User: alumno
Passwd: usuario

Session 1
Process Management

Departamento de Tecnología de los Computadores y las Comunicaciones
Universidad de Extremadura

Process Management

Use of Makefiles

```
$ cd ; mkdir ll_1 ; cd ll_1; mkdir calculator ; cd calculator
$ gedit root_of_2.c
```

```c
#include <stdio.h>        /* Header of function printf */
#include <stdlib.h>       /* Use of exit() */
#include "calculator.h"   /* Header of function square_root */

int main(void)
{
  printf("The square root of 2 is %f.\n", square_root(2.0));
  exit(0);
}
```

root_of_2.c

?

```
$ gedit calculator.c
```

```c
#include <math.h>
float add (float x, float y)
{
  return x+y;
}

float square_root (float x)
{
  return sqrt(x);
}
```

calculator.c

Implementation

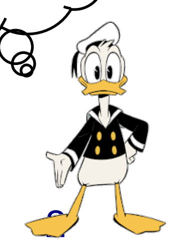
```
$ gedit calculator.h
```

```c
float add         (float x, float y);
float square_root (float x);
```

calculator.h

Interface

How do I compile this?



sqrt is in the mathematical library. That is why we need <math.h>

Use of Makefiles

```
$ gedit Makefile
```

Objective — Precedence — Rule

```
root_of_2: root_of_2.o calculator.o
<TAB>gcc root_of_2.o calculator.o -o root_of_2 -lm

root_of_2.o: root_of_2.c
<TAB>gcc -c root_of_2.c

calculator.o: calculator.c
<TAB>gcc -c calculator.c

clean:
<TAB>rm root_of_2 *.o
```

To link the mathematical library which contains the *sqtr* function

Makefile

A rule is *triggered* if at least one precedence is **more recent** than the objective (see **$** `ls -lt`)

- Now we have the source files that compose our program. **Next we build the executable.**
- To this end we must create what is known as a **makefile** file.
- A makefile is a file that provides with a set of **rules** about the steps to generate the final executable
- The left side example contains four rules

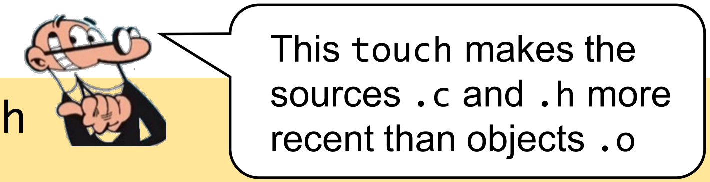
Process Management

Use of Makefiles

```
$ make
gcc -c root_of_2.c
gcc -c calculator.c
gcc -o root_of_2 root_of_2.o calculator.o -lm
```

```
$ ls
calculator.c  calculator.h  calculator.o  makefile  root_of_2
root_of_2.c root_of_2.o
$ make clean ; ls
calculator.c  calculator.h   makefile  root_of_2.c
$ make
$ ./root_of_2
The square root of 2 is 1.414214.
```

```
$ touch *.c *.h
$ make
```



Arguments of a program

We all know that *main* is the first function that will be called inside a program. Its prototype is:

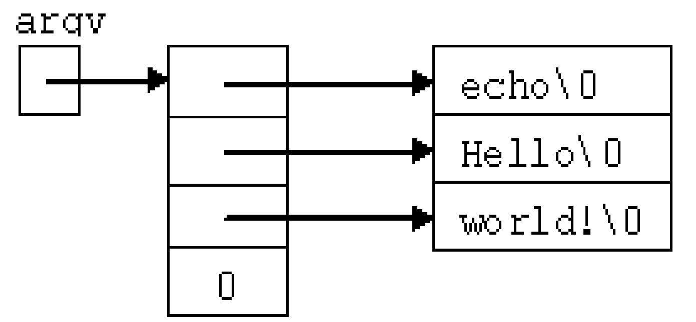
```
int main(int argc, char *argv[]);
```

If we type

$ echo Hello world!

