OPERATING SYSTEMS: OPERATING SYSTEM SERVICES

System calls

To remember…

Before classes	Class	After class

Prepare the prerequisites.

Study the material associated with the **bibliography**:
slides alone are not enough.
Please ask questions (especially after study).

Exercising skills:

- Perform all **exercises**.
- Carrying out the **practice notebooks** and **the practical exercises** progressively.

Recommended reading

Base

1. Carretero 2020:
   1. Cap. 6
2. Carretero 2007:
   1. Cap. 6.1 and 6.2

Suggested

1. Tanenbaum 2006:
   1. (es) Chap. 5
   2. (en) Chap. 5
2. Stallings 2005:
   1. 5.1, 5.2 and 5.3
3. Silberschatz 2006:
   1. 6.1, 6.2, 6.5 and 6.6

Contents

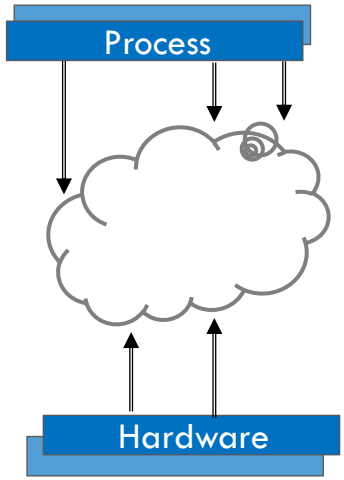
- Introduction to system calls
- System call mechanism
- Calls for services of:
  - Process management
  - Management of files and directories

Contents

Operating System executes…

Alejandro Calderón Mateos

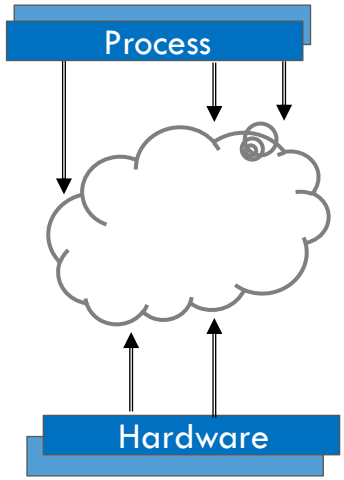
- During boot-up.
- Once the booting is complete, it executes in response to events:
  - **System call.**
  - Exception.
  - Hardware interruption.
- In kernel processes (firewall, etc.)



Events that activate the operating system

Alejandro Calderón Mateos

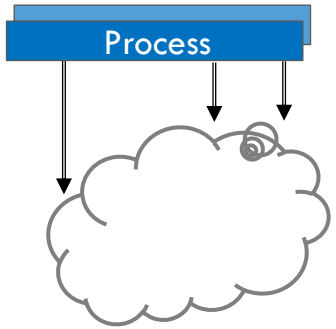
- System call.
  - **{ Source: "processes",**
    Function: "Request for services" }
- Exception.
  - { Source: "processes",
    Function: "Handling exceptional situations" }
- Hardware interruption.
  - { Source: "hardware",
    Function: "Request for hw. attention" }



System services

Alejandro Calderón Mateos

- Process management
- Memory management
- File management
- Device management
- Communication
- Maintenance

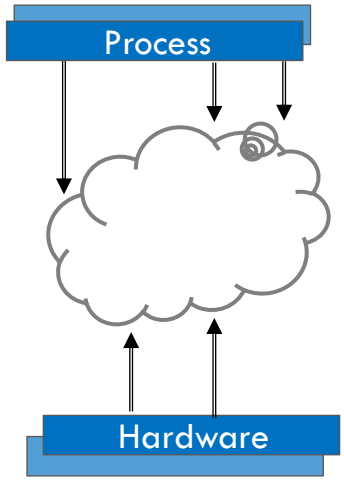


System calls…

summary

Alejandro Calderón Mateos

- During boot-up.
- **After startup,** it is executed **in response to events:**
  - **System call.**
    - { Source: "processes", Function: "**Request for services**" }
      - **Process management**
      - **Memory management**
      - **File management**
      - **Device management**
      - **Communication**
      - **Maintenance**
  - Exception.
    - { Source: "processes", Function: "Handling exceptions" }
  - Hardware interruption.
    - { Source: "hardware", Function: "Request for hw. attention" }
- In kernel processes (firewall, etc.)

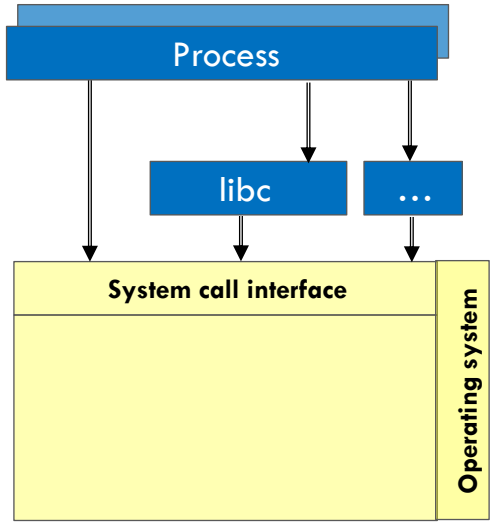


System calls versus…

- Commands are not system calls.
  - It is possible for a command line shell command (/bin/sh) to internally invoke the call.
  - E.g.: printf vs printf()
- Not every function in the system library is a system call.
  - Although it is possible for a library function to extend the functionalities of several system calls.
  - E.g.: sbrk() vs malloc()

System calls vs. system library

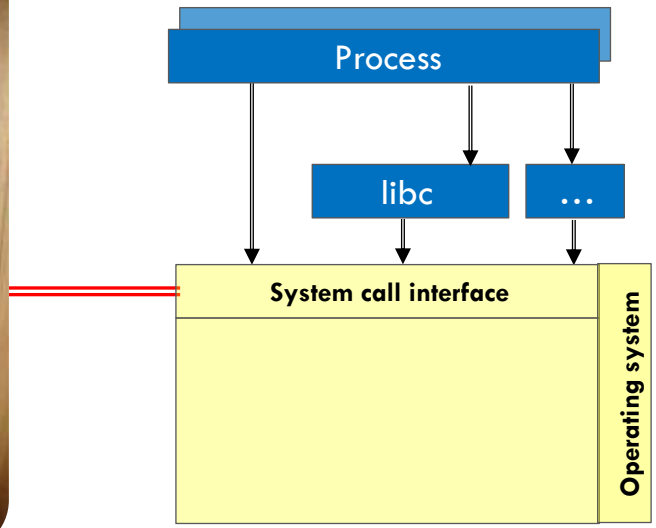
Alejandro Calderón Mateos



System calls vs. system library

Alejandro Calderón Mateos

"very basic house services"

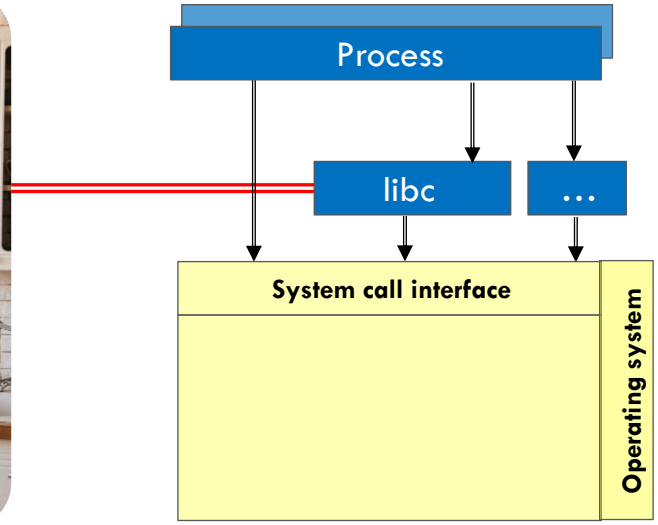


System calls vs. system library

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

Alejandro Calderón Mateos

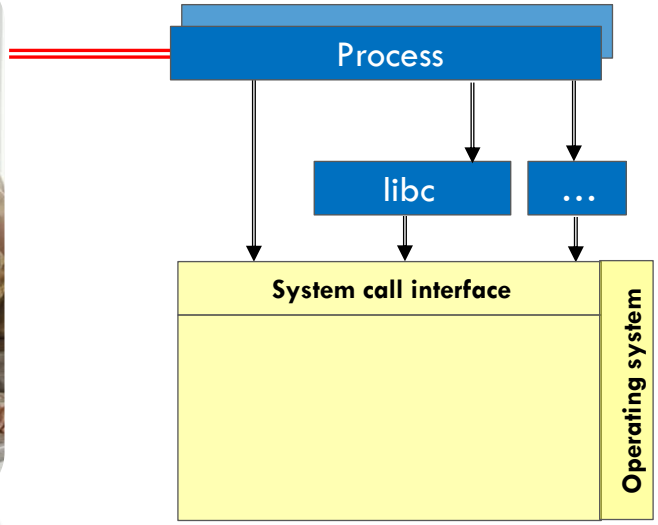
"basic house services"



System calls vs. system library

https://www.pexels.com/es-es/foto/hombre-en-camisa-de-vestir-blanca-sentado-al-lado-de-una-mujer-en-vestido-naranja-4262411

"people who use the services"



System calls vs. system library

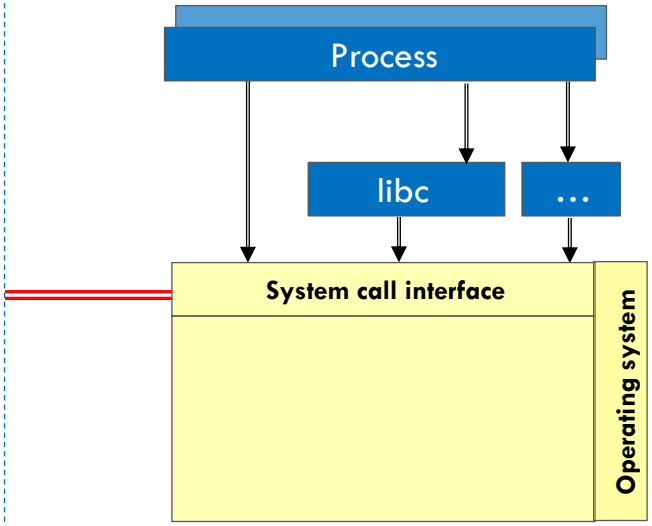
memory

Alejandro Calderón Mateos

```
#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);
- ...



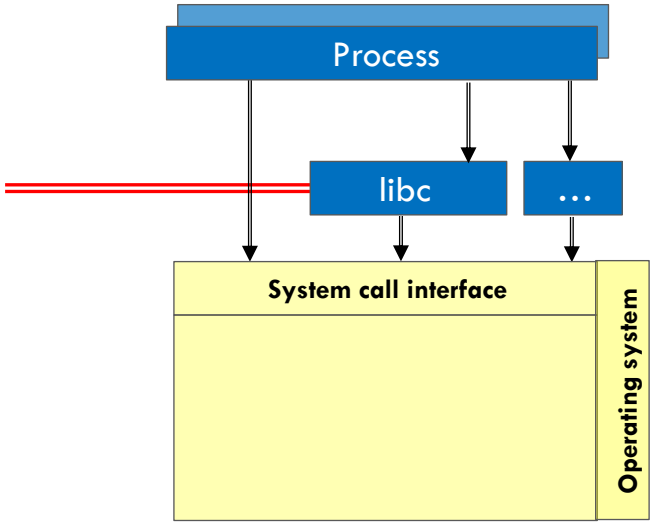
System calls vs. system library

memory

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, ...);
    ...
```

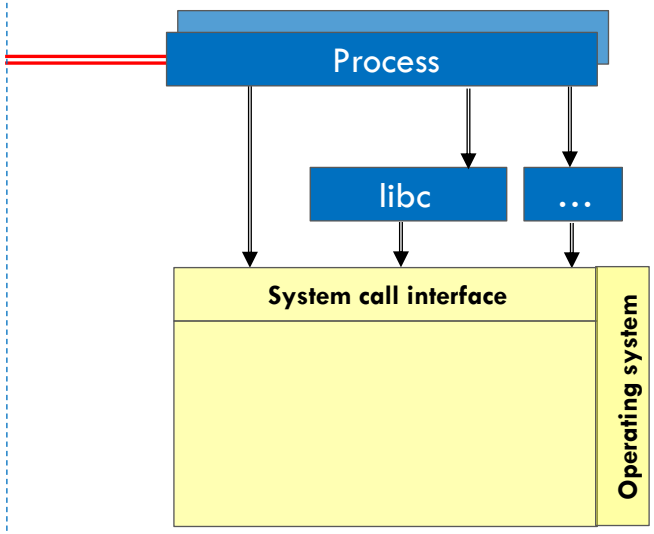


System calls vs. system library

memory

Alejandro Calderón Mateos

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

int main ( int argc, char *argv[] )
{
    int *ptr1 ;
    int i ;

