

# SISTEMAS OPERATIVOS: SERVICIOS DE LOS SISTEMAS OPERATIVOS



Llamadas al sistema

# ADVERTENCIA

- ❑ Las transparencias ayudan como simple gui3n de la clase pero no son los apuntes de la asignatura.
- ❑ El conocimiento exclusivo de este material no garantiza que el/la estudiante pueda alcanzar los objetivos de la asignatura.
- ❑ Se recomienda que el/la estudiante utilice todos los materiales bibliogr3ficos propuestos para complementar los conocimientos.

# Objetivos

- ❑ Comprender qué es un servicio del sistema operativo.
- ❑ Conocer las principales características de la interfaz POSIX.
- ❑ Conocer los principales servicios ofrecidos por POSIX (procesos y ficheros)
- ❑ Comprender los mecanismos que intervienen en una llamada al sistema.

# Contenidos

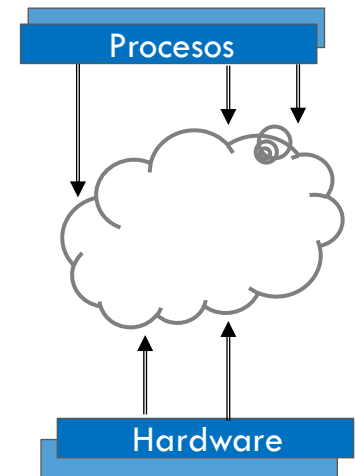
- Introducción a llamadas al sistema
- Mecanismo de llamada al sistema
- Llamadas para servicios de:
  - ▣ Gestión de procesos
  - ▣ Gestión de ficheros y directorios

# Contenidos

- Introducción a llamadas al sistema
- Mecanismo de llamada al sistema
- Llamadas para servicios de:
  - ▣ Gestión de procesos
  - ▣ Gestión de ficheros y directorios

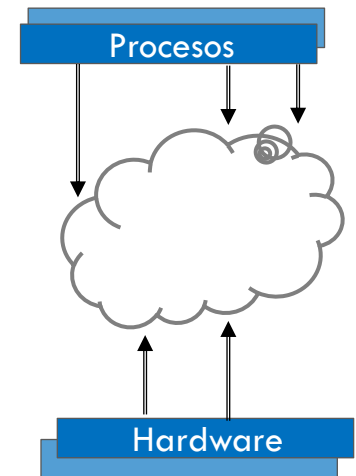
# Ejecución del sistema operativo

- Durante el arranque.
- Una vez finalizado el arranque, se ejecuta en respuesta a eventos:
  - Llamada al sistema.
  - Excepción.
  - Interrupción hardware.
- En procesos de núcleo (firewall, etc.)



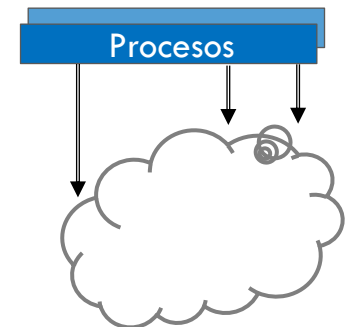
# Eventos que activan el sistema operativo

- **Llamada al sistema.**
  - ▣ { Origen: “procesos”,  
Función: “Petición de servicios” }
- **Excepción.**
  - ▣ { Origen: “procesos”,  
Función: “Tratar situaciones de excepción” }
- **Interrupción hardware.**
  - ▣ { Origen: “hardware”,  
Función: “Petición de atención del hw.” }



# Servicios del sistema

- Gestión de procesos
- Gestión de memoria
- Gestión de ficheros
- Gestión de dispositivos
- Comunicación
- Mantenimiento





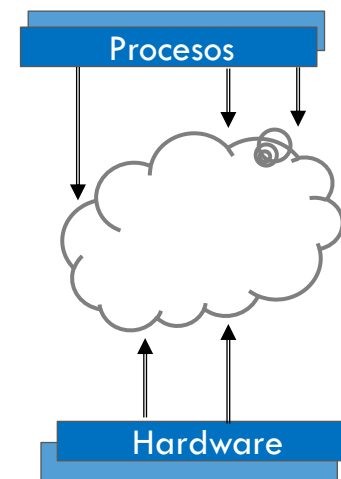
# Llamadas al sistema...

## resumen

9

Alejandro Calderón Mateos 

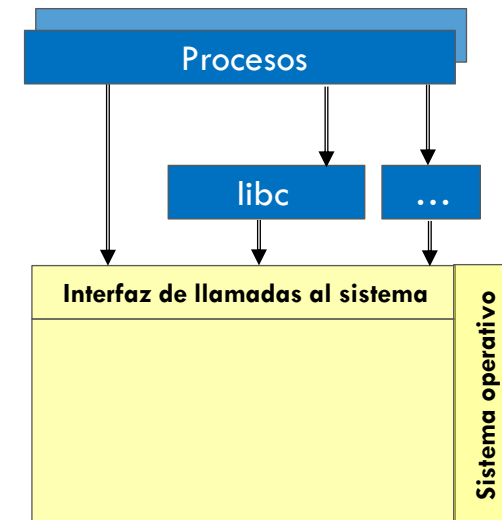
- Durante el arranque.
- **Tras el arranque, se ejecuta en respuesta a eventos:**
  - **Llamada al sistema.**
    - { Origen: “procesos”, Función: “**Petición de servicios**” }
      - **Gestión de procesos**
      - **Gestión de memoria**
      - **Gestión de ficheros**
      - **Gestión de dispositivos**
      - **Comunicación**
      - **Mantenimiento**
  - Excepción.
    - { Origen: “procesos”, Función: “Tratar excepciones” }
  - Interrupción hardware.
    - { Origen: “hardware”, Función: “Petición de atención del hw.” }
- En procesos de núcleo (firewall, etc.)



# Llamadas al sistema versus...

- Los **mandatos** no son llamadas al sistema.
  - ▣ Es posible que un mandato del *shell* en línea de mandatos (`/bin/sh`) invoque internamente la llamada.
  - ▣ Ej.: `printf` vs `printf()`
  
- No toda **función de la librería** de sistema es una llamada al sistema.
  - ▣ Aunque es posible que una función de librería extienda las funcionalidades de varias llamadas al sistema.
  - ▣ Ej.: `sbrk()` vs `malloc()`

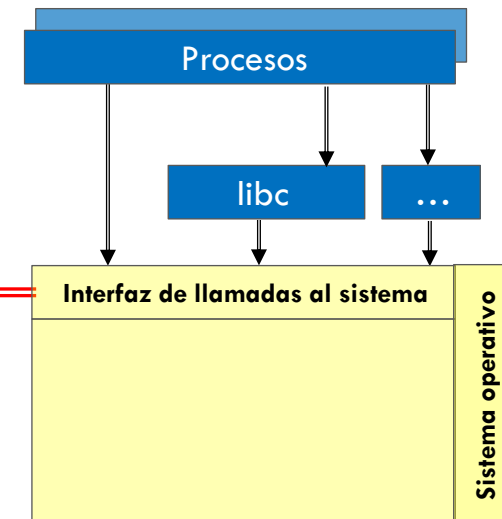
# Llamadas al sistema vs librería sistema



# Llamadas al sistema vs librería sistema



“servicios muy básicos de la casa”

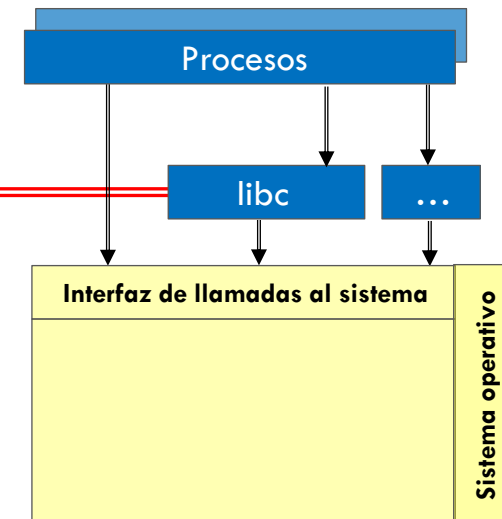


# Llamadas al sistema vs librería sistema

13

<https://www.pexels.com/es-es/foto/tablas-de-cortar-cerca-del-horno-debajo-del-capo-2062426/>

Alejandro Calderón Mateos 



# Llamadas al sistema vs librería sistema

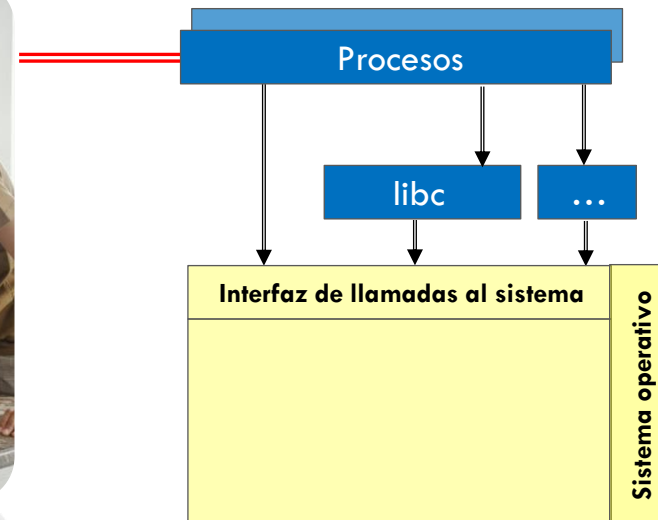
14

<https://www.pexels.com/es-es/foto/hombre-en-camisa-de-vestir-blanca-sentado-al-lado-de-una-mujer-en-vestido-naranja-426241>

Alejandro Calderón Mateos



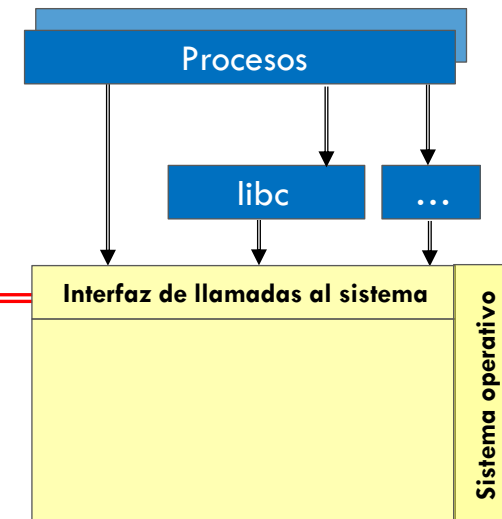
“personas que utilizan los servicios”



# Llamadas al sistema vs librería sistema memoria

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

- int brk (void \*);
- void \*sbrk (intptr\_t);
- int close (int);
- off\_t lseek (int, off\_t, int);
- ssize\_t read (int, void \*, size\_t);
- ssize\_t write (int, const void \*, size\_t);
- ...
- int open (const char \*path, int oflag, ... );
- int creat (const char \*path, mode\_t mode);
- ...



# Llamadas al sistema vs librería sistema memoria

16

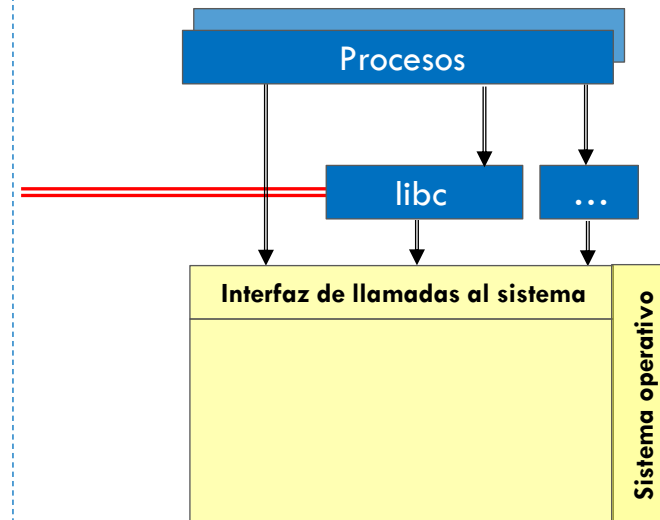
Alejandro Calderón Mateos 

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

- ❑ void \*malloc (unsigned long Size);
- ❑ void \*realloc (void \*Ptr, unsigned long NewSize);
- ❑ void \*calloc (unsigned short NItems, unsigned short SizeOfItems);
- ❑ void free (void \*Ptr);
- ❑ ...

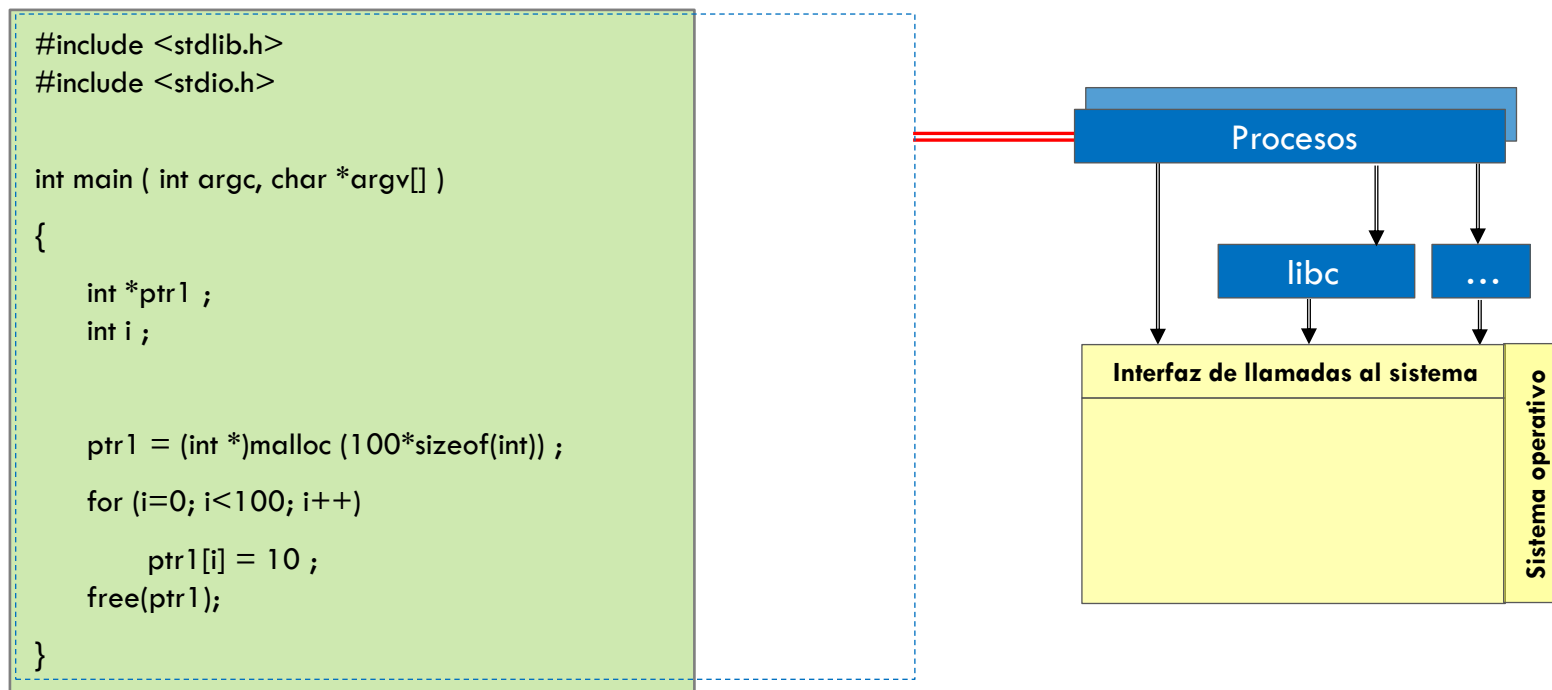
```
#include <stdio.h>
```

- ❑ FILE \* fopen (const char \*filename, const char \*opentype);
- ❑ int fclose (FILE \*stream);
- ❑ int feof(FILE \*fichero);
- ❑ int fseek ( FILE \* stream, long int offset, int origin );
- ❑ size\_t fread ( void \* ptr, size\_t size, size\_t count, FILE \* f);
- ❑ int fscanf(FILE \*f, const char \*formato, argumento, ...);
- ❑ size\_t fwrite(void \*ptr, size\_t size, size\_t neltos, FILE \*f);
- ❑ int fprintf(FILE \*f, const char \*fmt, arg1, ...);
- ❑ ...





# Llamadas al sistema vs librería sistema memoria



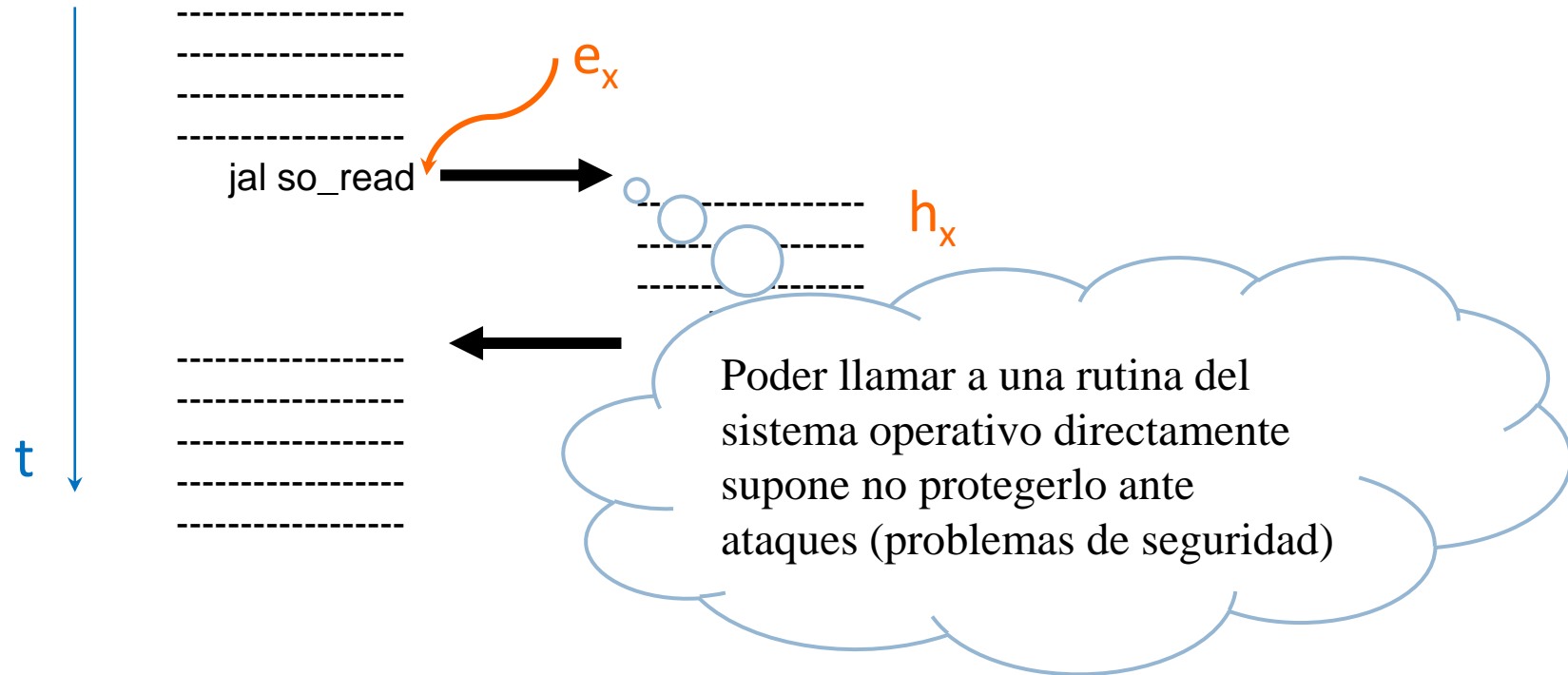
# Contenidos

- Introducción a llamadas al sistema
- Mecanismo de llamada al sistema
- Llamadas para servicios de:
  - ▣ Gestión de procesos
  - ▣ Gestión de ficheros y directorios

