TEMA 2.3. Gestión de Procesos

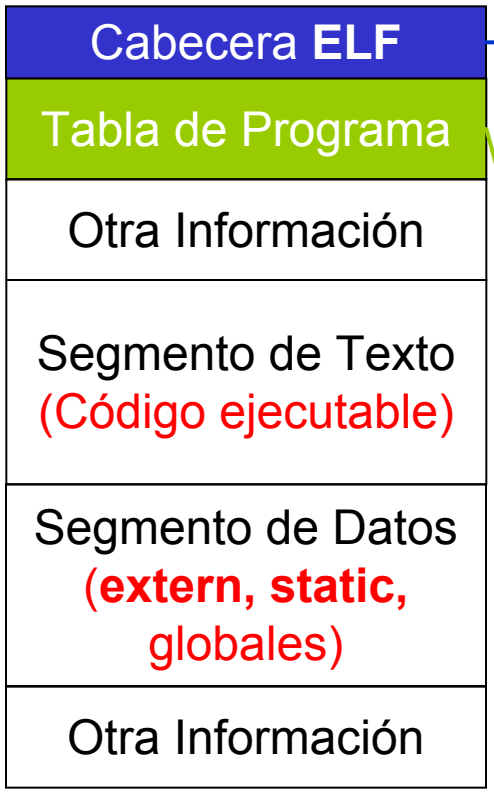
PROFESORES:
Rubén Santiago Montero
Eduardo Huedo Cuesta

OTROS AUTORES:
Ignacio Martín Llorente
Juan Carlos Fabero Jiménez

Estructura de un Programa

- Conjunto de instrucciones en código máquina y datos, almacenados en una imagen ejecutable en disco. Un programa es una entidad pasiva.

Executable and Linking Format (ELF)



Organización y Atributos

```
typedef struct {
Elf32_Half  e_type;
Elf32_Half  e_machine;
...
} Elf32_Ehdr;
```

- Objeto
- Ejecutable
- Objeto Dinámico
- CORE

- EM_SPARC
- EM_386
- EM_IA_64
- ...

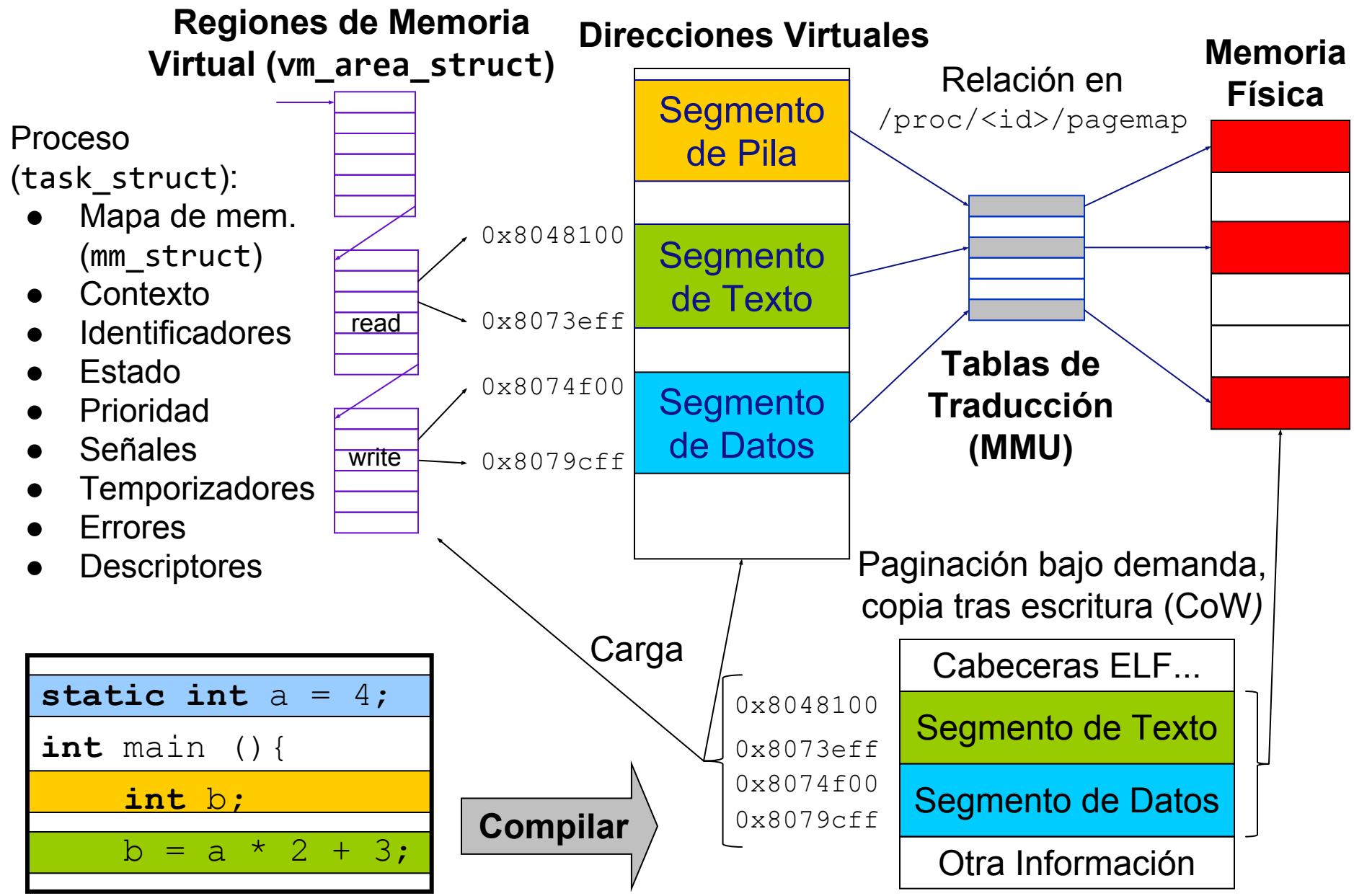
Información para la ejecución

```
typedef struct {
Elf32_Word  p_type;
Elf32_Addr  p_vaddr;
Elf32_Word  p_filesz;
Elf32_Word  p_memsz;
...
} Elf32_Phdr;
```

- PT_LOAD
- PT_DYNAMIC
- PT_PHDR
- ...