The **argc** argument passed to the main of echo is **3**

The **argv** passed is as figure shows




```
$ cd .. ; mkdir argv; cd argv
$ gedit arguments.c
$ gcc arguments.c -o arguments
$ ./arguments These are my arguments
```

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

int main(int argc, char *argv[])
{
  while(argc--)
   printf("%s ", *argv++);
  printf("\n");
  exit(EXIT_SUCCESS);
}
```

arguments.c

This program prints its arguments

System Calls and the Unix manual

To invoke a system call in your program it is necessary to know:

- Its name, its parameters and its return value.
- Also, the **header** file or files which need to be included in the program.

For instance, if we want to change the permissions of file *bridge.c*, so that every user has read and write access to it, we would invoke the *chmod* system call:

```
#include <sys/types.h>    /* This at the beginning of the program */
#include <sys/stat.h>
...
chmod("bridge.c", 0666);
...
```

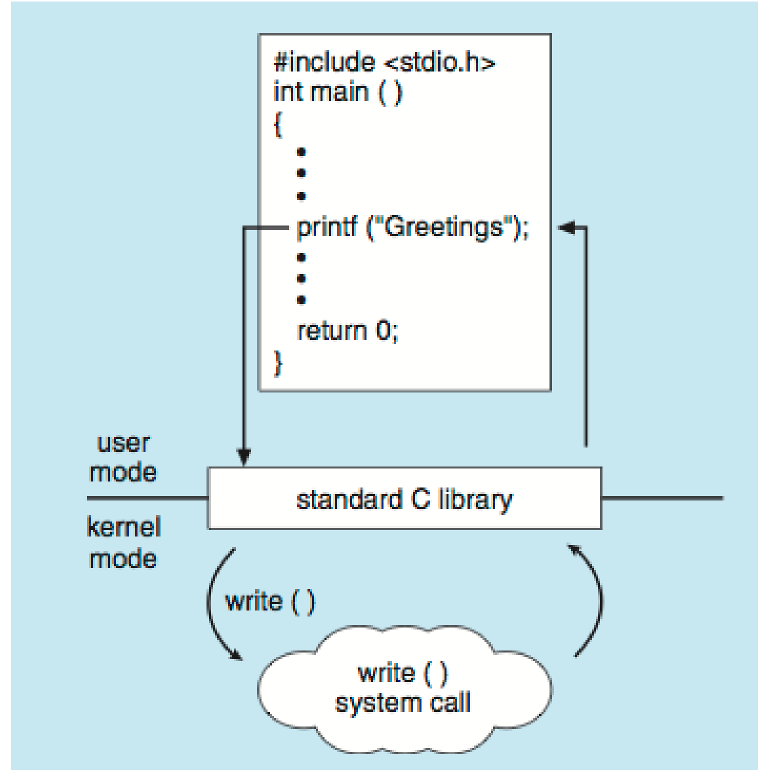
```
$ man -s 2 chmod
CHMOD(2)                Manual del Programador de Linux               CHMOD(2)

NOMBRE
       chmod, fchmod - cambia los permisos de un fichero

SINOPSIS
       #include <sys/types.h>
       #include <sys/stat.h>

       int chmod(const char *path, mode_t mode);
       int fchmod(int fildes, mode_t mode);
...
$ man -s 1 chmod
```

← *This is the* **chmod command**, *rather than the system call*

The manual pages of system calls are in the **section 2** of the manual

System Calls and the Unix manual

- All the functions of the **C library** (`printf` or `malloc`, for instance) internally invoke system calls.
- C library function `printf`, for instance, invokes the `write` system call, as shown in the figure:



The manual pages of the C library are in the **section 3**

```
$ man -s 3  printf
PRINTF(3)                Manual del Programador de Linux              PRINTF(3)

NOMBRE
       printf, fprintf, sprintf, vprintf, vfprintf, vsprintf - conversión de salida formateada

SINOPSIS
       #include <stdio.h>

       int printf(const char *format, ...);
...
```

Error conditions

- A system call may fail.
  For instance, the call chmod of slide 6 can fail because the file *bridge.c* does not exist, because we have not the suitable permissions to change the permissions of *bridge.c*, etc.
- It is the **responsibility of the programmer** to detect when an error has occurred. We know about the error because the system call returns a special value, usually -1 (minus 1).
- In any case, we should read the manual page to know about this value.
- To detect the error, we would do as follows:

```
#include <sys/types.h>
#include <sys/stat.h>
...
if (0 > chmod("bridge.c",0666)) {
  printf("Can not change the file permissions.\n");
  exit(1);
}
```

Error conditions

- To know the specific type of error produced we have to read a global variable known as `errno`
- The **manual page** informs on which value `errno` takes

  The manual page of `chmod` system call says:

```
$ man -s 2 chmod
...
ERRORS
...
EACCES     Search permission is denied on a component of the path prefix.
EFAULT     Pathname points outside your accessible address space.
EIO        An I/O error occurred.
ELOOP      Too many symbolic links were encountered in resolving pathname.
...
ENOENT     The file does not exist.
ENOMEM     Insufficient kernel memory was available.
ENOTDIR
```

```
#include <sys/stat.h>
#include <errno.h>
...
if (0 > chmod("bridge.c", 0666)) {
  if (errno == ENOENT)
    fprintf(stderr,"File bridge.c does not exist.\n");
  else
    fprintf(stderr,"Impossible to change the permissions");
  exit(1);
}
```

Error conditions

- Considering all and each of the possible value of **errno** is a *costly and tedious task*.
- The useful alternative is the **perror** C library function, which avoids all these tests.
- Function **perror** internally tests the **errno** variable and writes to standard error:
    1. A string that <u>we pass to it</u> as parameter,
    2. colon
    3. and **the system message describing the error**.
- We pass in string (1) the name of the program, the function in the program or any other reference helping the programmer to locate the line which raised the error.
- The former code may appear as follows, by using the function **perror**:

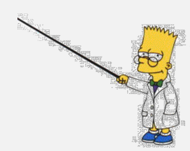
```
#include <sys/stat.h>
#include <stdio.h>
...
if (0 > chmod("bridge.c",0666)) {
  char my_Message[80];
  sprintf(my_Message,"%s, chmod, bridge.c", argv[0]);
  perror(my_Message);
  exit(1);
}
...
```

Reserve space for the user string

Build the user string

Pass the user string

Make sure you fully understand **sprintf**



Identifiers of users and processes system calls

Calls *getpid*, *getppid* and *getuid* obtain the process identifier (PID) , that of his parent (PPID), and its owner (UID), respectively:

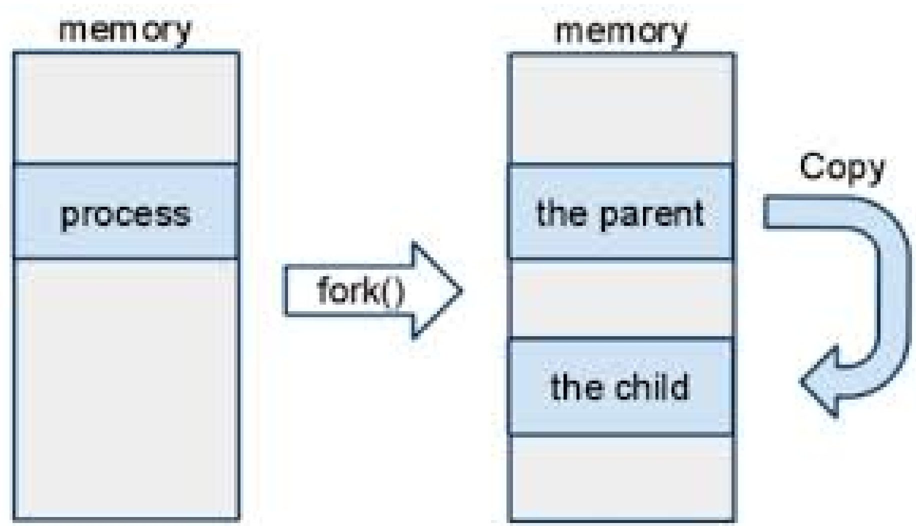
```
#include <sys/types.h>
#include <unistd.h>

pid_t getpid(void);
pid_t getppid(void);
uid_t getuid(void);
```

Example Next program prints the identifier of the invoking process, of its father and its owner.

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

int main(void)
{
  printf("ID de proceso:               %ld\n", (long)getpid());
  printf("ID de proceso padre:         %ld\n", (long)getppid());
  printf("ID de usuario propietario: %ld\n", (long)getuid());
  return 0;
}
```