# Ejecución petición de servicio no es una llamada a una función...

19

Alejandro Calderón Mateos 

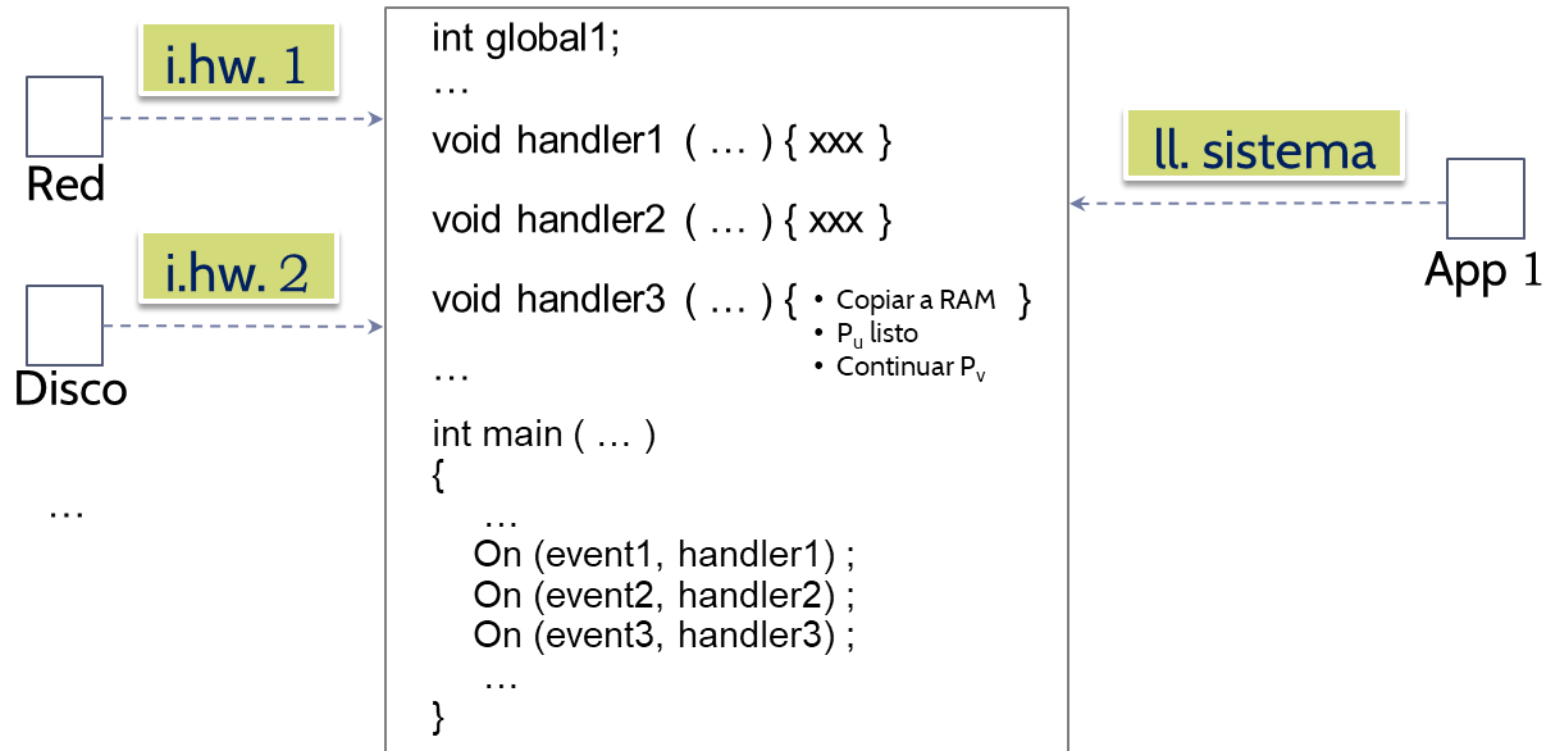


# Ejecución tratando eventos

## aspecto general

20

Alejandro Calderón Mateos

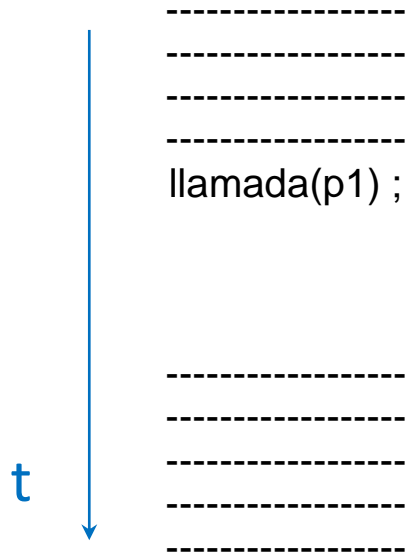


# Ejecución petición de servicio

## ejecución (general)

21

Alejandro Calderón Mateos

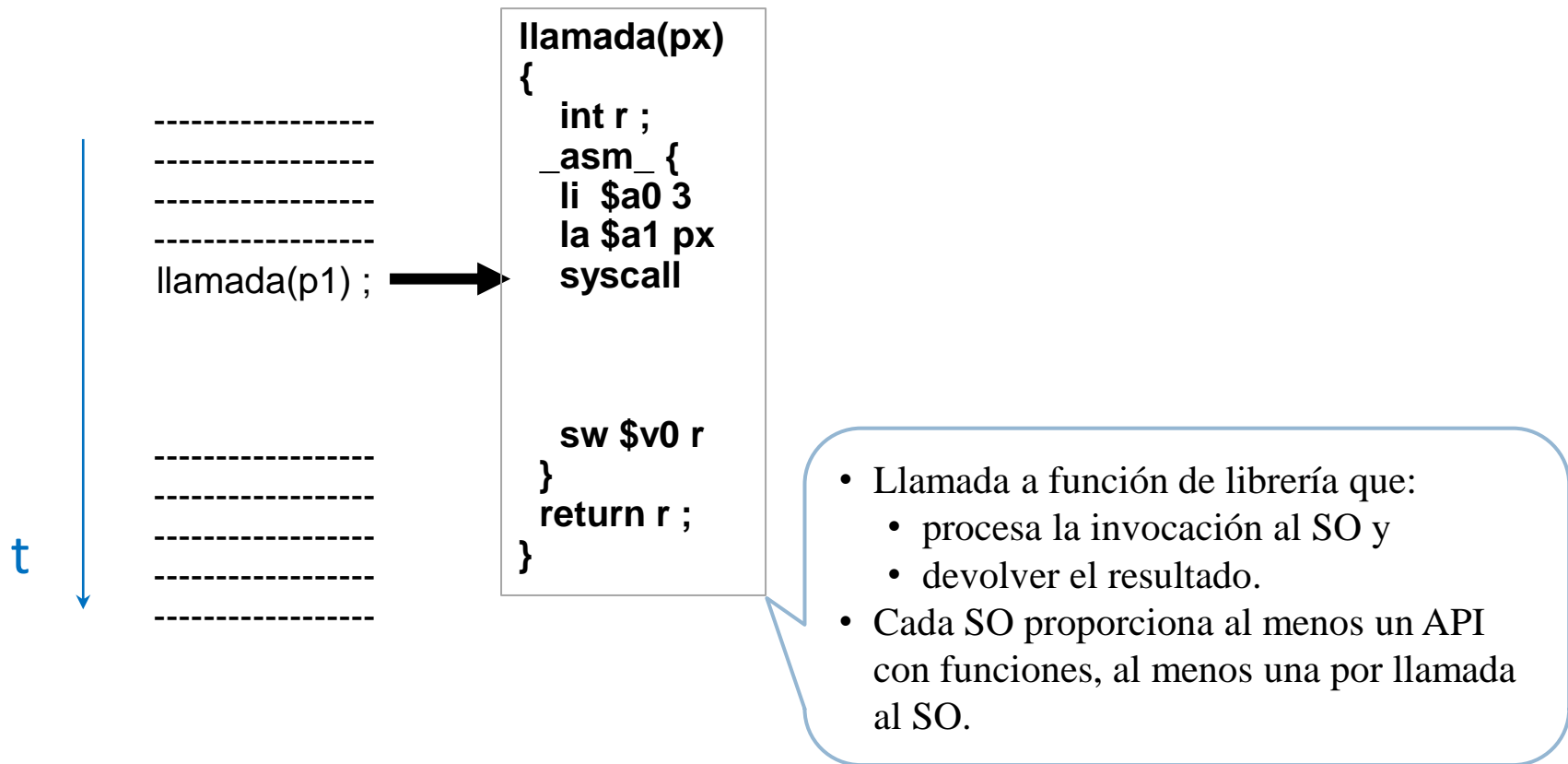


# Ejecución petición de servicio

## *ejecución (general)*

22

Alejandro Calderón Mateos 

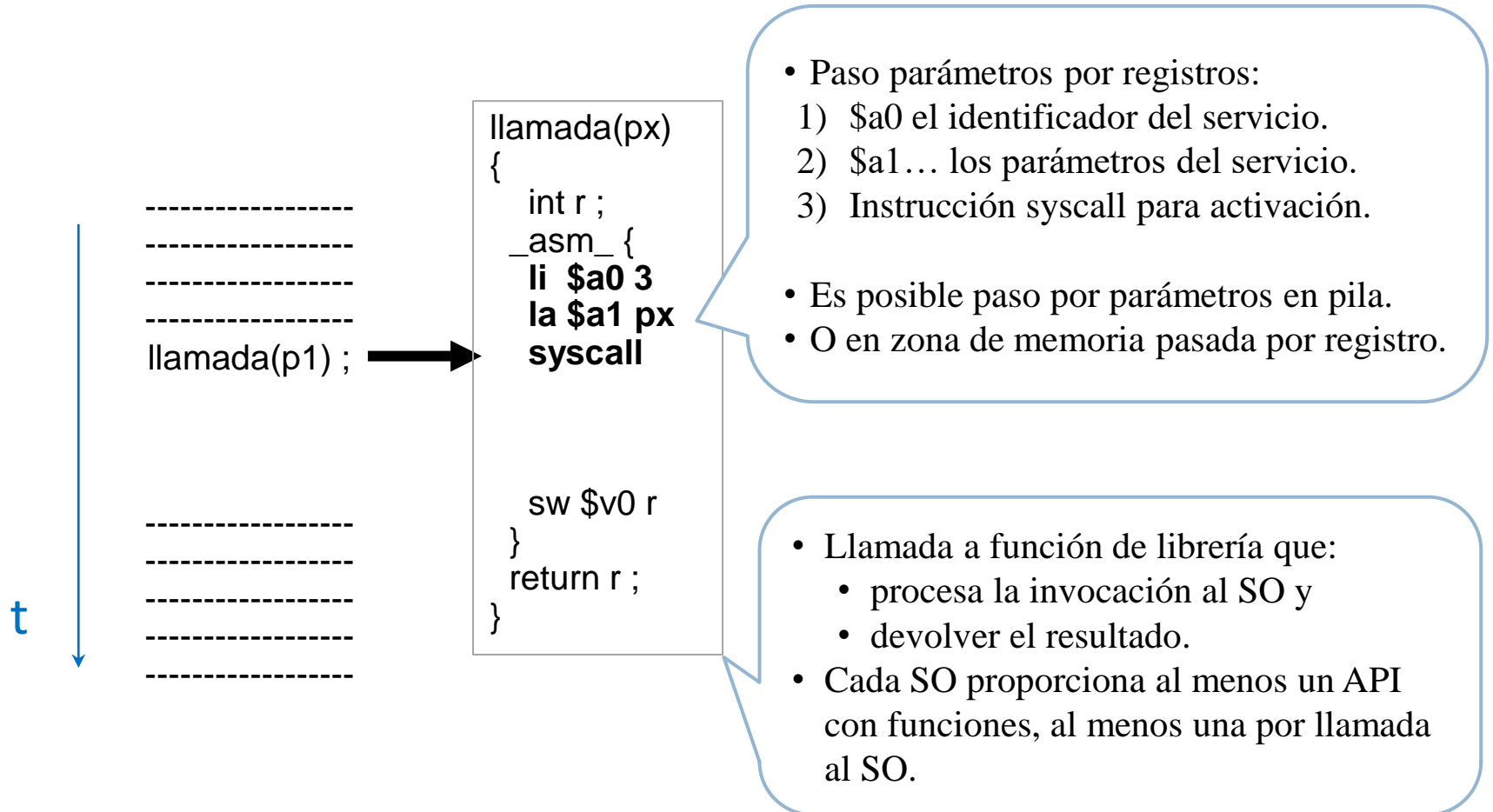


# Ejecución petición de servicio

## *ejecución (general)*

23

Alejandro Calderón Mateos 

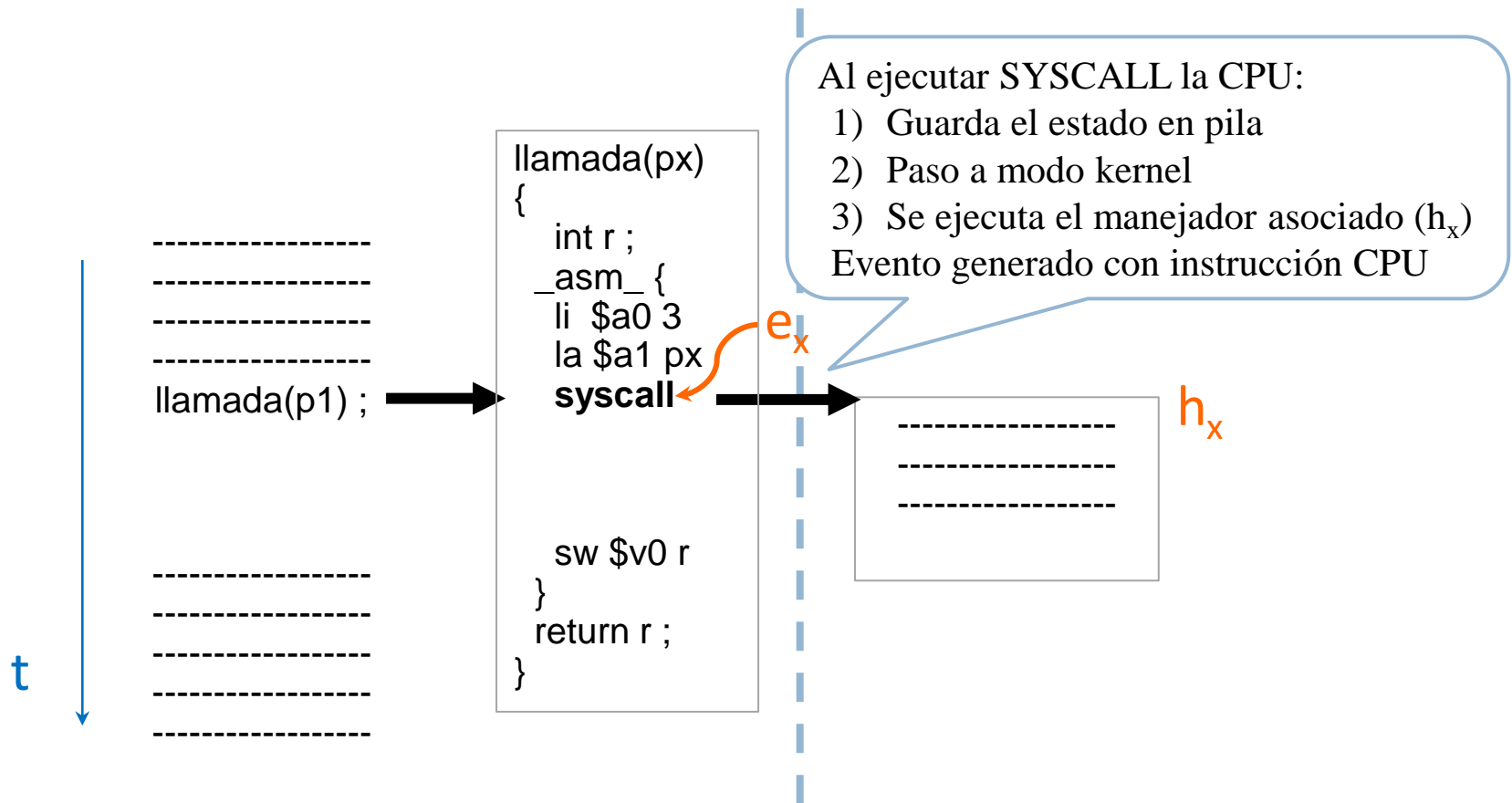


# Ejecución petición de servicio

## *ejecución (general)*

24

Alejandro Calderón Mateos



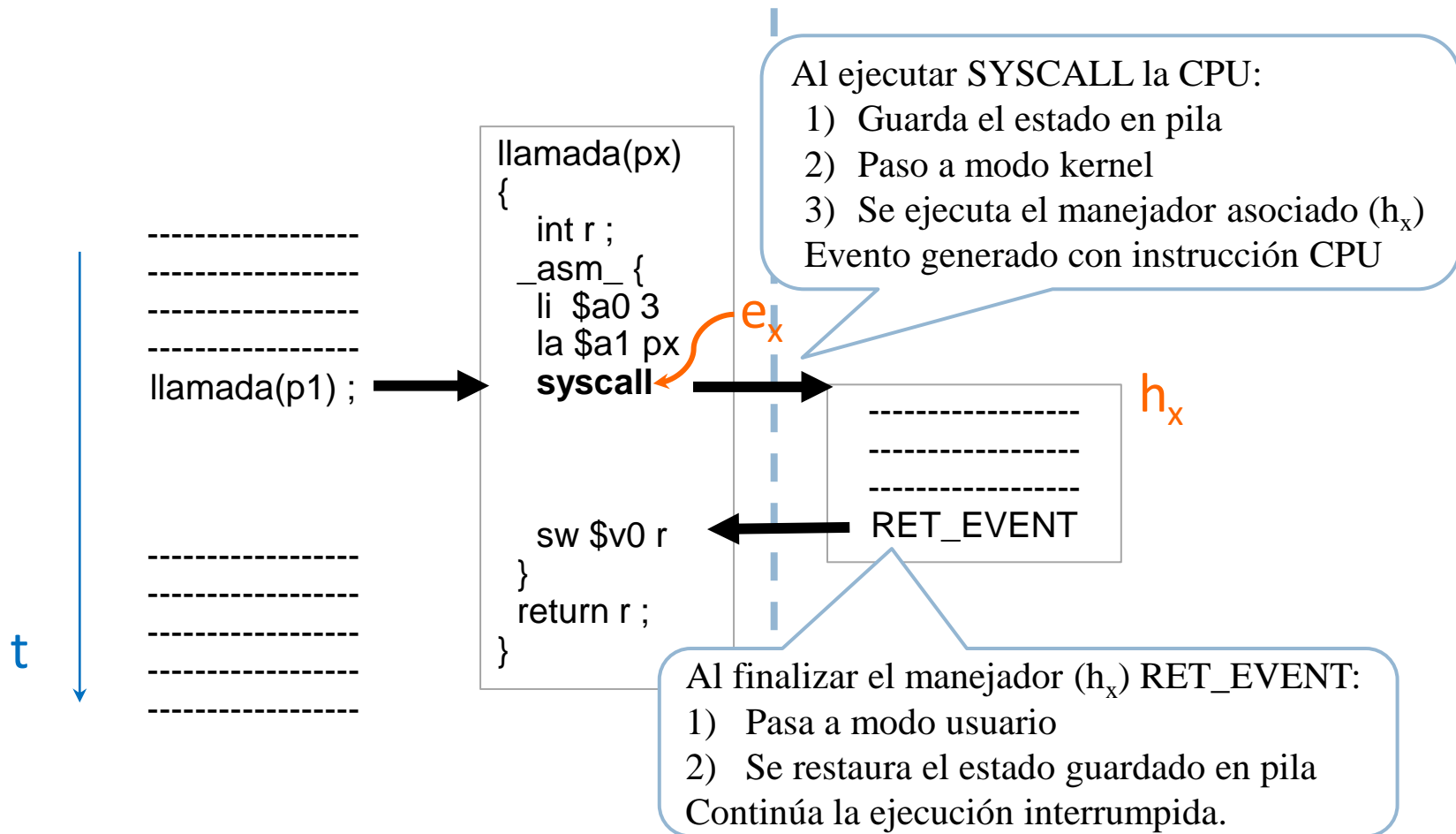


# Ejecución petición de servicio

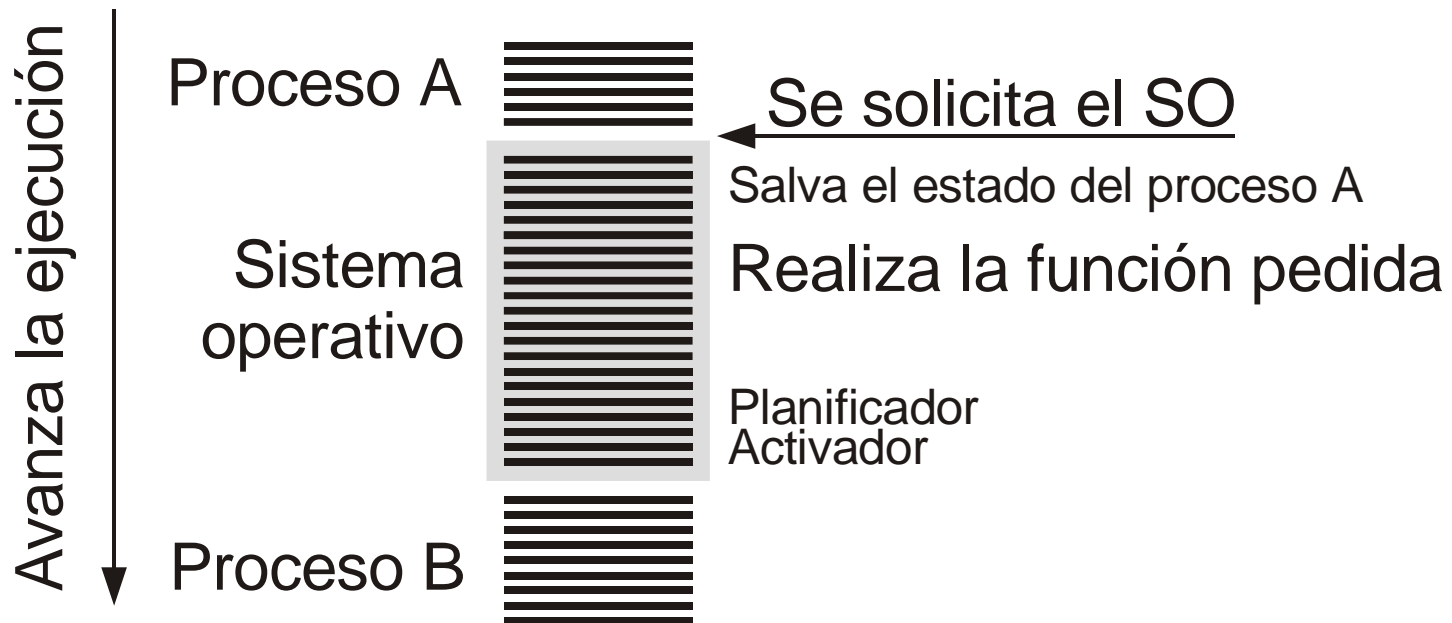
## ejecución (general)