Dirección Virtual del segmento

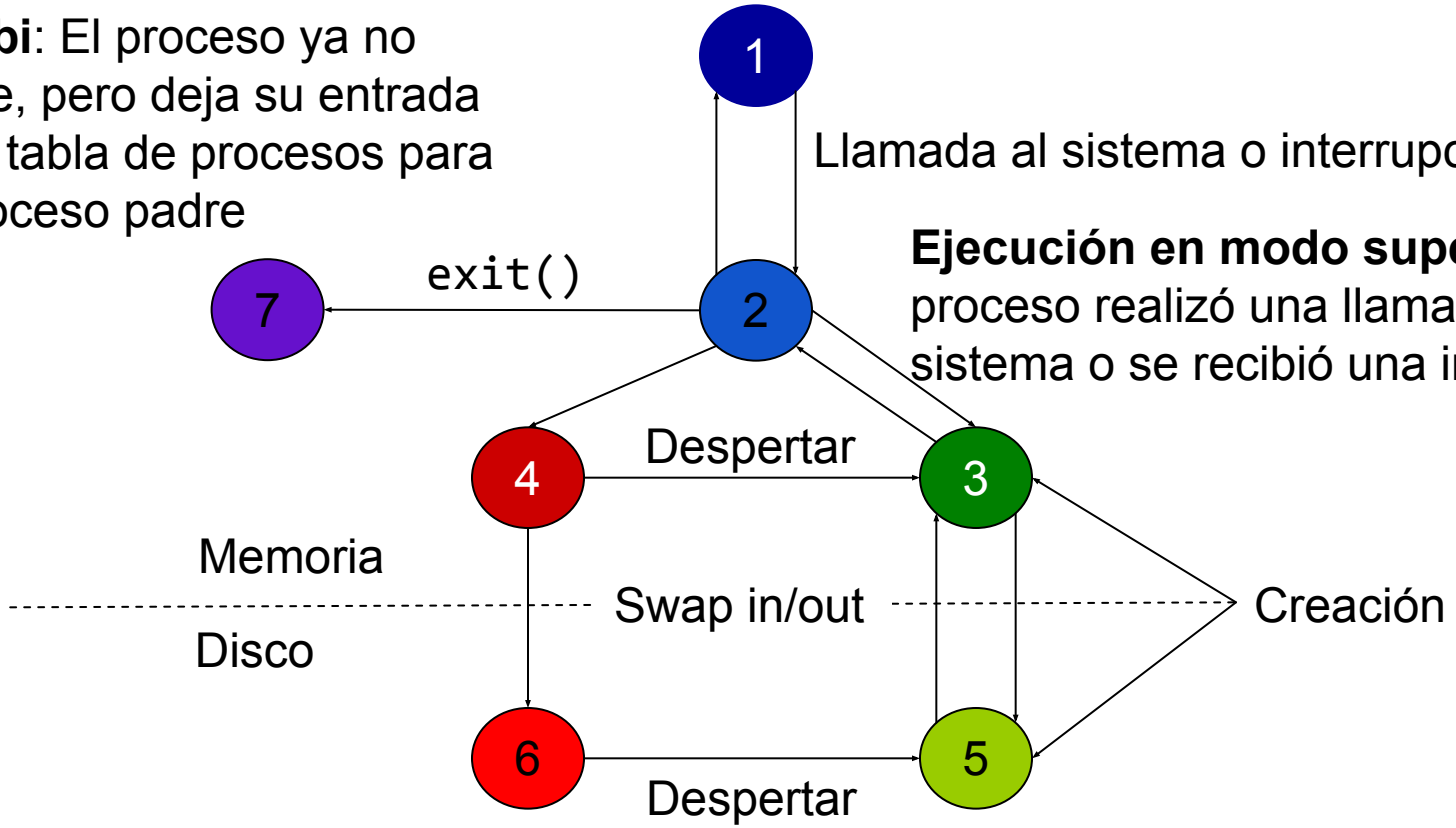
Estructura de un Proceso



Estados de un Proceso

Ejecución en modo usuario: El proceso está activo y ejecutándose

Zombi: El proceso ya no existe, pero deja su entrada en la tabla de procesos para el proceso padre

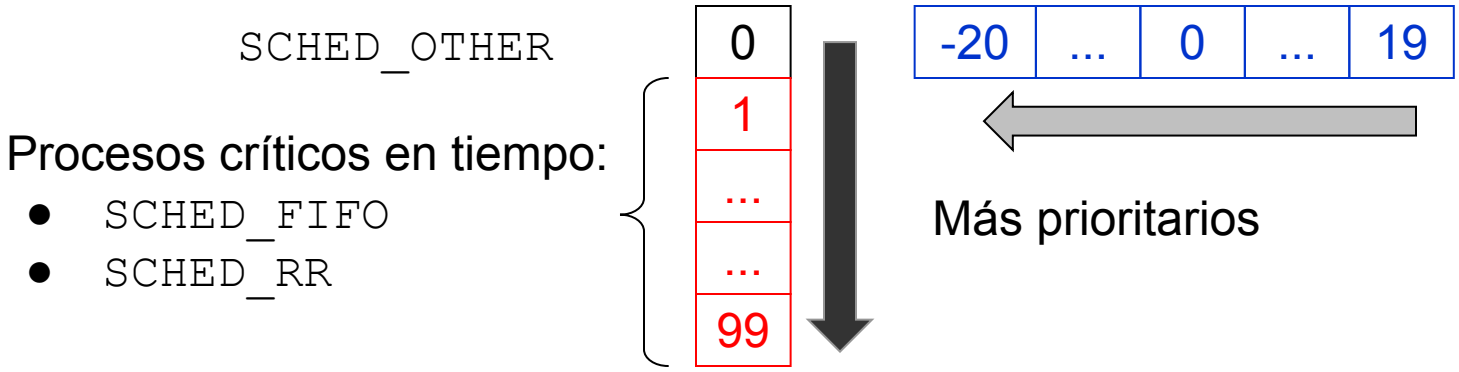


Espera: El proceso está esperando a que termine una operación de E/S

Preparado: El proceso está listo para ejecutarse, esperando que quede libre la CPU

