

Fundamentos de los Sistemas Operativos (FSO)

Departamento de Informática de Sistemas y Computadoras (DISCA)

Universitat Politècnica de València

Part 2: Process Management

Seminar 3

UNIX system calls for processes

f SO

DISCA



UNIVERSITAT
POLITÈCNICA
DE VALÈNCIA

- Goals:
 - To know the **services** provided by UNIX **to create processes**
 - To analyse **C language examples** using process related system calls
 - To know the **signal concept**
- Bibliografía:
 - “**UNIX System Programming**”, Kay A. Robbins, Steven Robbins. Prentice Hall. ISBN 0-13-042411-0

Process system call for:

- Identification
- Creation
- Waiting
- Ending
- Signals

UNIX process related system calls

fSO

	Processes
fork	Create a child process
exit	End process
wait	Wait for a process ending
exec	Execute a program
getpid	Get process ID attribute
getppid	Get parent ID attribute

	Signals
kill	Send signals
alarm	Generate an alarm (clock signal)
sigemptyset	Init a mask with no signals set
sigfillset	Init a mask with signals set
sigaddset	Append a signal to a signal set
sigdelset	Delete a signals in a signal set
sigismember	Check if a signals belongs to a signal set
sigprocmask	Check/Modify/Set a signal mask
sigaction	Capture/Manage a signal
sigsuspend	Wait signals capture

Process system call for:

- **Identification**
- Creation
- Waiting
- Ending
- Signals

- Every process must have an ID
- The creator process is the *parent* while the created process is the *child*. To know them we use:
 - Process ID (PID) `pid_t getpid(void);`
 - Parent process ID (PPID) `pid_t getppid(void);`

```
/** ej1_getpid.c ****/
#include <stdio.h>
#include <unistd.h>

int main(void)
{
    printf("\nProcess ID: %ld\n", (long)getpid());
    printf("Parent process ID: %ld\n", (long)getppid());
    while(1);
    return 0;
}
```

```
$ gcc -o ej1 ej1_getpid.c
$ ./ej1 &
[1] 2959
$
Process ID: 2959
Parent process ID: 1060
$ ps -l
```

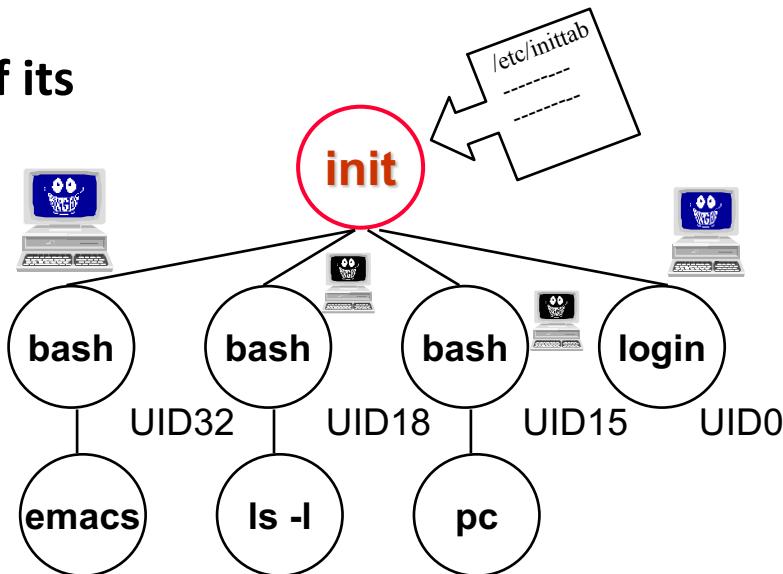
UID	PID	PPID	F	CPU	PRI	NI	SZ	RSS	WCHAN	S	ADDR	TTY	TIME	CMD
501	1060	1059	4006	0	31	0	2435548	1088	-	S	fffffff80136e3d50	ttys000	0:00.06	bash
501	2959	1060	4006	0	31	0	2434832	340	-	R	fffffff80140d8300	ttys000	0:04.65	./a.out

- Identification
- **Creation**
- Waiting
- Ending
- Signals

- Unix uses a copying mechanism for process creation

- The **child process is an exact replica of its parent process**
- The **child process inherits most of the attributes from its parent:**
 - Memory image
 - UID, GID
 - Current directory
 - Opened file descriptors
- Unix assigns an **identifier to every process called PID** at the time of its creation
- Every process knows the **identifier of its parent process, PPID**
- The child execution is concurrent with the parent and independent
- Processes are organized into a **hierarchical tree**

```
pid_t fork(void);
```