25

Alejandro Calderón Mateos



# Fases en la activación del Sistema Operativo



# Llamadas al sistema

## tratamiento en Linux (1 / 7)

27

Alejandro Calderón Mateos 

/usr/src/linux/arch/x86/kernel/traps.c

```
void __init trap_init(void)
{
    ...
    set_intr_gate(X86_TRAP_DE, divide_error);
    set_intr_gate(X86_TRAP_NP, segment_not_present);
    set_intr_gate(X86_TRAP_GP, general_protection);
    set_intr_gate(X86_TRAP_SPURIOUS, spurious_interrupt_bug);
    set_intr_gate(X86_TRAP_MF, coprocessor_error);
    set_intr_gate(X86_TRAP_AC, alignment_check);

#ifdef CONFIG_IA32_EMULATION
    set_system_intr_gate(IA32_SYSCALL_VECTOR, ia32_syscall);
    set_bit(IA32_SYSCALL_VECTOR, used_vectors);
#endif

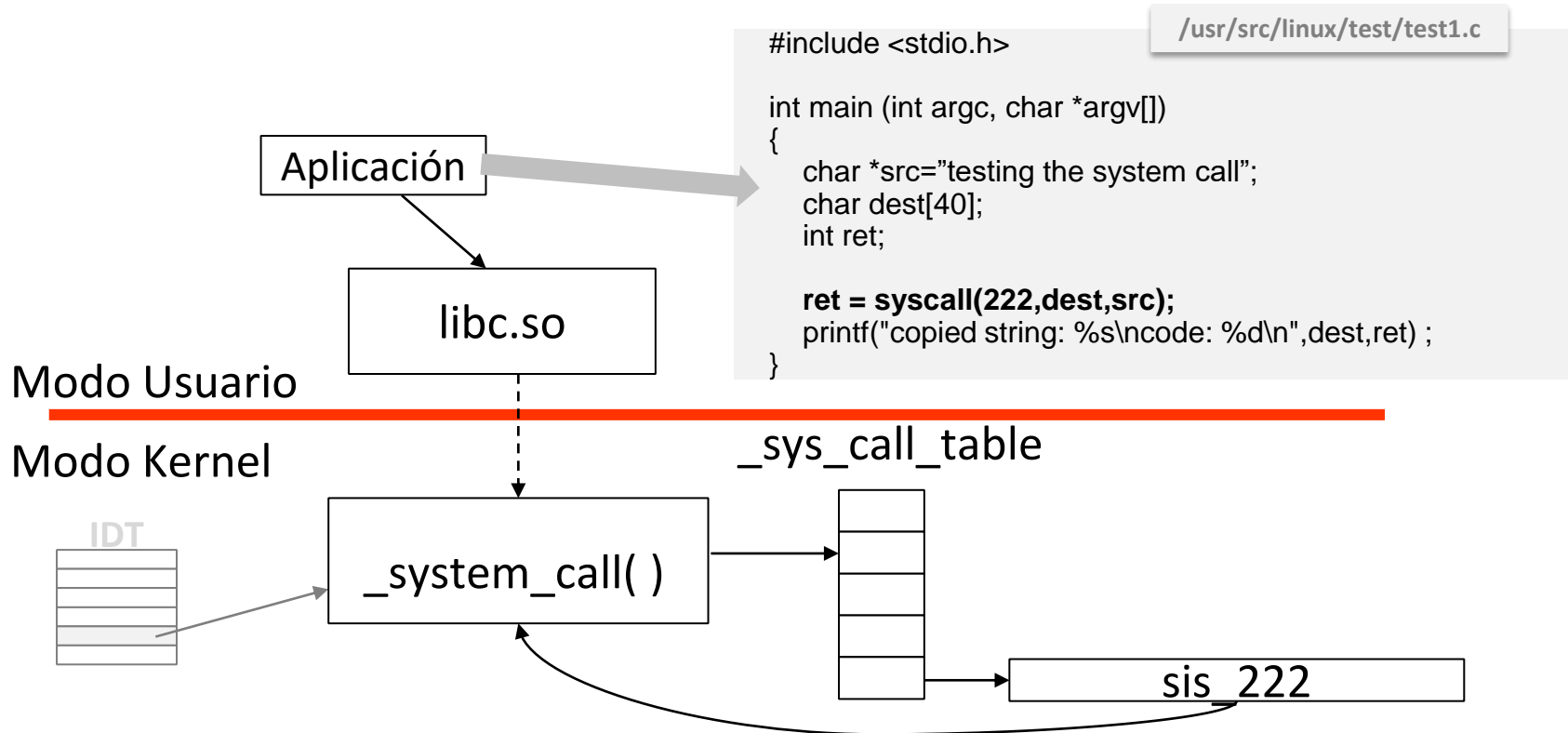
#ifdef CONFIG_X86_32
    set_system_trap_gate(SYSCALL_VECTOR, &system_call);
    set_bit(SYSCALL_VECTOR, used_vectors);
#endif
    ...
}
```

# Llamadas al sistema

## tratamiento en Linux (2/7)

28

Alejandro Calderón Mateos

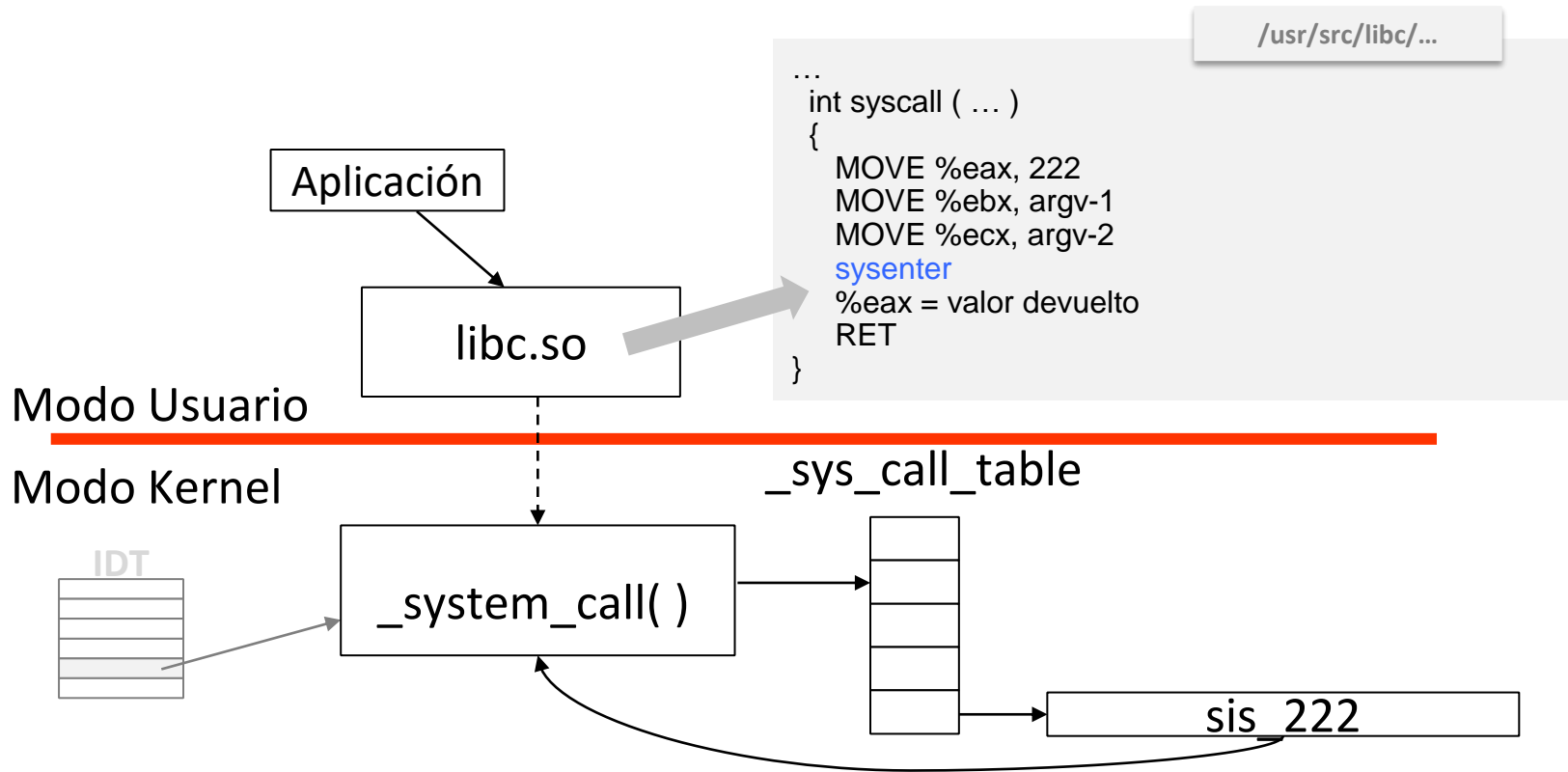


# Llamadas al sistema tratamiento en Linux (3/7)

- Cada servicio del SO se corresponde con una función (API el cto. de todas).
- Dicha función encapsula invocación al servicio: parámetros, trap, retornar...

29

Alejandro Calderón Mateos

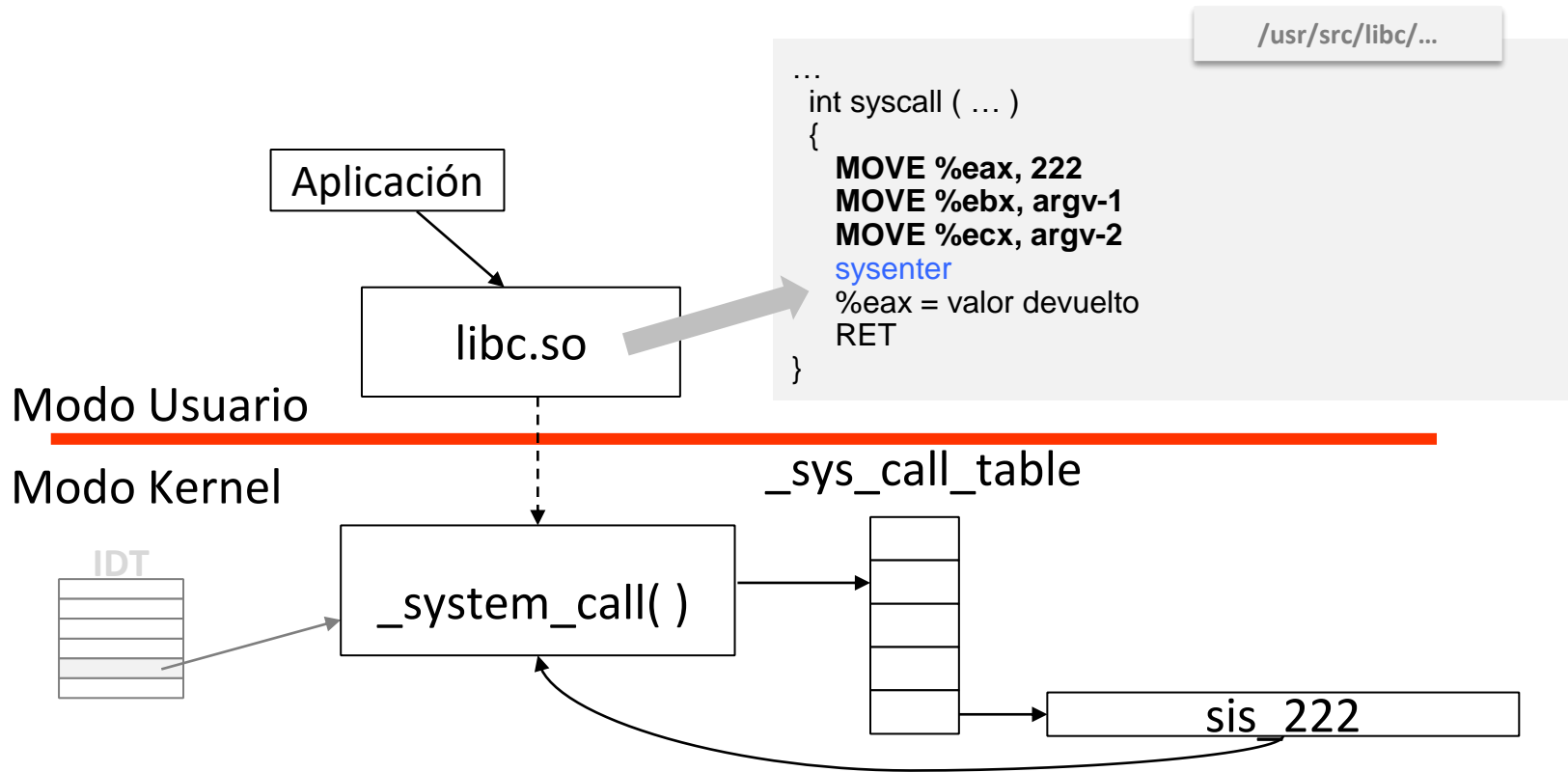


# Llamadas al sistema tratamiento en Linux (3/7)

- Paso de parámetros por registro, pila o zona de memoria pasada por registro.
- Parámetro 1: identificador de servicio

30

Alejandro Calderón Mateos

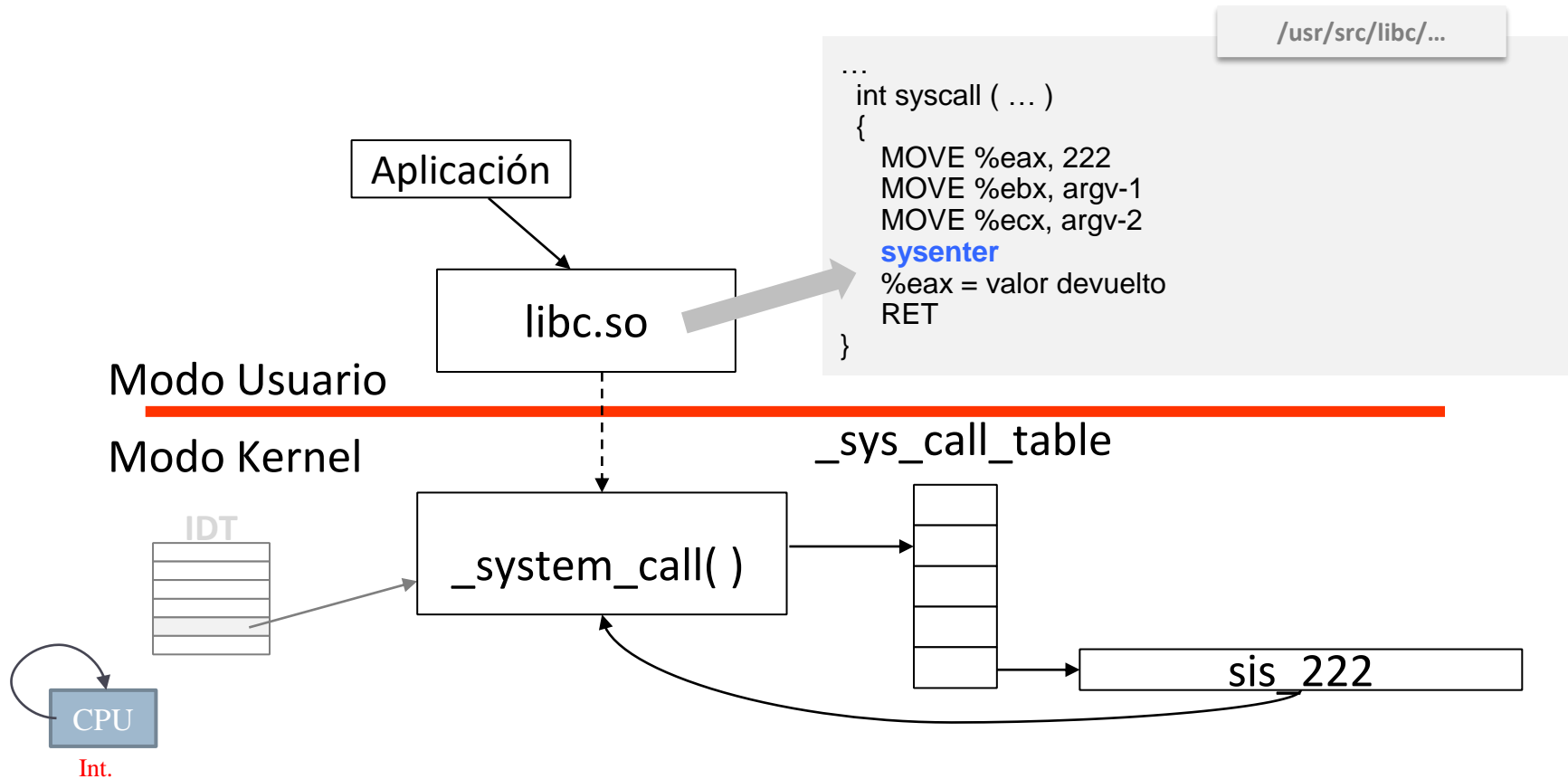


# Llamadas al sistema tratamiento en Linux (3/7)

- El trap (sysenter en CPU x86) es una instrucción que genera un evento con tratamiento similar a interrupción hardware.

31

Alejandro Calderón Mateos

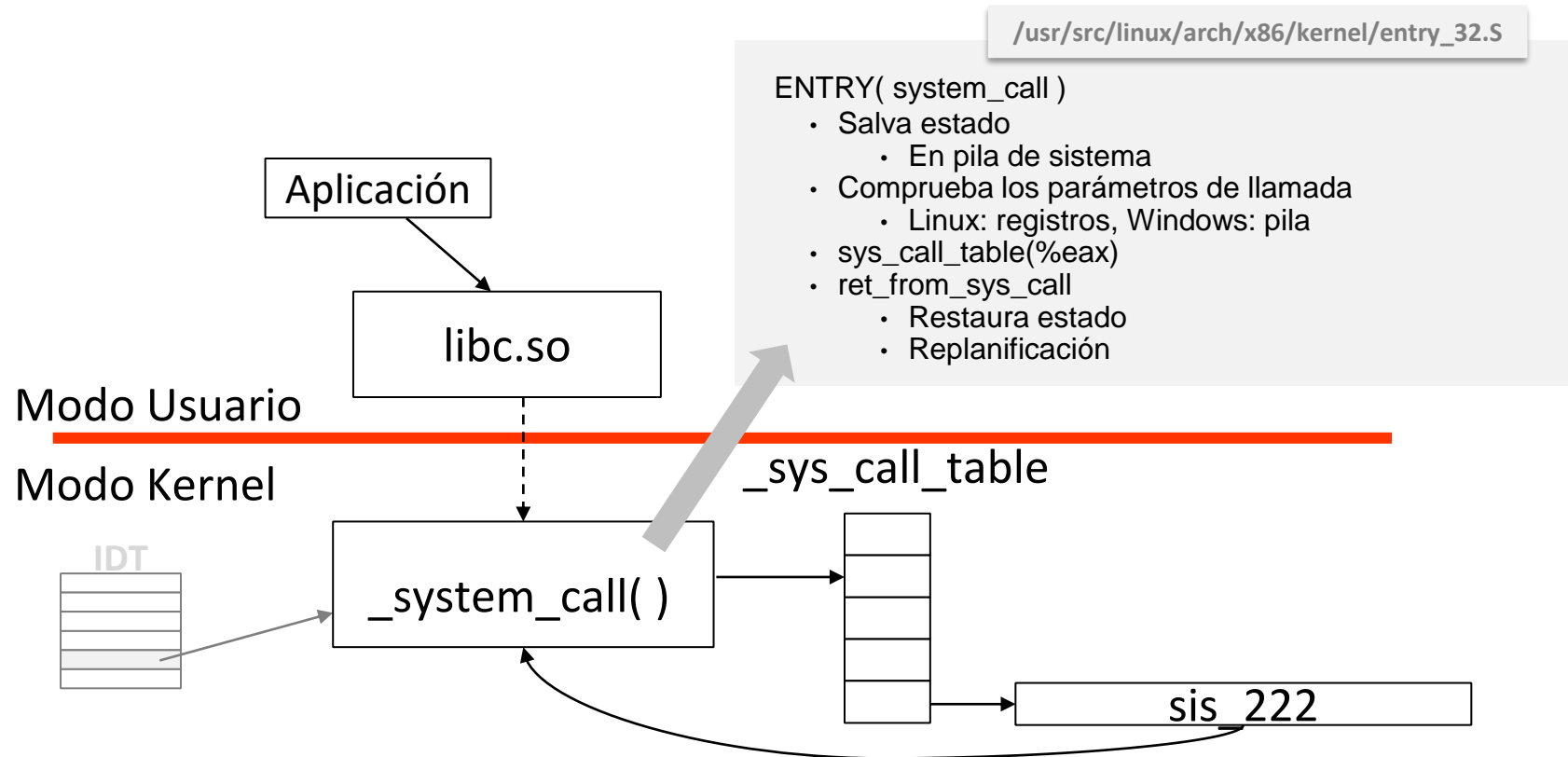


# Llamadas al sistema tratamiento en Linux (4/7)

- Comprueba parámetros, determina función en SO a partir del identificador (indexar en `_sys_call_table`) e invoca.