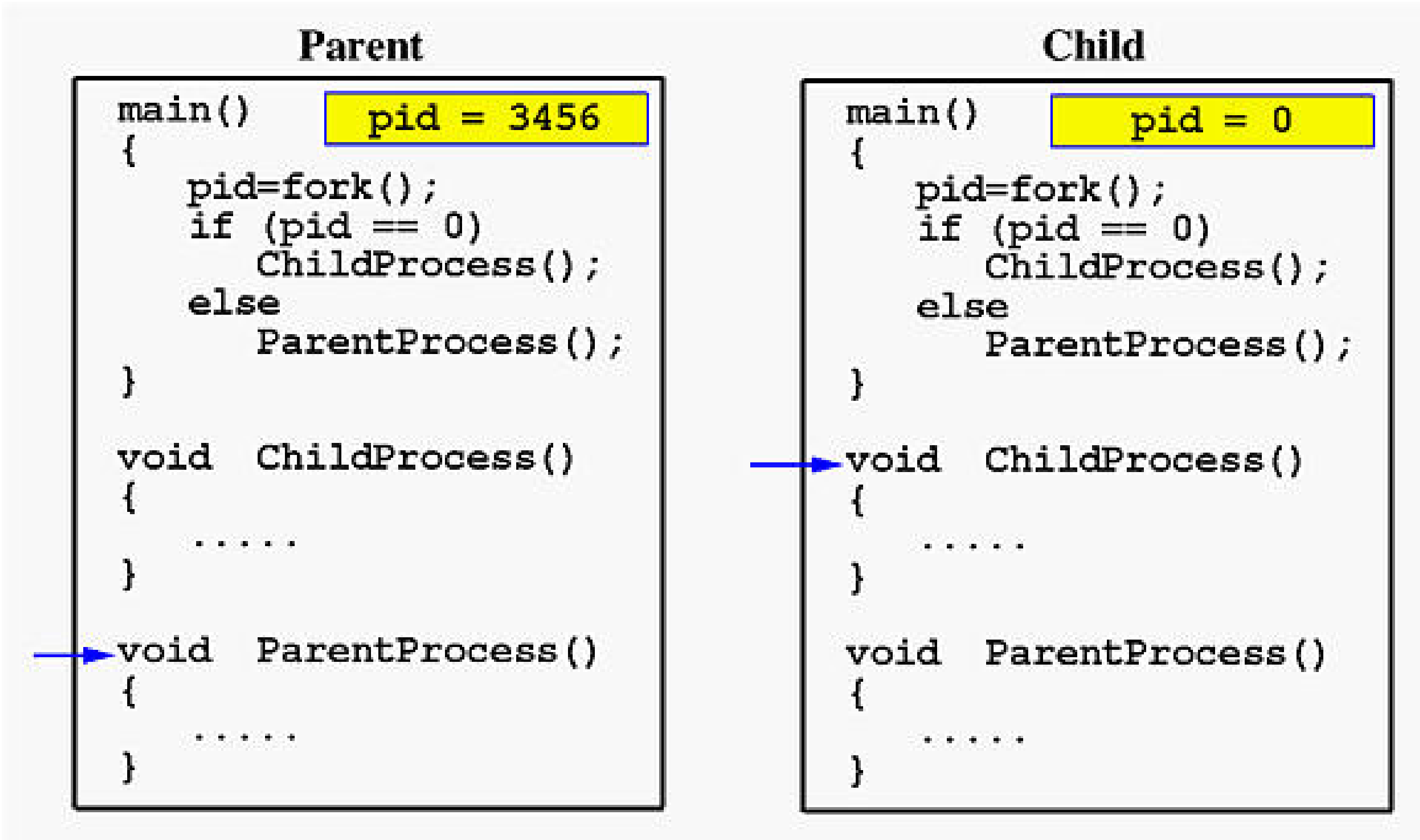
Fork system call

- A new process is created by the **fork** system call.
- The new process consists of a copy of the address space of the original process.



- Both processes continue their execution right after the system call `fork()`.
- Since both processes have identical but separate address spaces, those **variables** initialized before the fork() call have the same values in both address spaces

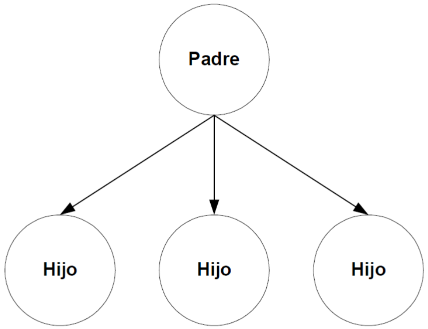
Fork system call



Both processes continue execution at the instruction after the fork(), **with one difference**: the return code for the fork() is **zero** for the new (child) process, whereas the **process identifier of the child** is returned to the parent.