- **fork:** process creation

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

pid_t fork(void)
```

- **Description**

- It creates a child process which is a “clon” from the parent: the child process inherits the most part of the parent attributes.
- NON Inherited attributes: PID, PPID, pendant signals, file locks and time/accounting.

- **Returning value**

- 0 to the child
- Child PID to the parent
- -1 to the parent if an error happens

- **Errors**

- Not enough resources to create the process

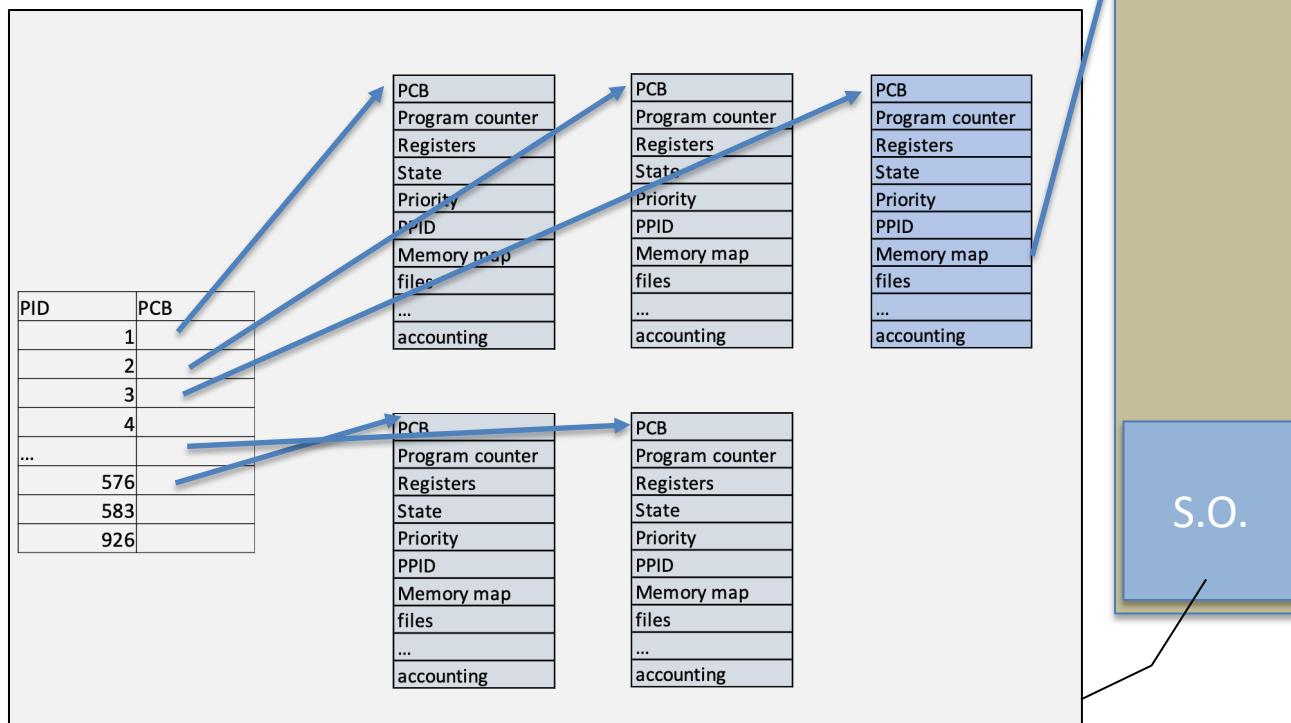
Creation

- **fork:** process creation

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

pid_t fork(void)
```