    ptr1 = (int *)malloc (100*sizeof(int)) ;
    for (i=0; i<100; i++)
        ptr1[i] = 10 ;
    free(ptr1);
}
```

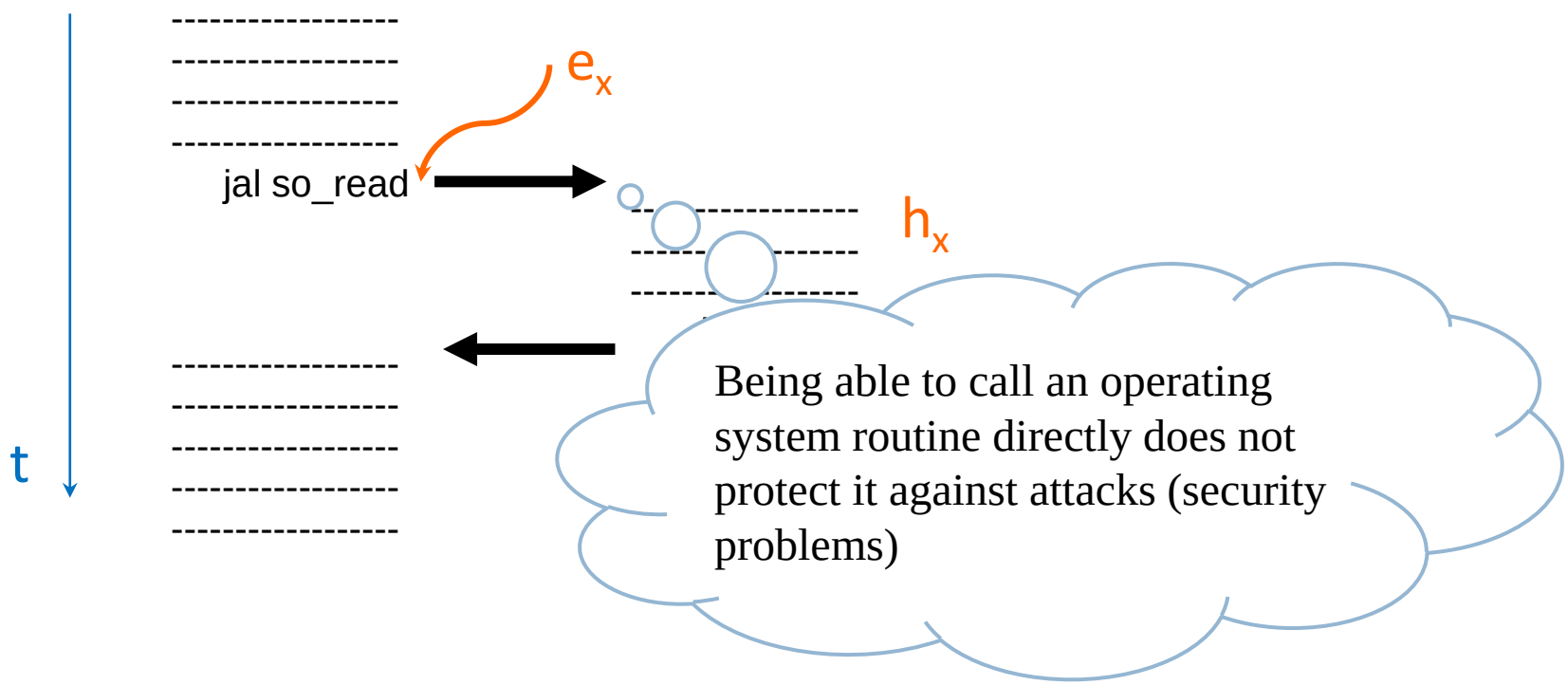


Contents

- Introduction to system calls
- System call mechanism
- Calls for services of:
  - Process management
  - Management of files and directories

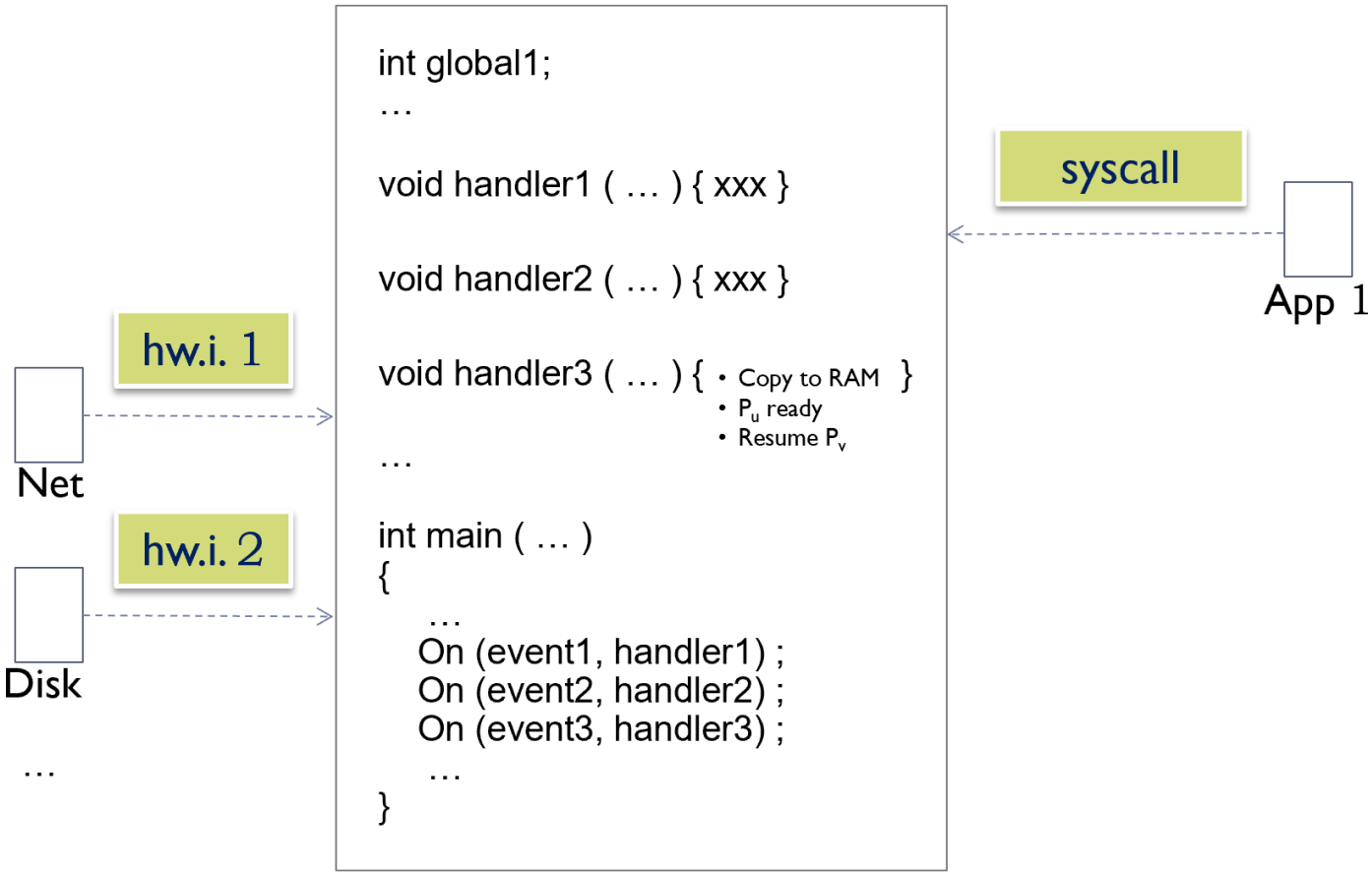
Execution of a service request
is not a function call…

Alejandro Calderón Mateos



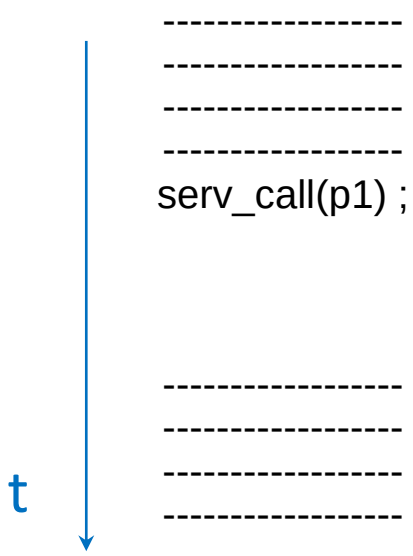
Execution by processing events
general aspects

Alejandro Calderón Mateos



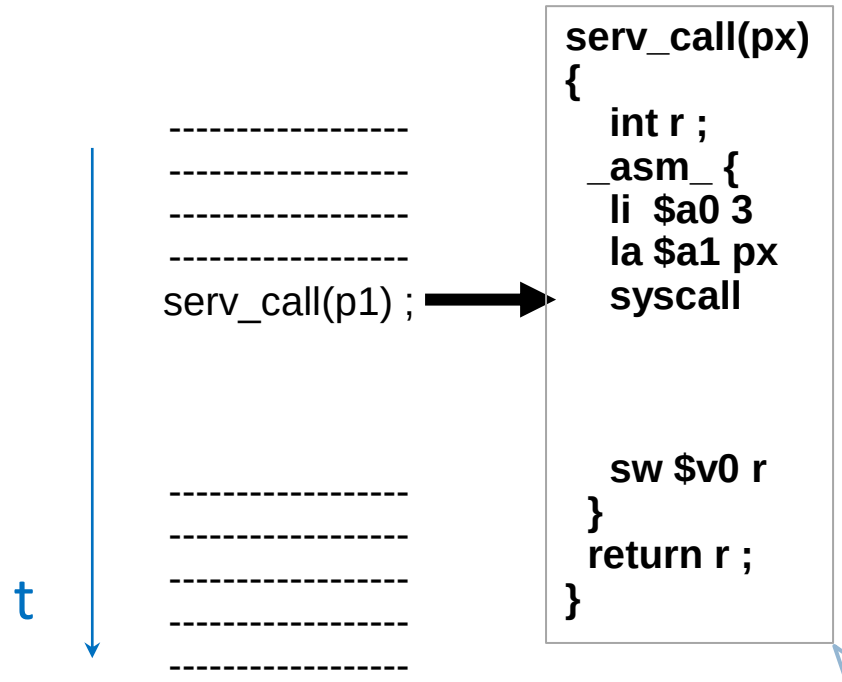
Execution of a service request
execution (general)

Alejandro Calderón Mateos



Execution of a service request
execution (general)

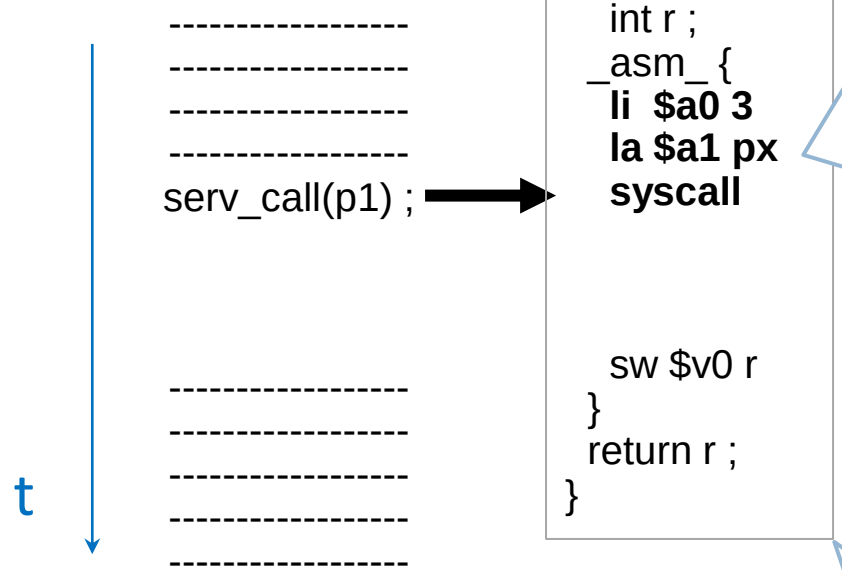
Alejandro Calderón Mateos



- Call to library function that:
  - processes the OS invocation and
  - return the result.
- Each OS provides at least one API with functions, at least one per OS call.

Execution of a service request
execution (general)

Alejandro Calderón Mateos

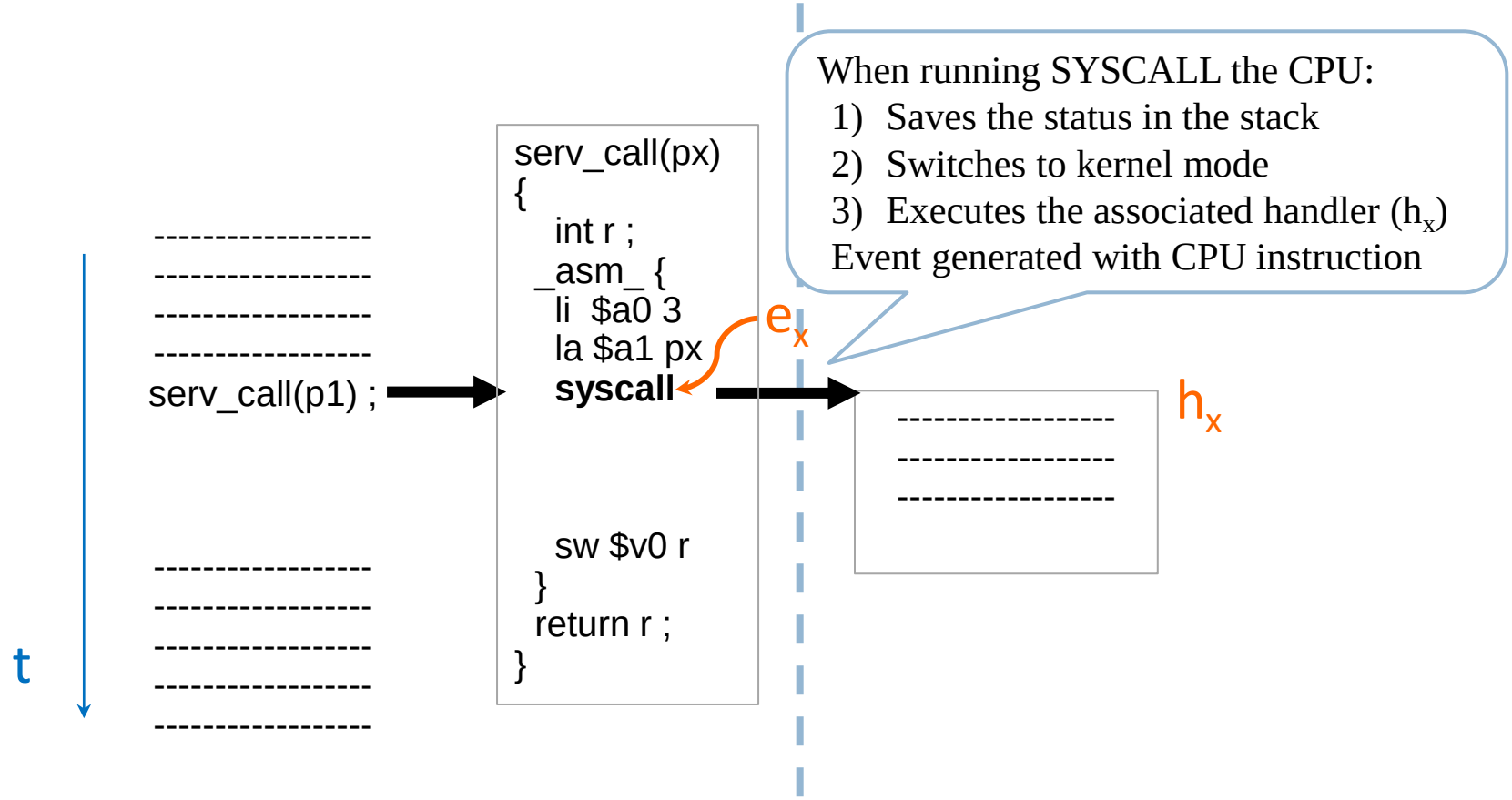


- Passing parameters by registers:
  1) $a0 service identifier.
  2) $a1… for service parameters.
  3) Syscall instruction for activation.
- Stack parameter passing is possible.
- Or in memory zone pointed by a register.

- Call to library function that:
  - processes the OS invocation and
  - return the result.
- Each OS provides at least one API with functions, at least one per OS call.

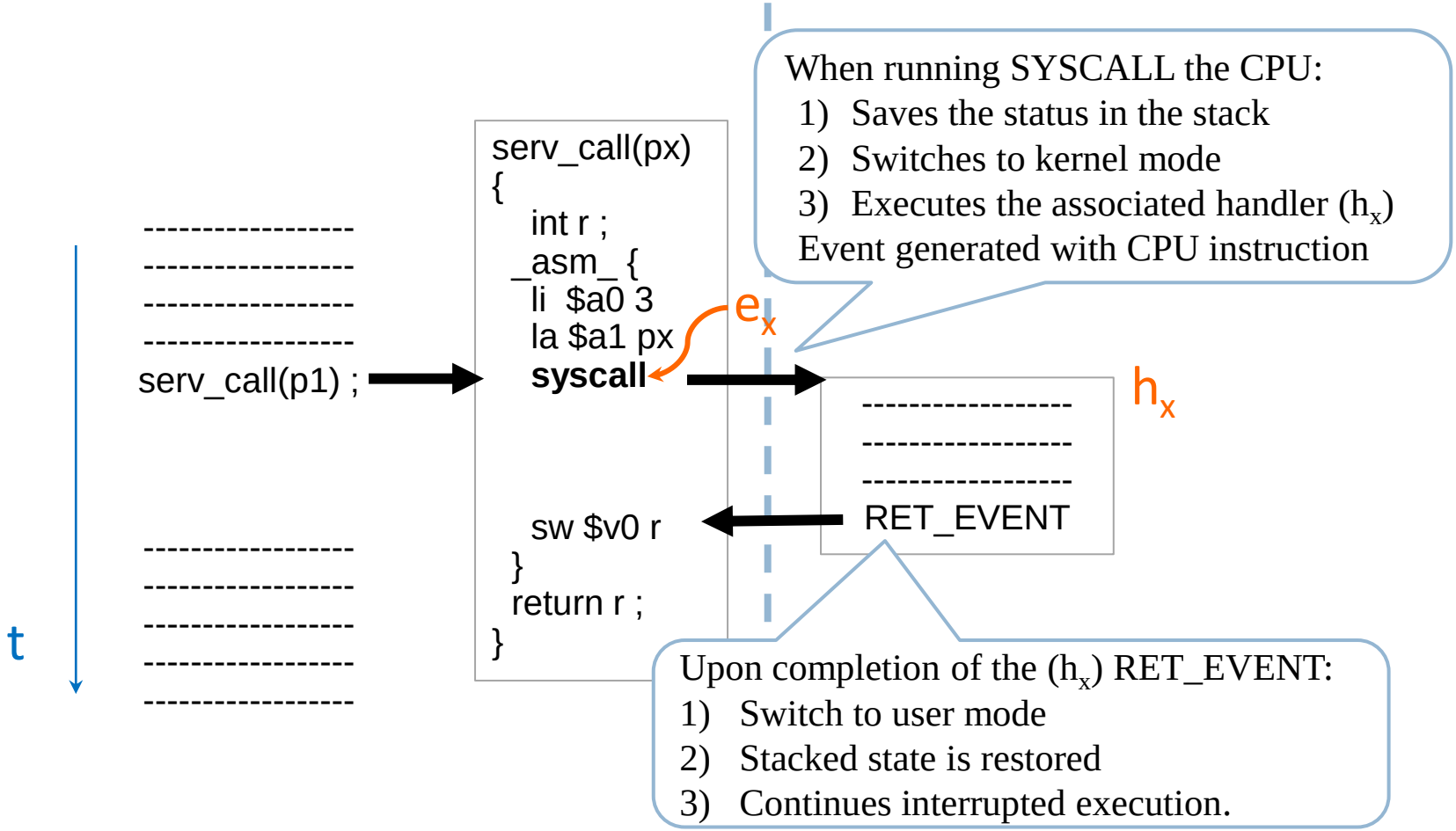
Execution of a service request
execution (general)

Alejandro Calderón Mateos

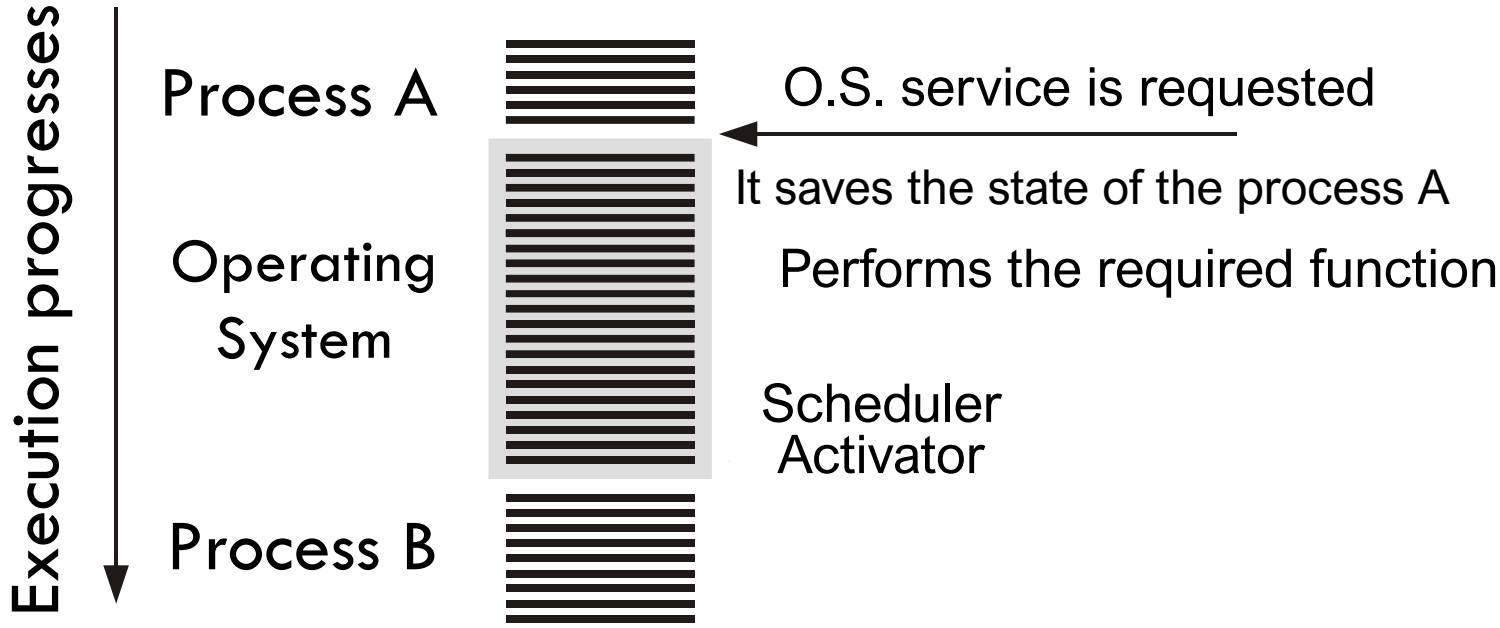


Execution of a service request
execution (general)

Alejandro Calderón Mateos



Phases in the activation of the Operating System



System calls
treatment in Linux (1/7)

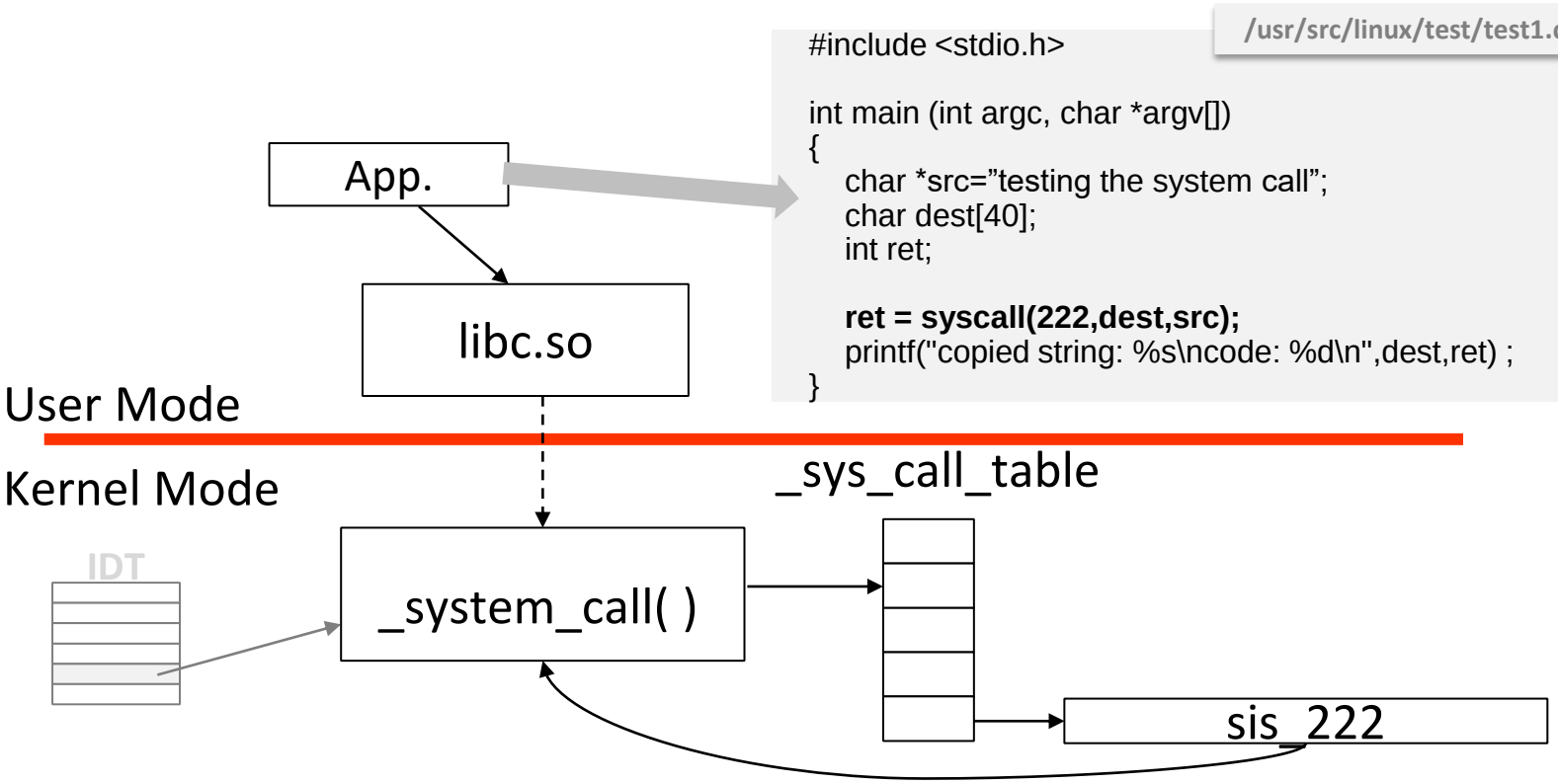
Alejandro Calderón Mateos

/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
...
```