32

Alejandro Calderón Mateos



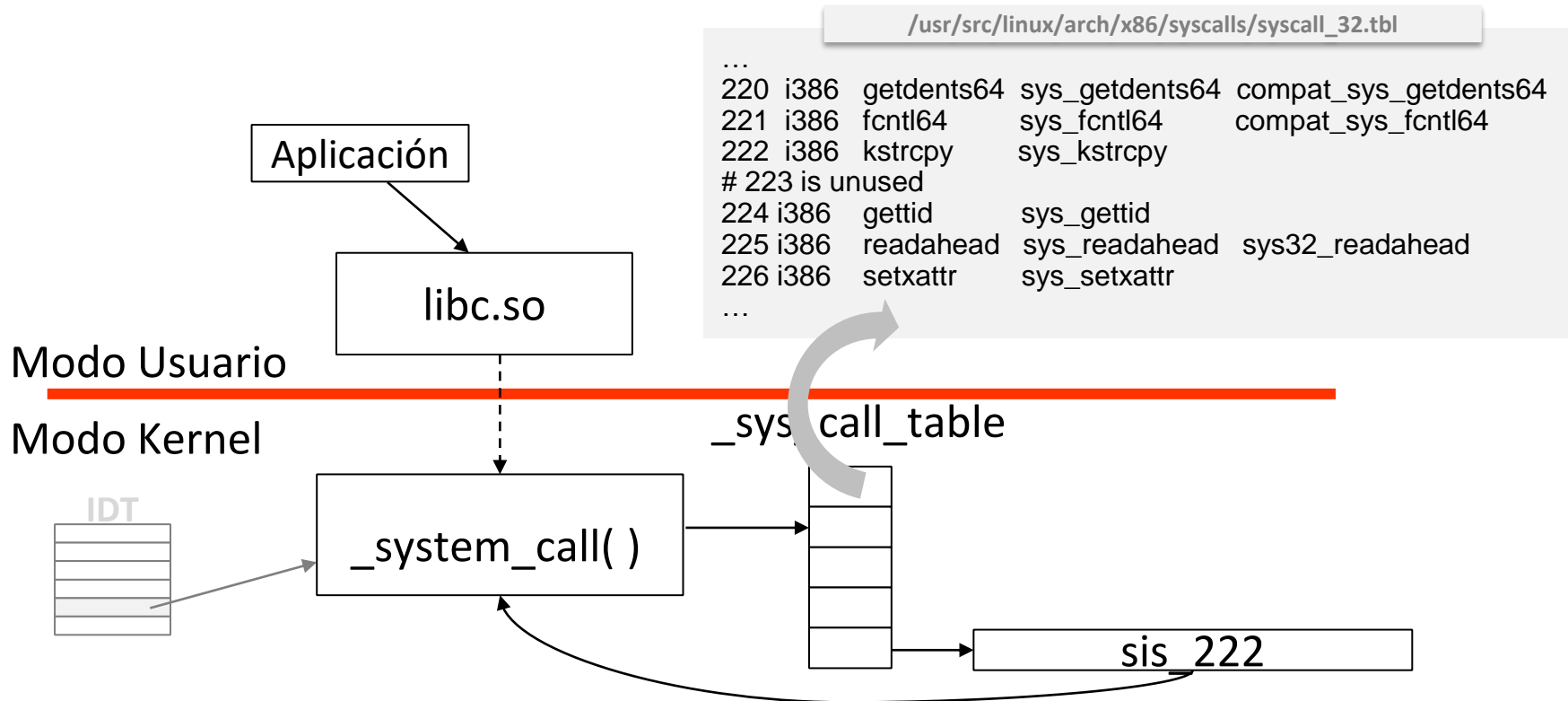


# Llamadas al sistema

## tratamiento en Linux (5/7)

33

Alejandro Calderón Mateos

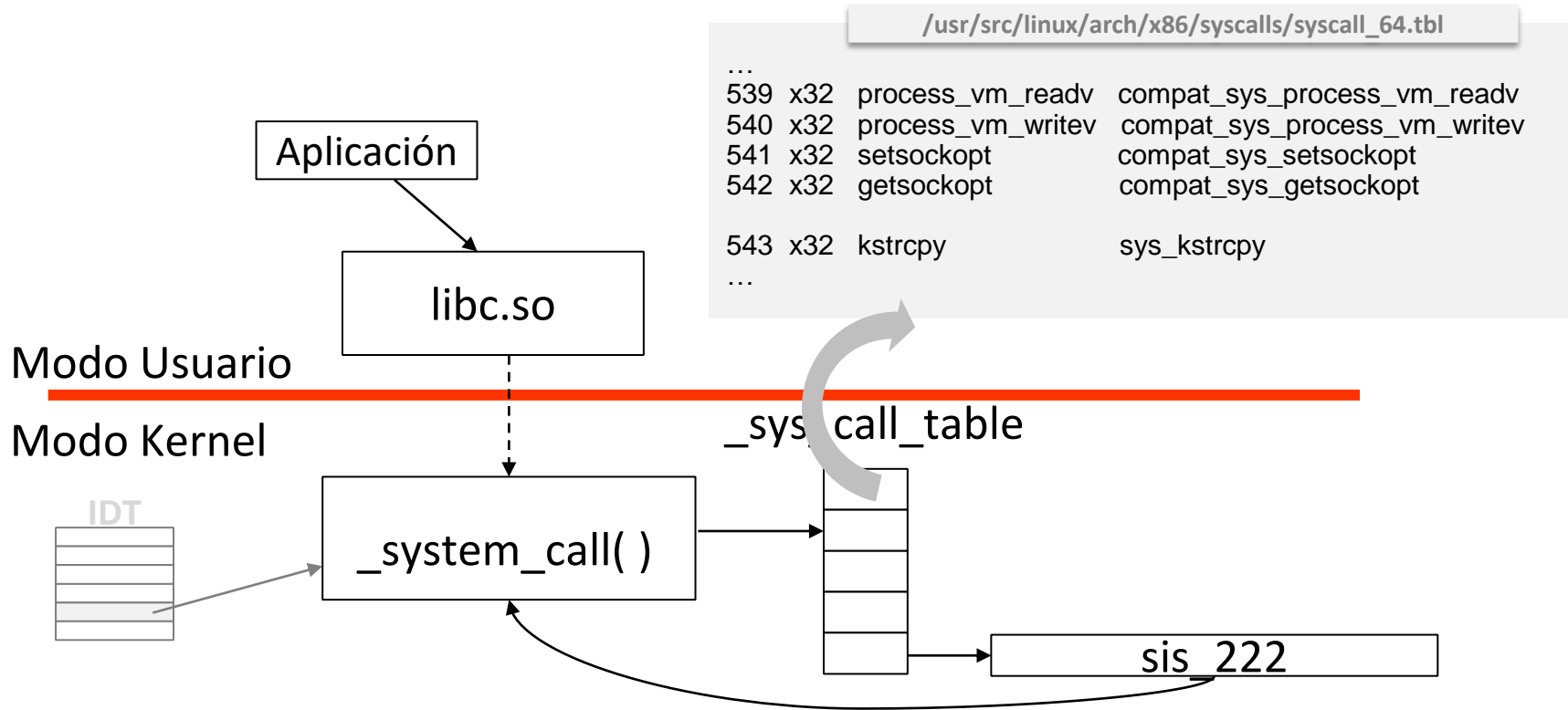


# Llamadas al sistema

## tratamiento en Linux (6/7)

34

Alejandro Calderón Mateos

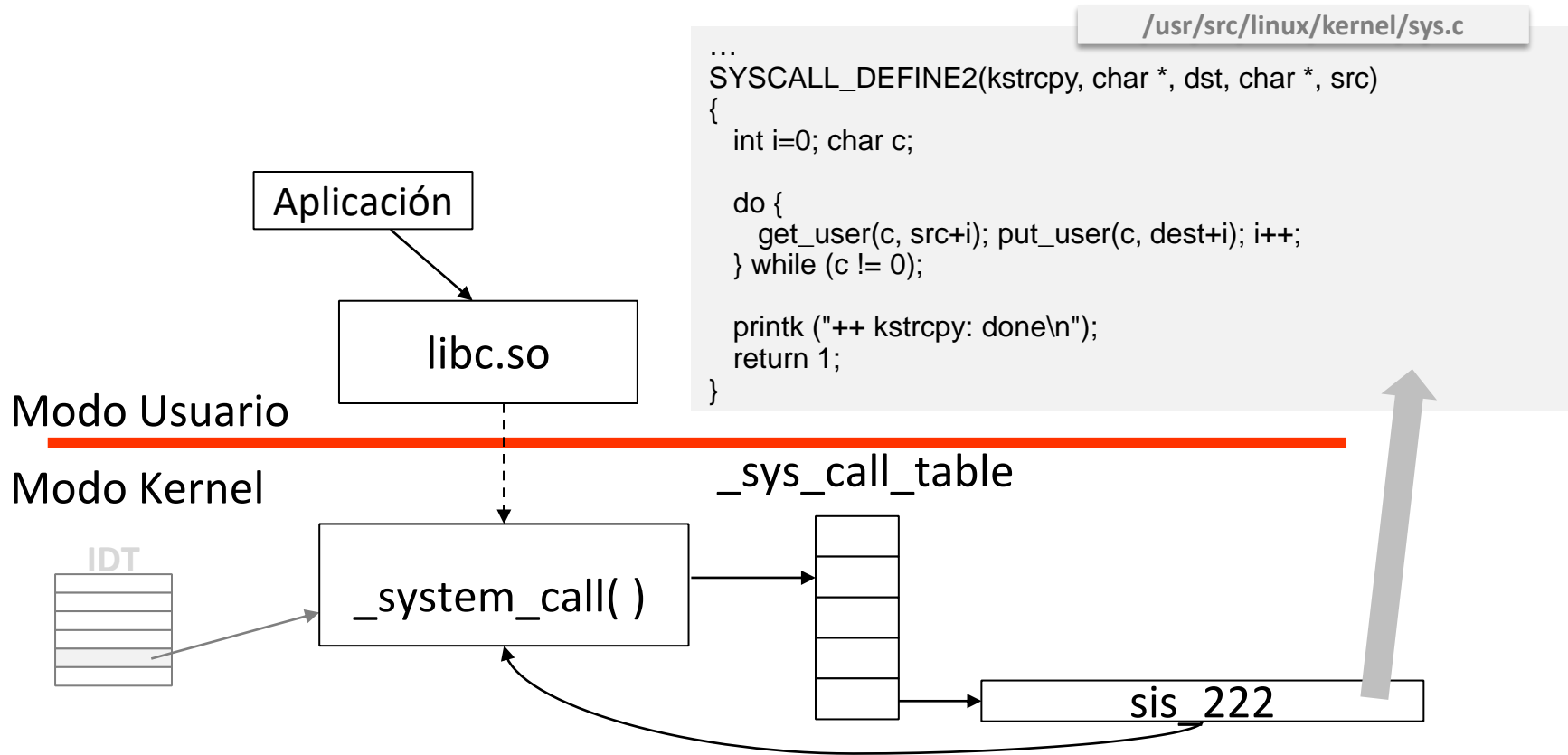


# Llamadas al sistema

## tratamiento en Linux (7/7)

35

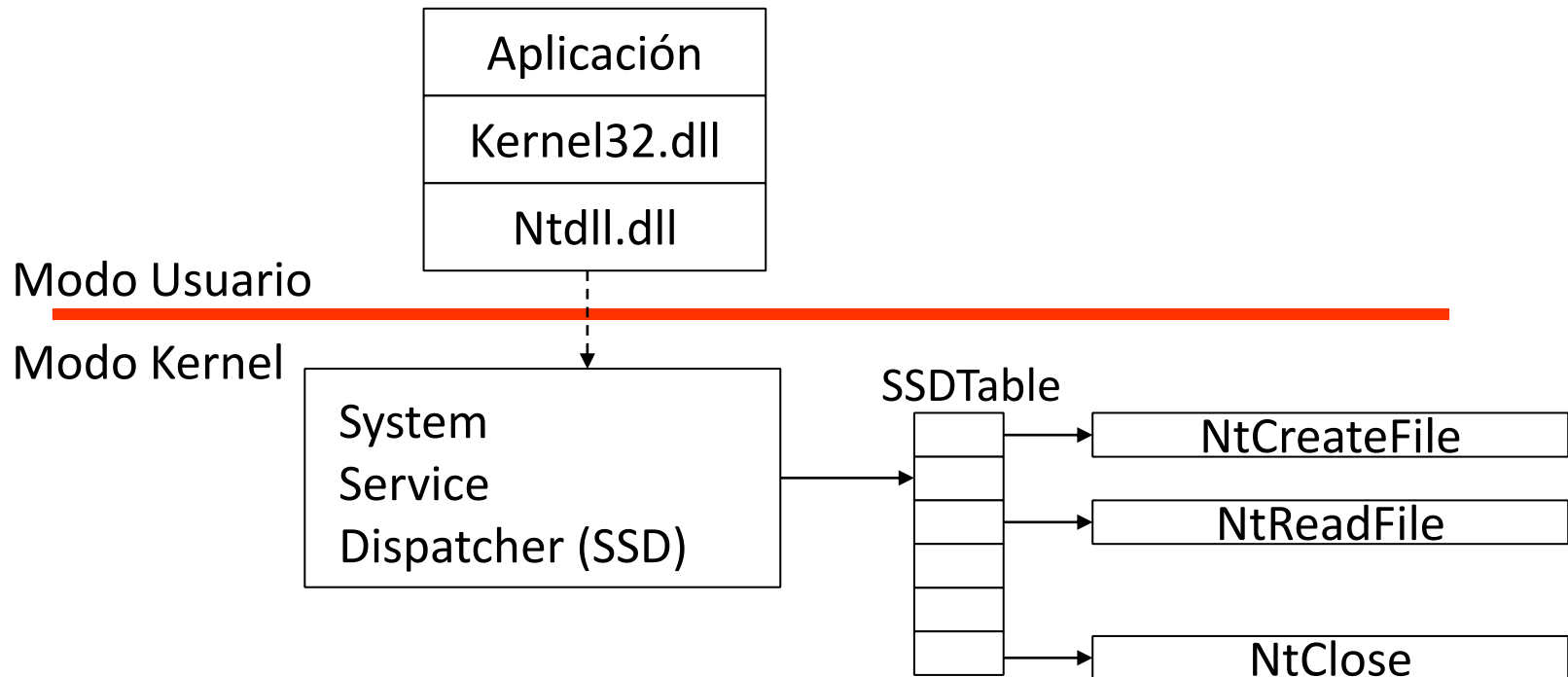
Alejandro Calderón Mateos



# Llamadas al sistema tratamiento en Windows

36

Sistemas operativos: una visión aplicada



# Interfaz del programador

- El conjunto de funciones que ofrecen los servicios del SO (encapsulando las llamadas) es la interfaz del programador.
  - ▣ Esta interfaz ofrece la visión que como máquina extendida tiene el usuario del sistema operativo
  - ▣ Mejor usar especificaciones de interfaces estándares.
- Cada sistema operativo puede ofrecer una o varias interfaces:
  - ▣ Linux: POSIX
  - ▣ Windows: Win32, POSIX

# Estándar POSIX

- ❑ Interfaz estándar de sistemas operativos de IEEE.
- ❑ **Objetivo:** portabilidad de las aplicaciones entre diferentes plataformas y sistemas operativos.
- ❑ **NO** es una implementación. Sólo define una interfaz
- ❑ Diferentes estándares
  - ❑ 1003.1 Servicios básicos del SO
  - ❑ 1003.1a Extensiones a los servicios básicos
  - ❑ 1003.1b Extensiones de tiempo real
  - ❑ 1003.1c Extensiones de procesos ligeros
  - ❑ 1003.2 Shell y utilidades
  - ❑ 1003.2b Utilidades adicionales

# Características de POSIX

- Nombres de funciones cortos y en letras minúsculas:
  - fork
  - read
  - close
- Las funciones normalmente devuelve 0 en caso de éxito o  $-1$  en caso de error.
  - Variable errno.
- Recursos gestionados por el sistema operativo se referencian mediante descriptores (números enteros)

# UNIX 03

- Single Unix Specification (SUS)
  - ▣ V1 (UNIX 95), V2 (UNIX 98), V3 (UNIX 03) y V4 (UNIX V7)
- Es una evolución que engloba a POSIX y otros estándares (X/Open XPG4, ISO C).
  - ▣ Incluye no solamente la interfaz de programación, sino también otros aspectos:
    - Servicios ofrecidos.
    - Intérprete de mandatos.
    - Utilidades disponibles.
- Ejemplo de UNIX 03: AIX, EulerOS, HP-UX, macOS



# Contenidos

41



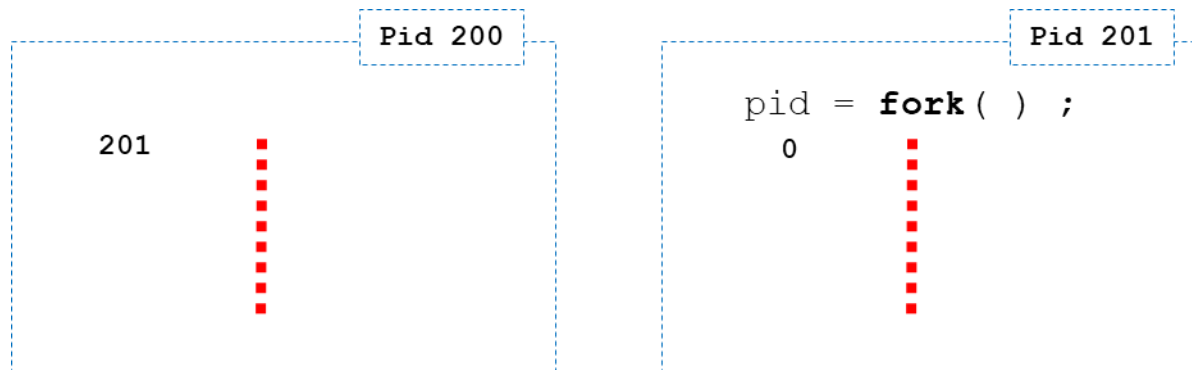
- Introducción a llamadas al sistema
- Mecanismo de llamada al sistema
- Llamadas para servicios de:
  - ▣ Gestión de procesos
  - ▣ Gestión de ficheros y directorios

# Gestión de procesos

- **Entendiendo el fork, exec, exit y wait**
- fork+exec+exit simple
- fork+exec+exit múltiple

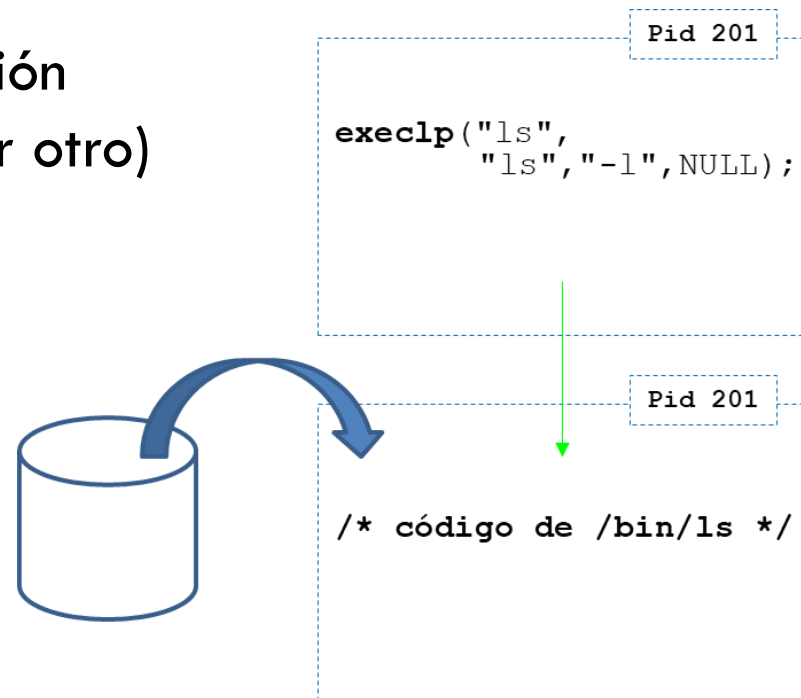
# Fork

- Crea un “clon” de un proceso:
  - Iguales **salvo** pequeñas diferencias:  
al padre se le devuelve el PID del hijo, y a el hijo cero.

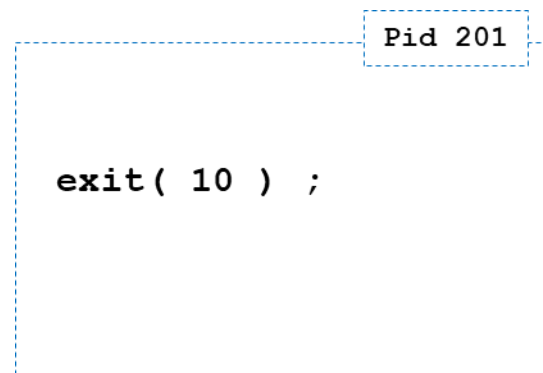


## □ Cambia la imagen de un proceso:

- Si todo va bien  
no se vuelve de esta función  
(el código se sustituye por otro)



- Finalizar la ejecución de un proceso
  - ▣ El parámetro es un valor entero que se suele usar como código de diagnóstico: si se ha ejecutado todo bien, ha habido algún problema leve, algún error grave, etc.



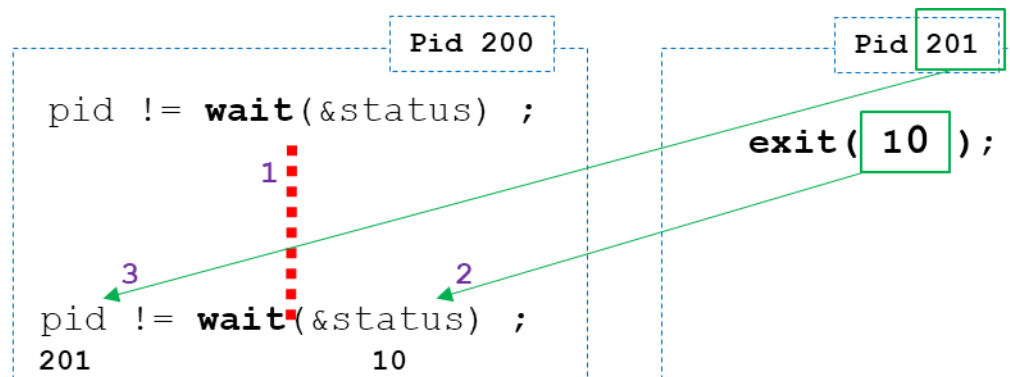
# Wait

49

Alejandro Calderón Mateos 

## □ Tiene tres efectos:

1. Bloquea la ejecución del padre hasta que alguno de sus hijos termine su ejecución.
2. Guarda en su parámetro el valor devuelto por el hijo.
3. Devuelve el pid del hijo que ha terminado.