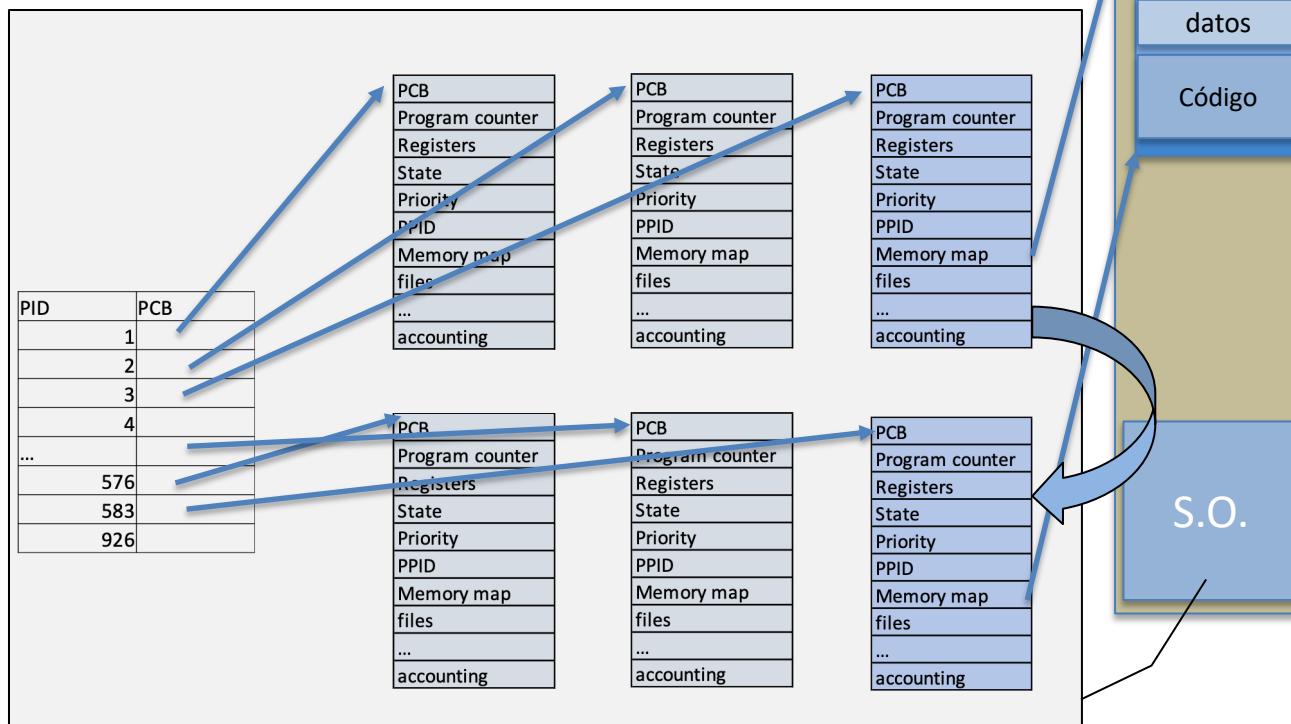
`res = fork(void)`



- **fork:** process creation

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

pid_t fork(void)
```



res = fork(void)
print res
res=583

res = fork(void)
print res
res=0

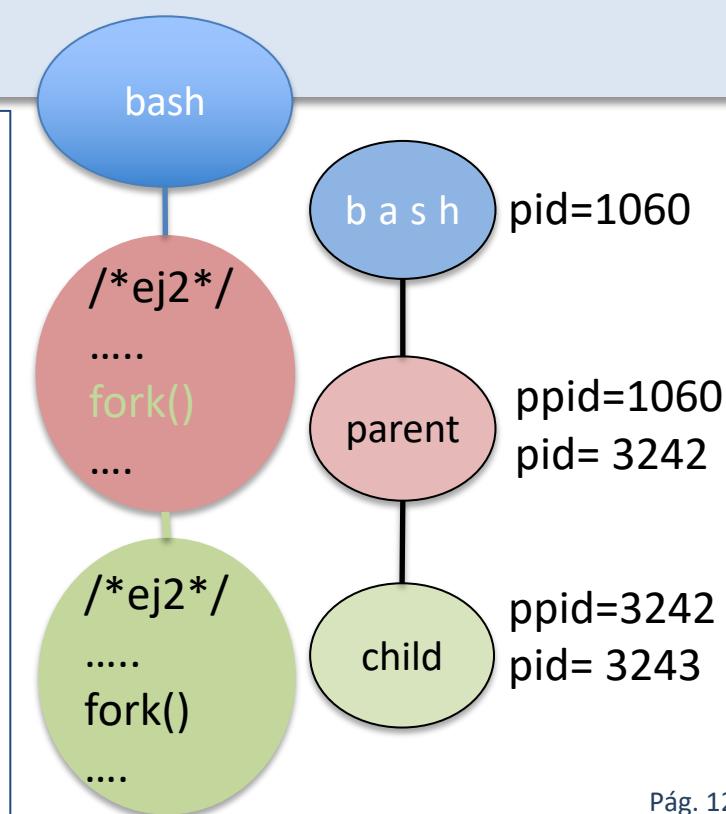
Creation

Parent and child resume execution with the instruction following `fork()`

