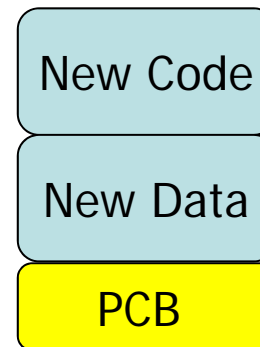
Parent



Child



New Process



exec system call

❖ 6 versions of exec system call

- execl, execlp, execl
- execv, execvp, execve

Type	Action
l (list)	Arguments are a list of strings
v (vector)	Arguments is a vector of strings <code>char **arguments</code>
p (path)	The executable filename is looked for in the directories listed in the environment variable <code>PATH</code>
e (environment)	The last argument is an environment vector <code>envp[]</code> which defines a set of new associations <code>strings name=value</code>

exec system call

```
#include <unistd.h>
int execl (char *path, char *arg0, ..., (char *)0);
int execlp (char *name, char *arg0, ..., (char *)0);
int execl (char *path, const char *arg0, ..., char *envp[]);
int execv (char *path, char *argv[]);
int execvp (char *name, char *arg[]);
int execve (char *path, char *arg[], char *envp[]);
```

❖ The return value is -1 in case of error

exec system call

❖ Arguments

- Pathname of the executable file
 - In the "p" versions the complete path is not necessary. The file must be in one of the directories listed in the environment variable PATH (echo \$PATH)
- Its argument list
 - The first argument is the **name** of the process (its argv[0])
 - The other argument of the list are the argument for the executable (argv[1], argv[2], etc).
- Possibly the environment vector

Examples

OK

whereis cp: /bin/cp

User defined name

```
execl("/bin/cp", "mycp", "./file1", "./file2", NULL);
```

OK

Alternative
termination

```
execl("/bin/cp", "cp", "./file1", "./file2", (char*)0);
```

NO

Path is missing

```
execl("cp", "File_copy", "./file1", "./file2", (char*)0);
```

OK

Default path (\$PATH)

```
execvp("cp", "cp", "./file1", "./file2", (char*)0);
```


exec system call

❖ `execv[p]`

- Uses a single argument: a vector of strings
 - The vector must be properly initialized

```
char *cmd[] = {  
    "ls",  
    "-laR",  
    ".",  
    (char *) 0  
};  
...  
execv ("/bin/ls", cmd);
```

Last argument must be a
NULL pointer

System call exec ()

❖ exec[lv]e

- Can provide to the executable a set of environment variables
 - Vector of strings
 - Without “e” the environment of the new process is inherited from the calling process

```
char *env[] = {  
    "USER=unknown",  
    "PATH=/tmp",  
    NULL  
};  
...  
execle (path, argv, ..., argn, 0, env);  
...  
execve (path, argv, env);
```

Considerations

- ❖ The `execed` process keeps all open file descriptor (including `stdin`, `stdout`, `stderr`)
 - This allow the process to inherit possible redirections previously set (e.g., by shell)
- ❖ Many kernel implement only system call **`execve`**
 - The other versions are macros that use this system call

Exercise

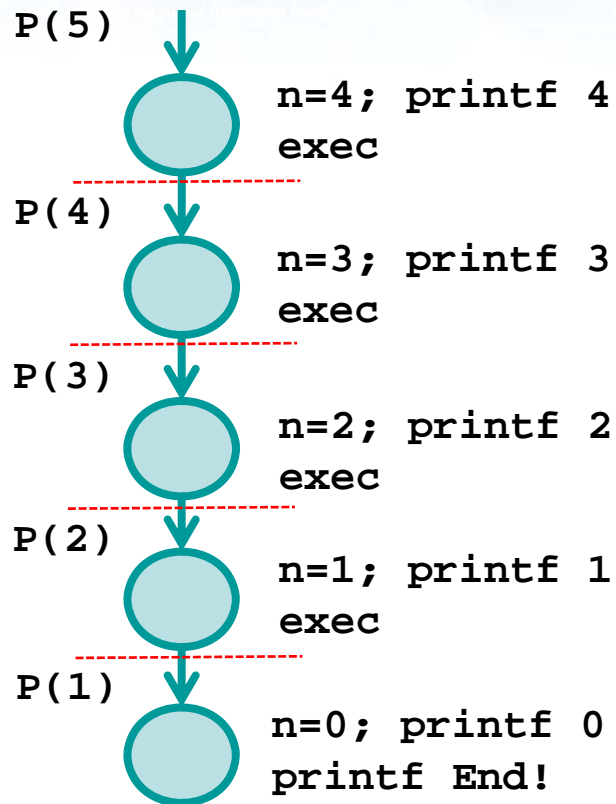
- ❖ Draw the process generation tree of the following C code segment , executed passing as its argument on the command line string "5"
- ❖ What does it display?
- ❖ Why?