Planificador

- Algoritmo del núcleo que determina el orden de ejecución de los procesos en función de la clase de planificación y de la prioridad de los procesos
- El sistema es expropiativo
- **Políticas de planificación** (ver `/usr/include/bits/sched.h`)
  - **`SCHED_OTHER:`** Política estándar de tiempo compartido con prioridad 0, que también considera el valor de *nice* del proceso
  - **`SCHED_FIFO:`** Política de tiempo real con prioridades entre 1 y 99, siempre expropiará a los procesos de la clase anterior
  - **`SCHED_RR:`** Igual que la anterior, pero a cada proceso se le aplica un cuanto de tiempo de ejecución



Planificador

<sched.h>
POSIX

- Consultar y establecer la política y los parámetros de planificación:

```
int sched_getscheduler(pid_t pid);
int sched_setscheduler(pid_t pid, int policy,
       const struct sched_param *p);

struct sched_param {
    int sched_priority;
    ...
};
```

  - `pid` es un PID (un valor 0 se refiere al proceso actual)
  - `policy` selecciona la política de planificación
  - `p` establece la nueva prioridad:
- Las llamadas afectan realmente al thread (el planificador maneja threads)
  - Todas las llamadas tienen su equivalente `pthread_*`
- Las llamadas `fork()` heredan los atributos de planificación
- El comando `chrt` (*change real-time*) ofrece acceso a esta funcionalidad

Planificador

<sched.h>
POSIX

- Obtener y establecer los parámetros de planificación:

```
int sched_getparam(pid_t pid,
        struct sched_param *p);
int sched_setparam(pid_t pid, const struct sched_param *p)
```

  - `pid` es un PID (0 para el proceso actual)
  - `p` para obtener o establecer la nueva prioridad

- Consultar los rangos de prioridades:

```
int sched_get_priority_max(int policy);
int sched_get_priority_min(int policy);
```

  - `policy` selecciona la política de planificación

- Errores en las llamadas de planificación:
  - `ESRCH`: No se pudo encontrar el proceso especificado por pid.
  - `EPERM`: El UID del proceso que realiza la llamada a `sched_setscheduler()`, ha de ser igual al UID del proceso especificado por PID. Sólo root puede establecer las políticas de “tiempo-real”
  - `EINVAL`: La política/prioridad no es válida

Planificador

<sys/time.h>
SV+BSD

- Obtener y establecer el valor de *nice* de un proceso:

```
int getpriority(int which, int who);
int setpriority(int which, int who, int prio);
```

  - `which` puede ser `PRIO_PROCESS`, `PRIO_PGRP` o `PRIO_USER`
  - `who` es un PID para `PRIO_PROCESS`, un PGID para `PRIO_PGRP` o un UID para `PRIO_USER`
    - 0 indica el proceso actual, el grupo de procesos del proceso actual o el UID real del proceso actual, respectivamente
  - `prio` es el valor del nuevo *nice* (0 por defecto) 19>`prio`>-20, valores más bajos corresponden a una mayor porción de CPU
- Los comandos `nice` y `renice` permiten acceder a esta funcionalidad
- Dado que `getpriority` puede devolver -1, para ver si se ha producido un error hay que consultar `errno`:
  - `ESRCH`: No se encontró el proceso,
  - `EINVAL`: Valor no válido en `which`
  - `EPERM`: El UID efectivo no permite modificar el *nice*
  - `EACCES`: Un proceso de usuario intentó reducir el *nice*

Atributos de un Proceso

<unistd.h>
SV+BSD+POSIX

- Obtener los identificadores de un proceso:

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

  - Cada proceso tiene un identificador único (Process IDentifier, PID) y, además, registra el proceso que lo creó (Parent PID, PPID)
- El comando principal para obtener la lista de procesos y sus atributos es `ps`
- Obtener el identificador de grupo de procesos o crear un grupo de procesos:

```
int setpgid(pid_t pid, pid_t pgid);
pid_t getpgid(pid_t pid);
```

  - Los procesos pertenecen a un grupo de procesos, con un identificador de grupo (PGID), cuyo principal uso es la distribución de señales
  - Si `pid` es 0, se usa el PID del proceso que llama
  - Si `pgid` es 0, se usa el PGID del proceso indicado en `pid`
- Códigos de error:
  - `EINVAL`: `pgid` < 0
  - `EPERM`: Violación de permisos
  - `ESRCH`: No existe el proceso

Atributos de un Proceso

<sys/time.h>
SV+BSD

- Crear una nueva sesión u obtener el identificador de sesión:

```
pid_t setsid(void);
pid_t getsid(pid_t pid);
```

  - Los grupos de procesos se pueden agrupar en sesiones, que se usan para gestionar el acceso al sistema:
    - El proceso de *login* crea una sesión
    - Todos los procesos y grupos del usuario pertenecen a esa sesión
    - En la desconexión todos los procesos de la sesión reciben la señal `SIGHUP`
  - Un proceso sólo puede crear una sesión nueva, y un grupo nuevo, de los que se convertirá el líder y tendrá un grupo igual (todos los procesos de un grupo deben estar en la misma sesión).
- El comando `setsid` también permite crear una nueva sesión (y un nuevo grupo asociado)

Recursos de un Proceso: Límites

<unistd.h>
<sys/types.h>
<sys/resource.h>

SV+BSD

- Obtener y establecer los límites del proceso:

```
int getrlimit(int resource,
        struct rlimit *rlim)
int setrlimit(int resource,
        const struct rlimit *rlim)

struct rlimit{
    int rlim_cur; /* Límite actual */
    int rlim_max; /* Límite máximo */
};
```

  - Los procesos tienen impuestos unos determinados límites en el uso de los recursos del sistema.
  - `resource` puede ser
    - `RLIMIT_CPU`: Max. tiempo de CPU (segundos)
    - `RLIMIT_FSIZE`: Max. tamaño de archivo (bytes)
    - `RLIMIT_DATA`: Max. tamaño del heap (bytes)
    - `RLIMIT_STACK`: Max. tamaño de pila (bytes)
    - `RLIMIT_CORE`: Max. tamaño de archivo core (bytes)
    - `RLIMIT_NPROC`: Max. número de procesos
    - `RLIMIT_NOFILE`: Max. número de descriptores de archivo