```
/**ej2_fork.c */
#include <stdio.h>
#include <sys/types.h>

int main(void)
{ printf("Process %ld creates another process\n", (long)getpid());
  res = fork();
  printf("Process %ld with parent %ld\n", (long)getpid(), (long)getppid());
  sleep(5);
  return 0;
}
```

```
$ ps
  PID TTY          TIME CMD
 1060 ttys000      0:00.07 -bash
$ gcc -o ej2 ej2_fork.c
$ ./ej2
Process 3242 creates another process
Process 3242 with parent 1060
Process 3243 with parent 3242
$
```



- Parent and child execute different code

```
// ej3_fork.c
#include <sys/types.h>
#include <unistd.h>

int main() {
    pid_t val;
    int var = 0;
    printf("PID before fork(): %d\n", (long) getpid());
    val = fork();
    if (val > 0) {
        printf("Parent PID: %d\n", (long) getpid());
        var++;
    } else {
        printf("Child PID: %d\n", (long) getpid());
    }
    printf("Process [%d]-> var=%d\n", (long) getpid(), var);
    return 0;
}
```

How many processes print this message?

What value(s) of “var” are shown?

- Example

```
// ej4_fork.c

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

int main(void)
{
    pid_t pid=fork();

    switch (pid) {
    case -1:
        printf("Could not create child process \n");
        break;
    case 0:
        printf("I am the child with PID %ld and my parent is %ld \n",
               (long)getpid(), (long)getppid());
        break;
    default:
        printf("I am the parent with PID %ld and my child is %ld \n",
               (long)getpid(), pid);
    }
    sleep(5);
    return 0;
}
```

```
$ gcc -o ej3 ej3_fork.c
$ ./ej3
```

I am the parent with PID 3702 and my child is 3704
I am the child with PID 3704 and my parent is 3702

- Creating processes in chain

```
// ej5_proc_chain.c
#include <stdio.h>
#include <sys/types.h>

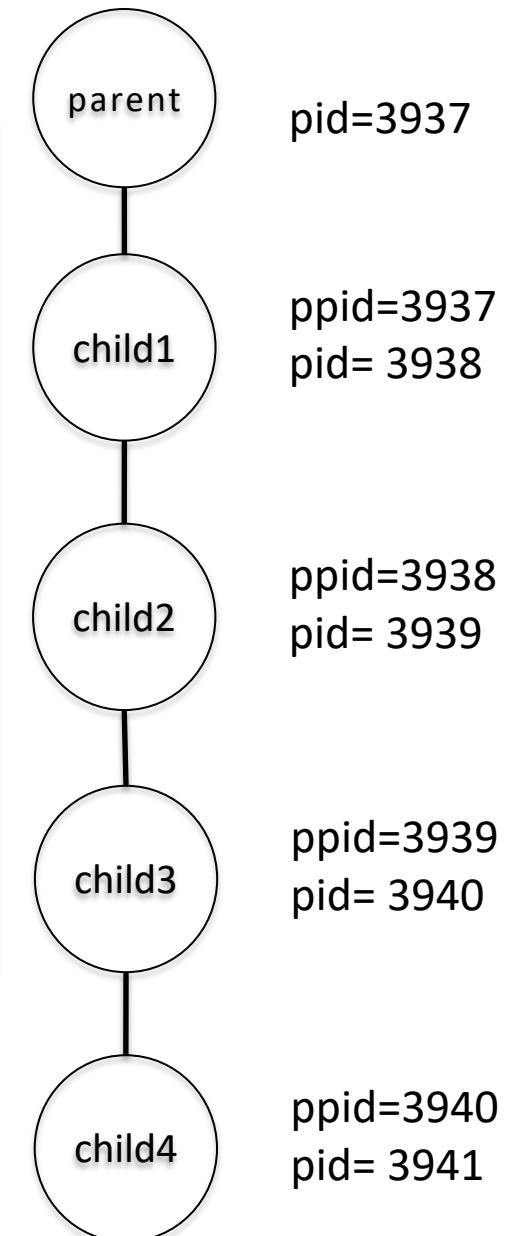
#define NPROCESSES 4

int main(void)
{
    pid_t pid;
    int i;

    for (i=0; i<NPROCESSES; i++) {
        pid=fork();
        if (pid!=0)
            break;
        printf("I am the child with PID %ld
               my parent is %ld\n",
               (long)getpid(), (long)getppid());
    }
    sleep(5);
    return 0;
}
```

Variants

if (pid > 0)
if (pid==0)
if (pid<0)