Fork system call

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

pid_t fork(void);
```



Example Next program *fork.c* produces a tree like that of the figure:

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main(int argc, char **argv)
{
  int   i;
  pid_t pid;
  for (i=0; i < 3; i++) {
    if (0 > (pid = fork())) {
        perror("");
        return(1);
    }
    if (pid == 0) {
      fprintf(stdout, "Child: My parent is %ld\n", (long)getppid());
      break;
    }
    fprintf(stdout, "Parent %ld: Created child %ld\n", (long)getpid(), (long)pid);
  }
  return 0;
}
```

fork.c

Makefile

```
all: fork
fork: fork.c
<TAB>gcc fork.c -o fork

clean:
<TAB>rm fork
```

```
$ cd .. ; mkdir fork; cd fork
$ gedit fork.c
$ gedit Makefile
$ make
$ ./fork
```

Exit system call

```
#include <stdlib.h>
void exit (int status);
```

status is between 0 and 255

Extend! *the fork/Makefile with the green code*

Example

```
all: fork exit1
fork: fork.c
<TAB>gcc fork.c -o fork
exit1 : exit1.c
<TAB>gcc exit1.c -o exit1

clean:
<TAB>rm fork exit1
```

fork/**Makefile**

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

int main(int argc, char **argv) {

   int dividend = 20;
   int divisor  = atoi(argv[1]);
   int quotient;

   if( divisor == 0) {
      fprintf(stderr, "Division by zero! Exiting...\n");
      exit(1);
   }

   quotient = dividend / divisor;
   fprintf(stderr, "Value of quotient : %d\n", quotient );

   exit(0);
}
```

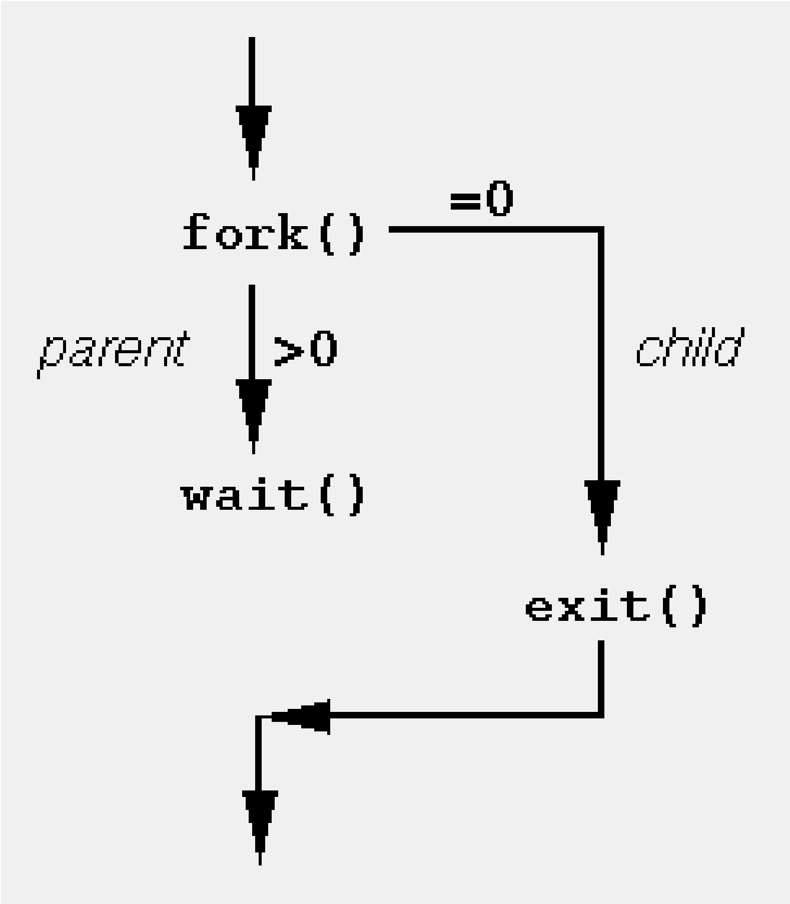
fork/**exit1.c**

```
$ gedit exit1.c
$ gedit Makefile
$ make
$ ./exit1 2
$ ./exit1 0
```

Wait system call

A parent can then create more children or, if it has nothing else to do while the child runs, it can issue a **wait** system call to suspend itself until the termination of a child:

```
main()
{
  pid = fork();
  if (pid == 0) {
    /* child */
    ...
    exit(code);
  }
  /* parent */
  wait(&status);
}
```



Process Management

Wait system call

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

pid_t wait(int *status);
```

`status` is an output parameter

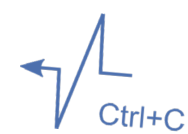
either the argument of a call to `exit` or the argument of `return` in main

If `status` is different from NULL, `wait` fills it with the exit value of the finished child. Once `wait` returns, we can apply macros to status to know precisely the cause of termination of the child:

`WIFEXITED``(status)`
If it returns *true* (other than zero), it indicates whether the child ended by calling *exit*.