Recursos de un Proceso: Uso de Recursos

<sys/time.h> <sys/resource.h>
SV+BSD

- Obtener el uso de los recursos del sistema por parte del proceso:

```
int getrusage(int who,
        struct rusage *usage);

struct rusage{
    struct timeval ru_utime; /* struct con time_t: */
    struct timeval ru_stime; /* tv_secs y tv_usecs */
    long   ru_maxrss;
    ...        /* ver /usr/include/bits/resource.h */
}
```

  - `who` puede ser
    - `RUSAGE_CHILDREN`: solo por los hijos
    - `RUSAGE_SELF`: por el proceso y sus hijos

<sys/times.h>
SV+BSD+POSIX

- Uso de CPU, como **ticks de reloj** respecto a un punto de referencia, usualmente el arranque del sistema:

```
clock_t times(struct tms *buffer);
```

  - Los ticks/segundo se pueden determinar con `sysconf(_SC_CLK_TCK)`

Recursos de un Proceso: Directorio de Trabajo

<unistd.h>
POSIX

- Obtener el **path absoluto** del directorio de trabajo:

```
char *getcwd(char *buffer, size_t size);
```

  - Es el directorio usado para toda ruta relativa en el proceso
  - Se devuelve en `buffer` de tamaño `size` (contando el carácter '\0' de fin de cadena)
  - Si la ruta requiere más espacio, la función retorna NULL y **errno = ERANGE**

- Cambiar el directorio de trabajo de un proceso:

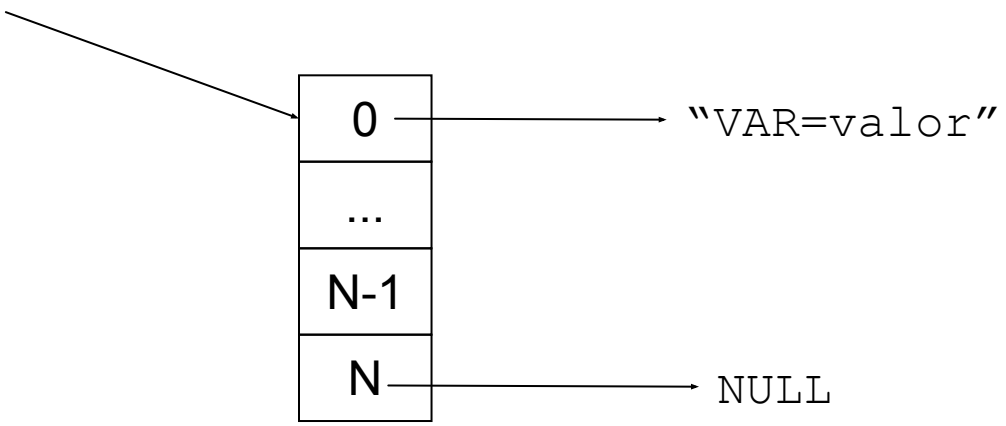
```
int chdir(const char *path);
```

Recursos de un Proceso: Entorno

- Los procesos se ejecutan en un determinado entorno, que en general se hereda del proceso padre
- Muchas aplicaciones limitan o controlan el entorno que pasan a los procesos o la forma en que inicializan su entorno, ej. `sudo` o la shell
- El entorno es un conjunto de cadenas de caracteres en la forma "VARIABLE = valor", por convenio las variables de entorno están en mayúsculas

```
extern char **environ;
```

- HOME
- USER
- PATH
- PWD
- SHELL
- ...



- Manejar el entorno con funciones de la librería estándar de C:

```
char *getenv(const char *name );
int setenv(const char *name, const char *value, int overwrite);
int unsetenv(const char *name);
```

Creación de Procesos: Ejecución de Programas

<stdlib.h>
ANSI C+POSIX

- Ejecutar un programa externo:

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

  - Ejecuta el comando especificado por `string`, mediante `/bin/sh -c string`, retornando al flujo del programa cuando termina
  - La llamada retorna cuando termina la ejecución del comando, a excepción de ejecución en *background*
  - En caso de que no se produzca error la llamada devuelve el código de finalización del programa
  - Si `string` es NULL, la función devuelve cero si la shell no está disponible

Creación de Procesos: Ejecución de Programas

- La familia de funciones `exec()` **ejecuta un programa en el proceso actual**
- El proceso actual se sustituye por la imagen del nuevo programa
- El primer argumento siempre es el programa

`<unistd.h>`
POSIX

	Ruta absoluta	Usa `PATH`	Nuevo entorno
Lista de argumentos	`execl()`	`execlp()`	`execle()`
Vector de argumentos	`execv()`	`execvp()`	`execve()`

```
int execl(const char *path, const char *a0,...);
int execlp(const char *path, const char *a0,...);
int execle(const char *path, const char *a0,...,
      char *const envp[]);
int execv(const char *path, char *const argv[]);
int execvp(const char *path, char *const argv[]);
int execve(const char *path, char *const argv[],
      char *const envp[]);
```

Creación de Procesos: fork()

<unistd.h>
SV+POSIX+BSD

- Crear un proceso hijo:

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

  - La función devuelve:
    - -1: Fallo
    - 0: Ejecutando el hijo
    - >0: Ejecutando el padre, el valor es el PID del hijo.
  - El nuevo proceso ejecuta el mismo código que el proceso padre
  - Los dos procesos creados tienen el mismo contexto de usuario:
    - Cada proceso dispone de un único identificador
    - El hijo recibe una copia de los descriptores de los ficheros abiertos por el padre
    - El hijo no hereda los cerrojos
    - El hijo no hereda las alarmas del padre
    - El conjunto de señales pendientes del hijo es nulo

Creación de Procesos: fork()

```
int main() {
  pid_t mi_pid, pid;

  pid = fork();

  switch (pid) {
     case -1:
       perror("fork");
       exit(-1);
     case 0:
       pid_t mi_pid = getpid();
       printf("Proceso hijo %i (padre: %i)\n",mi_pid,getppid());
       break;
     default:
       mi_pid = getpid();
       printf("Proceso padre %i (hijo %i)\n",mi_pid,pid);
       break;
  }

  return 0;
}
```