Exercise

Run with n=5

```
#include <stdio.h>
...
#include <unistd.h>
int main (int argc, char ** argv) {
    char str[10];
    int n;
    n = atoi(argv[1]) - 1;
    printf ("%d\n", n);
    if (n>0) {
        sprintf (str, "%d", n);
        execl (argv[0], argv[0], str, NULL);
    }
    printf ("End!\n");
    return 1;
}
```

Solution



```
int main (int argc, char ** argv) {  
    char str[10];  
    int n;  
    n = atoi(argv[1]) - 1;  
    printf ("%d\n", n);  
    if (n>0) {  
        sprintf (str, "%d", n);  
        execl (argv[0], argv[0], str, NULL);  
    }  
    printf ("End!\n");  
    return 1;  
}
```

Output

```
4  
3  
2  
1  
0  
End!
```

Exercise

- ❖ Draw the process generation tree of the following C code segment
- ❖ What does it display?
- ❖ Why?

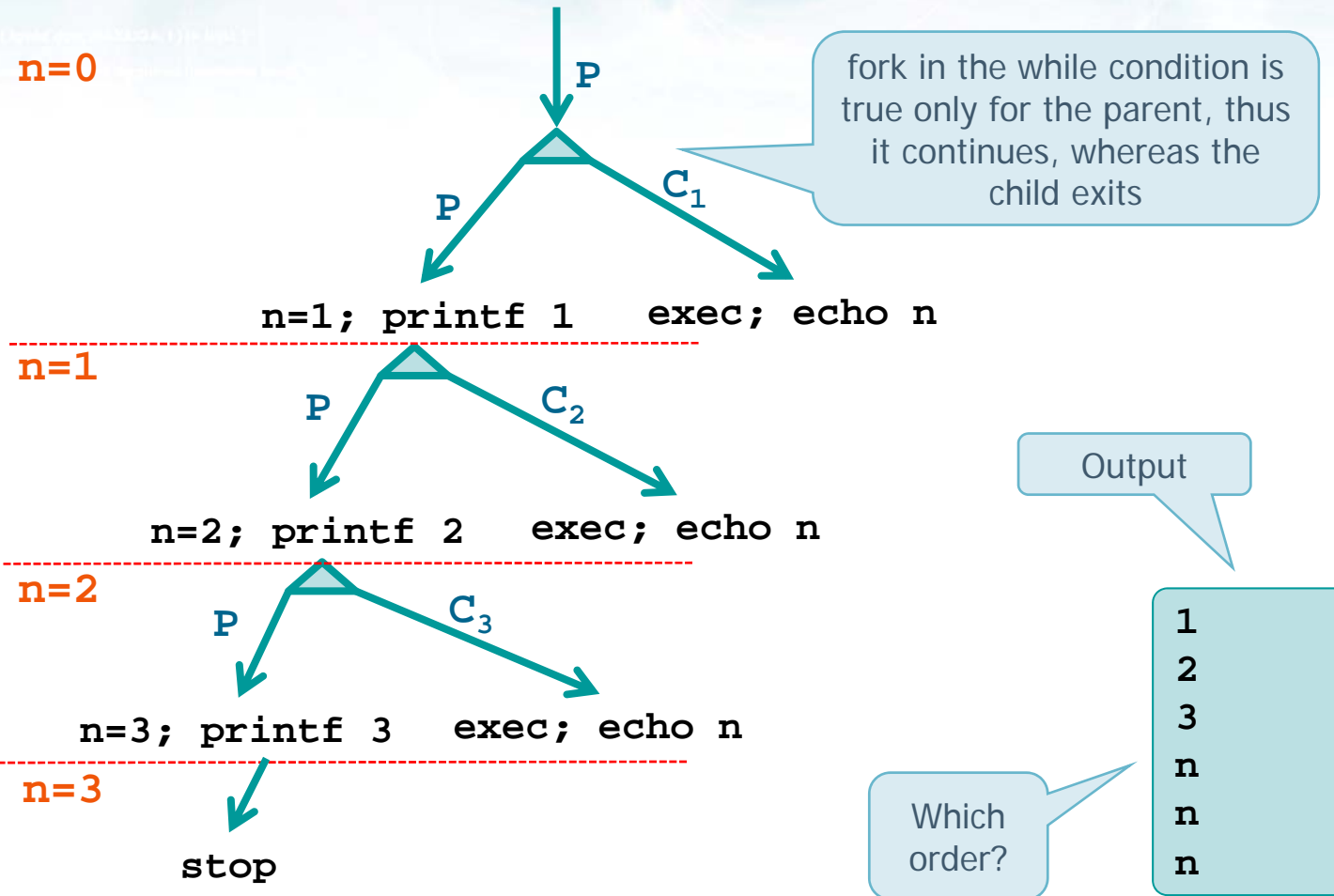
Exercise

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

int main(){
    int n;
    n=0;
    while (n<3 && fork()){
        if (!fork())
            execlp ("echo", "n++", "n", NULL);
        n++;
        printf ("%d\n", n);
    }
    return (1);
}
```

shell command
echo

Solution



Example

Program `./pgrm` exec itself if it receives as argument 1 or 2

```
...
n = atoi (argv[1]);
switch (n) {
    case 1:
        printf("#1:PID=%d;PPID=%d\n", getpid(), getppid());
        sleep (n*10);
        execlp ("./pgrm", "./Pgrm", "2", (char *) 0);
        break;
    case 2:
        printf("#2:PID=%d;PPID=%d\n", getpid(), getppid());
        sleep (n*10);
        execlp ("./pgrm", "myPgrm", "3", (char *) 0);
        break;
    default:
        printf("#3:PID=%d;PPID=%d\n", getpid(), getppid());
        sleep (n*10);
        break;
}
return (1);
```

Same pathname but
argv[0] (its name) changes

Example

Shell commands (in blue)

Run with n=1

PID does not change