```
$ ./a.out
I am the child with PID 3938 my parent is 3937
I am the child with PID 3939 my parent is 3938
I am the child with PID 3940 my parent is 3939
I am the child with PID 3941 my parent is 3940
```

- **exec()**

- The call to fork () creates a child who is a copy of the calling process
- What if we want to run a different program? Use **exec()** call
- There are different versions **exec** depending on the parameters specified:

```
#include <unistd.h>
int execl(const char *path, const char *arg0, ...,
           const char *argn, char * /*NULL*/);
int execlp(const char *file, const char *arg, ... ,
           const char *argn, char * /*NULL*/);
int execle(const char *path, const char *arg, ... ,
           const char *argn, char * /*NULL*/, char * const envp[]);

int execv(const char *path, char *const argv[] );
int execvp(const char *file, char *const argv[]);
int execve(const char *path, char *const argv[], char *const envp[]);
```

- Variant **I**: arguments are provided separately
- Variant **v**: arguments are provided through a pointer to a vector
- Variant **p**: the location of **file** is searched on the PATH
- Variant **e**: the environment is provided to the child through **envp**, it is not inherited from the parent

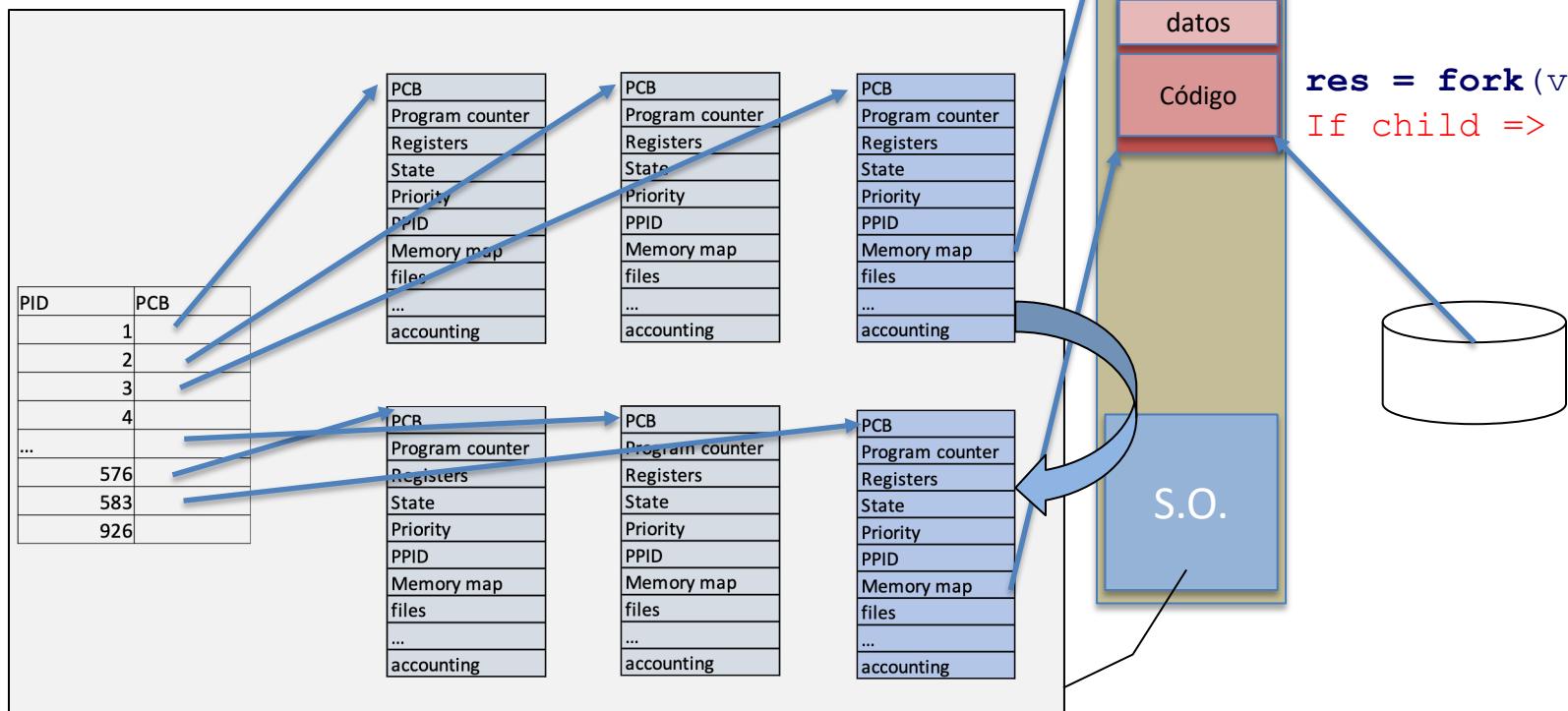
- **exec () features**

- It changes the process memory image by the one defined in the new executable file.
- **The executable file is specified by its name or its absolute path.**
- Some process attributes remain:
 - Signal handlers, except the ones captured and treated by the default action.
 - PID and PPID
 - Time accounting
 - File descriptors
 - Working directory, root directory and file creation mode mask
- If the SETUID bit of the executable file is set then EXEC sets as efective UID the UID of the executable file owner
 - The same happens with the SETGID bit
- Errors
 - Executable file doesn't exist or it is not recognized as executable
 - Permissions
 - Incorrect arguments
 - Not enough memory or resources
- Returning value
 - If EXEC returns to the calling program it is because an error happened, the returning value is -1

- **fork:** process creation

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

pid_t fork(void)
```