# Gestión de procesos

- Entendiendo el fork, exec, exit y wait
- **fork+exec+exit simple**
- fork+exec+exit múltiple

# Repaso

## fork() + exec()

51

Alejandro Calderón Mateos 

```
/* ejecutar el mandato ls -l */
#include <sys/types.h>
#include <stdio.h>

main() {
    pid_t pid;
    int status;

    pid = fork();
    if (pid == 0)
    {
        execvp("ls", "ls", "-l", NULL);
        exit(-1);
    }
    else
    {
        while (pid != wait(&status));
    }

    exit(0);
}
```



# Repaso

## fork() + exec()

52

Alejandro Calderón Mateos 

```
/* ejecutar el mandato ls -l */
#include <sys/types.h>
#include <stdio.h>

main() {
    pid_t pid;
    int status;

    pid = fork();
    if (pid == 0)
    {
        execlp("ls", "ls", "-l", NULL);
        exit(-1);
    }
    else
    {
        while (pid != wait(&status));
    }

    exit(0);
}
```

# Repaso

## fork() + exec()

53

Alejandro Calderón Mateos 

```
/* ejecutar el mandato ls -l */
#include <sys/types.h>
#include <stdio.h>

main() {
    pid_t pid;
    int status;

    pid = fork();
    if (pid == 0)
    {
        execvp("ls", "ls", "-l", NULL);
        exit(-1);
    }
    else
    {
        while (pid != wait(&status));
    }

    exit(0);
}
```

```
/* ejecutar el mandato ls -l */
#include <sys/types.h>
#include <stdio.h>

main() {
    pid_t pid;
    int status;

    pid = fork();
    if (pid == 0)
    {
        execvp("ls", "ls", "-l", NULL);
        exit(-1);
    }
    else
    {
        while (pid != wait(&status));
    }

    exit(0);
}
```

# Repaso

## fork() + exec()

54


Alejandro Calderón Mateos 

```
/* ejecutar el mandato ls -l */
#include <sys/types.h>
#include <stdio.h>

main() {
    pid_t pid;
    int status;

    pid = fork();
    if (pid == 0)
    {
        execlp("ls", "ls", "-l", NULL);
        exit(1);
    }
    else
    {
        while (pid != wait(&status));
    }

    exit(0);
}
```



```
/* ejecutar el mandato ls -l */
#include <sys/types.h>
#include <stdio.h>

main() {
    pid_t pid;
    int status;

    pid = fork();
    if (pid == 0)
    {
        execlp("ls", "ls", "-l", NULL);
        exit(1);
    }
    else
    {
        while (pid != wait(&status));
    }

    exit(0);
}
```



# Repaso


## fork() + exec()

55

Alejandro Calderón Mateos 

```
/* ejecutar el mandato ls -l */
#include <sys/types.h>
#include <stdio.h>


main() {
    pid_t pid;
    int status;

    pid = fork();
    if (pid == 0)
    {
        execvp("ls", "ls", "-l", NULL);
        exit(-1);
    }
    else
    {
         while (pid != wait(&status));
    }

    exit(0);
}
```

```
/* ejecutar el mandato ls -l */
#include <sys/types.h>
#include <stdio.h>

main() {
    pid_t pid;
    int status;

    pid = fork();
    if (pid == 0)
    {
         execvp("ls", "ls", "-l", NULL);
        exit(-1);
    }
    else
    {
        while (pid != wait(&status));
    }

    exit(0);
}
```

# Repaso


## fork() + exec()

56

Alejandro Calderón Mateos 

```
/* ejecutar el mandato ls -l */
#include <sys/types.h>
#include <stdio.h>


main() {
    pid_t pid;
    int status;

    pid = fork();
    if (pid == 0)
    {
        execvp("ls", "ls", "-l", NULL);
        exit(-1);
    }
    else
    {
 while (pid != wait(&status));
    }

    exit(0);
}
```

```
/* ejecutar el mandato ls -l */
#include <sys/types.h>
#include <stdio.h>

main() {
    pid_t pid;
    int status;

    pid = fork();
    if (pid == 0)
    {
 execvp("ls", "ls", "-l", NULL);
        exit(-1);
    }
    else
    {
        while (pid != wait(&status));
    }

    exit(0);
}
```

# Repaso


## fork() + exec()

57

Alejandro Calderón Mateos 

```
/* ejecutar el mandato ls -l */
#include <sys/types.h>
#include <stdio.h>


main() {
    pid_t pid;
    int status;

    pid = fork();
    if (pid == 0)
    {
        execvp("ls", "ls", "-l", NULL);
        exit(-1);
    }
    else
    {
 while (pid != wait(&status));
    }

    exit(0);
}
```

```
/* ejecutar el mandato ls -l */
#include <sys/types.h>
#include <stdio.h>

main() {
    pid_t pid;
    int status;

    pid = fork();
    if (pid == 0)
    {
 execvp("ls", "ls", "-l", NULL);
        exit(-1);
    }
    else
    {
        while (pid != wait(&status));
    }

    exit(0);
}
```

# Repaso


## fork() + exec()

58

Alejandro Calderón Mateos 

```
/* ejecutar el mandato ls -l */
#include <sys/types.h>
#include <stdio.h>

main() {
    pid_t pid;
    int status;

    pid = fork();
    if (pid == 0)
    {
        execvp("ls", "ls", "-l", NULL);
        exit(-1);
    }
    else
    {
         while (pid != wait(&status));
    }

    exit(0);
}
```

```
/* código del mandato ls */
#include <sys/types.h>
#include <stdio.h>

main() {

    /* código del ls */

    exit( 0 );
}
```

# Repaso

## wait() + exit()

59

Alejandro Calderón Mateos 

```
/* ejecutar el mandato ls -l */
#include <sys/types.h>
#include <stdio.h>

main() {
    pid_t pid;
    int status;

    pid = fork();
    if (pid == 0)
    {
        execvp("ls", "ls", "-l", NULL);
        exit(-1);
    }
    else
    {
        while (pid != wait(&status));
    }

    exit(0);
}
```

```
/* código del mandato ls */
#include <sys/types.h>
#include <stdio.h>

main() {

    /* código del ls */

    exit( 0 );
}
```



# Repaso

## wait() + exit()

60

Alejandro Calderón Mateos 

```
/* ejecutar el mandato ls -l */
#include <sys/types.h>
#include <stdio.h>

main() {
    pid_t pid;
    int status;

    pid = fork();
    if (pid == 0)
    {
        execvp("ls", "ls", "-l", NULL);
        exit(-1);
    }
    else
    {
        while (pid != wait(&status));
    }

    exit(0);
}
```

```
/* código del mandato ls */
#include <sys/types.h>
#include <stdio.h>

main() {

    /* código del ls */

    exit( 0 );
}
```

# Repaso

## wait() + exit()

61

Alejandro Calderón Mateos 

```
/* ejecutar el mandato ls -l */
#include <sys/types.h>
#include <stdio.h>

main() {
    pid_t pid;
    int status;

    pid = fork();
    if (pid == 0)
    {
        execlp("ls", "ls", "-l", NULL);
        exit(-1);
    }
    else
    {
        while (pid != wait(&status));
    }

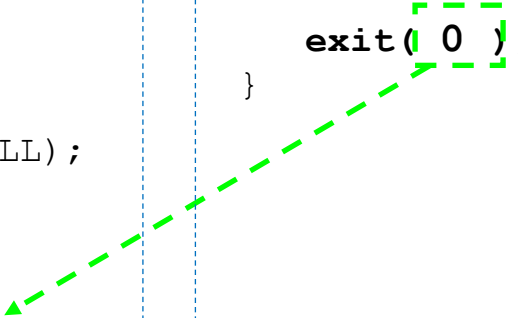
    exit(0);
}
```

```
/* código del mandato ls */
#include <sys/types.h>
#include <stdio.h>

main() {

    /* código del ls */

    exit( 0 );
}
```



# Repaso

## wait() + exit()

62

Alejandro Calderón Mateos 

```
/* ejecutar el mandato ls -l */
#include <sys/types.h>
#include <stdio.h>

main() {
    pid_t pid;
    int status;

    pid = fork();
    if (pid == 0)
    {
        execlp("ls", "ls", "-l", NULL);
        exit(-1);
    }
    else
    {
        while (pid != wait(&status));
    }

    exit(0);
}
```

# Repaso

## wait() + exit()

63

Alejandro Calderón Mateos 

```
/* ejecutar el mandato ls -l */
#include <sys/types.h>
#include <stdio.h>

main() {
    pid_t pid;
    int status;

    pid = fork();
    if (pid == 0)
    {
        execvp("ls", "ls", "-l", NULL);
        exit(-1);
    }
    else
    {
        while (pid != wait(&status));
    }

    exit(0);
}
```

# Repaso

`wait()` + `exit()`

# Gestión de procesos

65

Alejandro Calderón Mateos 

- Entendiendo el fork, exec, exit y wait
- fork+exec+exit simple
- **fork+exec+exit múltiple**

# Repaso

## múltiples procesos (bloqueante)

66

Alejandro Calderón Mateos 

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

main() {
    pid_t pid;
    int status;

    pid = fork();
    if (pid != 0) {
        while (pid != wait(&status));
    }
    else
    {
        /* hacer algo */
        exit(3);
    }

    pid = fork();
    if (pid != 0) {
        while (pid != wait(&status));
    }
    else
    {
        /* hacer algo */
        exit(3);
    }
}
```

# Repaso

## múltiples procesos (bloqueante)

67



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

main() {
    pid_t pid;
    int status;

    pid = fork();
    if (pid != 0) {
        while (pid != wait(&status));
    }
    else
    {
        /* hacer algo */
        exit(3);
    }

    pid = fork();
    if (pid != 0) {
        while (pid != wait(&status));
    }
    else
    {
        /* hacer algo */
        exit(3);
    }
}
```



# Repaso

## múltiples procesos (bloqueante)

68

Alejandro Calderón Mateos 

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

main() {
    pid_t pid;
    int status;

    pid = fork();
    if (pid != 0) {
        while (pid != wait(&status));
    }
    else
    {
        /* hacer algo */
        exit(3);
    }

    pid = fork();
    if (pid != 0) {
        while (pid != wait(&status));
    }
    else
    {
        /* hacer algo */
        exit(3);
    }
}
```

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

main() {
    pid_t pid;
    int status;

    pid = fork();
    if (pid == 0) {
        /* hacer algo */
        exit(3);
    }
    else
    {
        while (pid != wait(&status));
    }

    pid = fork();
    if (pid == 0) {
        /* hacer algo */
        exit(3);
    }
    else
    {
        while (pid != wait(&status));
    }
}
```

# Repaso

## múltiples procesos (bloqueante)

69

Alejandro Calderón Mateos 

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

main() {
    pid_t pid;
    int status;

    pid = fork();
    if (pid != 0) {
        while (pid != wait(&status));
    }
    else
    {
        /* hacer algo */
        exit(3);
    }

    pid = fork();
    if (pid != 0) {
        while (pid != wait(&status));
    }
    else
    {
        /* hacer algo */
        exit(3);
    }
}
```

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

main() {
    pid_t pid;
    int status;

    pid = fork();
    if (pid == 0) {
        /* hacer algo */
        exit(3);
    }
    else
    {
        while (pid != wait(&status));
    }

    pid = fork();
    if (pid == 0) {
        /* hacer algo */
        exit(3);
    }
    else
    {
        while (pid != wait(&status));
    }
}
```

# Repaso

## múltiples procesos (bloqueante)

70

Alejandro Calderón Mateos 

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

main() {
    pid_t pid;
    int status;

    pid = fork();
    if (pid != 0) {
        while (pid != wait(&status));
    }
    else
    {
        /* hacer algo */
        exit(3);
    }

    pid = fork();
    if (pid != 0) {
        while (pid != wait(&status));
    }
    else
    {
        /* hacer algo */
        exit(3);
    }
}
```

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

main() {
    pid_t pid;
    int status;

    pid = fork();
    if (pid == 0) {
        /* hacer algo */
        exit(3);
    }
    else
    {
        while (pid != wait(&status));
    }

    pid = fork();
    if (pid == 0) {
        /* hacer algo */
        exit(3);
    }
    else
    {
        while (pid != wait(&status));
    }
}
```

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

main() {
    pid_t pid;
    int status;

    pid = fork();
    if (pid == 0) {
        /* hacer algo */
        exit(3);
    }
    else
    {
        while (pid != wait(&status));
    }

    pid = fork();
    if (pid == 0) {
        /* hacer algo */
        exit(3);
    }
    else
    {
        while (pid != wait(&status));
    }
}
```

# Repaso

## múltiples procesos (bloqueante)

71

Alejandro Calderón Mateos 

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

main() {
    pid_t pid;
    int status;

    pid = fork();
    if (pid != 0) {
        while (pid != wait(&status));
    }
    else
    {
        /* hacer algo */
        exit(3);
    }

    pid = fork();
    if (pid != 0) {
        while (pid != wait(&status));
    }
    else
    {
        /* hacer algo */
        exit(3);
    }
}
```

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

main() {
    pid_t pid;
    int status;

    pid = fork();
    if (pid == 0) {
        /* hacer algo */
        exit(3);
    }
    else
    {
        while (pid != wait(&status));
    }

    pid = fork();
    if (pid == 0) {
        /* hacer algo */
        exit(3);
    }
    else
    {
        while (pid != wait(&status));
    }
}
```

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

main() {
    pid_t pid;
    int status;

    pid = fork();
    if (pid == 0) {
        /* hacer algo */
        exit(3);
    }
    else
    {
        while (pid != wait(&status));
    }

    pid = fork();
    if (pid == 0) {
        /* hacer algo */
        exit(3);
    }
    else
    {
        while (pid != wait(&status));
    }
}
```

# Repaso

## múltiples procesos (bloqueante)

72

Alejandro Calderón Mateos 

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

main() {
    pid_t pid;
    int status;

    pid = fork();
    if (pid == 0) {
        /* hacer algo */
        exit(3);
    }

    pid = fork();
    if (pid == 0) {
        /* hacer algo */
        exit(3);
    }

    while (pid != wait(&status));
}
```

# Repaso

## múltiples procesos (bloqueante)

73

Alejandro Calderón Mateos 

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

main() {
    pid_t pid;
    int status;

    pid = fork();
    if (pid == 0) {
        /* hacer algo */
        exit(3);
    }

    pid = fork();
    if (pid == 0) {
        /* hacer algo */
        exit(3);
    }

    while (pid != wait(&status));
}
```

# Repaso

## múltiples procesos (bloqueante)

74

Alejandro Calderón Mateos 

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

```
main() {
    pid_t pid;
    int status;
```

```
    pid = fork();
    if (pid == 0) {
        /* hacer algo */
        exit(3);
    }
```

```
    pid = fork();
    if (pid == 0) {
        /* hacer algo */
        exit(3);
    }
```

```
    while (pid != wait(&status));
```

```
}
```

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

```
main() {
    pid_t pid;
    int status;
```

```
    pid = fork();
    if (pid == 0) {
        /* hacer algo */
        exit(3);
    }
```

```
    pid = fork();
    if (pid == 0) {
        /* hacer algo */
        exit(3);
    }
```

```
    while (pid != wait(&status));
```

```
}
```

# Repaso

## múltiples procesos (bloqueante)

75

Alejandro Calderón Mateos 

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

```
main() {
    pid_t pid;
    int status;

    pid = fork();
    if (pid == 0) {
        /* hacer algo */
        exit(3);
    }

    pid = fork();
    if (pid == 0) {
        /* hacer algo */
        exit(3);
    }

    while (pid != wait(&status));
}
```

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

```
main() {
    pid_t pid;
    int status;

    pid = fork();
    if (pid == 0) {
        /* hacer algo */
        exit(3);
    }

    pid = fork();
    if (pid == 0) {
        /* hacer algo */
        exit(3);
    }

    while (pid != wait(&status));
}
```

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

```
main() {
    pid_t pid;
    int status;

    pid = fork();
    if (pid == 0) {
        /* hacer algo */
        exit(3);
    }

    pid = fork();
    if (pid == 0) {
        /* hacer algo */
        exit(3);
    }

    while (pid != wait(&status));
}
```



# Repaso

## múltiples procesos (bloqueante)

76

Alejandro Calderón Mateos 

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

```
main() {
    pid_t pid;
    int status;
```

```
    pid = fork();
    if (pid == 0) {
        /* hacer algo */
        exit(3);
    }
```

```
    pid = fork();
    if (pid == 0) {
        /* hacer algo */
        exit(3);
    }
```

```
    while (pid != wait(&status));
```

```
}
```

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

```
main() {
    pid_t pid;
    int status;
```

```
    pid = fork();
    if (pid == 0) {
        /* hacer algo */
        exit(3);
    }
```

```
    pid = fork();
    if (pid == 0) {
        /* hacer algo */
        exit(3);
    }
```

```
    while (pid != wait(&status));
```

```
}
```

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

```
main() {
    pid_t pid;
    int status;
```

```
    pid = fork();
    if (pid == 0) {
        /* hacer algo */
        exit(3);
    }
```

```
    pid = fork();
    if (pid == 0) {
        /* hacer algo */
        exit(3);
    }
```

```
    while (pid != wait(&status));
```

```
}
```

# Repaso

## múltiples procesos (bloqueante)

77

Alejandro Calderón Mateos 

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

```
main() {
    pid_t pid;
    int status;
```

```
    pid = fork();
    if (pid == 0) {
        /* hacer algo */
        exit(3);
    }
```

```
    pid = fork();
    if (pid == 0) {
        /* hacer algo */
        exit(3);
    }
```

```
    while (pid != wait(&status));
```

```
}
```

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

```
main() {
    pid_t pid;
    int status;
```

```
    pid = fork();
    if (pid == 0) {
        /* hacer algo */
        exit(3);
    }
```

```
    pid = fork();
    if (pid == 0) {
        /* hacer algo */
        exit(3);
    }
```

```
    while (pid != wait(&status));
```

```
}
```

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

```
main() {
    pid_t pid;
    int status;
```

```
    pid = fork();
    if (pid == 0) {
        /* hacer algo */
        exit(3);
    }
```

```
    pid = fork();
    if (pid == 0) {
        /* hacer algo */
        exit(3);
    }
```

```
    while (pid != wait(&status));
```

```
}
```

# Repaso

## múltiples procesos (bloqueante)

78

Alejandro Calderón Mateos 

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

```
main() {
    pid_t pid;
    int status;
```

```
    pid = fork();
    if (pid == 0) {
        /* hacer algo */
        exit(3);
    }
```

```
    pid = fork();
    if (pid == 0) {
        /* hacer algo */
        exit(3);
    }
```

```
    while (pid != wait(&status));
```

```
}
```

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

```
main() {
    pid_t pid;
    int status;
```

```
    pid = fork();
    if (pid == 0) {
        /* hacer algo */
        exit(3);
    }
```

```
    pid = fork();
    if (pid == 0) {
        /* hacer algo */
        exit(3);
    }
```

```
    while (pid != wait(&status));
```

```
}
```

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

```
main() {
    pid_t pid;
    int status;
```

```
    pid = fork();
    if (pid == 0) {
        /* hacer algo */
        exit(3);
    }
```

```
    pid = fork();
    if (pid == 0) {
        /* hacer algo */
        exit(3);
    }
```

```
    while (pid != wait(&status));
```

```
}
```

# Repaso

## múltiples procesos (bloqueante)

79

Alejandro Calderón Mateos 

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

```
main() {
    pid_t pid;
    int status;
```

```
    pid = fork();
    if (pid == 0) {
        /* hacer algo */
        exit(3);
    }
```

```
    pid = fork();
    if (pid == 0) {
        /* hacer algo */
        exit(3);
    }
```

```
    while (pid != wait(&status));
```

```
}
```

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

```
main() {
    pid_t pid;
    int status;
```

```
    pid = fork();
    if (pid == 0) {
        /* hacer algo */
        exit(3);
    }
```

```
    pid = fork();
    if (pid == 0) {
        /* hacer algo */
        exit(3);
    }
```

```
    while (pid != wait(&status));
```

```
}
```

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

```
main() {
    pid_t pid;
    int status;
```

```
    pid = fork();
    if (pid == 0) {
        /* hacer algo */
        exit(3);
    }
```

```
    pid = fork();
    if (pid == 0) {
        /* hacer algo */
        exit(3);
    }
```

```
    while (pid != wait(&status));
```

```
}
```

# Repaso

## múltiples procesos (bloqueante)

80

Alejandro Calderón Mateos 

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

```
main() {
    pid_t pid;
    int status;
```

```
    pid = fork();
    if (pid == 0) {
        /* hacer algo */
        exit(3);
    }
```

```
    pid = fork();
    if (pid == 0) {
        /* hacer algo */
        exit(3);
    }
```

```
    while (pid != wait(&status));
```

```
}
```

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

```
main() {
    pid_t pid;
    int status;
```

```
    pid = fork();
    if (pid == 0) {
        /* hacer algo */
        exit(3);
    }
```

```
    pid = fork();
    if (pid == 0) {
        /* hacer algo */
        exit(3);
    }
```

```
    while (pid != wait(&status));
```

```
}
```

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

```
main() {
    pid_t pid;
    int status;
```

```
    pid = fork();
    if (pid == 0) {
        /* hacer algo */
        exit(3);
    }
```

```
    pid = fork();
    if (pid == 0) {
        /* hacer algo */
        exit(3);
    }
```

```
    while (pid != wait(&status));
```

```
}
```

# Repaso

si 2º hijo termina antes que el 1º => zombie (padre no espera por él)

81

Alejandro Calderón Mateos



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

```
main() {
    pid_t pid;
    int status;
```

```
    pid = fork();
    if (pid == 0) {
        /* hacer algo */
        exit(3);
    }
```

```
    pid = fork();
    if (pid == 0) {
        /* hacer algo */
        exit(3);
    }
```

```
    while (pid != wait(&status));
```

```
}
```

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

```
main() {
    pid_t pid;
    int status;
```

```
    pid = fork();
    if (pid == 0) {
        /* hacer algo */
        exit(3);
    }
```

```
    pid = fork();
    if (pid == 0) {
        /* hacer algo */
        exit(3);
    }
```

```
    while (pid != wait(&status));
```

```
}
```

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

```
main() {
    pid_t pid;
    int status;
```

```
    pid = fork();
    if (pid == 0) {
        /* hacer algo */
        exit(3);
    }
```

```
    pid = fork();
    if (pid == 0) {
        /* hacer algo */
        exit(3);
    }
```

```
    while (pid != wait(&status));
```

```
}
```