```
> ./pgrm 1 &
[2] 2471
#1: PID=2471; PPID=2045
> ps -aux | grep 2471
user 2471 0.0 0.0 4192 352 pts/2 S 19:29 0:00 ./pgrm 1
#2: PID=2471; PPID=2045
> ps -aux | grep 2471
user 2471 0.0 0.0 4192 356 pts/2 S 19:29 0:00 ./Pgrm 2
#3: PID=2471; PPID=2045
> ps -aux | grep 2471
User 2471 0.0 0.0 4192 356 pts/2 S 19:29 0:00 myPgrm 3
[2]+  Exit 1  ./pgrm 1
```

The name changes

UNIX shell skeleton

❖ Command run in foreground

➤ <command>

```
while (TRUE) {
    write_prompt;
    read_command (command, parameters);
    if (fork() == 0)
        /* Child: Execute command */
        execve (command, parameters);
    else
        /* Parent: Wait child */
        wait (&status);
}
```

UNIX shell skeleton

❖ Command run in background

➤ `<command> &`

```
while (TRUE) {
    write_prompt;
    read_command (command, parameters);
    if (fork() == 0)
        /* Child: Execute command */
        execve (command, parameters);
    /* else */
        /* Parent: does not wait */
        /* wait (&status); */
}
```

Command execution

- ❖ It can be useful to execute a **shell command** from a process
 - For example for appending a date to a filename or to a file
- ❖ System call **system** solves this problem
 - It is defined by the standard ISO C and POSIX

system() system call

```
#include <stdlib.h>
```

```
int system (const char *string);
```

❖ System call system

- Forks a shell, which execute the string command, while the parent process waits the termination of the shell command
- Returns
 - -1 or 127 on error
 - The exit value of the shell that executed the command (with the format of **waitpid**)

Example

```
...  
system ("date");  
...
```

```
...  
system ("ls -laR");  
...
```

```
char str[L];  
...  
strcpy (str, "ls -la");  
system (str);  
...
```


system() implementation