Finalización de un Proceso

- Un proceso puede finalizar por dos motivos:
  - voluntariamente, llamando a `exit` (o `return` desde `main()`)
  - recibiendo una señal (hay múltiples causas)
- Terminar el proceso:

```
void _exit(int status);
```

  - `status` es el código de salida, que debe ser un número menor de 255
    - Convenio: 0 = éxito, 1 = error (no devolver `errno` ni -1)
    - 8 bits accesibles en `$?` o en el proceso padre vía `wait`

Finalización de un Proceso

- Esperar la finalización de algún hijo y liberar los recursos asociados:

```
pid_t wait(int *estado);
pid_t waitpid(pid_t pid, int *estado, int op);
```

  - `pid` puede ser:
    - <-1: Espera la finalización de un hijo cuyo PGID es `-pid`
    - -1: Igual a `wait`
    - 0: Espera la finalización de un hijo del grupo de procesos del padre
    - >0: Espera la finalización de un hijo con identificador `pid`
  - `op` puede ser:
    - `WNOHANG`: retorna sin esperar si no hay hijos que hayan terminado
    - `WUNTRACED`: retorna si el proceso ha sido detenido (no vía `ptrace`)
    - `WCONTINUED`: si un hijo detenido ha sido reanudado
  - Macros para consultar el código de terminación (`estado`):
    - `WIFEXITED(s)`: el hijo terminó normalmente vía `exit()`
    - `WEXITSTATUS(s)`: devuelve el código de salida
    - `WIFSIGNALED(s)`: el hijo terminó al recibir una señal
    - `WTERMSIG(s)`: número de la señal recibida

info. estado terminación (8 bits)	status del arg. de exit (8 bits)

Señales

- Las señales son **interrupciones software**, que informan a un proceso de la ocurrencia de un evento de forma **asíncrona**
- Las señales las **genera un proceso**, incluido el núcleo del sistema
- Las opciones en la ocurrencia de un evento son:
  - Bloquear la señal
  - Ignorar la señal
  - Invocar a una rutina de tratamiento por defecto, que en general termina con la ejecución del proceso
  - Invocar a una rutina de tratamiento propia
- Tipos de señales:
  - Terminación de procesos
  - Excepciones
  - Llamada de sistema
  - Generadas por proceso
  - Interacción con el terminal
  - Traza de proceso
  - Fuertemente dependientes del sistema (consultar `signal.h`)

Señales: System V (Ejemplos)

- **`SIGHUP`:** Desconexión de terminal. La envía un proceso a todos los de su grupo cuando finaliza su ejecución. Terminación del proceso (**F**)
- **`SIGINT`**: Interrupción. Se puede generar con la secuencia de teclas: Ctrl+C (**F**)
- **`SIGQUIT`**: Finalización. Se puede generar con la secuencia: Ctrl+\ (**F** y volcado de la memoria - core **C**)
- **`SIGILL`**: Instrucción ilegal (punteros a funciones mal gestionados) (**F** y **C**)
- **`SIGTRAP`**: Ejecución paso a paso, enviada después de cada instrucción (**F** y **C**).
- **`SIGKILL` (9):** Terminación brusca. No puede ignorarse (**F** y **C**)
- **`SIGBUS`**: Error de acceso a memoria (alineación o dirección no válida) (**F** y **C**)
- **`SIGSEGV`:** Violación de segmento *de datos* (**F** y **C**)
- **`SIGPIPE`**: Intento de escritura en un tubería sin lectores (**F**)
- **`SIGALARM`:** Despertador, contador a 0 (**F**)
- **`SIGTERM`:** Terminar proceso, el sistema advierte que el proceso debe terminar su ejecución (puede capturarse) (**F**)
- **`SIGUSR1`, `SIGUSR2`:** Señales de usuario, terminación
- **`SIGCHLD`:** Terminación del proceso hijo

`signal(7)`

Señales: Envío

- Enviar una señal a un proceso:

<signal.h>
SV+BSD+POSIX

```
int kill(pid_t pid, int signal);
```

  - `pid`: Identifica el proceso que recibirá la señal:
    - >0: Es el identificador del proceso
    - 0: Se envía a todos los procesos del grupo
    - -1: Se envía a todos los procesos (de mayor a menor), excepto a `init`
    - <-1: Se envía a todos los procesos del grupo con PGID igual a `-pid`
  - `signal`: La señal que se enviará (si es 0, se simula el envío)
- El comando `kill` proporciona acceso a esta llamada
- Llamadas equivalentes:

<signal.h>
ANSI-C

```
int raise(int signal);
int abort(void);
```

<stdlib.h>
SV+BSD+POSIX

  - `raise(signal)` ⇒ `kill(getpid(), signal)`
  - `abort()` ⇒ `kill(getpid(), SIGABRT)`

Señales: Envío. Ejemplo

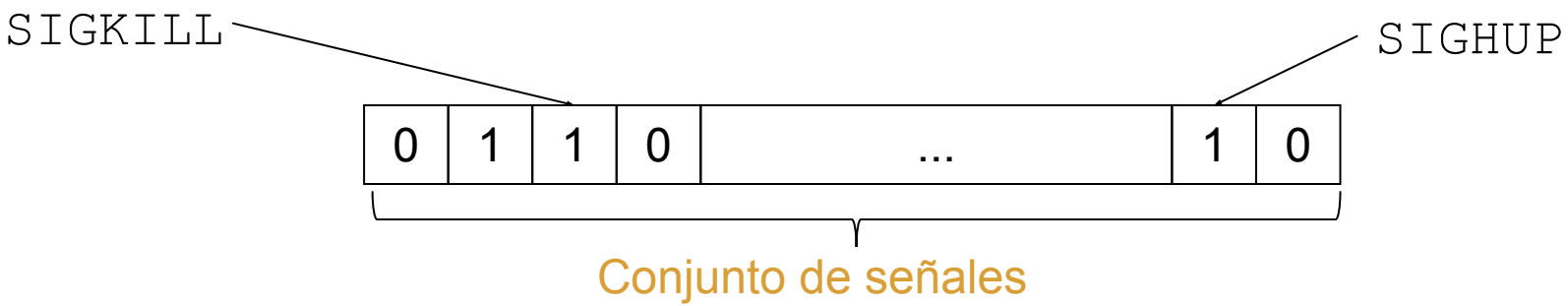
```
#include <signal.h>
#include <unistd.h>

int main()
{
    kill(getpid(), SIGABRT);
    return 0;
}
```

```
> ./abort_self
Aborted (core dumped)
```