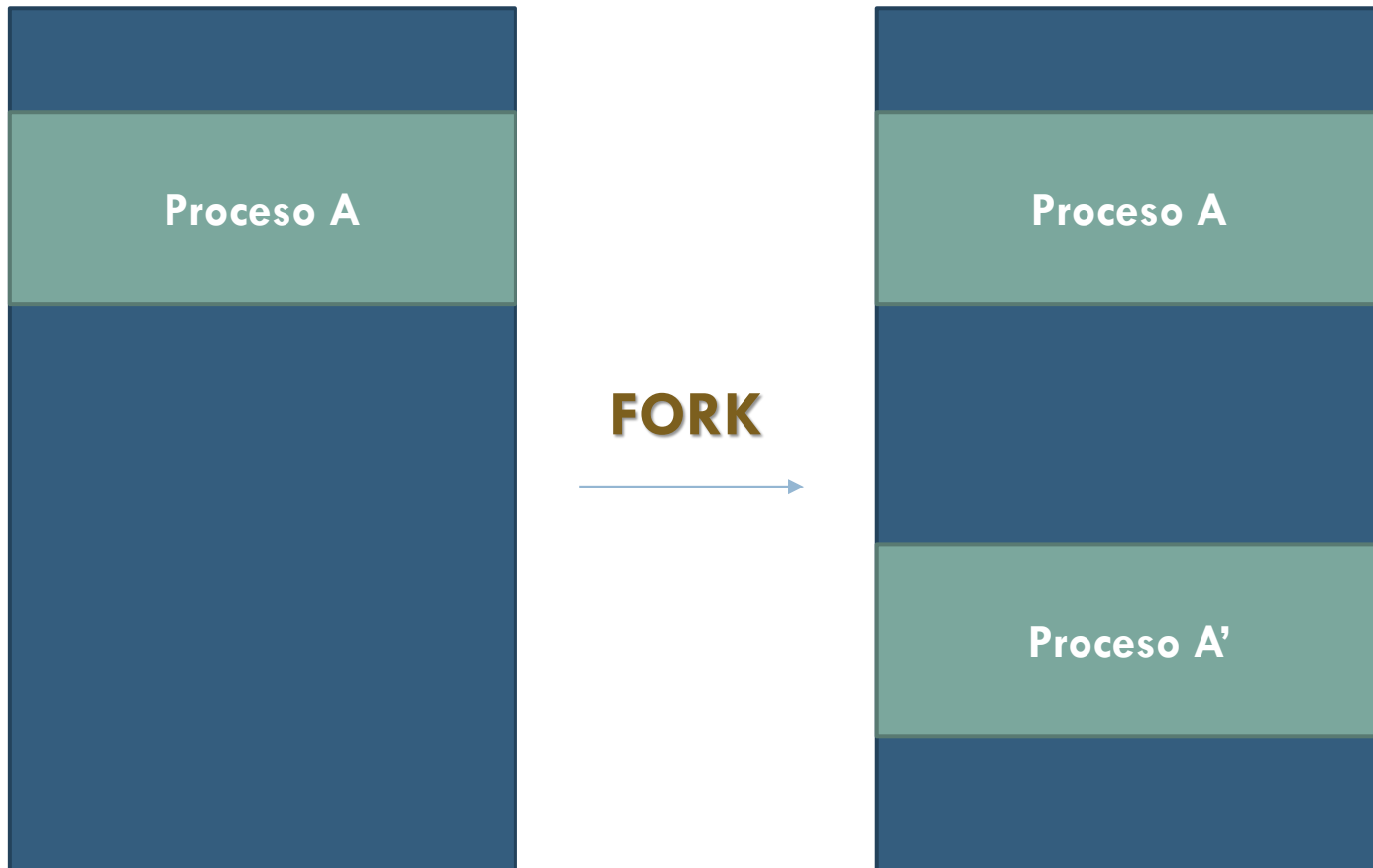
# Servicio fork

Servicio	<pre>#include &lt;unistd.h&gt;  pid_t fork(void);</pre>
Argumentos	
Devuelve	<ul style="list-style-type: none"><li>❑ -1 el caso de error.</li><li>❑ En el proceso padre: el identificador del proceso hijo.</li><li>❑ En el proceso hijo: 0</li></ul>
Descripción	<ul style="list-style-type: none"><li>❑ Duplica el proceso que invoca la llamada.</li><li>❑ Los procesos padre e hijo siguen ejecutando el mismo programa.</li><li>❑ El proceso hijo hereda los ficheros abiertos del proceso padre.<ul style="list-style-type: none"><li>❑ Se copian los descriptores de archivos abiertos.</li></ul></li><li>❑ Se desactivan las alarmas pendientes.</li></ul>

# Servicio fork

83

Sistemas operativos: una visión aplicada





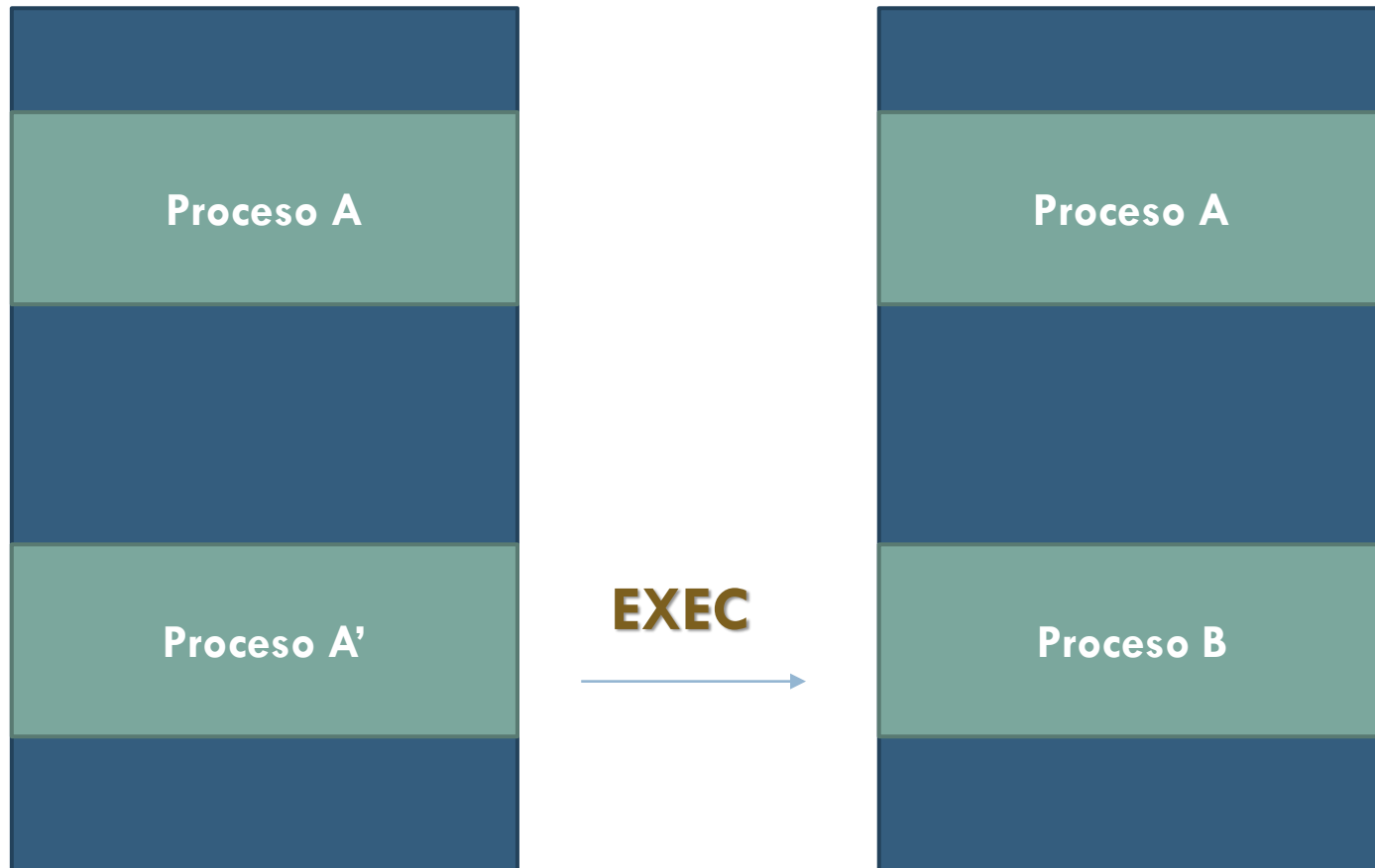
# Servicio exec

Servicio	<pre>#include &lt;unistd.h&gt; int <b>execl</b>(const char *path, const char *arg, ...); int <b>execv</b>(const char* path, char* const argv[]); int <b>execve</b>(const char* path, char* const argv[], char* const envp[]); int <b>execvp</b>(const char *file, char *const argv[]);</pre>
Argumentos	<ul style="list-style-type: none"><li>▣ path: Ruta al archivo ejecutable.</li><li>▣ file: Busca el archivo ejecutable en todos los directorios especificados por PATH</li></ul>
Devuelve	<ul style="list-style-type: none"><li>▣ Devuelve -1 en caso de error, <b>en caso contrario no retorna.</b></li></ul>
Descripción	<ul style="list-style-type: none"><li>▣ Cambia la imagen del proceso actual.</li><li>▣ El mismo proceso ejecuta otro programa.</li><li>▣ Los ficheros abiertos permanecen abiertos.</li><li>▣ Las señales con la acción por defecto seguirán por defecto, las señales con manejador tomarán la acción por defecto</li></ul>

# Servicio exec

85

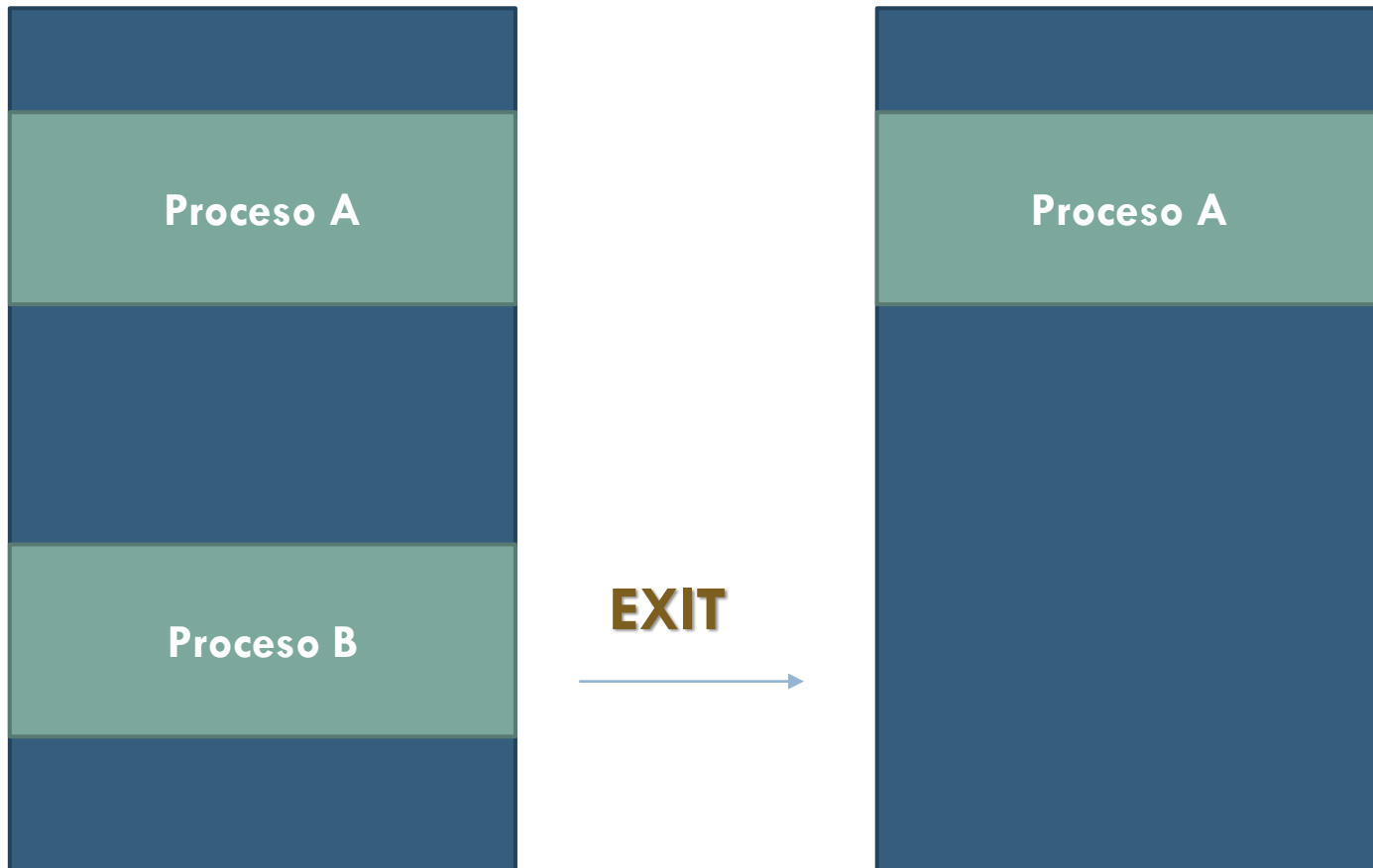
Sistemas operativos: una visión aplicada



# Servicio exit

Servicio	<pre>#include &lt;unistd.h&gt;  void <b>exit</b>(status);</pre>
Argumentos	<ul style="list-style-type: none"><li>❑ status: valor que el padre recupera en la llamada wait()</li></ul>
Devuelve	
Descripción	<ul style="list-style-type: none"><li>❑ Finaliza la ejecución del proceso.</li><li>❑ Se cierran todos los descriptores de ficheros abiertos.</li><li>❑ Se liberan todos los recursos del proceso.</li><li>❑ Se libera el BCP del proceso.</li></ul>

# Servicio exit

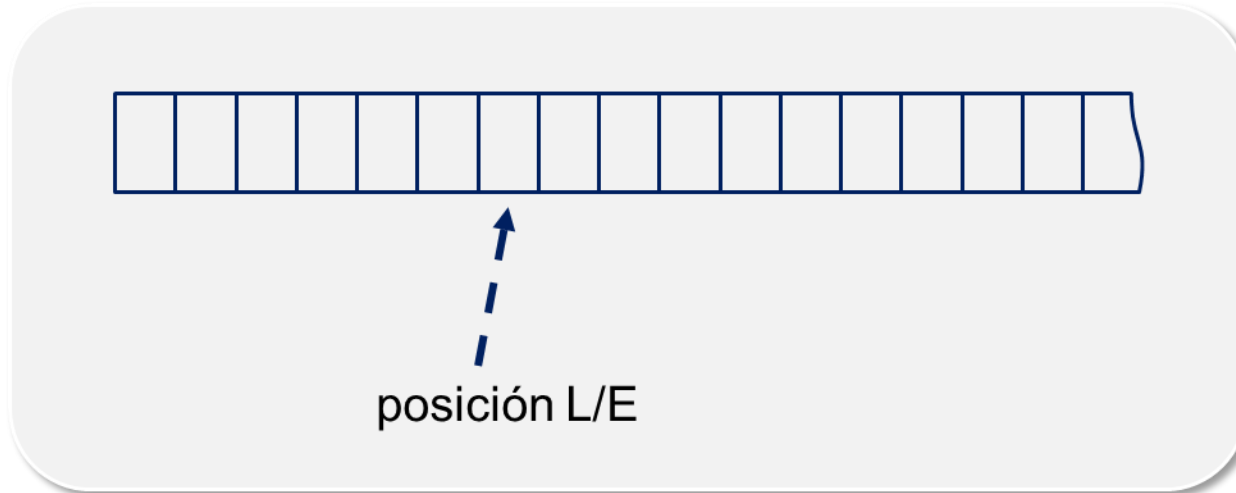


# Contenidos

- Introducción a llamadas al sistema
- Mecanismo de llamada al sistema
- Llamadas para servicios de:
  - Gestión de procesos
  - Gestión de ficheros y directorios

# Fichero o archivo

- Conjunto de información relacionada que ha sido definida por su creador/a.
- Habitualmente el contenido es representado por una secuencia o tira de bytes (visión lógica):



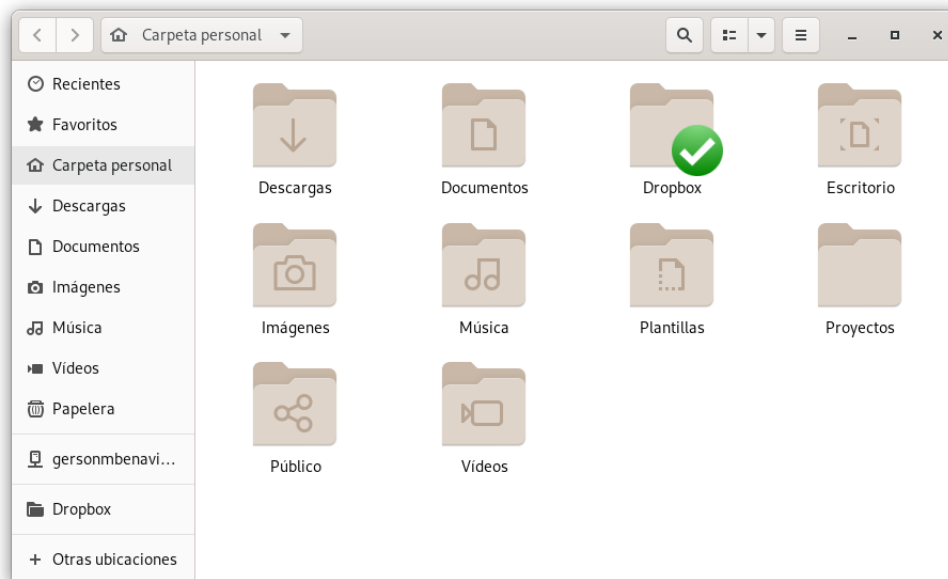
# Directorio (carpetas)

90

[https://es.wikipedia.org/wiki/GNOME\\_Archivos#/media/Archivo:GNOME\\_Archivos\\_3\\_36\\_3.png](https://es.wikipedia.org/wiki/GNOME_Archivos#/media/Archivo:GNOME_Archivos_3_36_3.png)

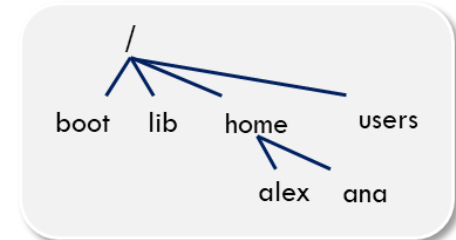
Alejandro Calderón Mateos 

- Estructura de datos que permite agrupar un conjunto de ficheros según el criterio del usuario/a.



# Nombre de ficheros y directorios

- **Nombres jerárquicos** para la identificación:
  - ▣ Lista de nombres hasta llegar al directorio/fichero.
  - ▣ Los nombres se separan con un carácter especial:
    - / en LINUX y \ en Windows



- **Nombres especiales de directorio:**
  - ▣ . Directorio actual o directorio de trabajo (Ej.: `cp /alex/correo.txt .`)
  - ▣ .. Directorio padre o directorio anterior (Ej.: `ls ..`)
  - ▣ ~ Directorio base del usuario+a en UNIX (Ej.: `ls -las ~` ; `ls -las $HOME`)
  - ▣ / Directorio raíz en UNIX (Ej.: `ls -las /`)
- **Dos tipos de nombrado usado:**
  - ▣ **Nombre absoluto** o completo (empieza por el directorio raíz)
    - `/usr/include/stdio.h` (linux)
    - `c:\usr\include\stdio.h` (windows)
  - ▣ **Nombre relativo** (es relativo al directorio actual, no empieza por raíz)
    - `stdio.h` **asumiendo que** `/usr/include` es el directorio actual.
    - `../include/stdio.h`



# Atributos típicos de un fichero/directorio

- **Nombre:** identificador para los usuarios del fichero/directorio (entrada).
- **Tipo:** tipo de entrada (para los sistemas que lo necesiten)
  - Ej.: extensión (.exe, .pdf, etc.)
  - **Tipos de fichero: normales, directorios, especiales.**
- **Localización:** identificador que ayuda a la localización de los bloques del dispositivo que pertenecen a la entrada.
- **Tamaño:** tamaño actual de la entrada.
- **Protección:** control de qué usuario/a puede leer, escribir, etc.
- **Día y hora:** instante de tiempo de último acceso, de creación, etc. que permite la monitorización del uso de la entrada.
- **Identificación** de usuario/a: identificador del creador/a, dueño/a, etc.

# Servicios POSIX para ficheros

## operaciones genéricas para ficheros

- `creat (...)` → **crear**: crea un fichero (dado nombre y atributos) y abre sesión.
- `open (...)` → **abrir**: abre una sesión con un fichero a partir de su nombre.
- `close (...)` → **cerrar**: cierra sesión de trabajo con un fichero abierto.
- `read (...)` → **leer**: lee datos de un fichero abierto a un almacén en memoria.
- `write (...)` → **escribir**: escribe a un fichero abierto desde un almacén en memoria.
- `lseek (...)` → **posicionar**: Mueve el apuntador usado para acceder al fichero, afectando a operaciones posteriores.
- `unlink (...)` → **borrar**: Borra un fichero a partir de su nombre.
  