```
int system (const char *cmd) {
    pid_t pid;
    int status;
    if (cmd == NULL)
        return(1);
    if ( (pid = fork()) < 0 ) {
        status = -1;
    } else if (pid == 0) {
        execl("/bin/sh", "sh", "-c", cmd, (char *) 0);
        _exit(127);
    } else {
        while (waitpid (pid, &status, 0) < 0)
            if (errno != EINTR) {
                status = -1;
                break;
            }
    }
    return(status);
}
```

Exercise

- ❖ Draw the process generation tree of the following C program, executed passing as its argument on the command line string "4"
- ❖ What does it display?
- ❖ Why?

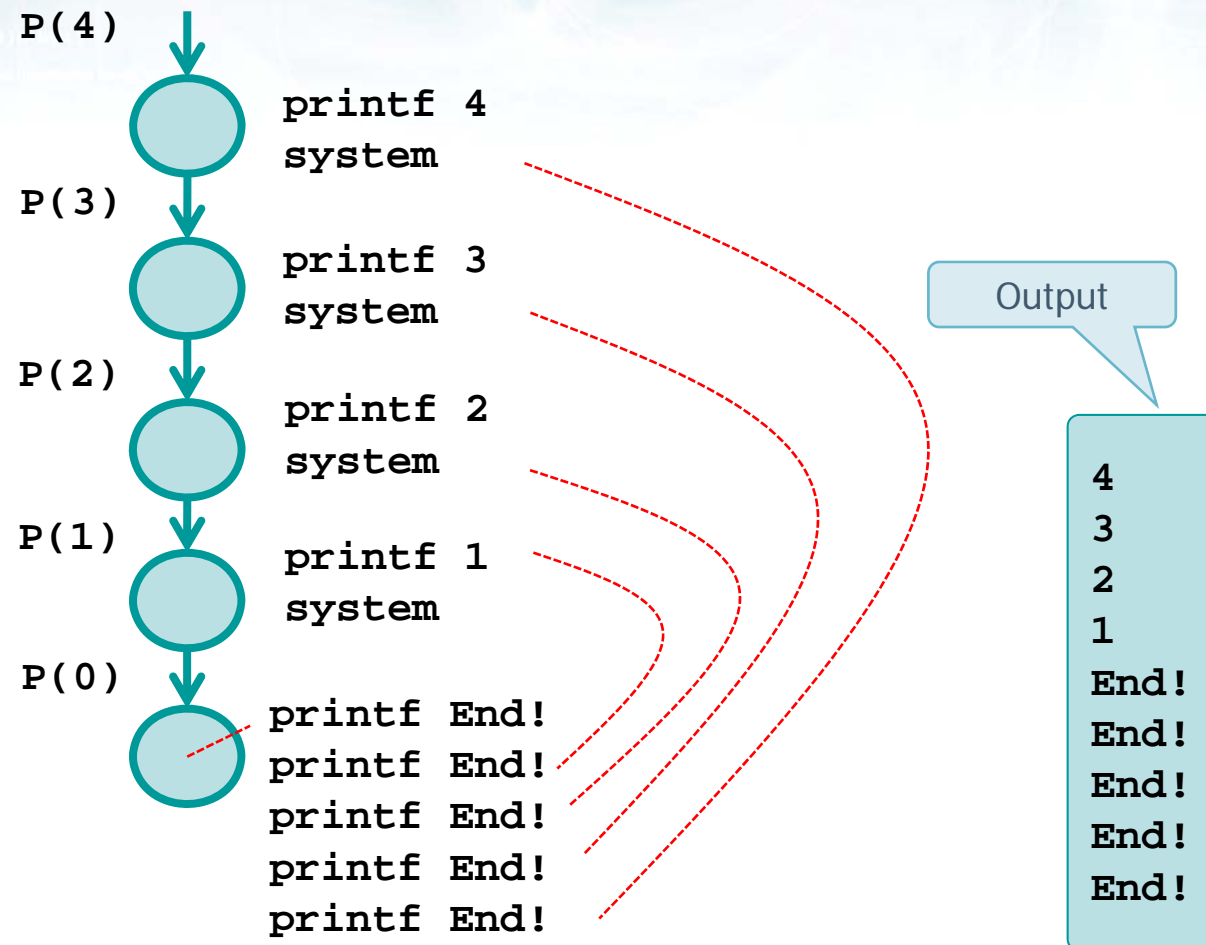
Esercizio

Run with n=4

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

int main(int argc, char ** argv){
    int n;
    char str[10];
    n = atoi (argv[1]);
    if (n>0) {
        printf ("%d\n", n);
        sprintf (str, "%s %d", argv[0], n-1);
        system (str);
    }
    printf("End!\n");
    return (1);
}
```

Solution



Exercise

- ❖ Draw the process generation tree of the following C code segment
- ❖ What does it display?
- ❖ Why?

Exercise

```
#include ...
int main () {
    char str[100];
    int i;
    for (i=0; i<2; i++){
        if (fork()!=0) {
            sprintf (str, "echo system with i=%d", i);
            system (str);
        } else {
            if (fork()==0) {
                sprintf (str, "exec with i=%d", i);
                execlp ("echo", "myPgrm", str, NULL);
            }
        }
    }
    return (0);
}
```