Señales: Conjuntos de Señales

- Gestión de señales:
  - **Individualmente**, por su nombre o identificador
  - Conjuntamente, usando **conjuntos de señales** (`sigset_t`)
- Implementación:
  - La representación es un tipo “opaco” que depende del sistema
  - Ver `/usr/include/bits/sigset.h`
  - Ejemplo: Cada bit representa una señal



Señales: Conjuntos de Señales

<signal.h>
POSIX

- Inicializar el conjunto al conjunto vacío, con todas las señales excluidas:

```
int sigemptyset(sigset_t *set);
```

- Inicializar el conjunto lleno, con todas las señales incluidas:

```
int sigfillset(sigset_t *set);
```

- Añadir y quitar señales del conjunto:

```
int sigaddset(sigset_t *set, int signal);
int sigdelset(sigset_t *set, int signal);
```

- Comprobar si una señal pertenece a un conjunto:

```
int sigismember(sigset_t *set, int signal);
```

```
#include <signal.h>
…
    sigset_t set;

    sigemptyset(&set);
    sigaddset(&set, SIGINT);
    sigaddset(&set, SIGQUIT);
…
```

Señales: Bloqueo

<signal.h>
POSIX

- Proteger regiones del código contra la recepción de una señal:

```
int sigprocmask(int how,
        const sigset_t *set, sigset_t *oset);
```

  - `how` define el comportamiento de la función
    - `SIG_BLOCK`: Añade el conjunto `set` al conjunto de señales actualmente bloqueadas (“OR”)
    - `SIG_UNBLOCK`: El conjunto `set` se retira del conjunto de señales bloqueadas. Puede desbloquearse una señal que previamente no estuviera bloqueada
    - `SIG_SETMASK`: Reemplaza el conjunto de señales actuales por `set`
  - `oset` almacena el conjunto previo de señales bloqueadas (distinto de NULL)
  - Las señales quedarán **bloqueadas** (máscara de señales), hasta que sean **explícitamente desbloqueadas**:

- Comprobar señales pendientes:

```
int sigpending(const sigset_t *set);
```

  - `set` es el conjunto de señales pendientes
  - `sigismember()` y `sigprocmask()` para determinar y tratar la señal

Señales: Bloqueo. Ejemplo

```
sigset_t blk_set;

sigemptyset(&blk_set);
sigaddset(&blk_set, SIGINT);
sigaddset(&blk_set, SIGQUIT);

sigprocmask(SIG_BLOCK, &blk_set, NULL);

/* Actualización de la base de datos, evitando la
corrupción de los datos */

sigprocmask(SIG_UNBLOCK, &blk_set, NULL);
```

Señales: Captura

- La **acción por defecto** en la recepción de una señal es la **finalización del proceso** o simplemente **ignorar la señal**
- Se puede **modificar el comportamiento**, instalando una función (manejador, *handler*) que se ejecute cuando se reciba la señal, evitando la finalización del proceso
- Obtener y establecer la acción asociada a una señal:

<signal.h>
POSIX

```
int sigaction(int signal,
        const struct sigaction *act,
        struct sigaction *oldact);

struct sigaction {
    void (*sa_handler) (int);
    sigset_t sa_mask;
    int sa_flags;
    ...
}
```

  - `signal` especifica la señal, a excepción de `SIGKILL` y `SIGSTOP`
  - `act` contiene el nuevo manejador para la señal (puede ser `NULL`)
  - `oldact` almacena el antiguo controlador de la señal (puede ser `NULL`)

Señales: Captura

- Campos de **`struct`** `sigaction`:
  - `sa_handler` es el nuevo manejador para la señal. Su valor puede ser:
    - `SIG_DFL`: Para el manejador por defecto
    - `SIG_IGN`: Para no atender la señal
    - Un puntero a una función: **`void`** `handler(`**`int`** `signal);`
  - `sa_mask` es el conjunto de señales que serán bloqueadas durante el tratamiento de la señal
    - Además, se bloquea la señal en cuestión (si no se indica el *flag* `SA_NODEFER`)
  - `sa_flags` modifica el comportamiento del proceso de gestión de la señal:
    - `SA_RESTART`: Reiniciar llamadas al sistema interrumpidas (en caso contrario terminan con `EINTR`) para compatibilidad con BSD
    - `SA_RESETHAND`: Restaurar el manejador por defecto tras tratar la señal
    - `SA_SIGINFO`: Pasar argumentos adicionales al manejador de la función

Señales: Captura

- El proceso suspende toda ejecución y llama al manejador, **restaurando** la ejecución en el **punto donde se produjo la señal**
- El **manejador** es una función del estilo:

```
void handler(int signal)
```

- Hay que tomar algunas precauciones en el manejador:
  - No usar variables del tipo `extern` o `static`
  - No usar **funciones no reentrantes**, como `malloc`, `free` o funciones en la librería `stdio`
  - Guardar y restaurar el valor de `errno`
- Como regla general, hacer lo menos posible en el manejador
  - Normalmente, fijar algún *flag* (declarado como `volatile`) y salir
- Tener en cuenta siempre que las señales son **asíncronas**

Señales: Captura

- Esperar la ocurrencia de una determinada señal, suspendiendo la ejecución del proceso:

```
int sigsuspend(const sigset_t *set);
```

  - La máscara de señales bloqueadas se sustituye temporalmente por el conjunto `set`, el proceso se suspende hasta que **una señal** que **no esté en la máscara** se produzca
  - Cuando se recibe la señal se **ejecuta el manejador** asociado a la señal y continúa la ejecución del proceso
  - Si se produce un error `EINTR`, se restaura la **máscara original**

- Alternativamente, suspender un proceso de forma más sencilla:

```
unsigned int sleep(unsigned int segundos);
```