- `fcntl (...)` → **control**: Permite manipular los atributos de un fichero.
- `dup (...)`
- `ftruncate (...)`
- `stat (...)`
- `utime (...)`

# Abstracción de fichero

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

int main ( int argc, char *argv[] )
{
    int fd1 ;
    char str1[10] ;
    int nb ;

    fd1 = open ("/tmp/txt1",
                O_CREAT|O_RDWR, S_IRWXU);
    if (-1 == fd1) {
        perror("open:");
        exit(-1);
    }

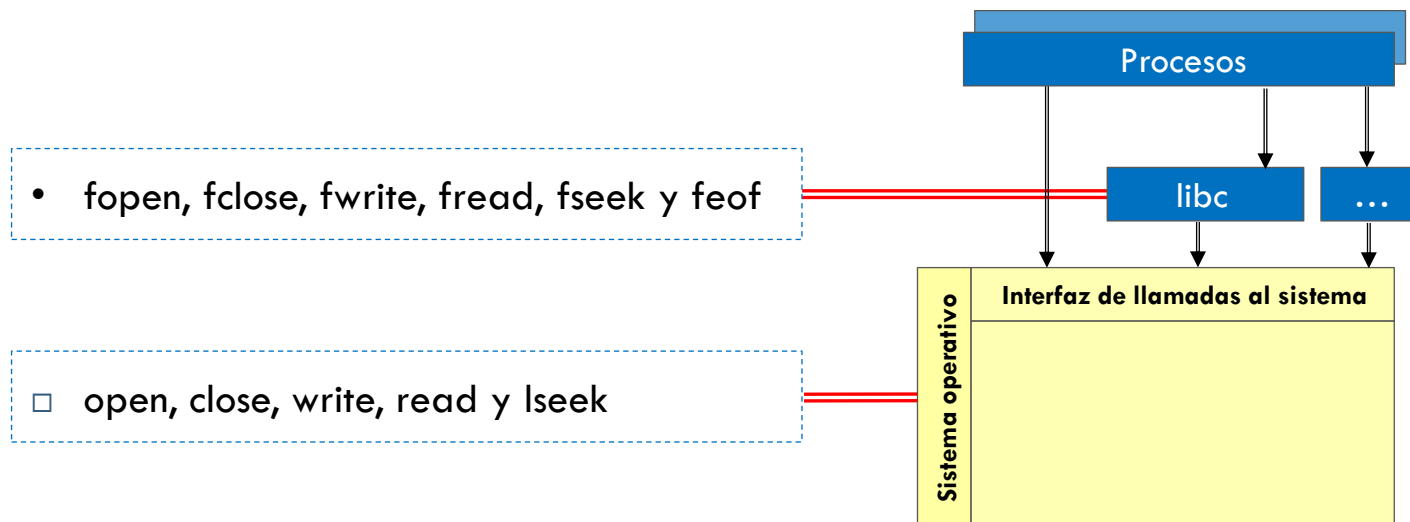
    strcpy(str1,"hola");
    nb = write (fd1,str1,strlen(str1));
    printf("bytes escritos = %d\n",nb);

    close (fd1);
    return (0) ;
}
```

- Se mantiene un **puntero** asociado a cada fichero abierto.
  - ▣ Indica la posición a partir de la cual se realizará la siguiente operación.
- La mayoría de operaciones usan **descriptores de ficheros**:
  - ▣ Identifica una sesión de trabajo con fichero:
    - Un número entero entre 0 y “64K”.
    - Se obtiene al abrir el fichero (open).
    - El resto de operaciones identifican el fichero por su descriptor.
  - ▣ Descriptores predefinidos:
    - 0: entrada estándar
    - 1: salida estándar
    - 2: salida de error

# Llamadas al sistema vs librería sistema

## sistema de ficheros



# Llamadas al sistema vs librería sistema

## escribir en fichero

96

Alejandro Calderón Mateos 

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

int main ( int argc, char *argv[] )
{
    int fd1 ;
    char str1[10] ;
    int nb ;

    fd1 = open ("/tmp/txt1",
                O_CREAT|O_RDWR, S_IRWXU);
    if (-1 == fd1) {
        perror("open:");
        exit(-1);
    }

    strcpy(str1,"hola");
    nb = write (fd1,str1,strlen(str1));
    printf("bytes escritos = %d\n",nb);

    close (fd1);
    return (0) ;
}
```

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

int main ( int argc, char *argv[] )
{
    FILE *fd1 ;
    char str1[10] ;
    int nb ;

    fd1 = fopen ("/tmp/txt2","w+");
    if (NULL == fd1) {
        printf("fopen: error\n");
        exit(-1) ;
    }

    strcpy(str1,"mundo");
    nb = fwrite (str1,strlen(str1),1,fd1);
    printf("items escritos = %d\n",nb);

    fclose (fd1) ;
    return (0) ;
}
```

# Llamadas al sistema vs librería sistema

## leer de fichero

97

Alejandro Calderón Mateos 

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

int main ( int argc, char *argv[] )
{
    int fd1 ;
    char str1[10] ;
    int nb, i ;

    fd1 = open ("/tmp/txt1", O_RDONLY);
    if (-1 == fd1) {
        perror("open:");
        exit(-1);
    }

    i=0;
    do {
        nb = read (fd1, &(str1[i]), 1);
        if (nb != 0) i++ ;
    } while (nb != 0) ;
    str1[i] = '\0';
    printf("%s\n", str1);

    close (fd1);
    return (0);
}
```

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

int main ( int argc, char *argv[] )
{
    FILE *fd1 ;
    char str1[10] ;
    int nb, i ;

    fd1 = fopen ("/tmp/txt2", "r");
    if (NULL == fd1) {
        printf("fopen: error\n");
        exit(-1) ;
    }

    i=0;
    do {
        nb = fread (&(str1[i]), 1, 1, fd1) ;
        i++ ;
    } while (nb != 0) ; /* feof() */
    str1[i] = '\0' ;
    printf("%s\n", str1);

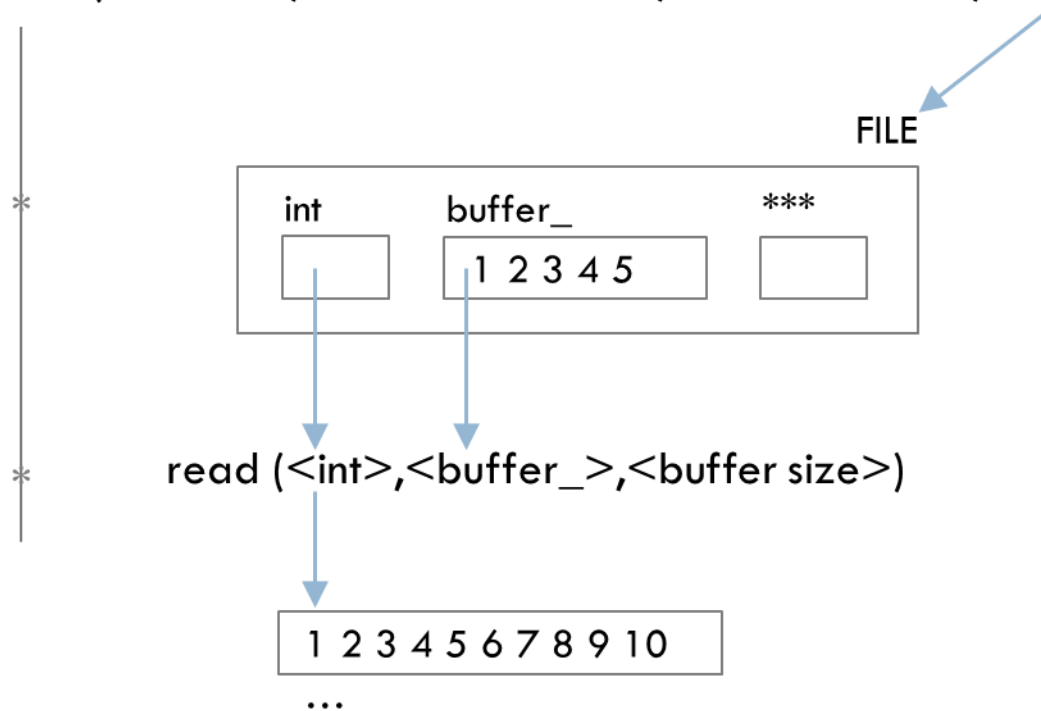
    fclose (fd1);
    return (0);
}
```

# Funcionalidad extendida

98

Alejandro Calderón Mateos 

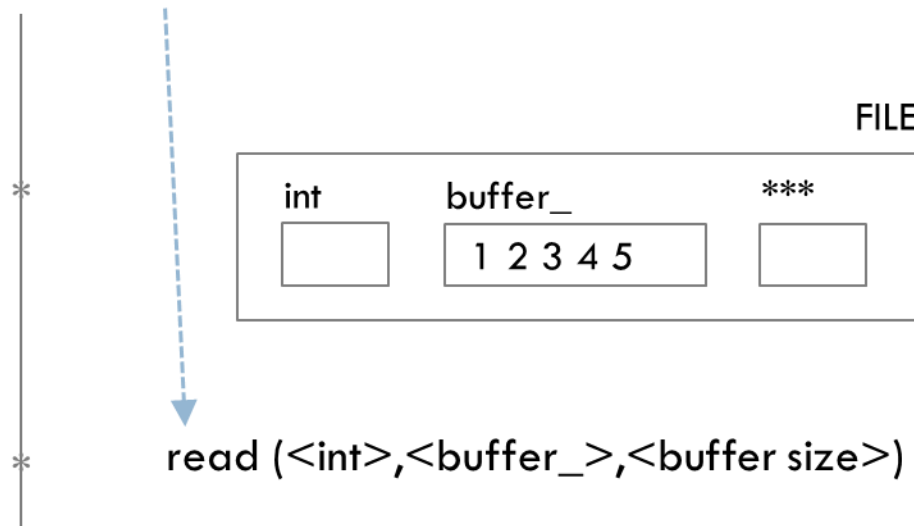
`fread (<buffer>, <size of one elto>, <num. of eltos>, <FILE *>)`



Un puntero a FILE contiene el descriptor de fichero y un buffer intermedio (principalmente)...

# Funcionalidad extendida

`fread (<buffer>, <size of one elto>, <num. of eltos>, <FILE *>)`



... de manera que cuando se pide la primera lectura, se realiza una lectura sobre el buffer (cuyo tamaño es mayor que el elemento pedido)...

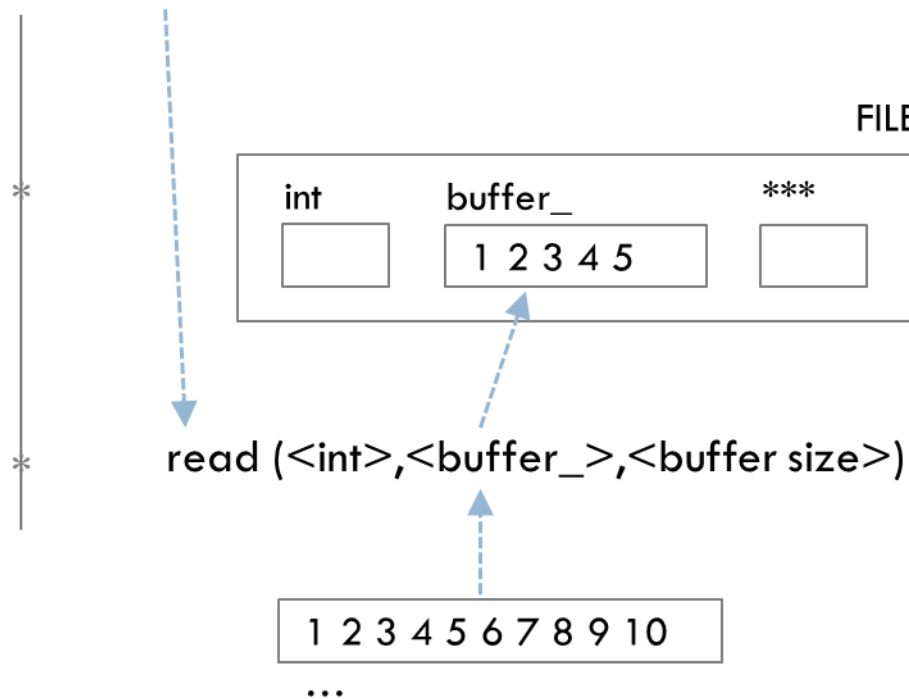


# Funcionalidad extendida

100

Alejandro Calderón Mateos 

`fread (<buffer>, <size of one elto>, <num. of eltos>, <FILE *>)`



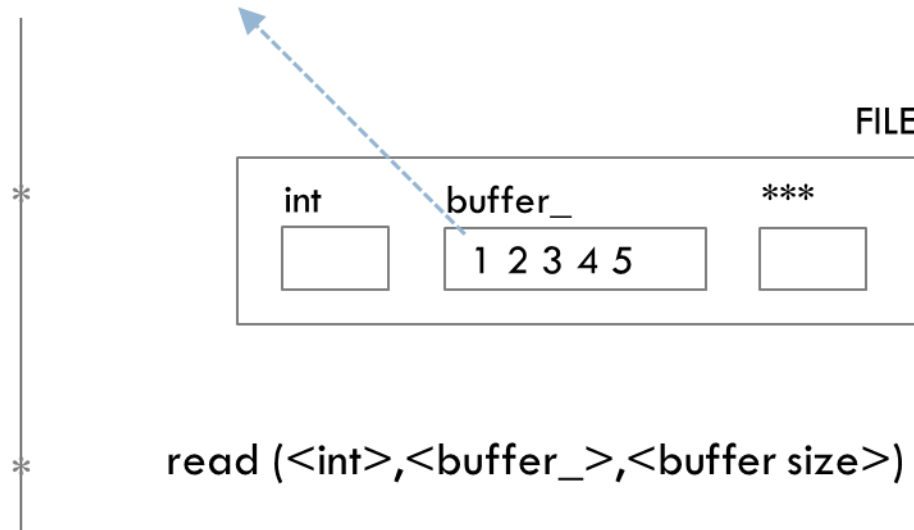
... los datos se cargan en el buffer y se copian la porción pedida al proceso que hace el fread...

# Funcionalidad extendida

101

Alejandro Calderón Mateos 

`fread (<buffer>, <size of one elto>, <num. of eltos>, <FILE *>)`



1 2 3 4 5 6 7 8 9 10  
...

...y la siguiente vez que se hace una lectura, si está en el buffer (memoria) se copia directamente de él. De esta forma se reduce las llamadas al sistema, lo que acelera la ejecución.

# Fichero o archivo: interfaz C99

102

Alejandro Calderón Mateos 

```
#include <stdlib.h>
#include <stdio.h>
#include <sys/time.h>

#define BSIZE 1024

int main ( int argc, char *argv[] )
{
    FILE *fd1 ; int i; double tiempo ;
    char buffer1[BSIZE] ;
    struct timeval ti, tf;

    gettimeofday(&ti, NULL);
    fd1 = fopen ("/tmp/txt2", "w+");
    if (NULL == fd1) {
        printf("fopen: error\n");
        exit(-1) ;
    }
    setbuffer(fd1,buffer1,BSIZE) ;
    for (i=0; i<8*1024; i++)
        fprintf(fd1,"%d",i);
    fclose (fd1) ;

    gettimeofday(&tf, NULL);
    tiempo= (tf.tv_sec - ti.tv_sec)*1000 +
            (tf.tv_usec - ti.tv_usec)/1000.0;
    printf("%g milisegundos\n", tiempo);
    return (0) ;
}
```

□ Compilar (gcc -o b b.c)  
y ejecutar con

▣ BSIZE=1024

▣ BSIZE=0

□ Resultados:

▣ BSIZE=1024

■ T=0.902 milisegundos

▣ BSIZE=0

■ T=14.866 milisegundos

**x 15**

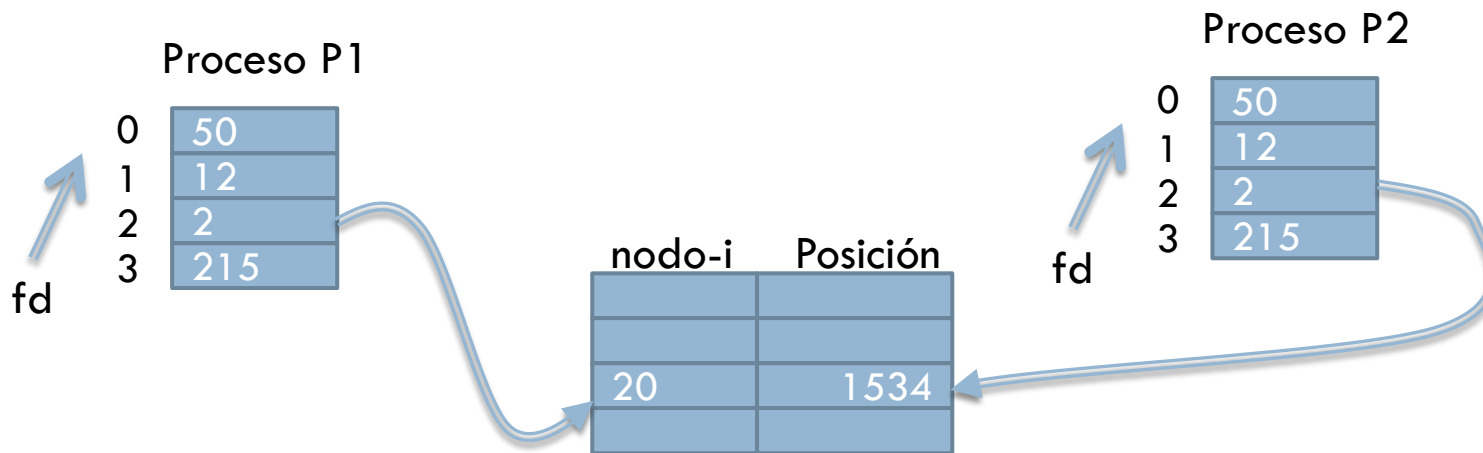
# Interacción entre procesos y ficheros

103

Sistemas operativos: una visión aplicada



- Cada proceso tiene asociada una tabla de ficheros abiertos.
- Cuando se duplica un proceso (fork):
  - ▣ Se duplica la tabla de archivos abiertos.
  - ▣ Se comparte la tabla intermedia de nodos-i y posiciones.



- **Protección:**
  - ▣ dueño      grupo      mundo
  - ▣ rwx        rwx        rwx
- **Ejemplos:** 755 indica rwxr-xr-x

# Ejemplo: Redirección (ls > fichero)

```
void main(void) {
    pid_t pid;
    int status, fd;

    close(1) ;

    fd = open("fichero", O_WRONLY|O_CREAT|O_TRUNC, 0644);
    if (fd < 0)    {
        perror("open");
        exit(-1);
    }
    pid = fork();
    // ...
}
```

# CREAT – Creación de fichero

Servicio	<pre>#include &lt;sys/types.h&gt; #include &lt;sys/stat.h&gt; #include &lt;fcntl.h&gt;  int <b>creat</b>(char *name, mode_t mode);</pre>
Argumentos	<ul style="list-style-type: none"><li>❑ name Nombre de fichero</li><li>❑ mode Bits de permiso para el fichero</li></ul>
Devuelve	Devuelve un descriptor de fichero ó -1 si error.
Descripción	<p>El fichero se abre para escritura:</p> <ul style="list-style-type: none"><li>❑ Si no existe crea un fichero vacío.<ul style="list-style-type: none"><li>■ UID_dueño = UID_efectivo</li><li>■ GID_dueño = GID_efectivo</li></ul></li><li>❑ Si existe lo trunca sin cambiar los bits de permiso.</li></ul>

# OPEN – Apertura de fichero

Servicio	<pre>#include &lt;sys/types.h&gt; #include &lt;sys/stat.h&gt; #include &lt;fcntl.h&gt;  int <b>open</b>(char *name, int flag, ...);</pre>
Argumentos	<ul style="list-style-type: none"><li>▣ name <b>nombre del fichero</b> (puntero al primer caracter).</li><li>▣ flags <b>opciones de apertura</b>:<ul style="list-style-type: none"><li>■ O_RDONLY <b>Sólo lectura</b></li><li>■ O_WRONLY <b>Sólo escritura</b></li><li>■ O_RDWR <b>Lectura y escritura</b></li><li>■ O_APPEND <b>Posicionar el puntero de acceso al final del fichero abierto</b></li><li>■ O_CREAT <b>Si existe no tiene efecto. Si no existe lo crea</b></li><li>■ O_TRUNC <b>Trunca si se abre para escritura</b></li></ul></li></ul>
Devuelve	Un descriptor de fichero ó -1 si hay error.
Descripción	Apertura de fichero (o creación con O_CREAT).

# CREAT y OPEN

## □ Ejemplos:

```
fd = creat("datos.txt", 0744);
```

```
fd = open ("datos.txt",  
           O_WRONLY | O_CREAT | O_TRUNC, 0744);
```

```
fd = open("/home/patricia/datos.txt");
```

```
fd = open("/home/patricia/datos.txt",  
           O_WRONLY | O_CREAT | O_TRUNC, 0740);
```



# CLOSE – Cierre de fichero

Servicio	<pre>#include &lt;unistd.h&gt;  int <b>close</b>(int fd);</pre>
Argumentos	fd descriptor de fichero.
Devuelve	Devuelve 0 ó -1 si error.
Descripción	El proceso cierra la session de trabajo con el fichero, y el descriptor pasa a estar libre.

# UNLINK – Borrado de fichero

Servicio	<pre>#include &lt;unistd.h&gt;  int <b>unlink</b>(const char* path);</pre>
Argumentos	<code>path</code> nombre del fichero
Devuelve	Devuelve 0 ó -1 si error.
Descripción	Decrementa el contador de enlaces del fichero. Si el contador es 0, borra el fichero y libera sus recursos.

# READ – Lectura de fichero

Servicio	<pre>#include &lt;sys/types.h&gt;  ssize_t <b>read</b>(int fd, void *buf, size_t n_bytes);</pre>
Argumentos	<ul style="list-style-type: none"><li>❑ <code>fd</code> descriptor de fichero</li><li>❑ <code>buf</code> zona donde almacenar los datos</li><li>❑ <code>n_bytes</code> número de bytes a leer</li></ul>
Devuelve	Número de bytes realmente leídos ó -1 si error.
Descripción	<ul style="list-style-type: none"><li>❑ Transfiere <code>n_bytes</code>. Puede leer menos datos de los solicitados si se rebasa el fin de fichero o se interrumpe por una señal.</li><li>❑ Después de la lectura se incrementa el puntero del fichero con el número de bytes realmente transferidos.</li></ul>

# WRITE – Escritura de fichero

Servicio	<pre>#include &lt;sys/types.h&gt;  ssize_t <b>write</b>(int fd, void *buf, size_t n_bytes);</pre>
Argumentos	<ul style="list-style-type: none"><li>❑ <code>fd</code> descriptor de fichero</li><li>❑ <code>buf</code> zona de datos a escribir</li><li>❑ <code>n_bytes</code> número de bytes a escribir</li></ul>
Devuelve	Número de bytes realmente escritos ó -1 si error.
Descripción	<ul style="list-style-type: none"><li>❑ Transfiere <code>n_bytes</code>. Puede escribir menos datos de los solicitados si se rebasa el tamaño máximo de un fichero o se interrumpe por una señal.</li><li>❑ Después de la escritura se incrementa el puntero del fichero con el número de bytes realmente transferidos.</li><li>❑ Si se rebasa el fin de fichero el fichero aumenta de tamaño.</li></ul>

# LSEEK – Movimiento del puntero de posición

112

Sistemas operativos: una visión aplicada



Servicio	<pre>#include &lt;sys/types.h&gt; #include &lt;unistd.h&gt;  off_t <b>lseek</b>(int fd, off_t offset, int whence);</pre>
Argumentos	<ul style="list-style-type: none"><li>❑ fd <b>Descriptor de fichero</b></li><li>❑ offset <b>desplazamiento</b></li><li>❑ whence <b>base del desplazamiento</b></li></ul>
Devuelve	La nueva posición del puntero ó -1 si error.
Descripción	<ul style="list-style-type: none"><li>❑ Coloca el puntero de acceso asociado a fd</li><li>❑ La nueva posición se calcula:<ul style="list-style-type: none"><li>■ SEEK_SET posición = offset</li><li>■ SEEK_CUR posición = posición actual + offset</li><li>■ SEEK_END posición = tamaño del fichero + offset</li></ul></li></ul>

# Ejemplo: Copia de un fichero en otro (1/3)

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

#define BUFSIZE 512

int main(int argc, char **argv)
{
    int fd_ent, fd_sal;
    char buffer[BUFSIZE];
    int n_read;

    /* abre el fichero de entrada */
    fd_ent = open(argv[1], O_RDONLY);
    if (fd_ent < 0) {
        perror("open");
        exit(-1);
    }

    /* crea el fichero de salida */
    fd_sal = creat(argv[2], 0644);
    if (fd_sal < 0) {
        close(fd_ent);
        perror("open");
        exit(-1);
    }

    /* bucle de lectura del fichero de entrada */
    while ((n_read = read(fd_ent, buffer, BUFSIZE)) > 0)
    {
        /* escribir el buffer al fichero de salida */
        if (write(fd_sal, buffer, n_read) < n_read) {
            perror("write2");
            close(fd_ent); close(fd_sal); exit(-1);
        }
    }

    if (n_read < 0) {
        perror("read");
        close(fd_ent); close(fd_sal); exit(-1);
    }

    close(fd_ent); close(fd_sal);
    return 0;
}
```