`WEXITSTATUS``(status)`
Returns the *eight* least significant bits of status. Logically, it can only be applied if `WIFEXITED` returns true.

`WIFSIGNALED``(status)`
If it returns *true*, the child is terminated because killed by a signal.



`WTERMSIG``(status)`
Returns the name of the signal that caused the death of the child process. Logically, this macro can only be applied if `WIFSIGNALED` returned true.

Wait system call

Example

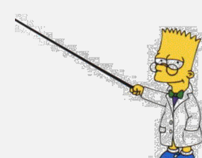
```
all: fork exit1 wait
wait : wait.c
<TAB>gcc wait.c -o wait

clean:
<TAB>rm fork exit1 wait
```

fork/**Makefile**

Keep extending *the fork/Makefile with the green code*

```
$ gedit wait.c
$ gedit Makefile
$ make
$ ./wait
```

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/wait.h>
int main()
{
  int number, statval;
  printf("%d: I'm the parent !\n", getpid());
  if(fork() == 0) {
    number = 10;
    printf("PID %d: exiting with number %d\n", getpid(), number);
    exit(number);
  }
  else {
    printf("PID %d: waiting for child\n", getpid());
    wait(&statval);
    if(WIFEXITED(statval))
      printf("Child's exit code %d\n", WEXITSTATUS(statval));
    else
      printf("Child did not terminate with exit\n");
  }
  return 0;
}
```

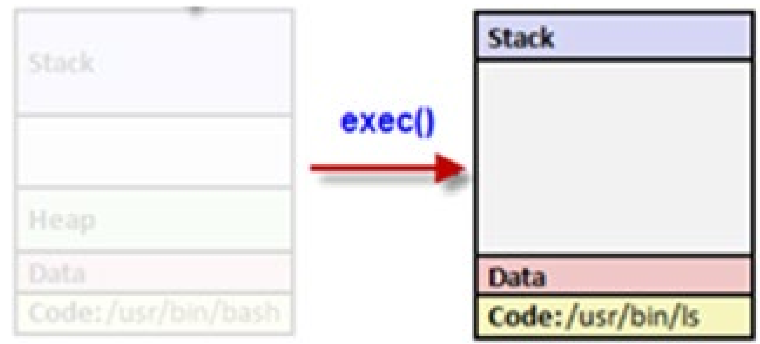


This should return 10

fork/**wait.c**

Exec system call

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




```
#include <unistd.h>
int execl(char *path,
          char *arg0, ..., char *argn,
          NULL);
```

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

int main() {
  int status;
  printf ("Mi lista de procesos\n");
  if (0 > execl("/bin/ps", "ps", "ux", NULL)) {
    fprintf(stderr, "Error en exec %d\n", errno);
    exit(1);
  }
  printf ("Fin de mi lista de procesos\n");
  exit(0);
}
```

execl.c

```
all: fork exit1 wait execl
execl: execl.c
<TAB>gcc execl.c -o execl

clean:
<TAB>rm fork exit1 wait execl
```