System calls
treatment in Linux (2/7)

Alejandro Calderón Mateos



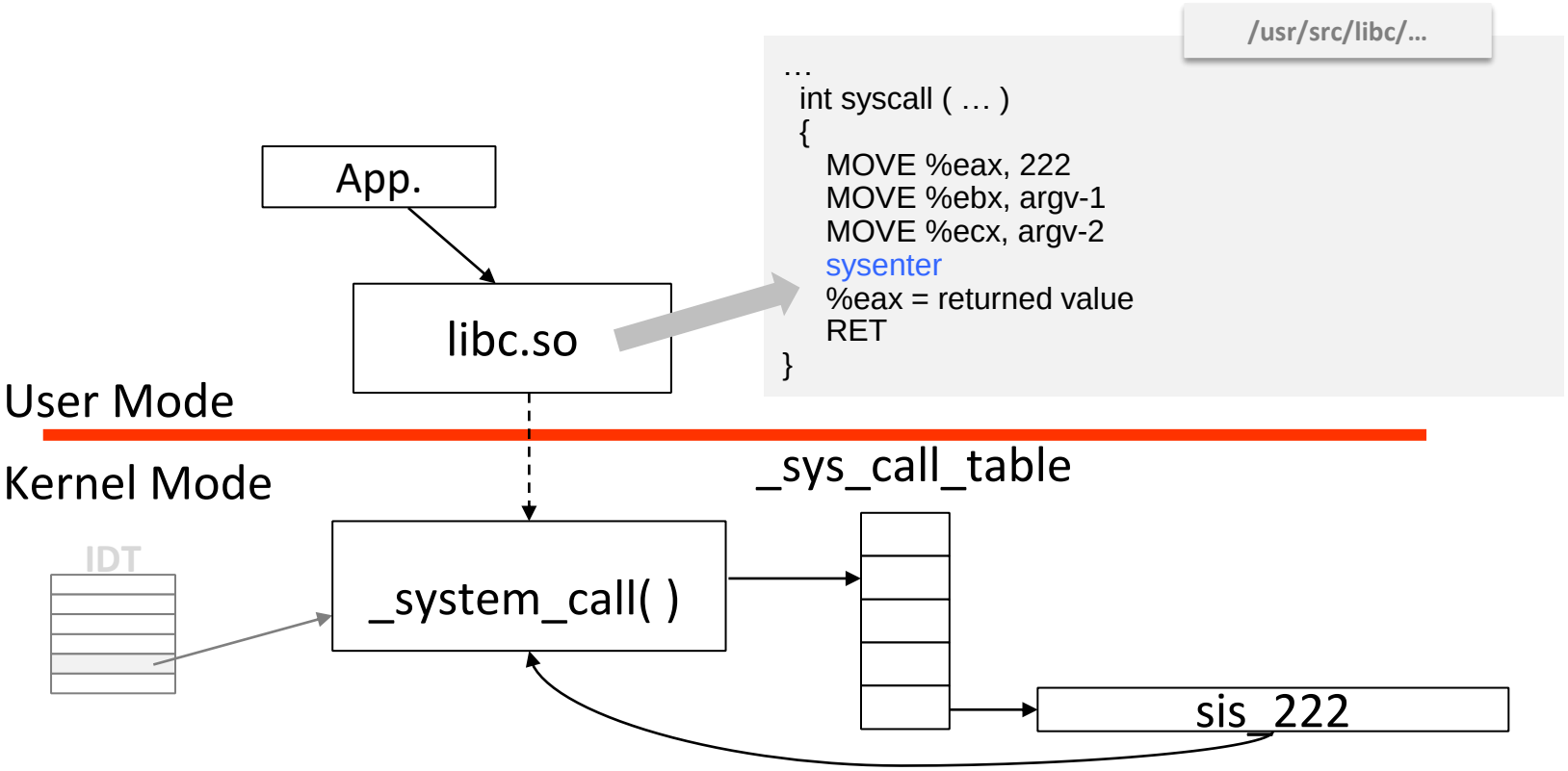
/usr/src/linux/test/test1.c

```
#include <stdio.h>

int main (int argc, char *argv[])
{
    char *src="testing the system call";
    char dest[40];
    int ret;

    ret = syscall(222,dest,src);
    printf("copied string: %s\ncode: %d\n",dest,ret) ;
}
```

System calls
treatment in Linux (3/7)

- Each O.S. service corresponds to a function (from API).
- This function encapsulates invocation of the service: parameters, trap, return …

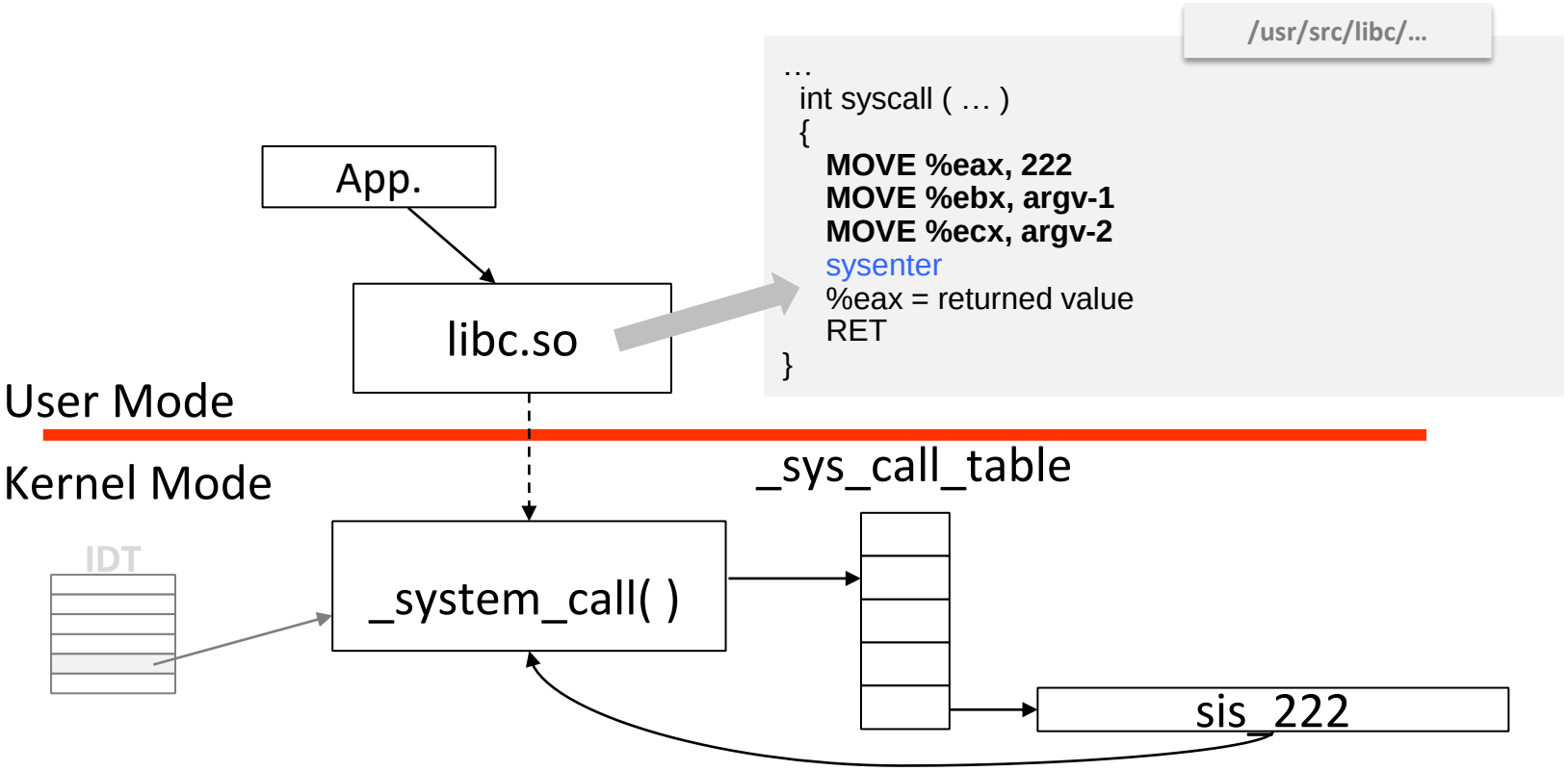
Alejandro Calderón Mateos



System calls
treatment in Linux (3/7)

- Parameter passing by register, stack or memory zone passed by register.
- Parameter 1: service identifier

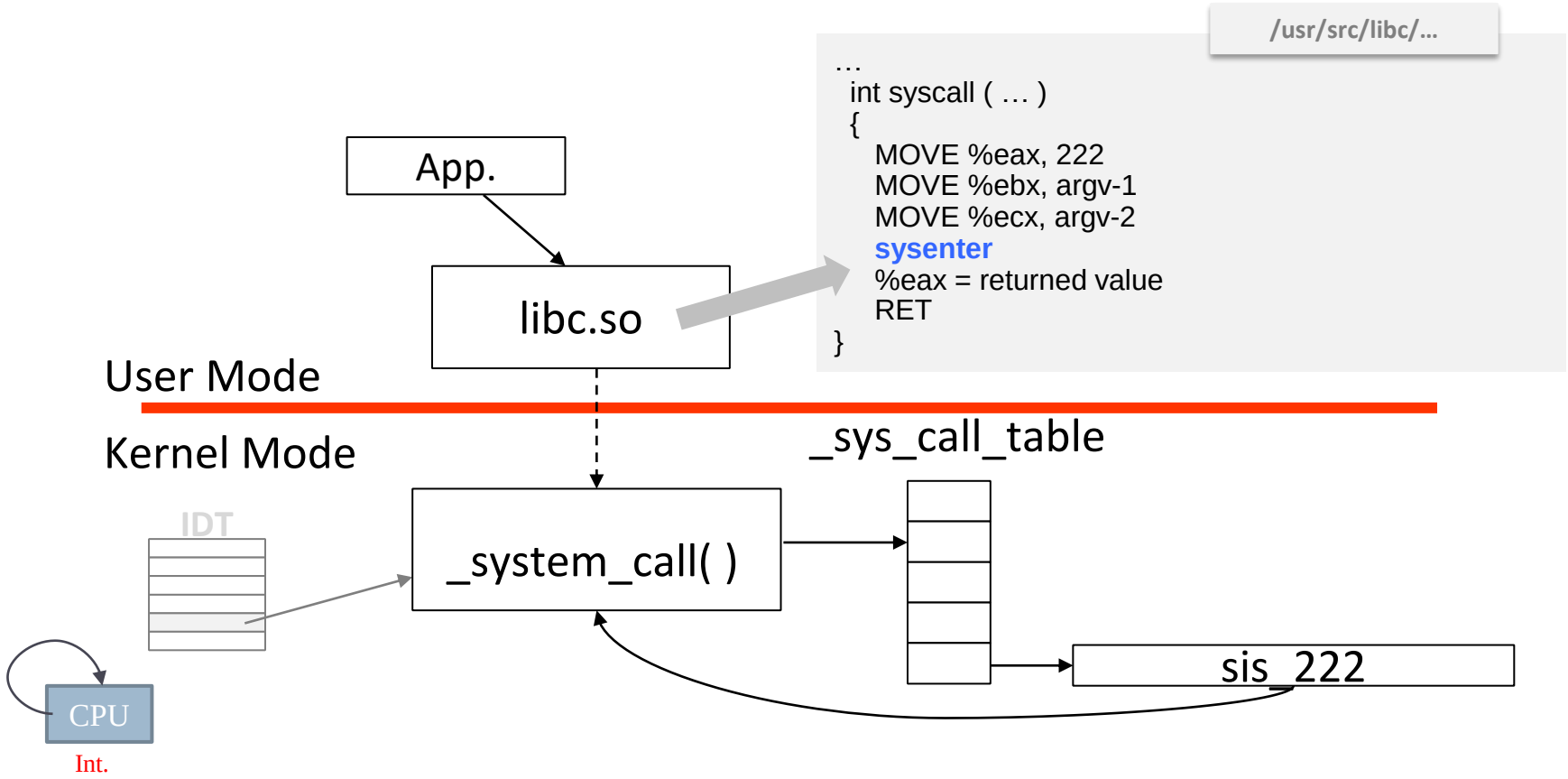
Alejandro Calderón Mateos



System calls
treatment in Linux (3/7)

- The trap (sysenter on x86 CPUs) is an instruction that generates an event with hardware interrupt-like treatment.

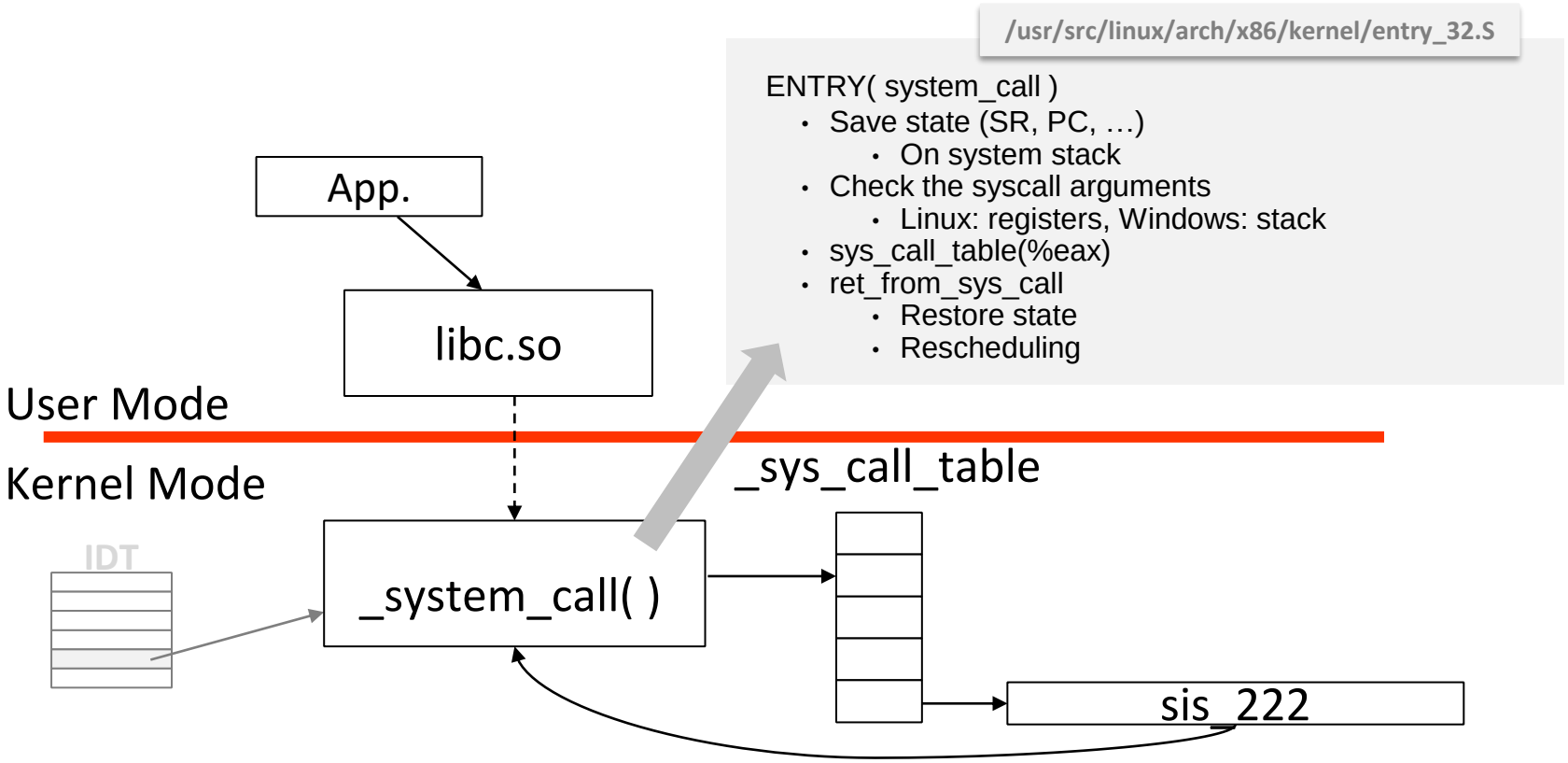
Alejandro Calderón Mateos



System calls
treatment in Linux (4/7)

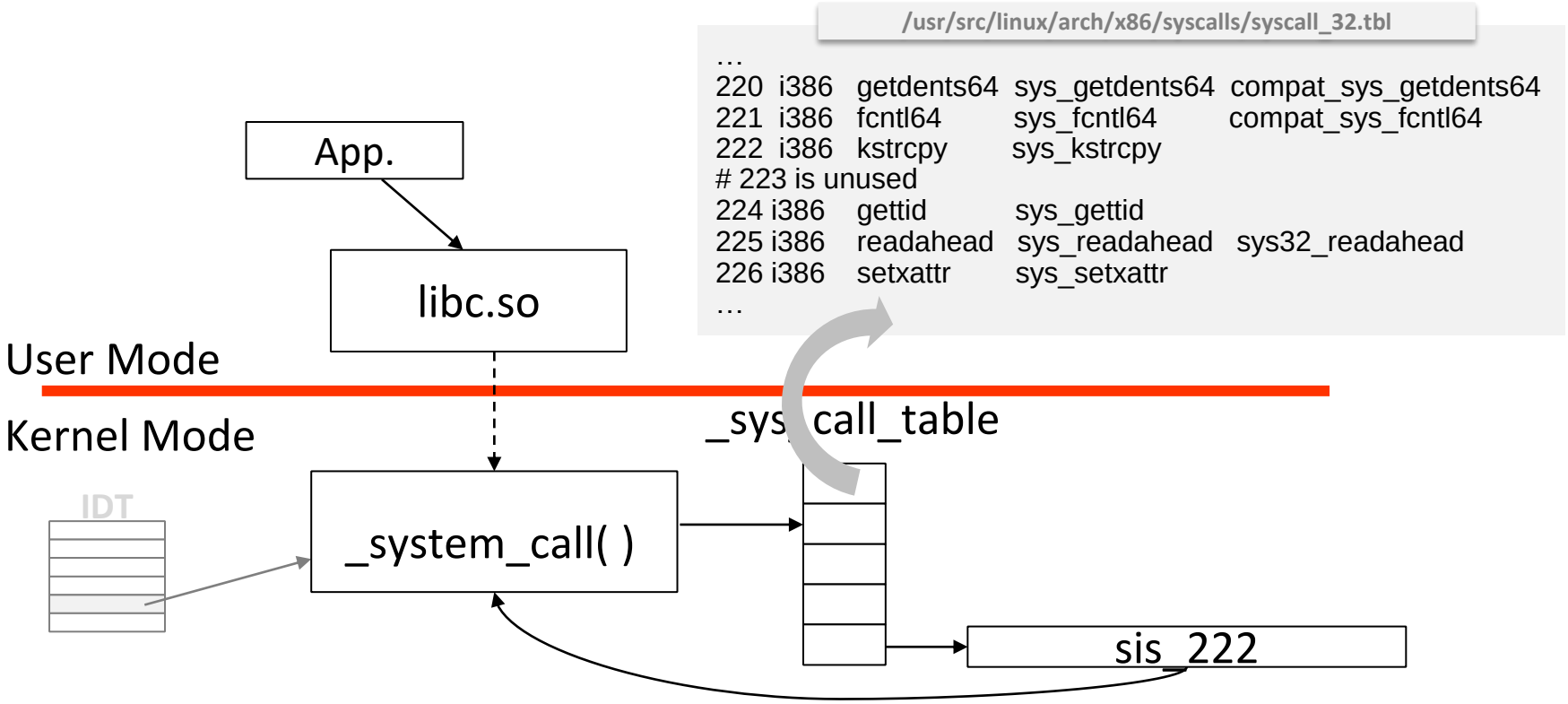
- Checks parameters, determines function in O.S. from the identifier (index in _sys_call_table) and invokes.