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

#define BUFSIZE 512

int main(int argc, char **argv)
{
    int fd_ent, fd_sal;
    char buffer[BUFSIZE];
    int n_read;
```

# Ejemplo: Copia de un fichero en otro (2/3)

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

#define BUFSIZE 512

int main(int argc, char **argv)
{
    int fd_ent, fd_sal;
    char buffer[BUFSIZE];
    int n_read;

    /* abre el fichero de entrada */
    fd_ent = open(argv[1], O_RDONLY);
    if (fd_ent < 0) {
        perror("open");
        exit(-1);
    }

    /* crea el fichero de salida */
    fd_sal = creat(argv[2], 0644);
    if (fd_sal < 0) {
        close(fd_ent);
        perror("open");
        exit(-1);
    }

    /* bucle de lectura del fichero de entrada */
    while ((n_read = read(fd_ent, buffer, BUFSIZE)) > 0)
    {
        /* escribir el buffer al fichero de salida */
        if (write(fd_sal, buffer, n_read) < n_read) {
            perror("write2");
            close(fd_ent); close(fd_sal); exit(-1);
        }
    }

    if (n_read < 0) {
        perror("read");
        close(fd_ent); close(fd_sal); exit(-1);
    }

    close(fd_ent); close(fd_sal);
    return 0;
}
```

```
/* abre el fichero de entrada */
fd_ent = open(argv[1], O_RDONLY);
if (fd_ent < 0) {
    perror("open");
    exit(-1);
}

/* crea el fichero de salida */
fd_sal = creat(argv[2], 0644);
if (fd_sal < 0) {
    close(fd_ent);
    perror("open");
    exit(-1);
}
```

# Ejemplo: Copia de un fichero en otro (3/3)

115

Sistemas operativos: una visión aplicada



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

#define BUFSIZE 512

int main(int argc, char **argv)
{
    int fd_ent, fd_sal;
    char buffer[BUFSIZE];
    int n_read;

    /* abre el fichero de entrada */
    fd_ent = open(argv[1], O_RDONLY);
    if (fd_ent < 0) {
        perror("open");
        exit(-1);
    }

    /* crea el fichero de salida */
    fd_sal = creat(argv[2], 0644);
    if (fd_sal < 0) {
        close(fd_ent);
        perror("open");
        exit(-1);
    }

    /* bucle de lectura del fichero de entrada */
    while ((n_read = read(fd_ent, buffer, BUFSIZE)) > 0)
    {
        /* escribir el buffer al fichero de salida */
        if (write(fd_sal, buffer, n_read) < n_read) {
            perror("write2");
            close(fd_ent); close(fd_sal); exit(-1);
        }
    }

    if (n_read < 0) {
        perror("read");
        close(fd_ent); close(fd_sal); exit(-1);
    }

    close(fd_ent); close(fd_sal);
    return 0;
}
```

```
/* bucle de lectura del fichero de entrada */
while ((n_read = read(fd_ent, buffer, BUFSIZE)) > 0)
{
    /* escribir el buffer al fichero de salida */
    if (write(fd_sal, buffer, n_read) < n_read) {
        perror("write2");
        close(fd_ent); close(fd_sal);
        exit(-1);
    }
}

if (n_read < 0) {
    perror("read");
    close(fd_ent); close(fd_sal);
    exit(-1);
}

close(fd_ent); close(fd_sal);
return 0;
}
```



# FCNTL – Modificación de atributos

Servicio	<pre>#include &lt;sys/types.h&gt;  int <b>fcntl</b>(int fildes, int cmd /* arg*/ ...);</pre>
Argumentos	<ul style="list-style-type: none"><li>▣ <code>fildes</code> descriptor de ficheros</li><li>▣ <code>cmd</code> mandato para modificar atributos, puede haber varios.</li></ul>
Devuelve	0 para éxito ó -1 si error.
Descripción	Modifica los atributos de un fichero abierto

# DUP – Duplicación de descriptor de fichero

Servicio	<pre>#include &lt;unistd.h&gt;  int <b>dup</b>(int fd);</pre>
Argumentos	<ul style="list-style-type: none"><li>▣ <code>fd</code> descriptor de fichero</li></ul>
Devuelve	Un descriptor de fichero que comparte todas las propiedades del <code>fd</code> ó -1 si error.
Descripción	<ul style="list-style-type: none"><li>▣ Crea un nuevo descriptor de fichero que tiene en común con el anterior:<ul style="list-style-type: none"><li>■ Accede al mismo fichero</li><li>■ Comparte el mismo puntero de posición</li><li>■ El modo de acceso es idéntico.</li></ul></li><li>▣ El nuevo descriptor tendrá el menor valor numérico posible.</li></ul>

# FTRUNCATE – Asignación e espacio a un fichero

Servicio	<pre>#include &lt;unistd.h&gt;  int <b>ftruncate</b>(int fd, off_t length);</pre>
Argumentos	<ul style="list-style-type: none"><li>▣ fd descriptor de fichero</li><li>▣ length nuevo tamaño del fichero</li></ul>
Devuelve	Devuelve 0 ó -1 si error.
Descripción	El nuevo tamaño del fichero es length. Si length es 0 se trunca el fichero.

# STAT – Información sobre un fichero

Servicio	<pre>#include &lt;sys/types.h&gt; #include &lt;sys/stat.h&gt;  int <b>stat</b>(char *name, struct stat *buf); int <b>fstat</b>(int fd, struct stat *buf);</pre>
Argumentos	<ul style="list-style-type: none"><li>▣ name nombre del fichero</li><li>▣ fd descriptor de fichero</li><li>▣ buf puntero a un objeto de tipo struct stat donde se almacenará la información del fichero.</li></ul>
Devuelve	Devuelve 0 ó -1 si error.
Descripción	Obtiene información sobre un fichero y la almacena en una estructura de tipo struct stat

# STAT – Información sobre un fichero

```
struct stat {  
    mode_t    st_mode;    /* modo del fichero */  
    ino_t     st_ino;     /* número del fichero */  
    dev_t     st_dev;     /* dispositivo */  
    nlink_t   st_nlink;   /* número de enlaces */  
    uid_t     st_uid;     /* UID del propietario */  
    gid_t     st_gid;     /* GID del propietario */  
    off_t     st_size;    /* número de bytes */  
    time_t    st_atime;   /* último acceso */  
    time_t    st_mtime;   /* última modificacion */  
    time_t    st_ctime;   /* último modificacion de datos */  
};
```

# STAT – Información sobre un fichero

## □ Comprobación del tipo de fichero aplicado a `st_mode`:

<code>S_ISDIR(s.st_mode)</code>	Cierto si directorio
<code>S_ISCHR(s.st_mode)</code>	Cierto si especial de caracteres
<code>S_ISBLK(s.st_mode)</code>	Cierto si especial de bloques
<code>S_ISREG(s.st_mode)</code>	Cierto si fichero normal
<code>S_ISFIFO(s.st_mode)</code>	Cierto si pipe o FIFO

# UTIME – Alteración de atributos de fecha

Servicio	<pre>#include &lt;sys/stat.h&gt; #include &lt;utime.h&gt;  int <b>utime</b>(char *name, struct utimbuf *times);</pre>
Argumentos	<ul style="list-style-type: none"><li>▣ <code>name</code> nombre del fichero</li><li>▣ <code>times</code> estructura con las fechas de último acceso y modificación.<ul style="list-style-type: none"><li>■ <code>time_t actime</code> fecha de acceso</li><li>■ <code>time_t mctime</code> fecha de modificación</li></ul></li></ul>
Devuelve	Devuelve 0 ó -1 si error.
Descripción	Cambia las fechas de último acceso y última modificación según los valores de la estructura <code>struct utimbuf</code>

# Servicios POSIX para directorios

## □ Visión lógica:

- Un directorio es un fichero con registros tipo “estructura DIR”
- Existen llamadas para trabajar con los registros de un directorio.
- Particularidades:
  - SOLO SE LEE de un directorio, NO SE PUEDE ESCRIBIR DESDE PROGRAMA.
  - ¡Ojo! Al ser el nombre de cada entrada del directorio de longitud variable no se pueden manipular como registros de longitud fija

### □ “estructura DIR”:

- d\_ino;       // *Nodo\_i*
- d\_off;       // *Posición en el fichero del elemento del directorio*
- d\_reclen;   // *Tamaño del directorio*
- d\_type;     // *Tipo del elemento*
- d\_name[0]; // *Nombre del fichero **de longitud variable***



# Servicios POSIX para directorios

- **DIR \*opendir**(const char \**dirname*);
  - ▣ Abre el directorio y devuelve un puntero al principio de tipo DIR
- **int readdir**(DIR \**dirp*, struct dirent \**entry*, struct dirent \*\**result*);
  - ▣ Lee la siguiente entrada de directorio y la devuelve en una struct dirent
- **long int telldir**(DIR \**dirp*);
  - ▣ Indica la posición actual del puntero dentro del archivo del directorio
- **void seekdir**(DIR \**dirp*, long int *loc*);
  - ▣ Avanza desde la posición actual hasta la indicada en “loc”. Nunca saltos atras.
- **void rewinddir**(DIR \**dirp*);
  - ▣ Resetea el puntero del archivo y lo pone otra vez al principio
- **int closedir**(DIR \**dirp*);
  - ▣ Cierra el archivo del directorio

# Ejemplo de trabajo con directorios

125

Alejandro Calderón Mateos 

```
#include <sys/types.h>
#include <dirent.h>
#include <stdlib.h>
#include <stdio.h>

int main ( int argc, char *argv[] )
{
    DIR *dirp;
    struct dirent *direntp;

    // list entries of "." directory
    dirp = opendir(".");
    if (dirp == NULL) {
        perror("Error: "); exit(1);
    }

    while ((direntp = readdir(dirp)) != NULL) {
        printf("{ i-node:%ld,\t offset: %ld, \t long:%ld,\t name:%s }\n",
               direntp->d_ino, direntp->d_off, direntp->d_reclen, direntp->d_name);
    }
    closedir(dirp);
}
```

# Contenidos

126

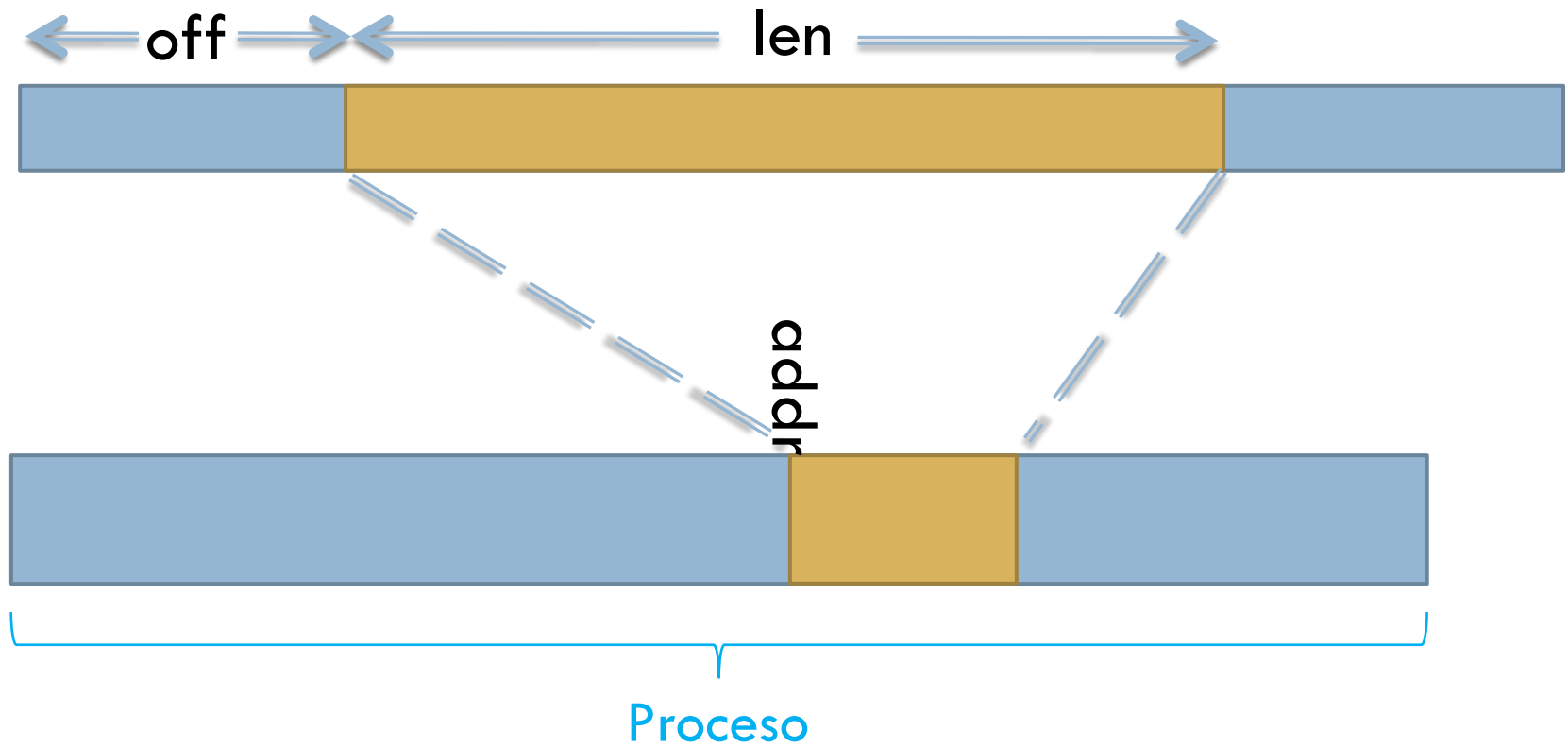


- Introducción a llamadas al sistema
- Mecanismo de llamada al sistema
- Llamadas para servicios de:
  - ▣ Gestión de procesos
  - ▣ Gestión de ficheros y directorios
    - Proyección de fichero

# Proyección POSIX

127

Sistemas operativos: una visión aplicada



# Proyección POSIX: mmap

Servicio	<pre>void *mmap(void *addr, size_t len,            int prot, int flags,            int fildes, off_t off);</pre>
Argumentos	<ul style="list-style-type: none"><li>▣ <code>addr</code> dirección donde proyectar. Si <code>NULL</code> el SO elige una.</li><li>▣ <code>len</code> especifica el número de bytes a proyectar.</li><li>▣ <code>prot</code> el tipo de acceso (lectura, escritura o ejecución).</li><li>▣ <code>flags</code> especifica información sobre el manejo de los datos proyectados (compartidos, privado, etc.).</li><li>▣ <code>fildes</code> representa el descriptor de fichero del fichero o descriptor del objeto de memoria a proyectar.</li><li>▣ <code>off</code> desplazamiento dentro del fichero a partir del cual se realiza la proyección.</li></ul>
Devuelve	Devuelve la dirección de memoria donde se ha proyectado el fichero.
Descripción	Establece una proyección entre el espacio de direcciones de un proceso y un descriptor de fichero u objeto de memoria compartida.

# Proyección POSIX: mmap

- `int prot`: Tipos de protección:
  - ▣ `PROT_READ`: Se puede leer.
  - ▣ `PROT_WRITE`: Se puede escribir.
  - ▣ `PROT_EXEC`: Se puede ejecutar.
  - ▣ `PROT_NONE`: No se puede acceder a los datos.
- `int flags`: Propiedades de una región de memoria:
  - ▣ `MAP_SHARED`: La región es compartida. Las modificaciones afectan al fichero. Los procesos hijos comparten la región.
  - ▣ `MAP_PRIVATE`: La región es privada. El fichero no se modifica. Los procesos hijos obtienen duplicados no compartidos.
  - ▣ `MAP_FIXED`: El fichero debe proyectarse en la dirección especificada por la llamada.

# Proyección POSIX: munmap

Servicio	<code>void <b>munmap</b>(void *addr, size_t len);</code>
Argumentos	<ul style="list-style-type: none"><li>▣ <code>addr</code> dirección donde está proyectado.</li><li>▣ <code>len</code> especifica el número de bytes proyectados.</li></ul>
Devuelve	Nada.
Descripción	Desproyecta parte del espacio de direcciones de un proceso comenzando en la dirección <code>addr</code> .

# Ejemplo: copia de un fichero (1 / 2)

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

int main() {
    int i, fd1, fd2;
    struct stat dstat;
    char * vec1, *vec2, *p, *q;

    fd1 = open("f1", O_RDONLY);
    fd2 = open("f2", O_CREAT|O_TRUNC|O_RDWR, 0640);
    fstat(fd1, &dstat);
    ftruncate(fd2, dstat.st_size);

    vec1=mmap(0, bstat.st_size,
        PROT_READ, MAP_SHARED, fd1, 0);
    vec2=mmap(0, bstat.st_size,
        PROT_READ, MAP_SHARED, fd2, 0);

    close(fd1); close(fd2);

    p=vec1; q=vec2;
    for (i=0; i<dstat.st_size; i++) {
        *q++ = *p++;
    }

    munmap(fd1, bstat.st_size);
    munmap(fd2, bstat.st_size);

    return 0;
}
```

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

```
int main()
{
    int i, fd1, fd2;
    struct stat dstat;
    char * vec1, *vec2, *p, *q;

    fd1 = open("f1", O_RDONLY);
    fd2 = open("f2", O_CREAT|O_TRUNC|O_RDWR, 0640);
    fstat(fd1, &dstat);
    ftruncate(fd2, dstat.st_size);
```



# Ejemplo: copia de un fichero (2/2)

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

int main() {
    int i, fd1, fd2;
    struct stat dstat;
    char * vec1, *vec2, *p, *q;

    fd1 = open("f1", O_RDONLY);
    fd2 = open("f2", O_CREAT|O_TRUNC|O_RDWR, 0640);
    fstat(fd1, &dstat);
    ftruncate(fd2, dstat.st_size);

    vec1=mmap(0, bstat.st_size,
        PROT_READ, MAP_SHARED, fd1, 0);
    vec2=mmap(0, bstat.st_size,
        PROT_READ, MAP_SHARED, fd2, 0);

    close(fd1); close(fd2);

    p=vec1; q=vec2;
    for (i=0; i<dstat.st_size; i++) {
        *q++ = *p++;
    }

    munmap(fd1, bstat.st_size);
    munmap(fd2, bstat.st_size);

    return 0;
}
```

```
vec1=mmap(0, bstat.st_size,
           PROT_READ, MAP_SHARED, fd1, 0);
vec2=mmap(0, bstat.st_size,
           PROT_READ, MAP_SHARED, fd2, 0);

close(fd1); close(fd2);

p=vec1; q=vec2;
for (i=0; i<dstat.st_size; i++) {
    *q++ = *p++;
}

munmap(fd1, bstat.st_size);
munmap(fd2, bstat.st_size);

return 0;
}
```

# Ejemplo: contar el número de blancos en un fichero (1 / 2)

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

int main() {
    int fd;
    struct stat dstat;
    int i, n;
    char c,
    char * vec;

    fd = open("datos.txt", O_RDONLY);
    fstat(fd, &dstat);
    vec = mmap(NULL, dstat.st_size,
               PROT_READ, MAP_SHARED, fd, 0);
    close(fd);
    c = vec;
    for (i=0; i<dstat.st_size; i++) {
        if (*c==' ') {
            n++;
        }
        c++;
    }
    munmap(vec, dstat.st_size);
    printf("n=%d, \n", n);
    return 0;
}
```

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

```
int main() {
    int fd;
    struct stat dstat;
    int i, n;
    char c,
    char * vec;
```

# Ejemplo: contar el número de blancos en un fichero (2/2)

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

int main() {
    int fd;
    struct stat dstat;
    int i, n;
    char c,
    char * vec;

    fd = open("datos.txt", O_RDONLY);
    fstat(fd, &dstat);
    vec = mmap(NULL, dstat.st_size,
               PROT_READ, MAP_SHARED, fd, 0);
    close(fd);
    c = vec;
    for (i=0; i<dstat.st_size; i++) {
        if (*c==' ') {
            n++;
        }
        c++;
    }
    munmap(vec, dstat.st_size);
    printf("n=%d, \n", n);
    return 0;
}
```

```
fd = open("datos.txt", O_RDONLY);
fstat(fd, &dstat);
vec = mmap(NULL, dstat.st_size,
           PROT_READ, MAP_SHARED, fd, 0);
close(fd);
c = vec;
for (i=0; i<dstat.st_size; i++) {
    if (*c==' ') { n++; }
    c++;
}
munmap(vec, dstat.st_size);
printf("n=%d, \n", n);
return 0;
}
```

# SISTEMAS OPERATIVOS: SERVICIOS DE LOS SISTEMAS OPERATIVOS



Llamadas al sistema