  - Suspende la ejecución del proceso durante los segundos especificados, o hasta que se reciba una señal que deba ser tratada

Señales: Alarmas

<unistd.h>
SV+BSD+POSIX

- Fijar una alarma:

```
unsigned int alarm(unsigned int secs);
```

  - Usa el temporizador `ITIMER_REAL` para programar una señal `SIGALARM`
  - `secs`: Número de segundos a los que se fijará el temporizador (si es cero, no se planifica ninguna nueva alarma)
  - En cualquier caso, cualquier alarma programada previamente se cancela
  - Devuelve el valor de segundos restantes para que se produzca el final de la cuenta (0 si no hay ninguna fijada)
  - Debe instalarse previamente el controlador
  - No mezclar con sleep o cualquier otra función que use el mismo temporizador ej. `setitimer()`
  - No heredadas por `fork()`, pero sí por `exec`

Señales: Alarmas

<sys/time.h>
SV+BSD

- Consultar o fijar alarmas asociadas a otros temporizadores:

```
int getitimer(int which,
        struct itimerval *value);
int setitimer(int which, struct itimerval *value,
        struct itimerval *ovalue);

struct itimerval {
    struct timeval it_interval; /* Intervalo */
    struct timeval it_value;    /* Tiempo que queda */
}
```

  - `which` selecciona el temporizador
    - `ITIMER_REAL`: Tiempo real (*wall-clock*), genera `SIGALARM`
    - `ITIMER_VIRTUAL`: Tiempo de CPU en modo usuario, genera `SIGVTALARM`
    - `ITIMER_PROF`: Tiempo de CPU en modo usuario y sistema, genera `SIGPROF`

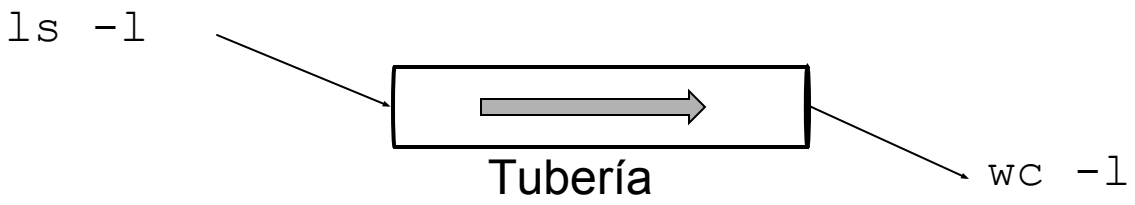
Comunicación entre Procesos. Tuberías

Introducción

- Mecanismos de sincronización:
  - Ejecución de procesos/threads en el mismo sistema
    - **Señales** (sólo para procesos)
    - Ficheros con **cerrojos**
    - Mutex y variables de condición (solo para threads de un proceso)
    - Semáforos (System V IPC)
    - Colas de mensajes (System V IPC)
  - Ejecución de procesos en distintos sistemas
    - Basados en **sockets** (paso de mensajes, colas de mensajes…)
- Compartición de datos entre procesos:
  - Ejecución de procesos en el mismo sistema
    - Memoria compartida (System V IPC)
    - **Tuberías sin nombre** (pipes)
    - **Tuberías con nombre** (FIFOs)
    - Colas de mensajes (System V IPC)
    - Basados en **ficheros**
  - Ejecución de procesos en distintos sistemas
    - Basados en **sockets**

Tuberías sin Nombre

- Soporte para **comunicación unidireccional** entre dos procesos
- El sistema las **trata** a todos los efectos **como ficheros:**
  - i-nodo
  - Descriptores
  - Tabla de ficheros del sistema y proceso
  - Operaciones de E/S típicas
  - Heredadas de padres a hijos
- **Sincronización** realizada por parte del **núcleo**
- Acceso tipo **FIFO** (*first-in-first-out*)
- La tubería **reside** en **memoria principal**