Alejandro Calderón Mateos



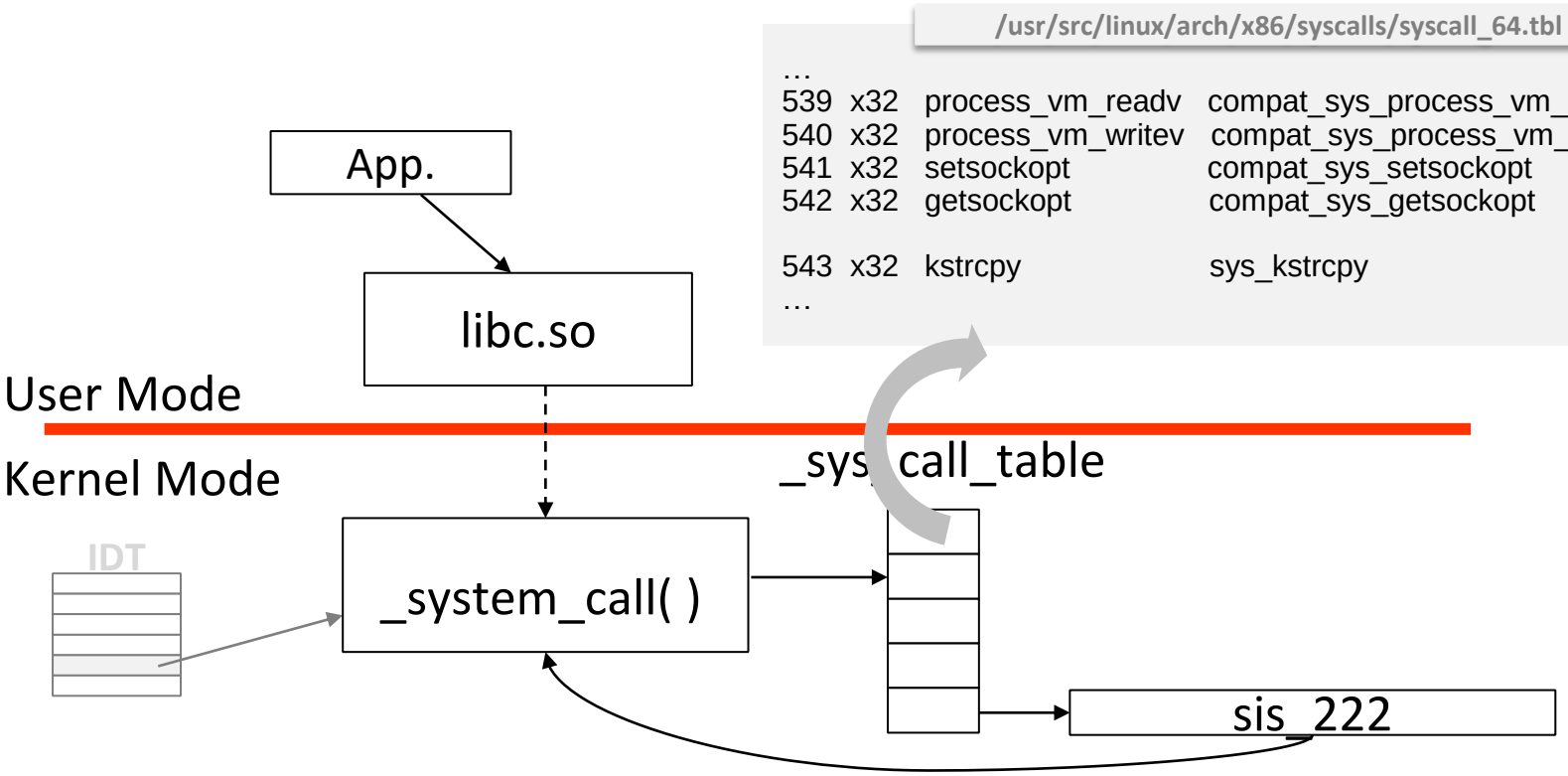
System calls
treatment in Linux (5/7)

Alejandro Calderón Mateos



System calls
treatment in Linux (6/7)

Alejandro Calderón Mateos

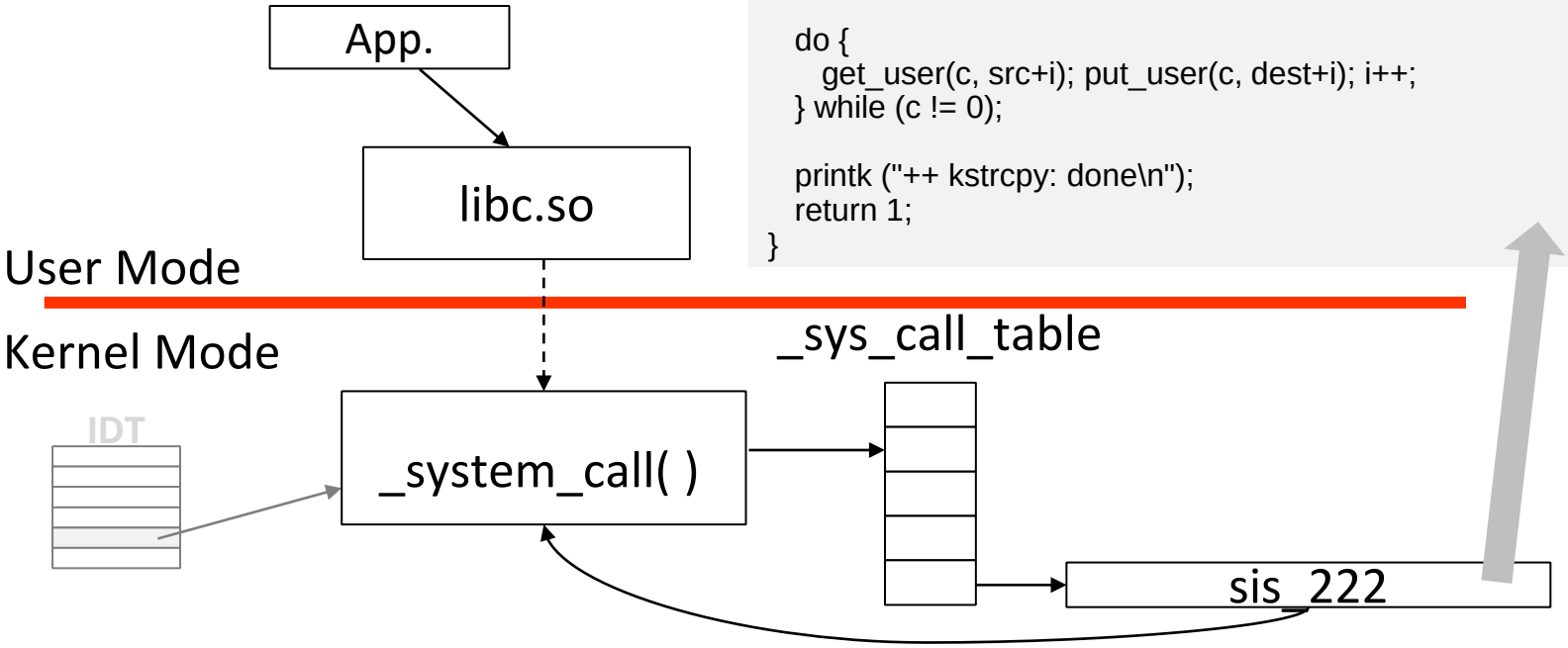


System calls
treatment in Linux (7/7)

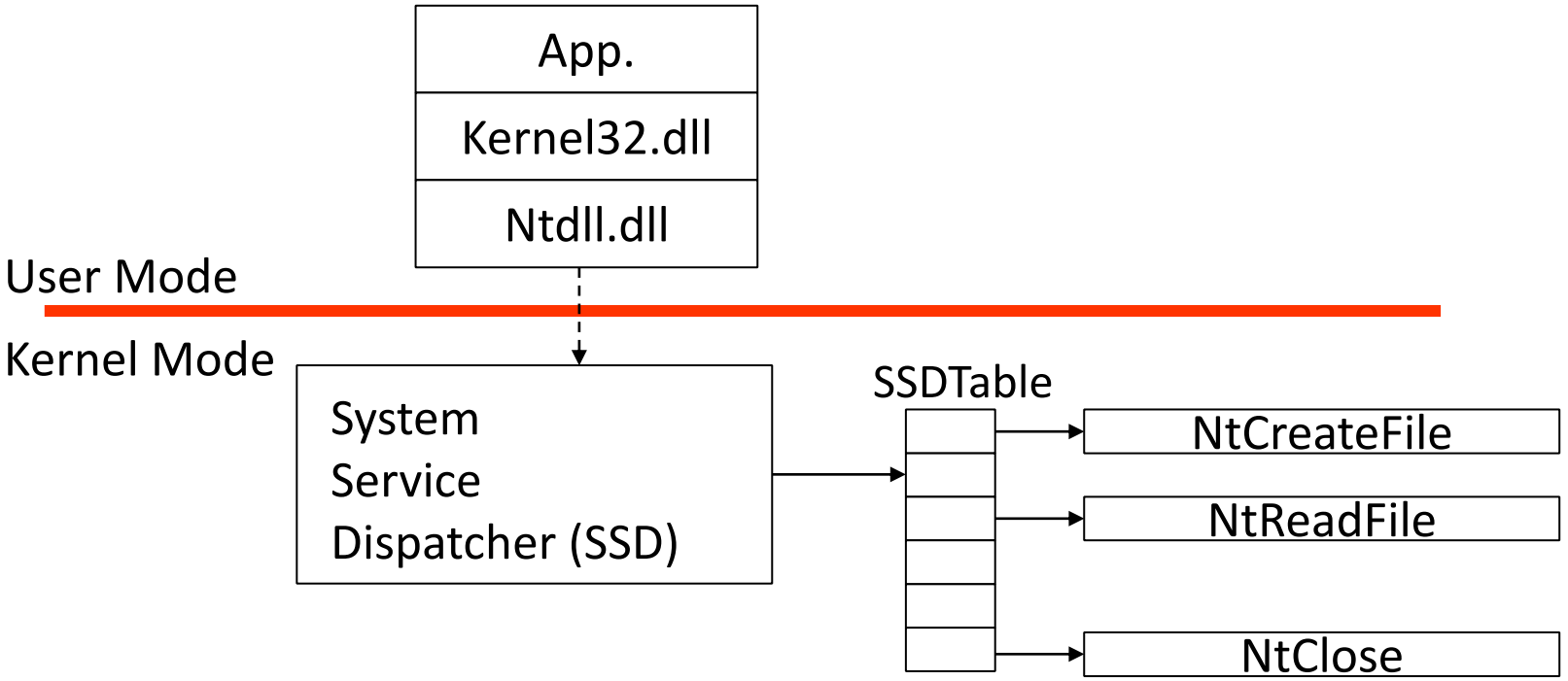
Alejandro Calderón Mateos

/usr/src/linux/kernel/sys.c

```
…
SYSCALL_DEFINE2(kstrcpy, char *, dst, char *, src)
{
   int i=0; char c;

   do {
      get_user(c, src+i); put_user(c, dest+i); i++;
   } while (c != 0);

   printk ("++ kstrcpy: done\n");
   return 1;
}
```



System calls
treatment in Windows



Programmer interface

- The set of functions that provide the OS services (encapsulating the calls) is the programmer's interface.
  - This interface provides the user's view of the operating system as an extended machine.
  - Best to use standard interface specifications.
- Each operating system may offer one or more interfaces:
  - Linux: POSIX
  - Windows: Win32, POSIX

POSIX standard

- IEEE Standard Operating System Interface.
- **Goal**: portability of applications across different platforms and operating systems.
- It is **NOT** an implementation. It only defines an interface.
- Different standards:
  - 1003.1 Basic OS services
  - 1003.1a Extensions to basic services
  - 1003.1b Real-time extensions
  - 1003.1c Lightweight process extensions (threads)
  - 1003.2 Shell and utilities
  - 1003.2b Additional utilities

POSIX features

- Short, lowercase function names:
  - fork
  - read
  - close
- The functions normally return 0 in case of success or -1 in case of error.
  - errno variable.
- Resources managed by the operating system are referenced by descriptors (integers)

UNIX 03

- Single Unix Specification (SUS)
  - V1 (UNIX 95), V2 (UNIX 98), V3 (UNIX 03) & V4 (UNIX V7)
- It is an evolution that encompasses POSIX and other standards (X/Open XPG4, ISO C).
  - It includes not only the programming interface, but also other aspects:
    - Services offered.
    - Mandate interpreter.
    - Available utilities.
- Example of UNIX 03: AIX, EulerOS, HP-UX, macOS

Contents

- Introduction to system calls
- System call mechanism
- Calls for services of:
  - Process management
  - Management of files and directories

Process management

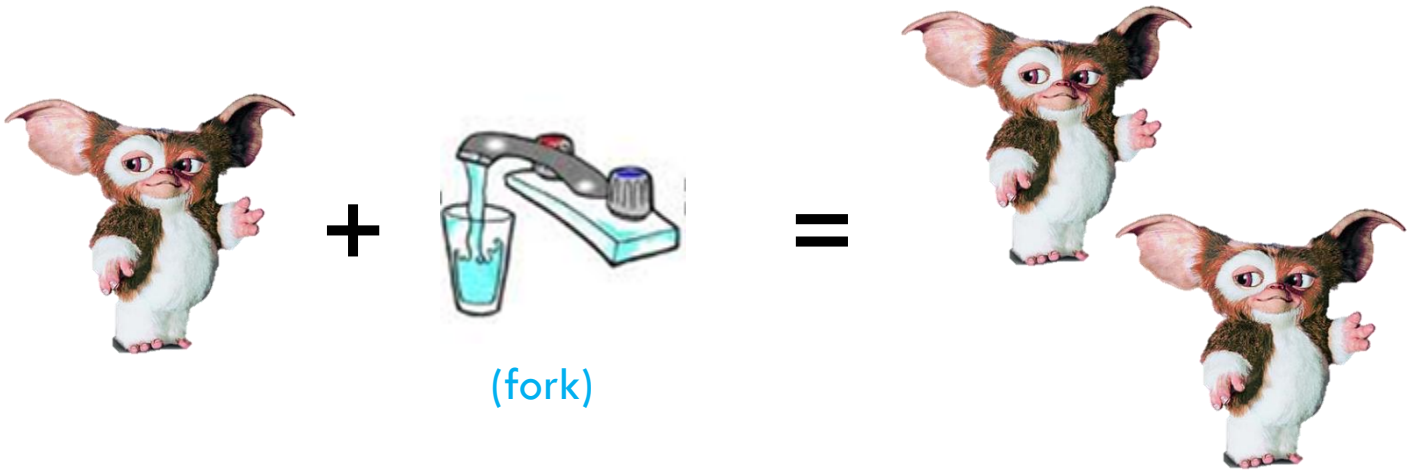
Alejandro Calderón Mateos

- **Understanding fork, exec, exit and wait**
- simple fork+exec+exit
- multiple fork+exec+exit

Fork

Alejandro Calderón Mateos

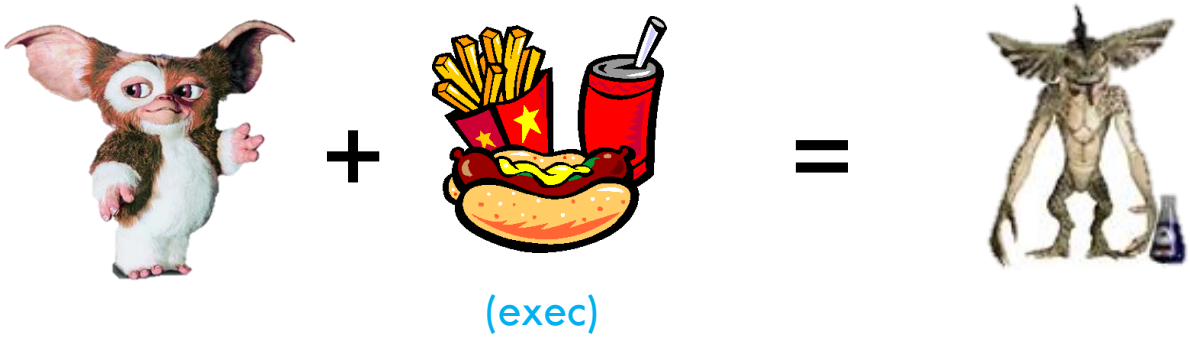
- Create a "clone" of a process



Exec

Alejandro Calderón Mateos

- Changes the image of a process

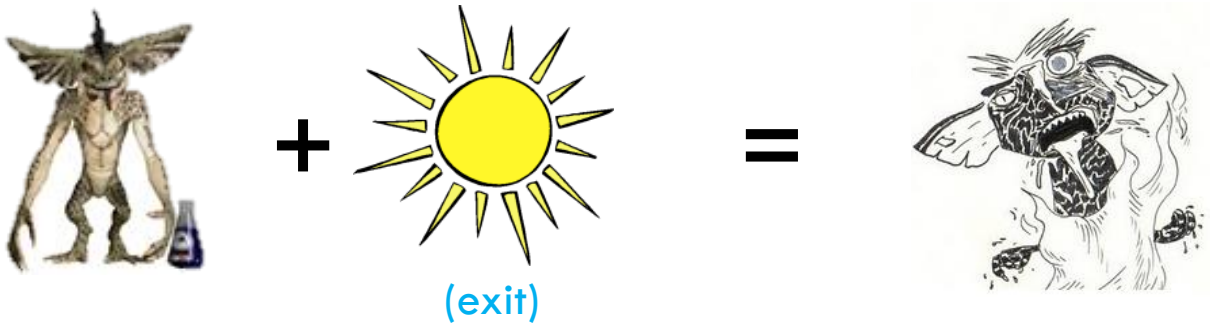


(exec)