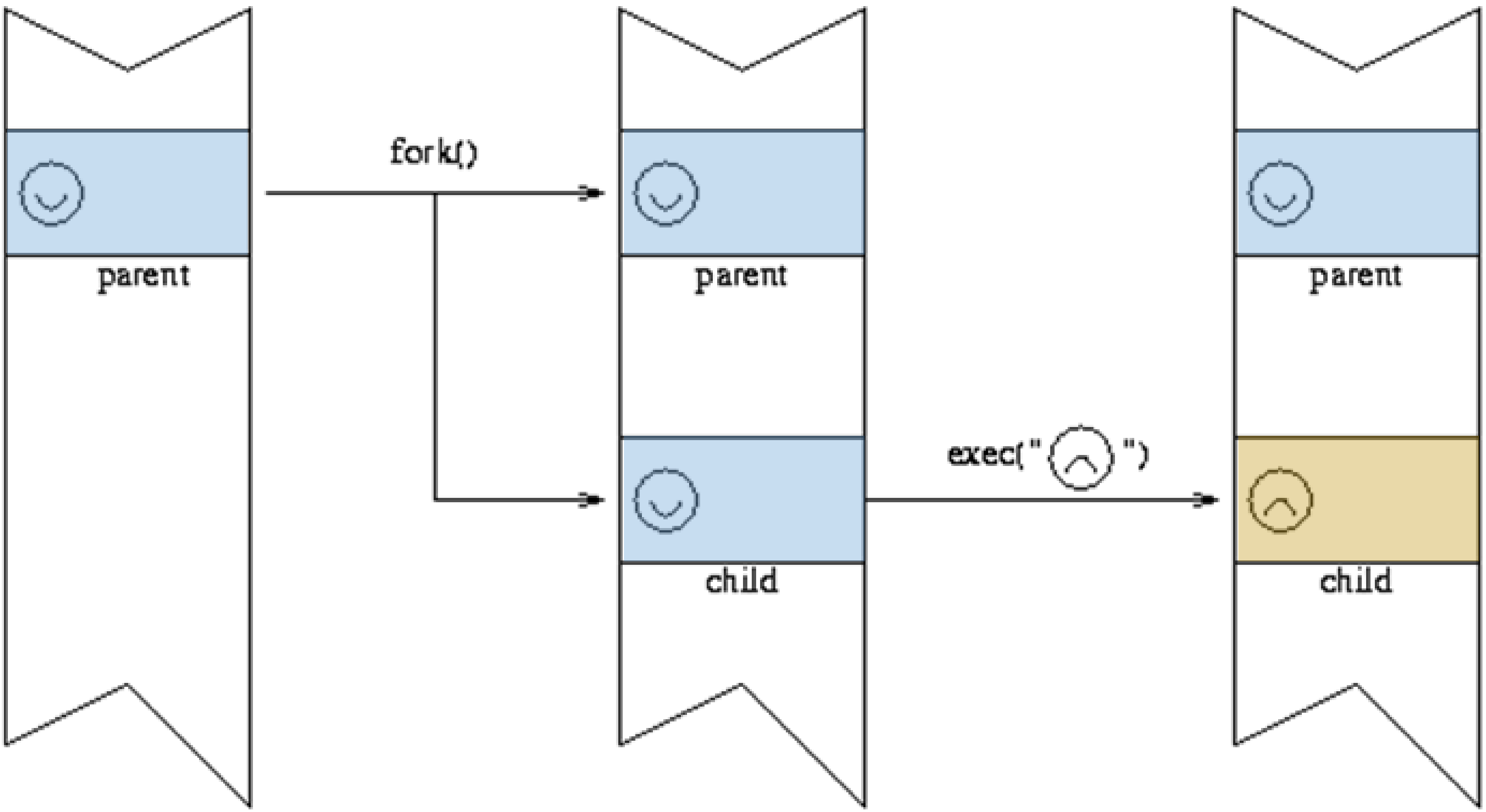
fork/**Makefile**

```
$ gedit execl.c
$ gedit Makefile
$ make
$ ./execl
```