- Exec() example: the child process lists the current directory

```
// ej6_exec.c
#include <stdio.h>
#include <sys/types.h>

int main(void)
{
    int status;
    pid_t pid=fork();

    char* arguments [] = { "ls", "-l", 0 };

    switch (pid) {
    case -1:
        printf("The child process could not be created \n");
        break;
    case 0:
        printf("I am the child with PID %ld, the current directory content is: \n",
               (long)getpid());
        if (execvp("ls",arguments)==-1){
            printf("Error in exec\n");
            exit(0);
        }
        break;
    default:
        printf("I am the parent process with PID %ld and my child is %d.\n",
               (long)getpid(), pid)
    }
    return 0;
}
```

Another way:

```
execl("/bin/ls", "ls","-l",NULL)
```

- Identification
- Creation
- **Waiting**
- Ending
- Signals

- The parent has to wait for its child to finish:

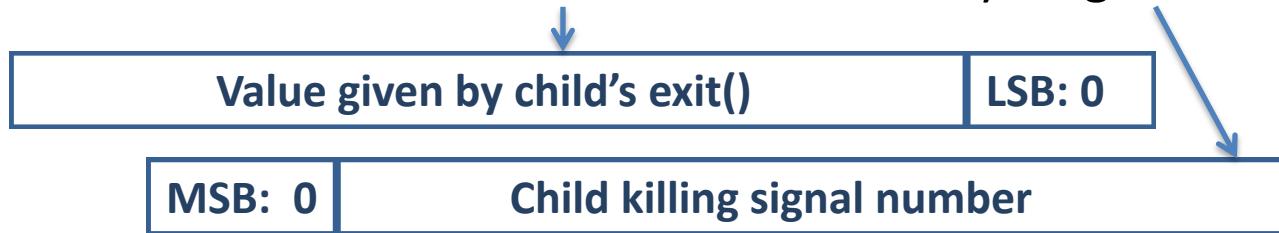
```
#include <sys/types.h>
#include <sys/wait.h>
pid_t wait(int *status);
pid_t waitpid(pid_t pid, int *status, int options);
```

- `wait` and `waitpid` stop the calling process execution until a child ends, or until the calling process receives a signal

- `wait()` call

```
pid_t wait(int *status);
```

- It suspends the execution of the calling process until any of its children finishes
 - If there is any zombie child then `wait` returns immediately
- **status:** is not a NULL pointer, it contains information about child termination:
 - Child finished with `exit` or child finished by a `signal`



- **Returning value:**
 - -1 if error or no children
 - Otherwise the finished child PID

Waiting

- **Wait() example:**

```
// ej7_wait.c
#include <stdio.h>
#include <sys/types.h>
#include <errno.h>

int main(void)
{
    int status;
    pid_t pid=fork();

    switch (pid) {
    case -1:
        printf(" The child process could not be created \n");
        break;
    case 0:
        printf("I am the child process with PID %ld and my parent is %ld\n",
               (long)getpid(), (long)getppid());
        sleep(20);
        printf("I have finished \n");
        break;
    default:
        printf("I am the parent process with PID %ld and my child is %d.
               Waiting ...", (long)getpid(), pid);
        if (wait(&status) != -1)
            printf("My child has ended ok \n");
    }
    return 0;
}
```