Exit

Alejandro Calderón Mateos

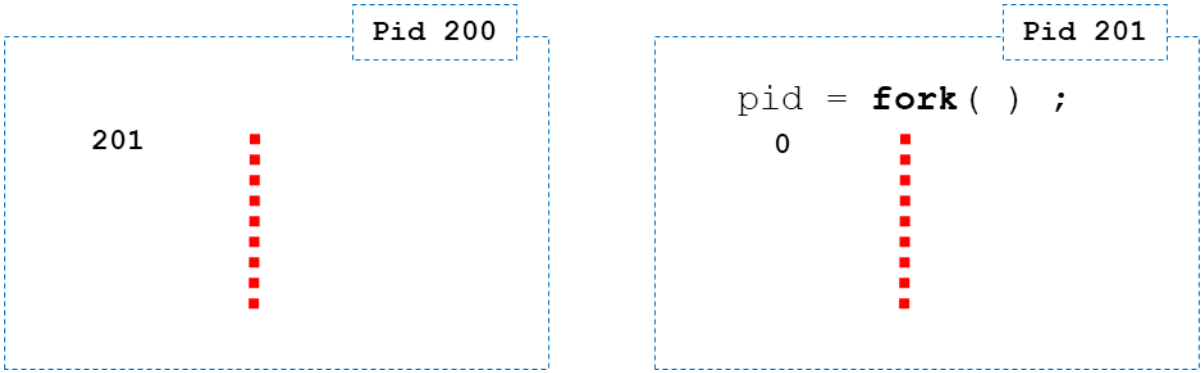
- End the execution of a process



Fork

Alejandro Calderón Mateos

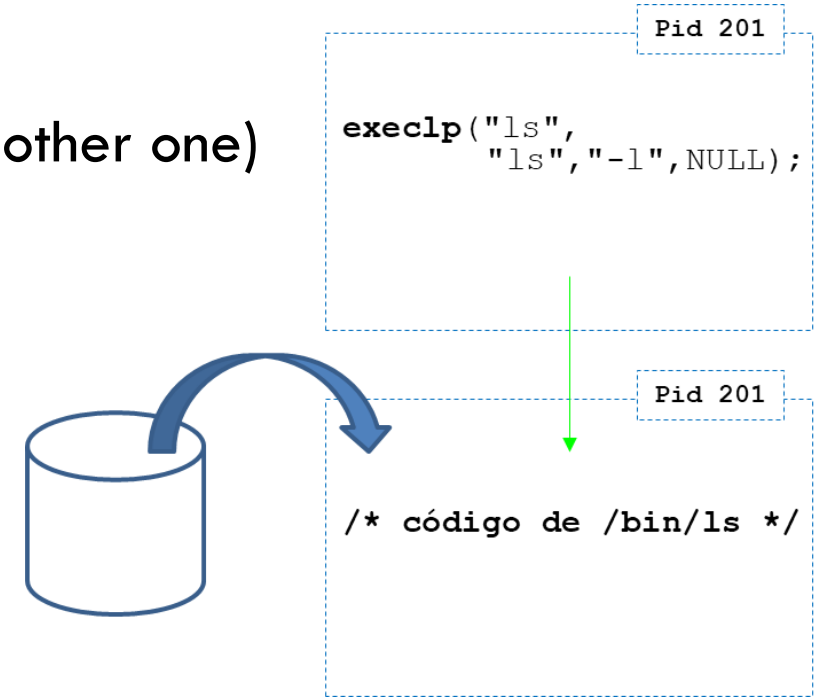
- Creates a "clone" of a process:
  - Same except for small differences:
    the father gets back the PID of the son, and the son gets back zero.



Exec

Alejandro Calderón Mateos

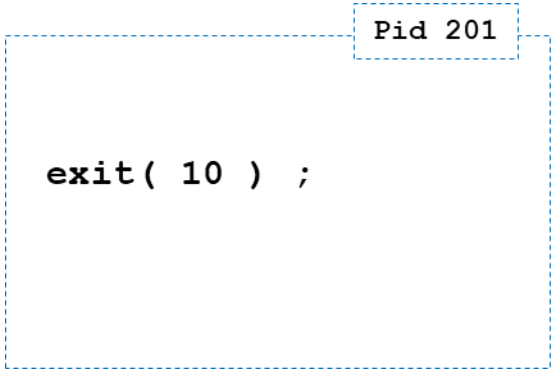
- Changes the image of a process:
  - If all goes well,
    this function does not return
    (the code is replaced by another one)



Exit

Alejandro Calderón Mateos

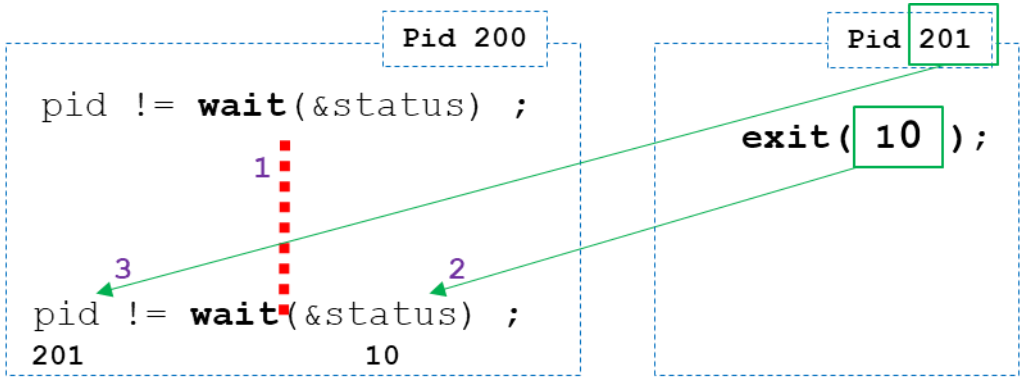
- End execution of a process
  - The parameter is an integer value that is often used as a diagnostic code: if everything ran well, if there was a minor problem, if there was a major error, etc.



Wait

Alejandro Calderón Mateos

- Has three effects:
  1. Block the execution of the father until one of his children completes his execution.
  2. Stores in its parameter the returned value by the child.
  3. Returns the pid of the terminated child.



Process management

- Understanding fork, exec, exit and wait
- **simple fork+exec+exit**
- multiple fork+exec+exit

Review

fork() + exec()

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);
}
```

Review

fork() + exec()

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);
}
```

Review

fork() + exec()

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);
}
```

Review

fork() + exec()

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("/bin/ls","ls","-l",NULL);
      exit(-1);
   }
   else
   {
      while (pid != wait(&status));
   }

   exit(0);
}
```

Father

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

main() {
   pid_t pid;
   int status;

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

   exit(0);
}
```

Son

Review

fork() + exec()

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);
}
```

Review

fork() + exec()

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);
}
```

Review

fork() + exec()

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);
}
```

Review

fork() + exec()

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() {

    /* ls code */

    exit( 0 );
}
```

Review

wait() + exit()


```
/* 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() {

    /* ls code */

    exit( 0 );
}
```

Review

wait() + exit()


```c
/* 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);
}
```

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

main() {

    /* ls code */

    exit( 0 );
}
```

Review

wait() + exit()

Alejandro Calderón Mateos

```c
/* 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);
}
```

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

main() {

    /* ls code */

    exit( 0 );
}
```

Review

wait() + exit()

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);
}
```

0

Review

wait() + exit()

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);
}
```

Review

wait() + exit()

Alejandro Calderón Mateos

Process management

Alejandro Calderón Mateos

- Understanding fork, exec, exit and wait
- simple fork+exec+exit
- **multiple fork+exec+exit**

Review

multiple processes (blocking)

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);
    }
}
```

Review

multiple processes (blocking)


```
#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);
    }
}
```

Review

multiple processes (blocking)

Alejandro Calderón Mateos

```c
#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);
    }
}
```

```c
#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));
    }
}
```

Review

multiple processes (blocking)

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));
    }
}
```

Review

multiple processes (blocking)

Alejandro Calderón Mateos

```c
#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);
    }
}
```

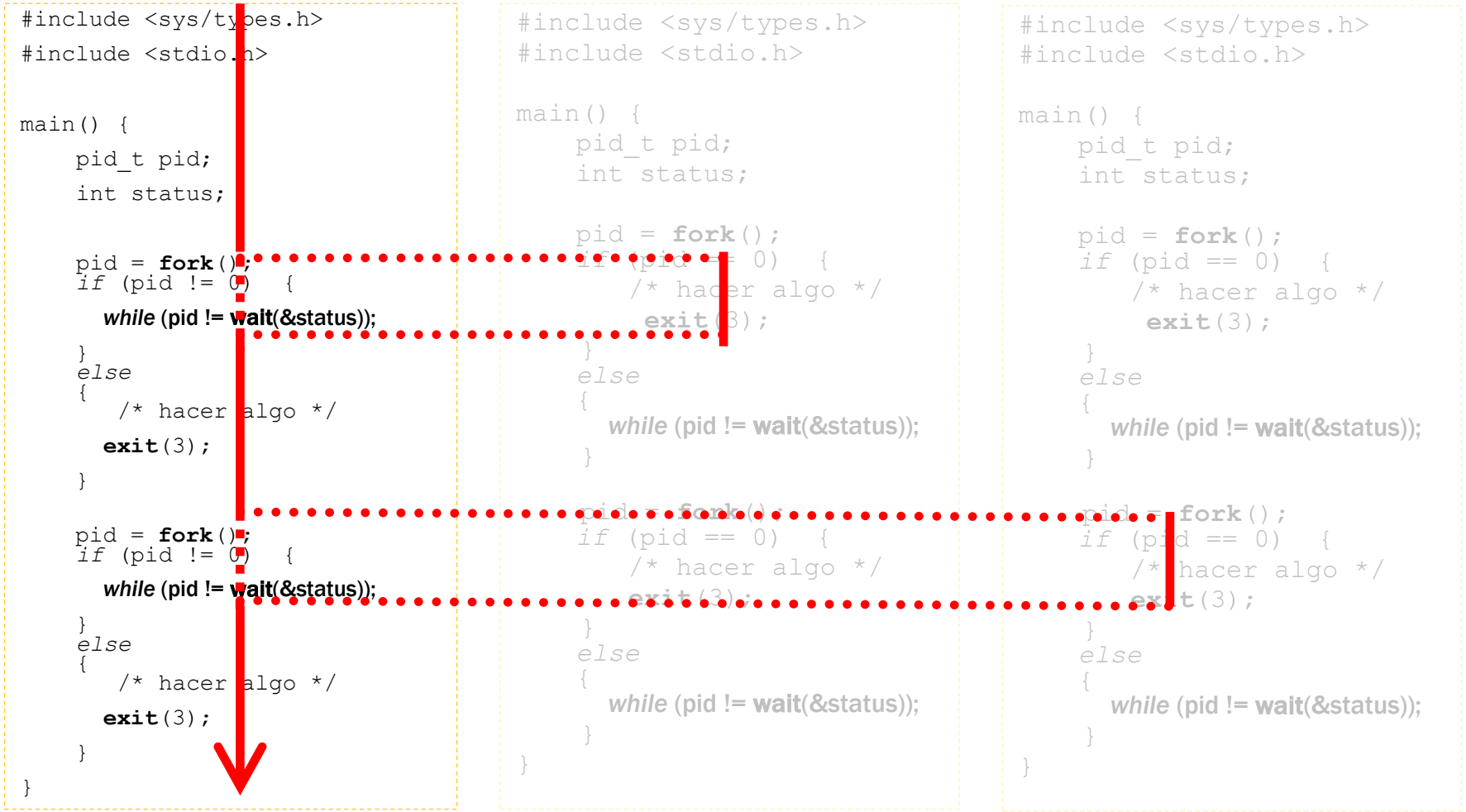
```c
#include <sys/types.h>
#include <stdio.h>

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));
    }
}
```

```c
#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));
    }
}
```

Review

multiple processes (blocking)

Alejandro Calderón Mateos



Review

multiple processes (blocking)

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));
}
```

Review

multiple processes (blocking)

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));
}
```

Review

multiple processes (blocking)

Alejandro Calderón Mateos

```c
#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));
}
```

```c
#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));
}
```

Review

multiple processes (blocking)

Alejandro Calderón Mateos

```c
#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));
}
```

```c
#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));
}
```

```c
#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));
}
```

Review

multiple processes (blocking)

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));
}
```

Review

multiple processes (blocking)

Alejandro Calderón Mateos

```c
#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));
}
```

```c
#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));
}
```

```c
#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));
}
```

Review

multiple processes (blocking)

Alejandro Calderón Mateos

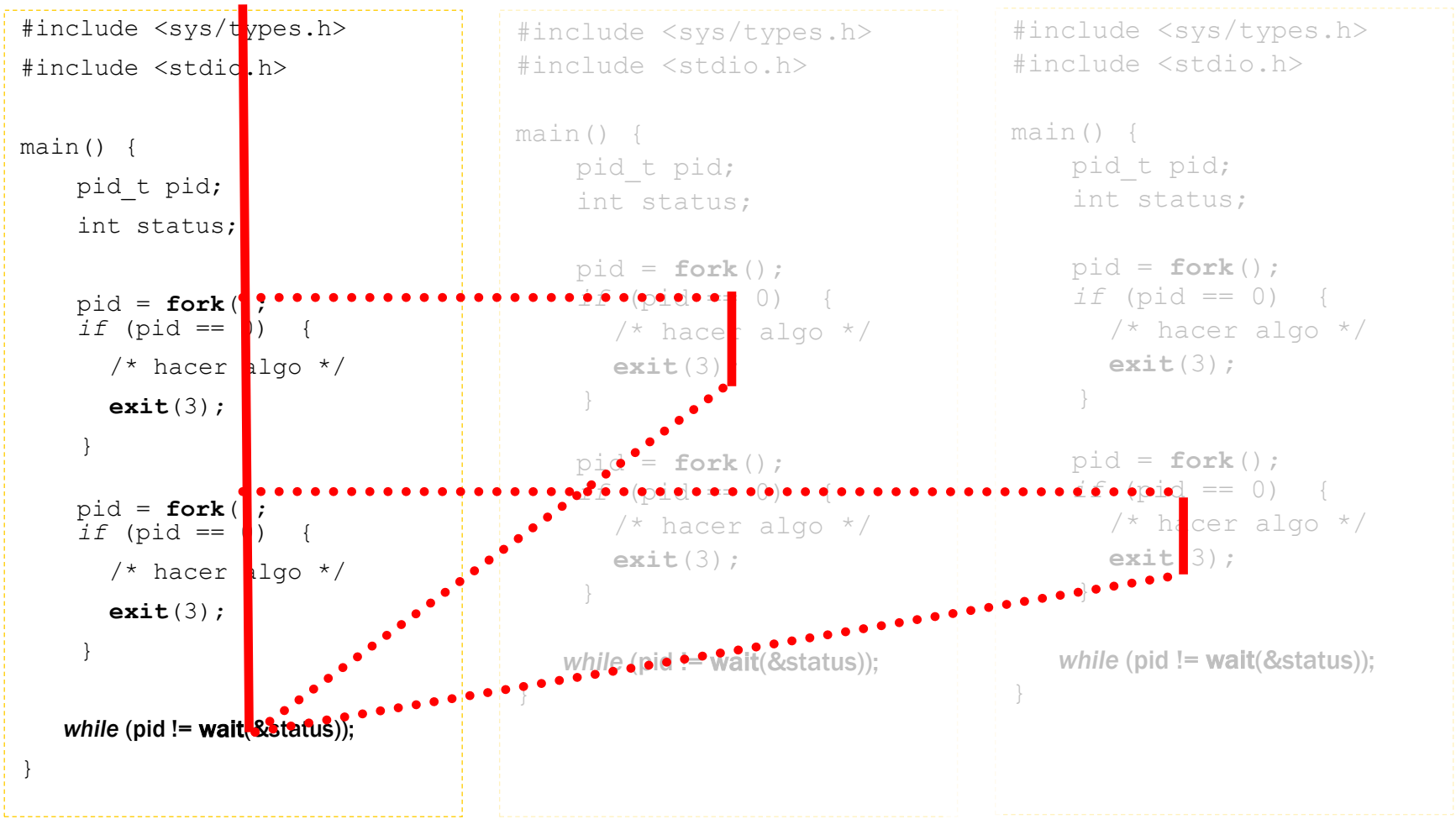
```
#include <sys/types.h>
#include <stdio.h>

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));
}
```