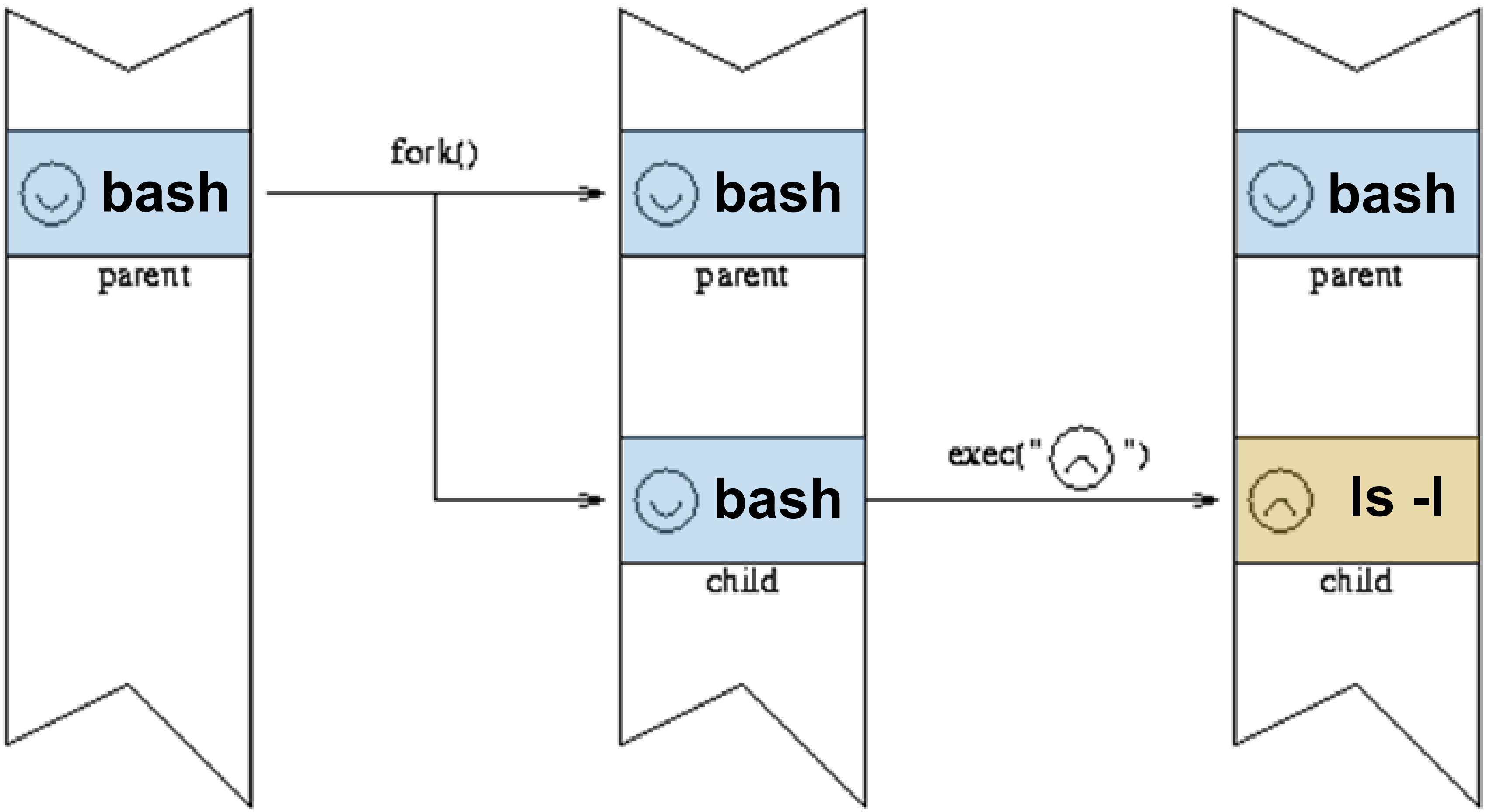
Why does not appear on the screen?

Exec system call

After a **fork** system call, the child typically invokes the **exec** system call to replace the process's memory space with a new program.



Exec system call



Exercises

Exercise 1

Realice un programa un programa *./exercise1/exhaustFork.c.* Invoca la llamada al sistema *fork* en un blucle hasta que *fork* falla porque se ha terminado la memoria del sistema. Llegado el caso, muestre el número de réplicas que sí han podido crearse. El hijo invoca sleep(16) y termina.

Exercise 2

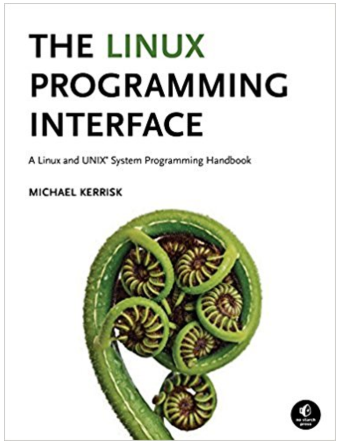
Realice un programa *./exercise2/launch.c.* Lee de la entrada estándar (mediante la función de la biblioteca C *scanf*) la ruta de un programa y cree un proceso hijo para ejecutar dicho programa. Pruebe a lanzar el programa *arguments.c* de la transparencia 5.

Utilice la función `basename` de la biblioteca C para extraer de la ruta del fichero ejecutable el nombre del mismo.

Home Exercise 3

Realice un programa denominado *./exercise3/four.c*. Creará cuatro procesos, A, B, C y D, de forma que A sea padre de B, B sea padre de C, y C sea padre de D. D debe invocar la llamada al sistema *execl* para mutar en el nuevo proceso *ps*, de modo que la salida en pantalla muestre la relación de parentesco entre los cuatro procesos. Utilice *man ps*.

References



**The Linux Programming Interface: A Linux and UNIX System Programming Handbook,
1st Edition,** [Michael Kerrisk]