- `waitpid()` call

```
pid_t waitpid(pid_t pid, int *status, int options);
```

- It waits for a particular child
- Parameters:
 - **pid**: PID of the child to wait for. If its value is -1 it waits until the first child ends (like `wait`)
 - **status**: like `wait()`
 - **options**:
 - Value 0: blocking (common value)
 - WNOHANG: non blocking
 - **Returning value**:
 - 0 : no child ended (non-blocking version),
 - -1 : error
 - > 0 : pid of the returning child

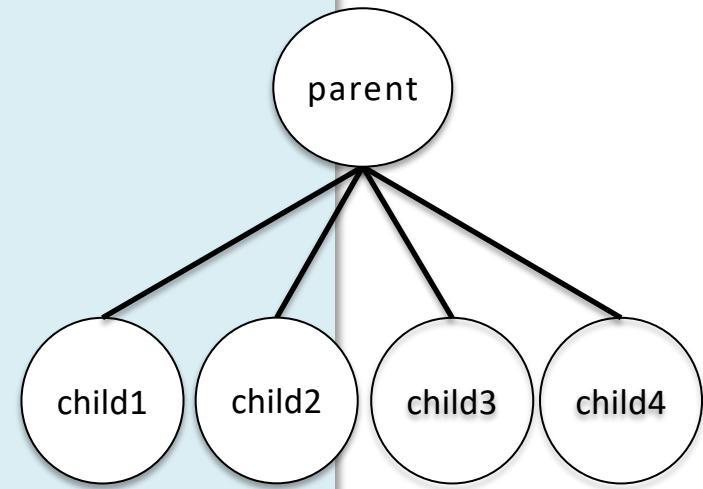
- `waitpid()` example: fan process creation

```
// ej8_waitpid.c
#include <stdio.h>
#include <sys/types.h>

#define NPROCESSES 4

int main(void)
{
    pid_t pid[NPROCESSES];
    int i, status;

    for (i=0; i<NPROCESSES; i++) {
        pid[i]=fork();
        if (pid[i]==0) {
            printf("I am the child %ld my parent is %ld\n",
                   (long)getpid(), (long)getppid());
            sleep(10);
            exit(0);
        }
    }
    // Now wait for the third child
    if (waitpid(pid[2],&status,0)==pid[2])
        printf(" My third child has finished \n");
    return 0;
}
```



```
$gcc -o ej8 ej8_waitpid.c
```

- Identification
- Creation
- Waiting
- **Ending**
- Signals

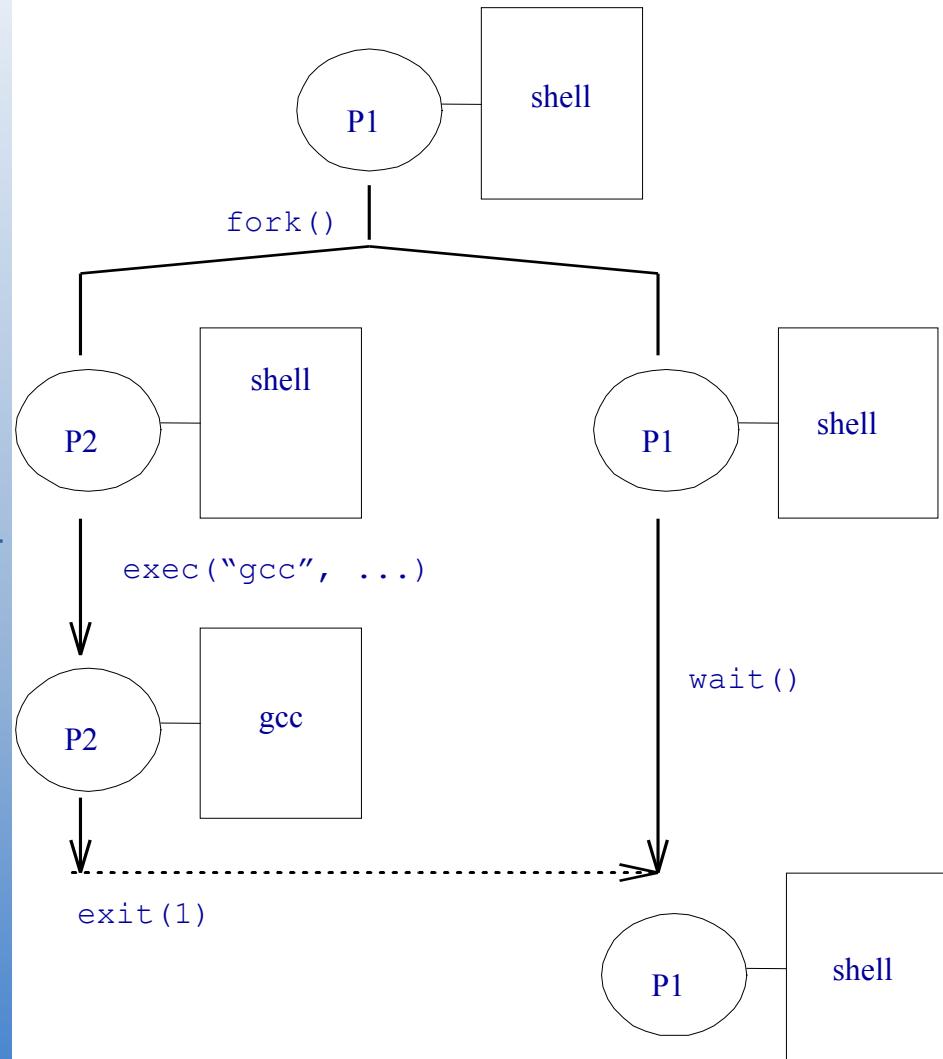
- A process ends completely when:
 - The process itself ends (normally or abnormally) **AND**
 - Its parent process does a wait call
- The normal termination is done by calling to **exit**

```
void exit (int status)
```