Review

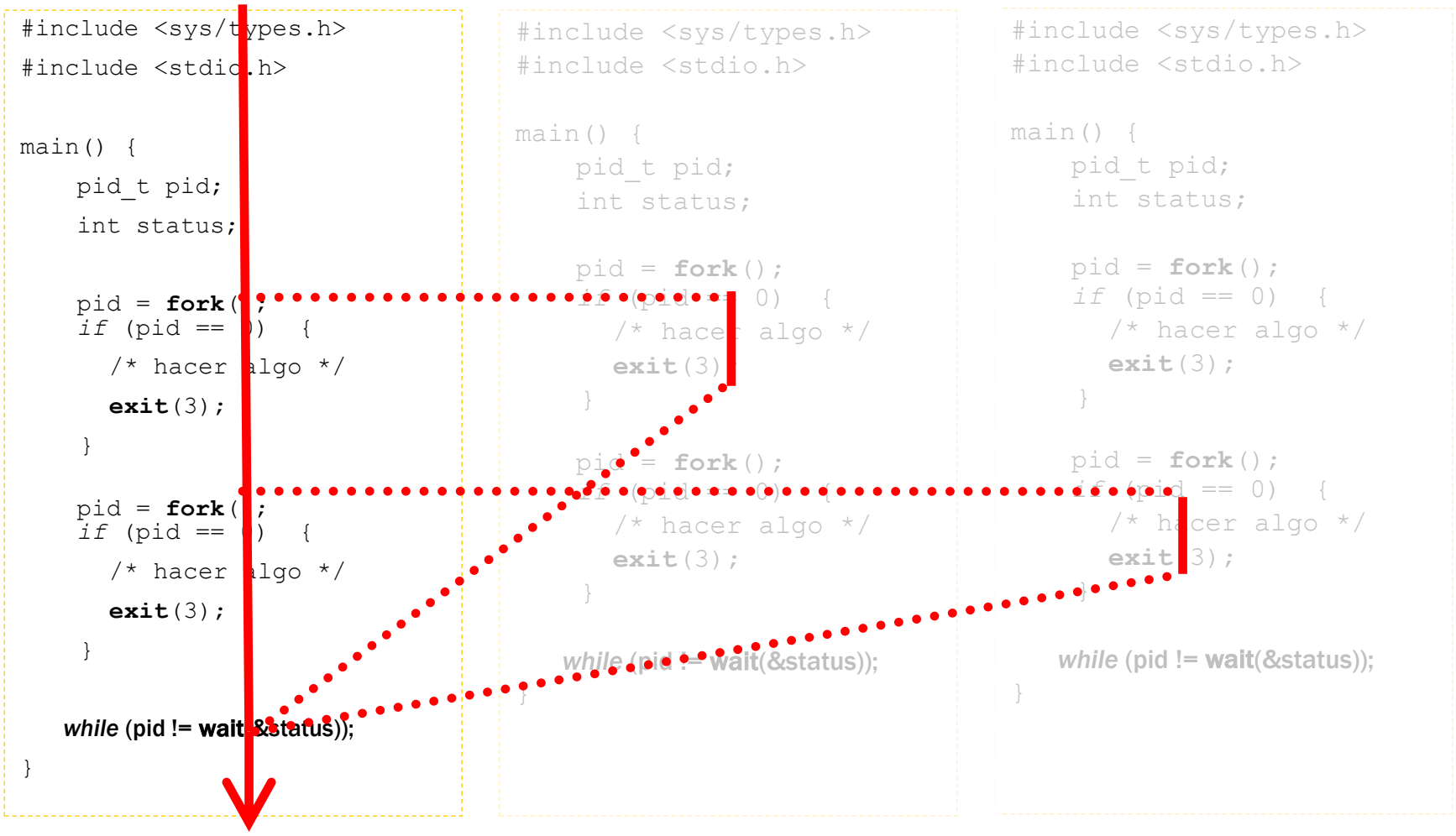
multiple processes (blocking)



Review

multiple processes (blocking)

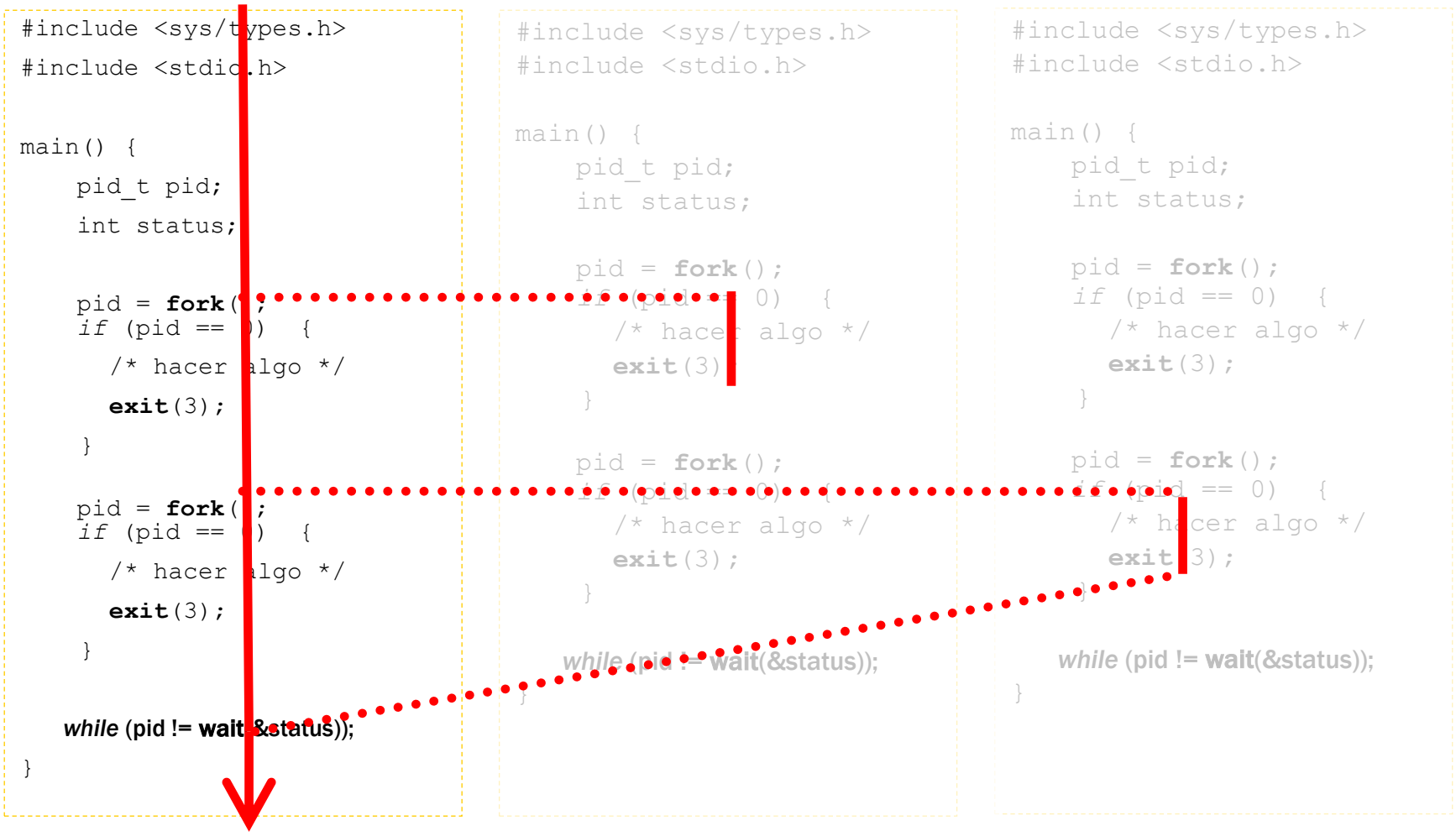
Alejandro Calderón Mateos



Review

if 2nd son ends before 1st son => zombie (father does not wait for him)

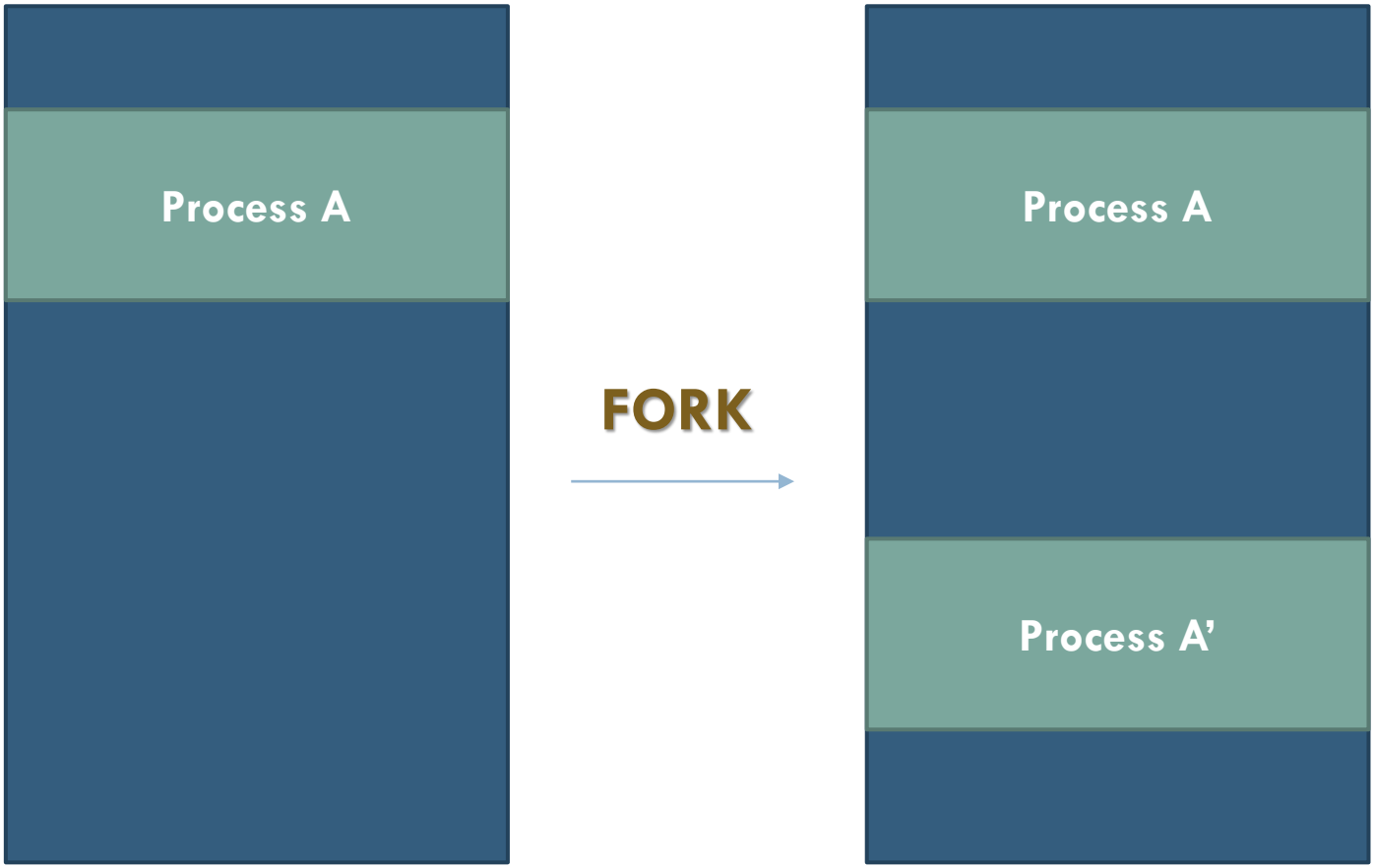
Alejandro Calderón Mateos



Fork service

Service	`#include <unistd.h>` **`pid_t fork(void);`**
Arguments	
Returns	▫ -1 in case of error. ▫ In the parent process: the identifier of the child process. ▫ In the child process: 0
Description	▫ Duplicates the process that invokes the call. ▫ The parent and child processes continue executing the same program. ▫ The child process inherits the open files from the parent process. ▫ The descriptors of open files are copied. ▫ Pending alarms are deactivated.

Fork service

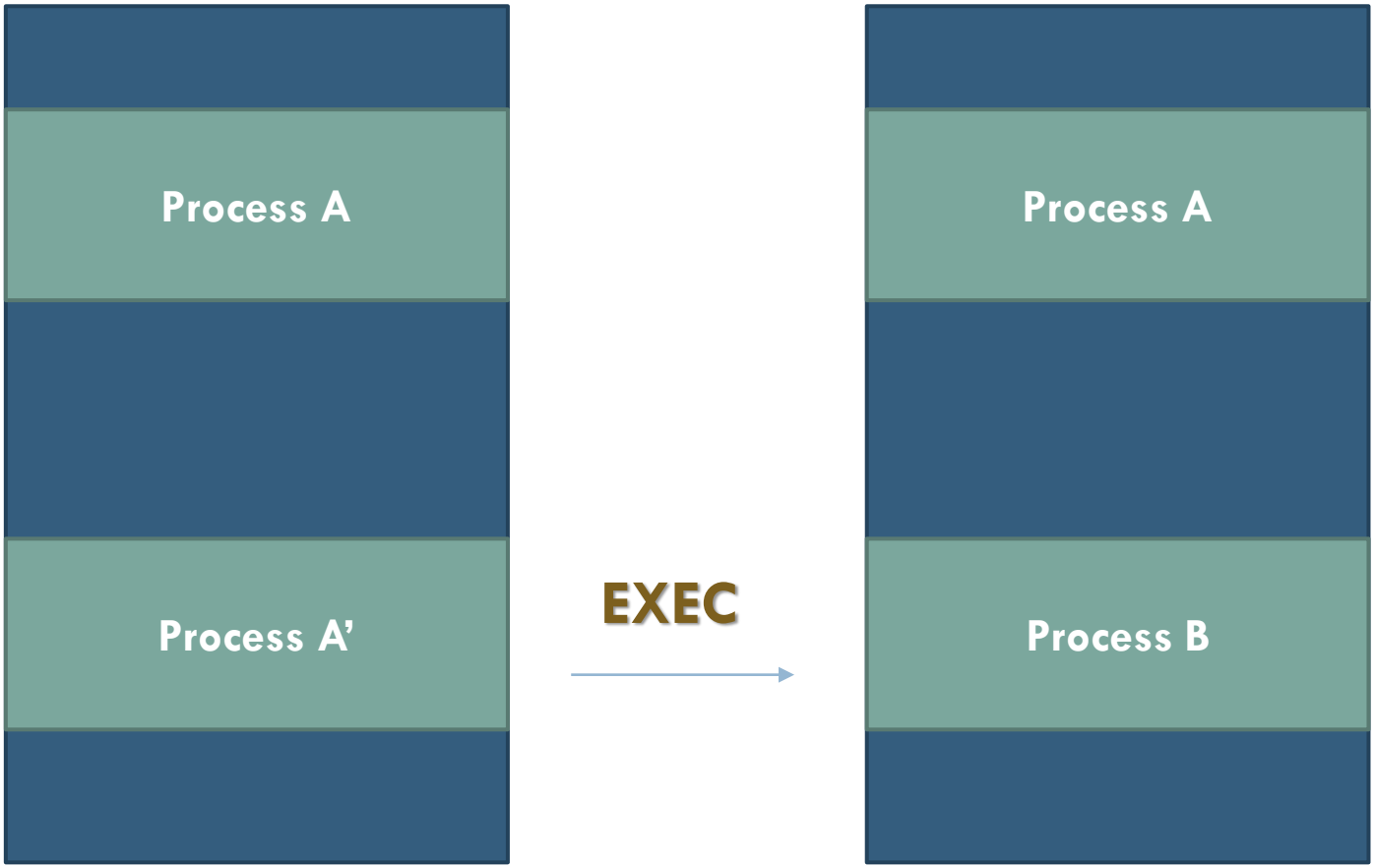


Exec service

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

Arguments	▫ `path`: Path to the executable file. ▫ `file`: Looks for the executable file in all directories specified by PATH
Returns	▫ Returns -1 in case of error, **otherwise no return.**
Description	▫ Changes the image of the current process. ▫ The same process executes another program. ▫ Open files remain open. ▫ Signals with the default action will continue by default, signals with handler will take the default action.

Exec service

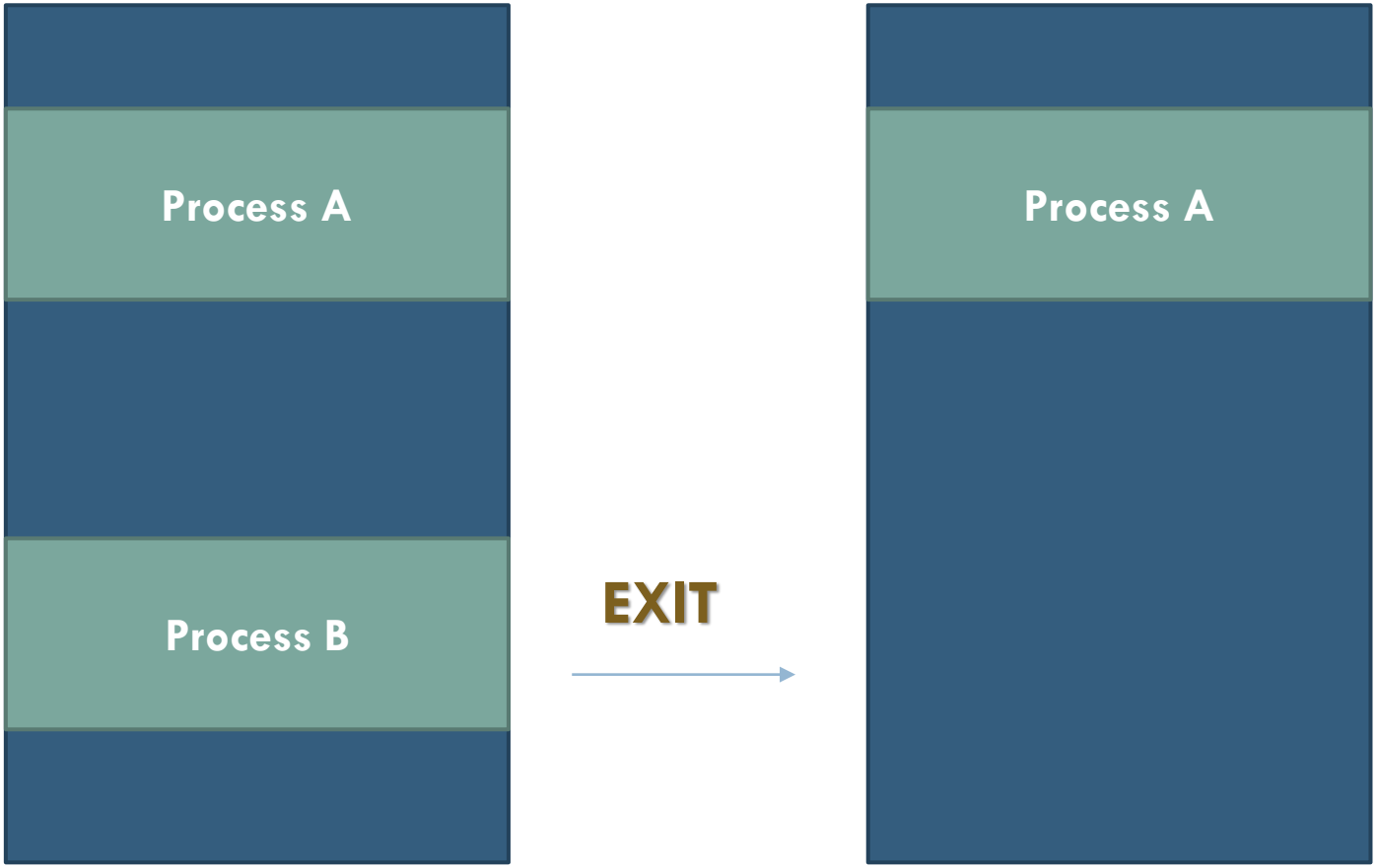


Exit service

Service	`#include <unistd.h>`
	
	`void exit(status);`

Arguments	▫ status: value retrieved by the parent in the wait() call
Returns	
Description	▫ The execution of the process ends. ▫ All open file descriptors are closed. ▫ All process resources are released. ▫ The PCB (process control block) of the process is released

Exit service



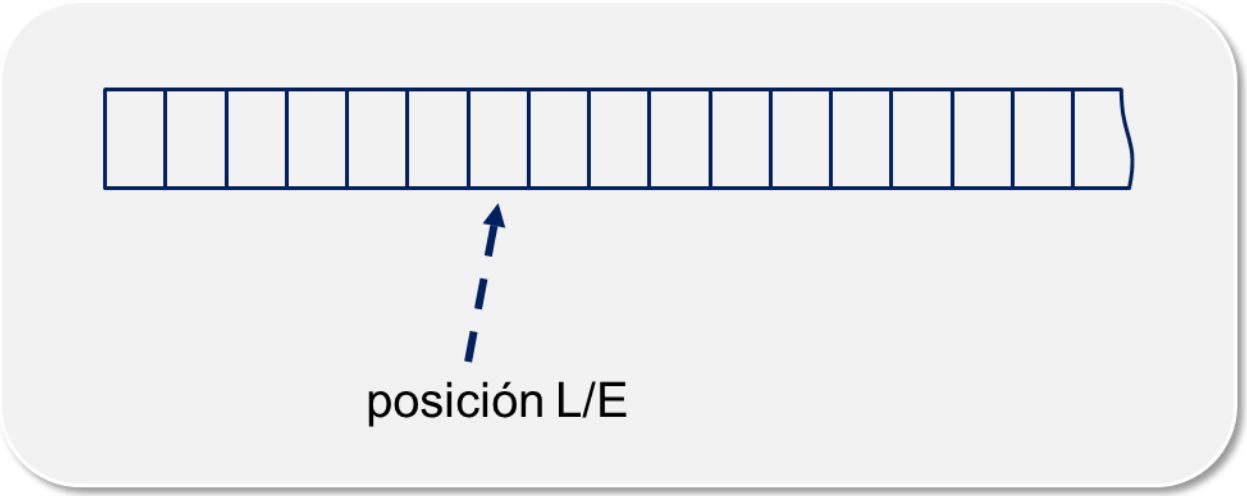
Contents

- Introduction to system calls
- System call mechanism
- Calls for services of:
  - Process management
  - Management of files and directories

File

Alejandro Calderón Mateos

- Set of related information
  that has been defined by its creator.
- Usually the content is represented by
  a sequence or strip of bytes (logical view):

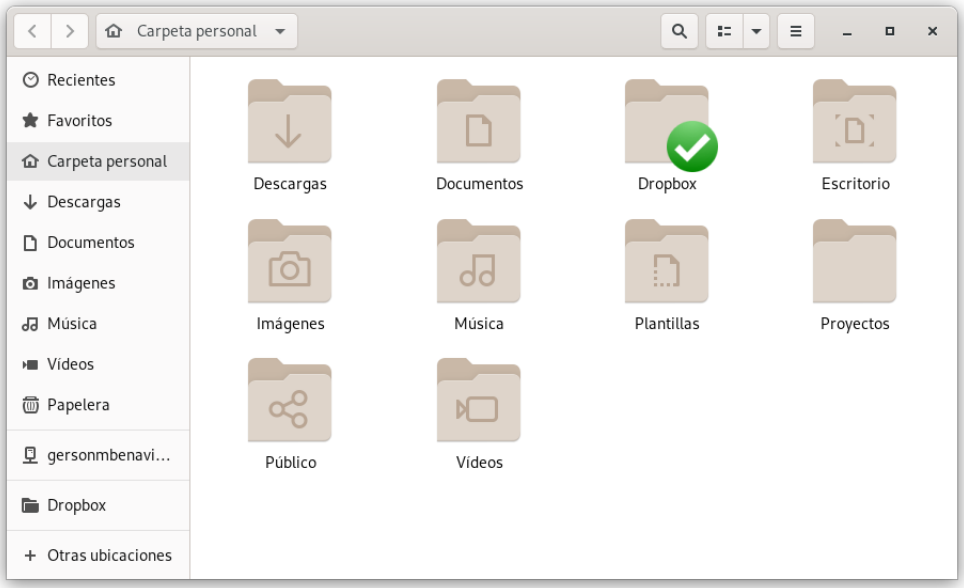


Directory (folder)

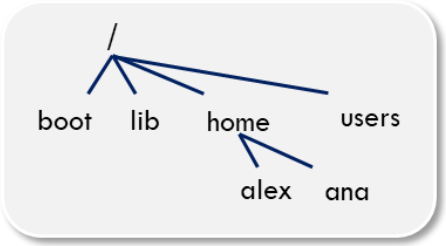
https://es.wikipedia.org/wiki/GNOME_Archivos#/media/Archivo:GNOME_Archivos_3_36_3.png

Alejandro Calderón Mateos

- Data structure that allows grouping a set of files according to the user's criteria



Name of files and directories



- Hierarchical names for identification:
  - List of names until the directory/file is reached.
  - Names are separated by a special character:
    - / in LINUX and \ in Windows
- Special directory names:
  - . Current directory or working directory (E.g.: cp /alex/correo.txt .)
  - .. Parent directory or previous directory (E.g.: ls ..)
  - ~ User home directory in UNIX (E.g.: ls –las ~ ; ls –las $HOME)
  - / Root directory in UNIX (E.g.: ls –las /)
- Two types of used name:
  - Absolute or **Full name** (begins with the root directory)
    - `/usr/include/stdio.h` (linux)
    - `c:\usr\include\stdio.h` (windows)
  - **Relative name** (is relative to the current directory, does not begin with root)
    - `stdio.h` assuming that `/usr/include` is the current directory.
    - `../include/stdio.h`

Typical attributes of a file/directory

- Name: identifier for the users of the file/directory (entry).
- Type: type of input (for systems that require it)
  - E.g.: extension (.exe, .pdf, etc.)
  - **File types: normal, directories, specials.**
- Location: identifier that helps to locate the device blocks that belong to the input.
- Size: current size of the entry.
- Protection: control of which user can read, write, etc.
- Day and time: time instant of last access, creation, etc. that allows monitoring the use of the entry.
- User identification: identifier of the creator, owner, etc.

POSIX services for files

generic operations for files

- creat (…) → **create:** creates a file (given name and attributes) and opens session.
- open (…) → **open:** opens a session with a file from its name.
- close (…) → **close:** closes work session with an open file.
- read (…) → **read:** reads data from a file open to a memory area.
- write (…) → **write:** writes to an open file from a memory area.
- lseek (…) → **seek:** Moves the pointer used to access the file, affecting subsequent operations.
- unlink (…) → **delete:** Deletes a file from its name.

- fcntl (…) → **file control:** Allows to manipulate the attributes of a file.
- dup (…)
- ftruncate (…)
- stat (…)
- utime (…)

File abstraction

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) ;
}
```

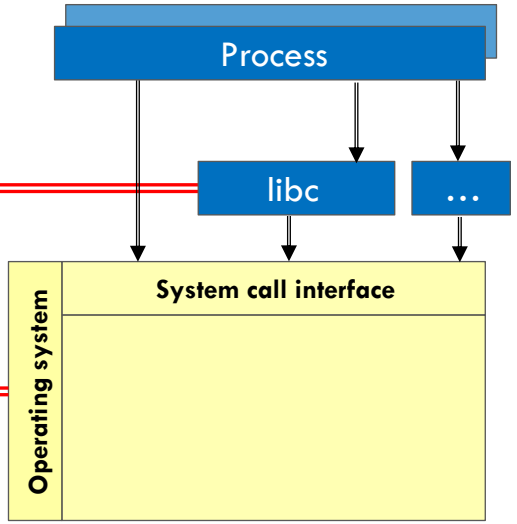
- A pointer associated with each open file is maintained.
  - Indicates the position from which the following operation is to be carried out.
- Most operations use file descriptors:
  - Identifies a work session with a file:
    - A number between 0 and “64K”.
    - Obtained when opening the file (open).
    - The rest of the operations identify the file by its descriptor.
  - Pre-defined descriptors:
    - 0: standard input
    - 1: standard output
    - 2: error output

System calls vs. system library

file system

Alejandro Calderón Mateos

- fopen, fclose, fwrite, fread, fseek y feof
- open, close, write, read y lseek



System calls vs. system library

write to file

Alejandro Calderón Mateos

```c
#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) ;
}
```

```c
#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) ;
}
```

System calls vs. system library

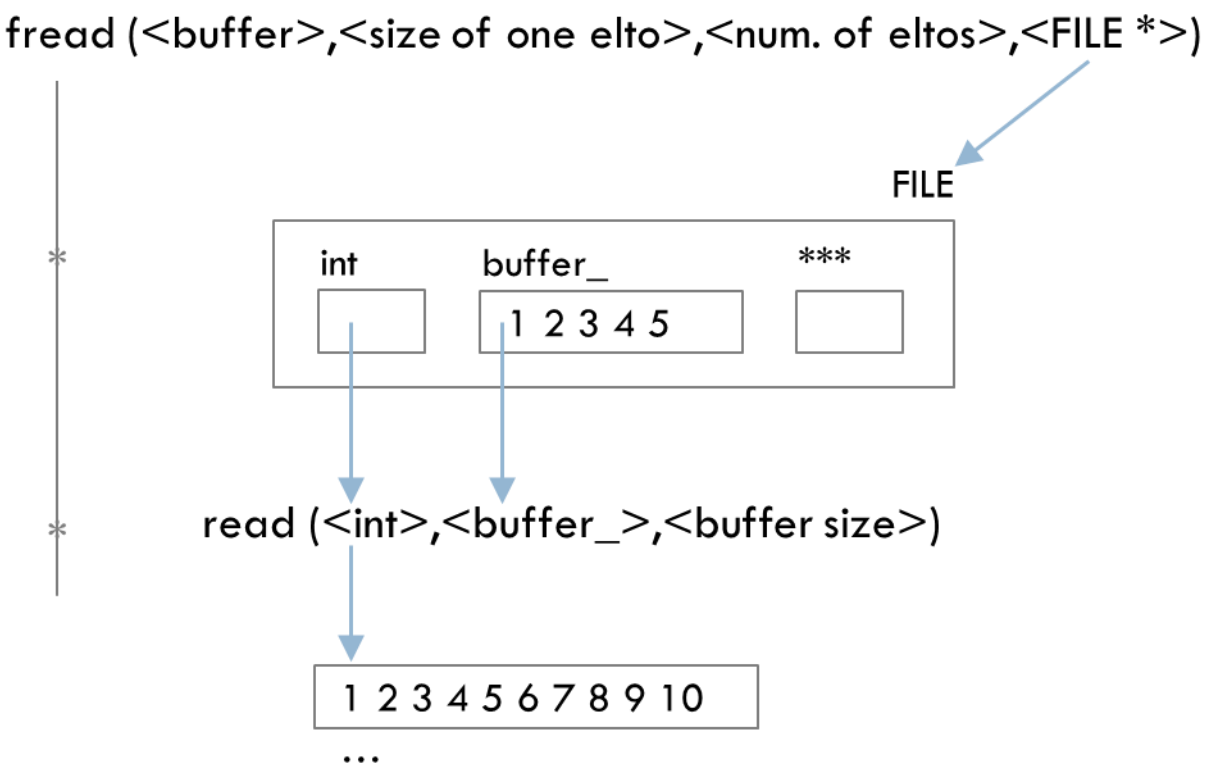
read from file

Alejandro Calderón Mateos

```c
#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);
}
```

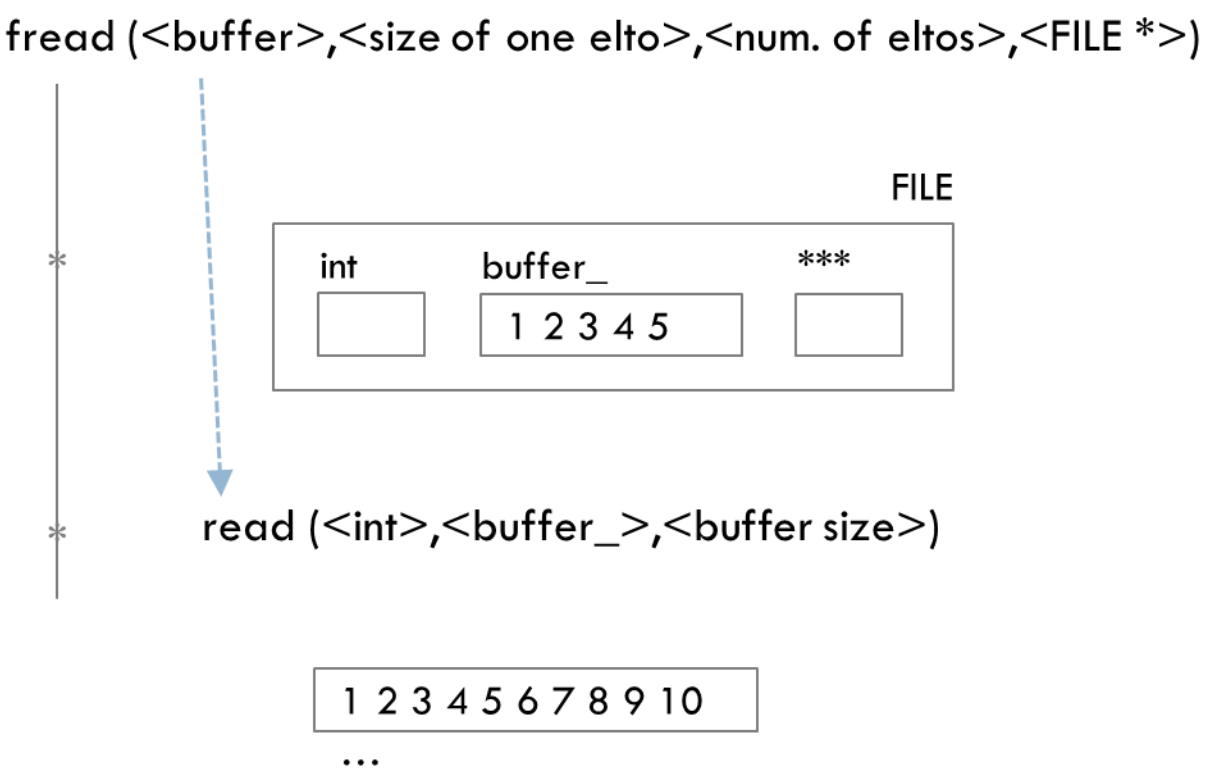
```c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>


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);
}
```

Extended functionality



A pointer to FILE contains the file descriptor and an intermediate buffer (mainly)...

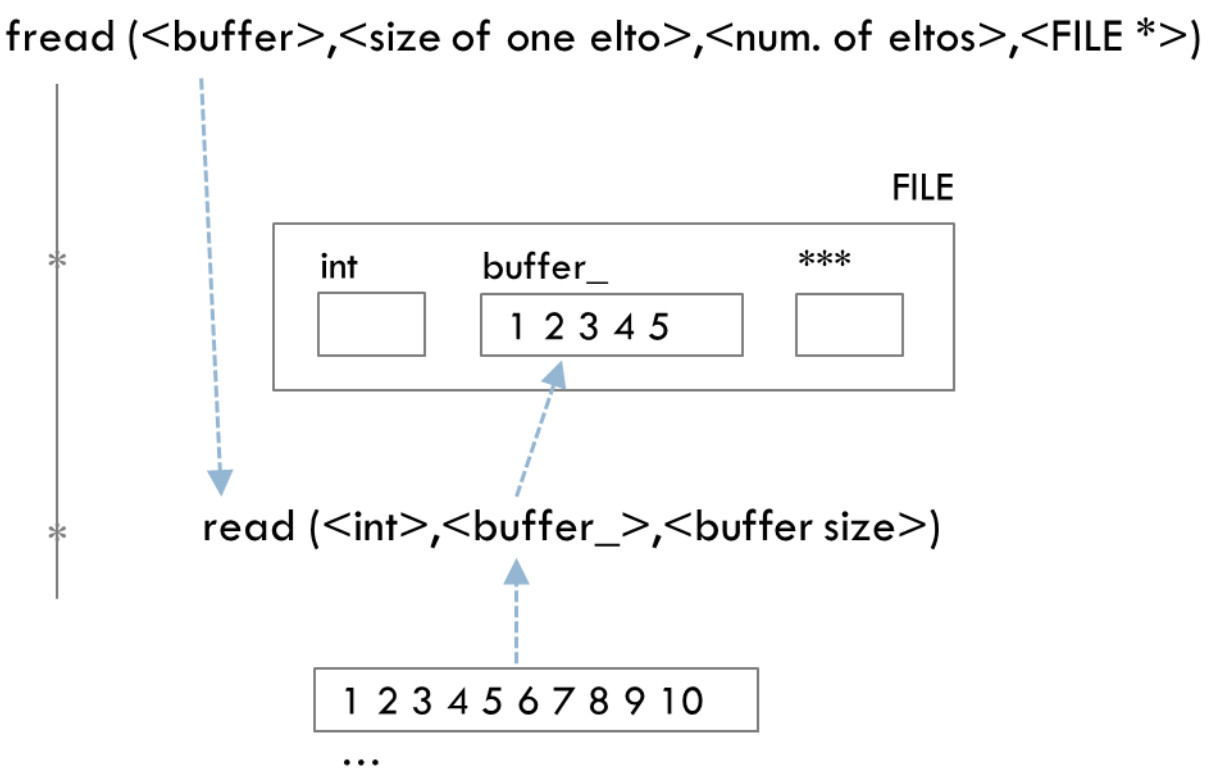
Extended functionality



... so that when the first read is requested, a read is performed on the buffer (the size of which is larger than the requested element) ...

Extended functionality

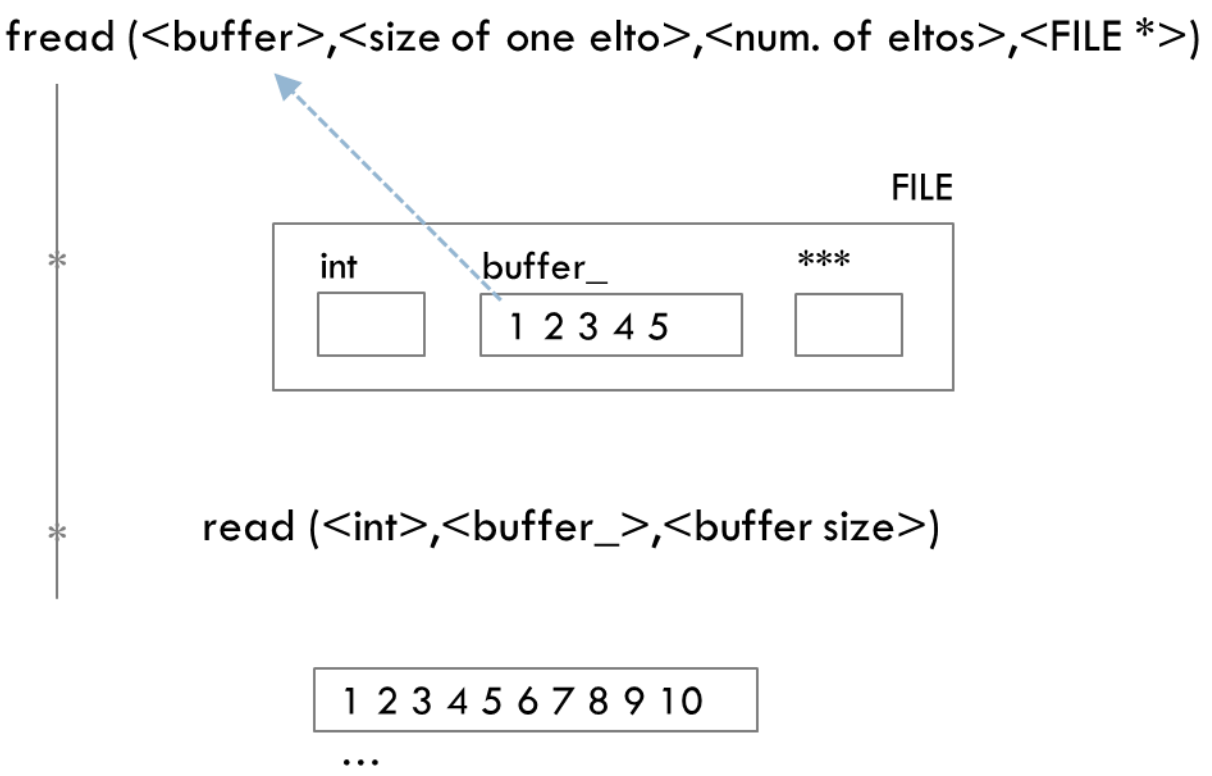
Alejandro Calderón Mateos



… the data is loaded into the buffer and the requested portion is copied to the freading process …

Extended functionality

Alejandro Calderón Mateos



... and the next time a read is made, if it is in the buffer (memory) it is copied directly from it.
This reduces the number of system calls, which speeds up execution.

File: C99 interface

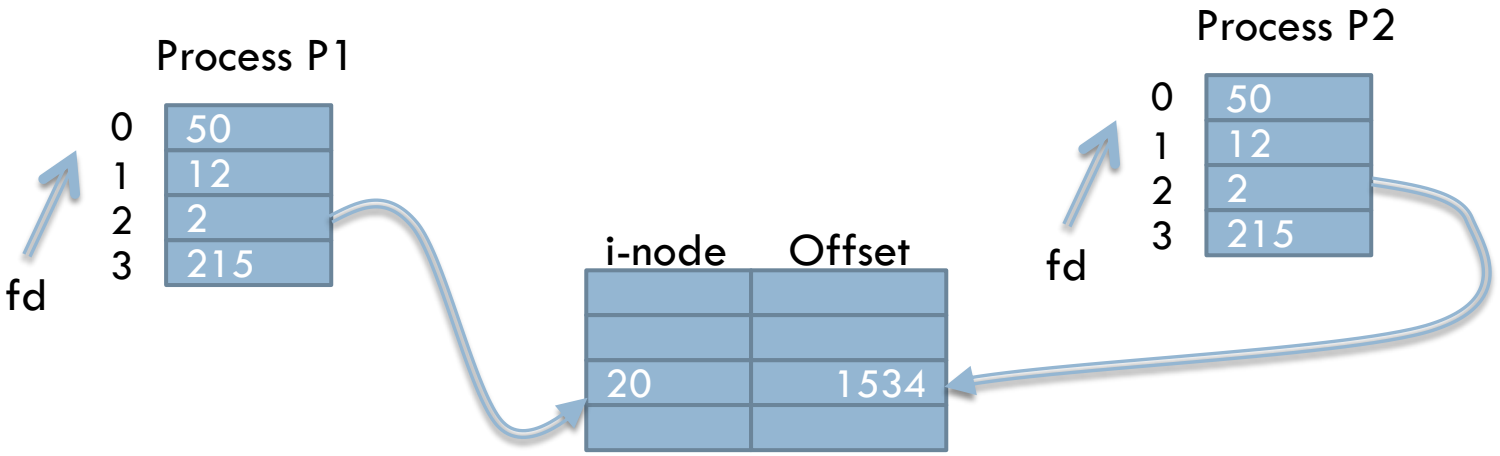