- The status parameter is used to communicate to the parent how the child process ends
- By convention, this value is typically 0 if the process ends normally and any other value means abnormal termination
- The parent process can obtain this value through the system call **wait**

- **Abnormal ending:**
 - The process is ended by the operating system because of an error condition (memory boundary violation, arithmetic errors) or by some other process initiative
 - Signals
- **Zombie:** If the process ends before his parent calls `wait()`:
- **Orphan:** If the parent process terminates before the child:
 - An orphan process is adopted by process root `init`

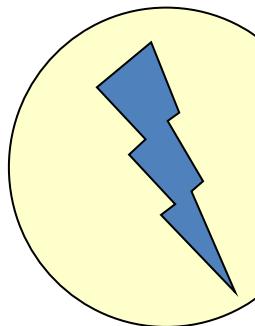
- Simplified structure of UNIX shell



```
while(TRUE) {  
    print_prompt();  
    read_command(command, parameters);  
  
    p=fork(); /* creates a child */  
  
    if (p != 0) { /* parent code */  
        waitpid(-1, &status, 0); /* waiting for  
        the child */  
  
    } else { /* child code */  
        exec(command, params, 0); /* memory image  
        change */  
        error("Command cannot be executed");  
        exit(1);  
    }  
}
```

- Identification
- Creation
- Waiting
- Ending
- **Signals**

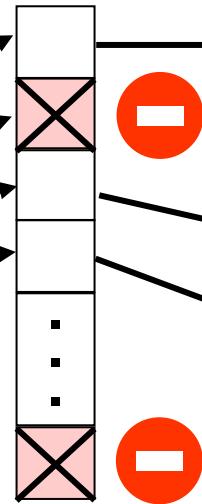
Signals generation



- Terminal
- Errors
- Software

`kill(...)`
`alarm(..)`

Signal mask



Signals handling

`handler(...)`

`SIG_IGN`

`SIG_DFL`

Handler executes

Ignors

Default action
(exit)

Mask definition

`sigemptyset(...)`
`sigfillset(..)`
`sigaddset(..)`
`sigdelset(..)`
`sigismember(...)`

Mask setting

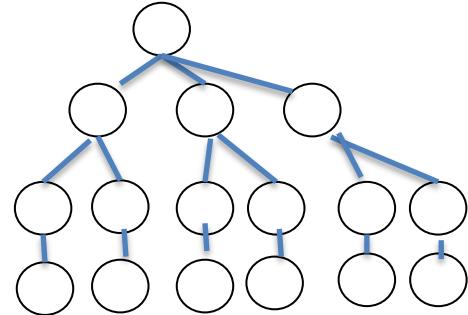
`sigprocmask(...)`

Handler setting

`sigaction(...)`
Signal waiting
`pause(...)`
`sigsuspend(...)`

- A signal is the **mechanism used by the OS to inform processes about certain events**
- They all follow the same pattern:
 - It is generated due to the occurrence of an event
 - It is supplied to the process
 - It should be treated by the default handling routine or by the one defined by a specific process
- A signal can:
 - Be handled: a handling routine must be installed
 - Be masked: its handling is delayed
 - Be ignored: its occurrence if not informed

- Write a program that build a process tree like the figure.



- Write a modification to wait the parent process to the termination of its own childs