```c
#include <stdlib.h>
#include <stdio.h>
#include <sys/time.h>

#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) ;
}
```

- Compile (gcc –o b b.c) and execute with:
  - BSIZE=1024
  - BSIZE=0
- Results:
  - BSIZE=1024
    - T=0.902 milliseconds
  - BSIZE=0
    - T=14.866 milliseconds

X 15

Interaction between processes and files

- Each process has an associated table of open files.
- When a process is duplicated (fork):
  - The table of open files is duplicated.
  - The intermediate table of i-nodes and offset is shared.



- Protection:
  - `owner group world`
  - `rwx rwx rwx`
- Example: `755` indicates `rwxr-xr-x`

Example: redirection (ls > file)


```
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 – Creation of file

Service	`#include <sys/types.h>`
	`#include <sys/stat.h>`
	`#include <fcntl.h>`
	
	`int creat(char *name, mode_t mode);`

Arguments	▫ `name` File name ▫ `mode` Permission bits for the file
Return	Returns a file descriptor or -1 if error.
Description	The file is opened for writing: ▫ If it does not exist, create an empty file. ▪ UID_owner = UID_actual ▪ GID_owner = GID_actual ▫ If it exists, truncate it without changing the permission bits.

OPEN – Opening a file

Service	`#include <sys/types.h>`
	`#include <sys/stat.h>`
	`#include <fcntl.h>`
	
	`int open(char *name, int flag, ...);`

Arguments	▫ `name` file name (pointer to the first character). ▫ `flags` opening options: ▪ `O_RDONLY` Read only ▪ `O_WRONLY` Writing only ▪ `O_RDWR` Reading and writing ▪ `O_APPEND` Position the access pointer at the end of the open file ▪ `O_CREAT` If it exists it has no effect. If it does not exist, it creates it ▪ `O_TRUNC` Truncated if opened for writing
Return	A file descriptor or -1 in case of error.
Description	File opening (or creation with O_CREAT).

CREAT and OPEN

- Examples:

```
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 – Closing file

Service	`#include <unistd.h>`
	
	`int close(int fd);`

Arguments	`fd` file descriptor.
Return	Return 0 or -1 if error.
Description	The process closes the work session with the file, and the descriptor becomes free.

UNLINK – Deletion of file

Service	`#include <unistd.h>`
	
	`int unlink(const char* path);`

Arguments	`path` file name to be unlinked
Return	Return 0 or -1 if error.
Description	Decrements the link counter of the file. If the counter is 0, it deletes the file and frees its resources.

READ – File reading

Service	`#include <sys/types.h>`
	
	`ssize_t` **`read`**`(int fd, void *buf, size_t n_bytes);`

Arguments	▫ `fd` file descriptor ▫ `buf` area where to store the data ▫ `n_bytes` number of bytes to read
Return	Number of bytes actually read or -1 if error.
Description	▫ Transfers `n_bytes`. May read less data than requested if the end of file is exceeded or interrupted by a signal. ▫ After reading, the file pointer is incremented by the number of bytes actually transferred.

WRITE – File writing

Service	`#include <sys/types.h>`
	
	`ssize_t` **`write`**`(int fd, void *buf, size_t n_bytes);`

Arguments	▫ `fd` file descriptor ▫ `buf` data area to be written ▫ `n_bytes` number of bytes to write
Return	Number of bytes actually written or -1 if error.
Description	▫ Transfers `n_bytes`. Can write less data than requested if the maximum size of a file is exceeded or interrupted by a signal. ▫ After writing, the file pointer is incremented by the number of bytes actually transferred. ▫ If the end of file is exceeded, the file increases in size.

LSEEK – Movement of the position pointer

Service	`#include <sys/types.h>`
	`#include <unistd.h>`
	
	`off_t lseek(int fd, off_t offset, int whence);`

Arguments	▫ `fd` File descriptor ▫ `offset` displacement ▫ `whence` base of the displacement
Return	The new pointer position or -1 if error.
Description	▫ Places the access pointer associated with `fd` ▫ The new position is calculated: ▪ `SEEK_SET` new_offset = offset ▪ `SEEK_CUR` new_offset = actual location + offset ▪ `SEEK_END` new_offset = file size + offset

Example: Copying one file to another(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;

 /* opens the input file*/
  fd_ent = open(argv[1], O_RDONLY);
  if (fd_ent < 0)  {
      perror("open");
      exit(-1);
  }

  /* creates the output file*/
  fd_sal = creat(argv[2], 0644);
  if (fd_sal < 0)  {
      close(fd_ent);
      perror("open");
      exit(-1);
  }

  /* loop for reading the input file*/
  while ((n_read = read(fd_ent, buffer, BUFSIZE)) > 0)
  {
      /* write the buffer to the output file*/
      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;
```

Example: Copying one file to another(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;

 /* opens the input file*/
  fd_ent = open(argv[1], O_RDONLY);
  if (fd_ent < 0)  {
      perror("open");
      exit(-1);
  }

  /* creates the output file*/
  fd_sal = creat(argv[2], 0644);
  if (fd_sal < 0)  {
      close(fd_ent);
      perror("open");
      exit(-1);
  }

  /* loop for reading the input file*/
  while ((n_read = read(fd_ent, buffer, BUFSIZE)) > 0)
  {
      /* write the buffer to the output file*/
      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;
}
```

```
/* opens the input file*/
fd_ent = open(argv[1], O_RDONLY);
if (fd_ent < 0)  {
    perror("open");
    exit(-1);
}

/* creates the output file*/
fd_sal = creat(argv[2], 0644);
if (fd_sal < 0)  {
    close(fd_ent);
    perror("open");
    exit(-1);
}
```

Example: Copying one file to another(3/3)


```
#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;

 /* opens the input file*/
  fd_ent = open(argv[1], O_RDONLY);
  if (fd_ent < 0)  {
      perror("open");
      exit(-1);
  }

  /* creates the output file*/
  fd_sal = creat(argv[2], 0644);
  if (fd_sal < 0)  {
      close(fd_ent);
      perror("open");
      exit(-1);
  }

  /* loop for reading the input file*/
  while ((n_read = read(fd_ent, buffer, BUFSIZE)) > 0)
  {
      /* write the buffer to the output file*/
      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;
}
```

```
/* loop for reading the input file*/
while ((n_read = read(fd_ent,buffer,BUFSIZE))>0)
{
    /* write the buffer to the output file*/
    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 – Modifying attributes

Service	`#include <sys/types.h>`
	
	`int fcntl(int fildes, int cmd /* arg*/ ...);`

Arguments	▫ `fildes` file descriptor ▫ `cmd` to modify attributes, there may be several.
Return	0 for success or -1 if error.
Description	Modify the attributes of an open file

DUP – Duplication of file descriptor

Service	`#include <unistd.h>`
	
	`int dup(int fd);`

Arguments	▫ `fd` file descriptor
Return	UA file descriptor that shares all the properties of the `fd` or -1 if error.
Description	▫ Creates a new file descriptor that has in common with the previous descriptor: ■ Accesses the same file. ■ Shares the same position pointer. ■ The access mode is identical. ▫ The new descriptor will have the smallest possible numeric value.

FTRUNCATE – Allocation of space to a file

Service	`#include <unistd.h>`
	
	`int ftruncate(int fd, off_t length);`

Arguments	▫ `fd` file descriptor ▫ `length` new file size
Return	Return 0 or -1 if error.
Description	The new file size is `length`. If `length` is 0 the file is truncated.

STAT – Information about a file

Service	`#include <sys/types.h>`
	`#include <sys/stat.h>`
	
	`int stat(char *name, struct stat *buf);`
	`int fstat(int fd, struct stat *buf);`

Arguments	▫ `name` file name ▫ `fd` file descriptor ▫ `buf` pointer to an object of type `struct stat` where the file information will be stored
Return	Return 0 or -1 if error.
Description	Gets information about a file and stores it in a struct stat structure.

STAT – Information about a file


```
struct stat {
    mode_t  st_mode;  /* file mode */
    ino_t   st_ino;   /* file identificator */
    dev_t   st_dev;   /* device */
    nlink_t st_nlink; /* number of links*/
    uid_t   st_uid;   /* Owner UID */
    gid_t   st_gid;   /* Owner's GID */
    off_t   st_size;  /* number of bytes */
    time_t  st_atime; /* last access */
    time_t  st_mtime; /* last modification */
    time_t  st_ctime; /* last data modification */
};
```

STAT – Information about a file

- Checking the file type applied to `st_mode`:

`S_ISDIR(s.st_mode)`	True if directory
`S_ISCHR(s.st_mode)`	True if special character
`S_ISBLK(s.st_mode)`	True if special block
`S_ISREG(s.st_mode)`	True if normal file
`S_ISFIFO(s.st_mode)`	True if pipe or FIFO

UTIME – Date attribute alteration

Service	`#include <sys/stat.h>`
	`#include <utime.h>`
	
	`int utime(char *name, struct utimbuf *times);`

Arguments	▫ `name` file name ▫ `times` structure with last access and modification dates. ▪ `time_t actime` date of access ▪ `time_t mctime` modification date
Return	Return 0 or -1 if error.
Description	Changes the last access and last modification dates according to the values of the `struct utimbuf`

POSIX services for directories

- **Logical view:**
  - A directory is a file with records of type "DIR structure".
  - There are calls to work with the records in a directory.
  - Particularities:
    - ONLY READS FROM A DIRECTORY, CANNOT WRITE FROM PROGRAM
    - Caution! As the name of each directory entry is of variable length, they cannot be manipulated as fixed-length records.

- "estructura DIR":
  - d_ino; // *I-node*
  - d_off; // *Position in the file of the directory element*
  - d_reclen; // *Size of the directory*
  - d_type; // *Type of element*
  - d_name[0]; // ***Variable-length*** *file name*

POSIX services for directories

- DIR ***opendir**(const char **dirname*);
  - Opens the directory and returns a pointer to the beginning of type DIR
- int **readdir**(DIR **dirp*, struct dirent **entry*, struct dirent ***result*);
  - Reads the following directory entry and returns it in a struct dirent
- long int **telldir**(DIR **dirp*);
  - Indicates the current position of the pointer inside the directory file
- void **seekdir**(DIR **dirp*, long int *loc*);
  - Advances from the current position to the position indicated in "loc". Never jump backwards.
- void **rewinddir**(DIR **dirp*);
  - Resets the file pointer and puts it back to the beginning of the file
- int **closedir**(DIR **dirp*);
  - Closes the directory file

Example of work with directories

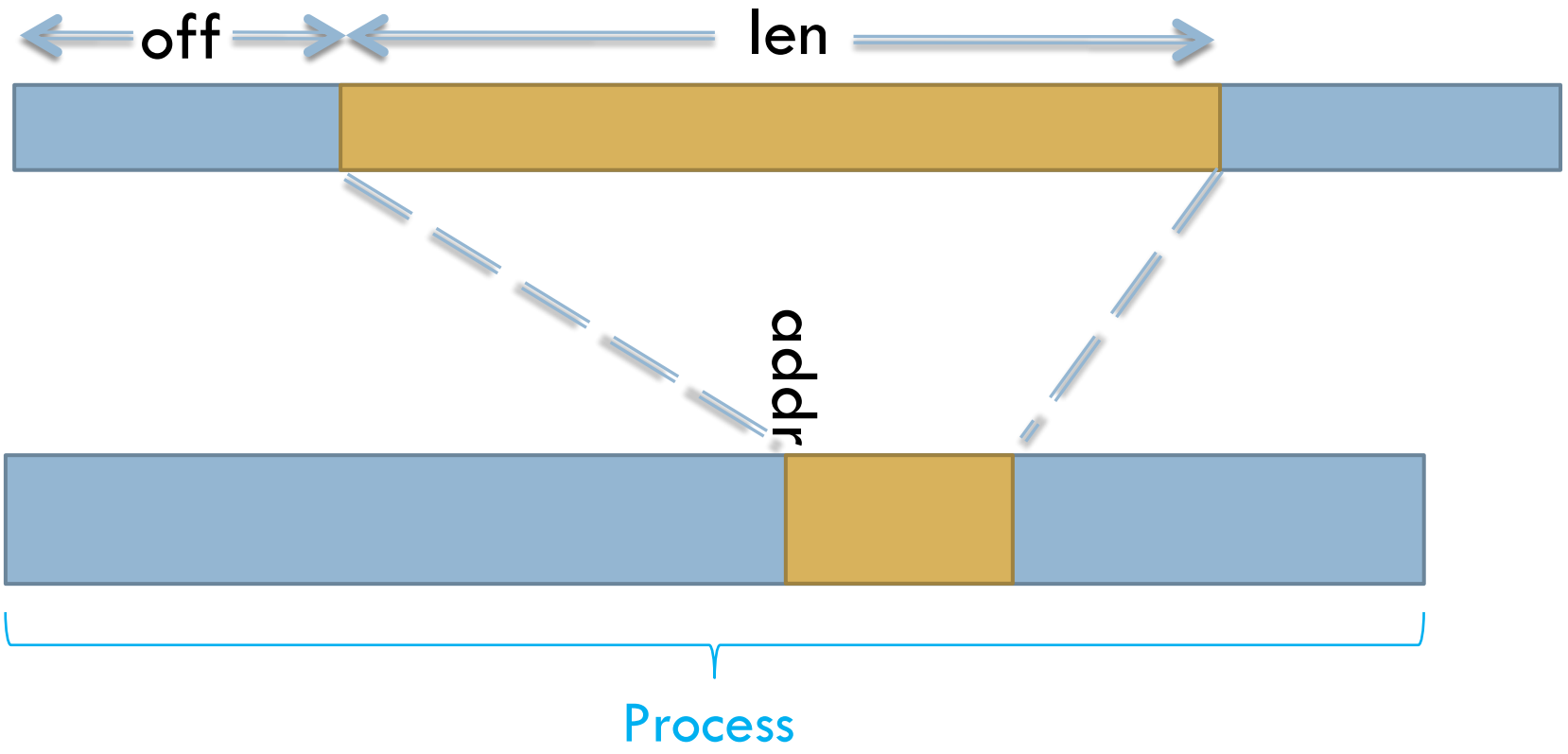
Alejandro Calderón Mateos

```
#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:%d,\t name:%s }\n",
                    direntp->d_ino, direntp->d_off, direntp->d_reclen, direntp->d_name);
    }
    closedir(dirp);
}
```

Contents

- Introduction to system calls
- System call mechanism
- Calls for services of:
  - Process management
  - Management of files and directories
    - File memory mapping

POSIX memory mapping



POSIX mapping: mmap

Service	`void *mmap(void *addr, size_t len, int prot, int flags, int fildes, off_t off);`
Arguments	▫ `addr` direction to project. If `NULL` the SO chooses one. ▫ `len` specifies the number of bytes to map. ▫ `prot` the type of access (read, write or execution). ▫ `flags` specifies information about the handling of the projected data (shared, private, etc.). ▫ `fildes` represents the file descriptor of the file or descriptor of the memory object to map into memory. ▫ `off` displacement within the file from which the mapping is performed.
Return	Return the memory address where the file has been projected.
Description	Establishes a projection between the address space of a process and a file descriptor or shared memory object.

POSIX mapping: mmap

- `int prot`: Types of protection:
  - PROT_READ: You can read.
  - PROT_WRITE: You can write.
  - PROT_EXEC: You can execute.
  - PROT_NONE: Unable to access data.
- `int flags`: Properties of a memory region:
  - MAP_SHARED: The region is shared. Modifications affect the file. Child processes share the region.
  - MAP_PRIVATE: The region is private. The file is not modified. Child processes get unshared duplicates.
  - MAP_FIXED: The file must be projected to the address specified by the call.

POSIX mapping: munmap

Service	`void munmap(void *addr, size_t len);`
Arguments	▫ `addr` address where it is mapped. ▫ `len` specifies the number of mapped bytes.
Return	Nothing.
Description	Unmap part of the address space of a process starting at the `addr` address.

Example: copying a file (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);
```

Example: copying a file (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;
}
```

Example: count the number of blanks in a file (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;
```

Example: count the number of blanks in a file (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;
}
```

OPERATING SYSTEMS: OPERATING SYSTEM SERVICES

System calls