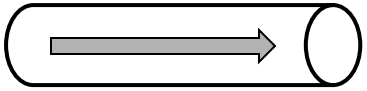


Tuberías sin Nombre

- Crear una tubería

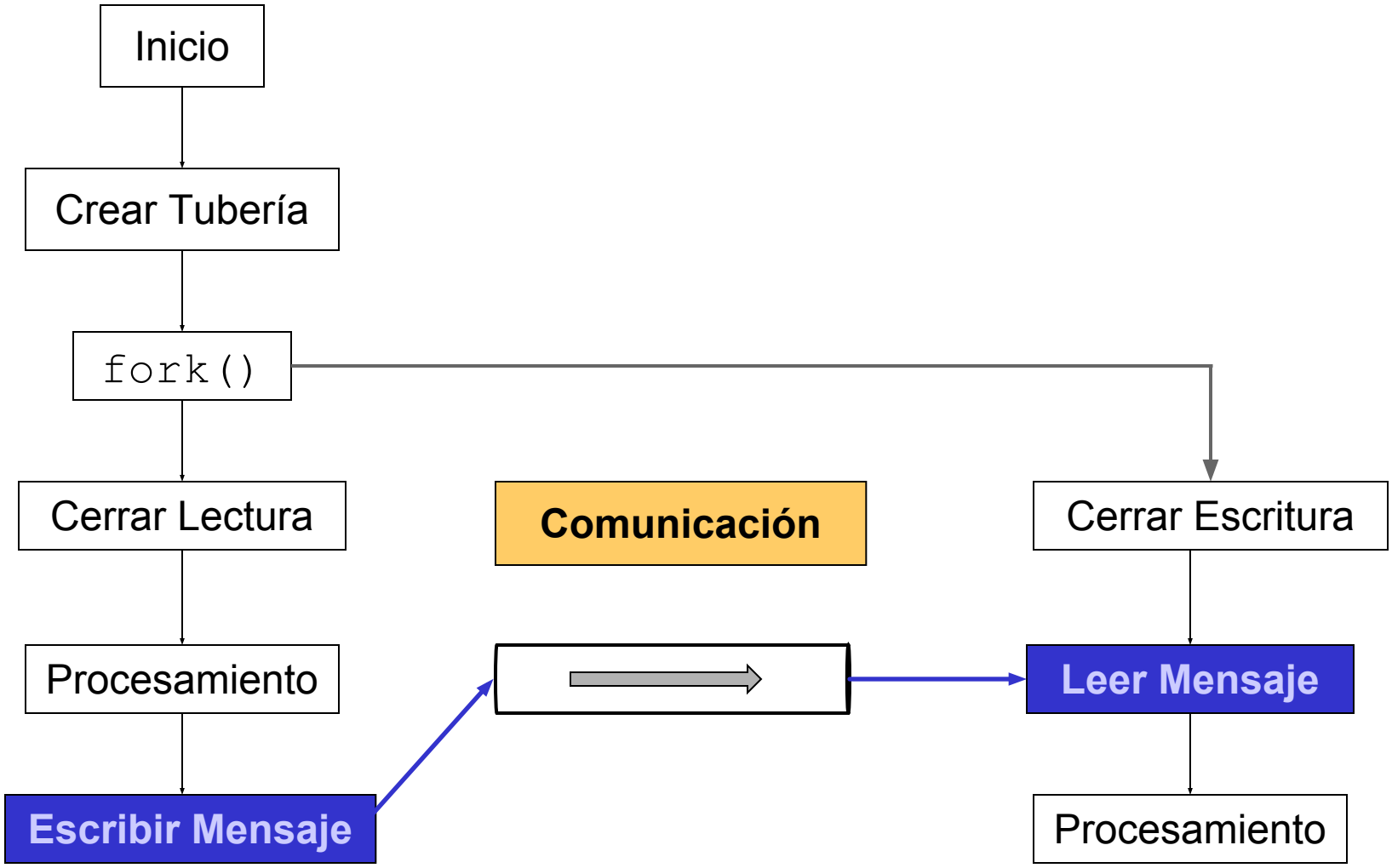
```
int pipe(int descriptor[2]);
```

<unistd.h>
SV+BSD+POSIX

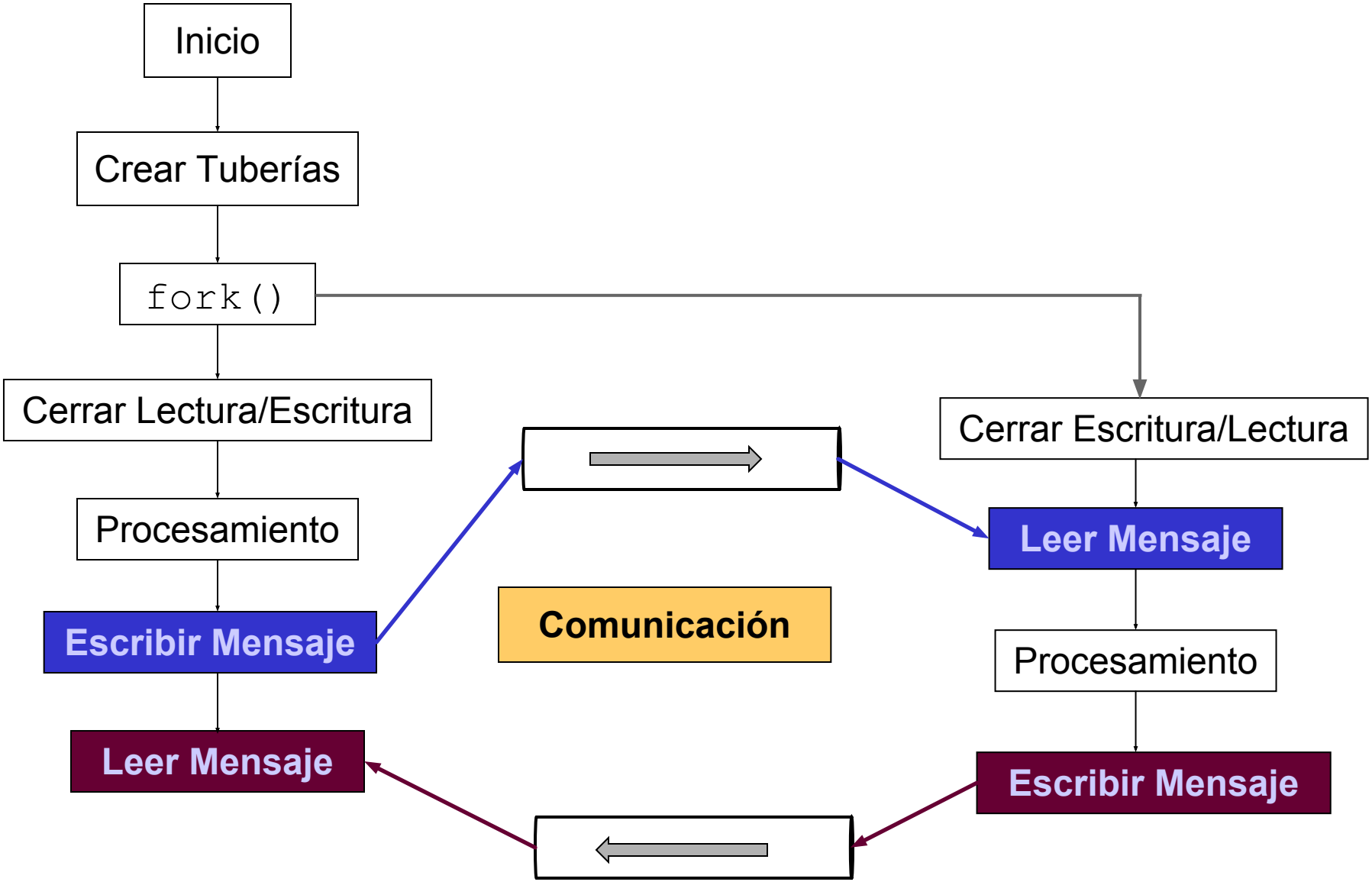
Escritura: `descriptor[1]`  Lectura: `descriptor[0]`

  - Si la tubería se llena, las llamadas a `write()` quedarán bloqueadas
  - Errores:
    - **EMFILE:** Demasiados descriptores.
    - **ENFILE:** Demasiados ficheros en el sistema.
    - **EFAULT:** Array de descriptores no válido.

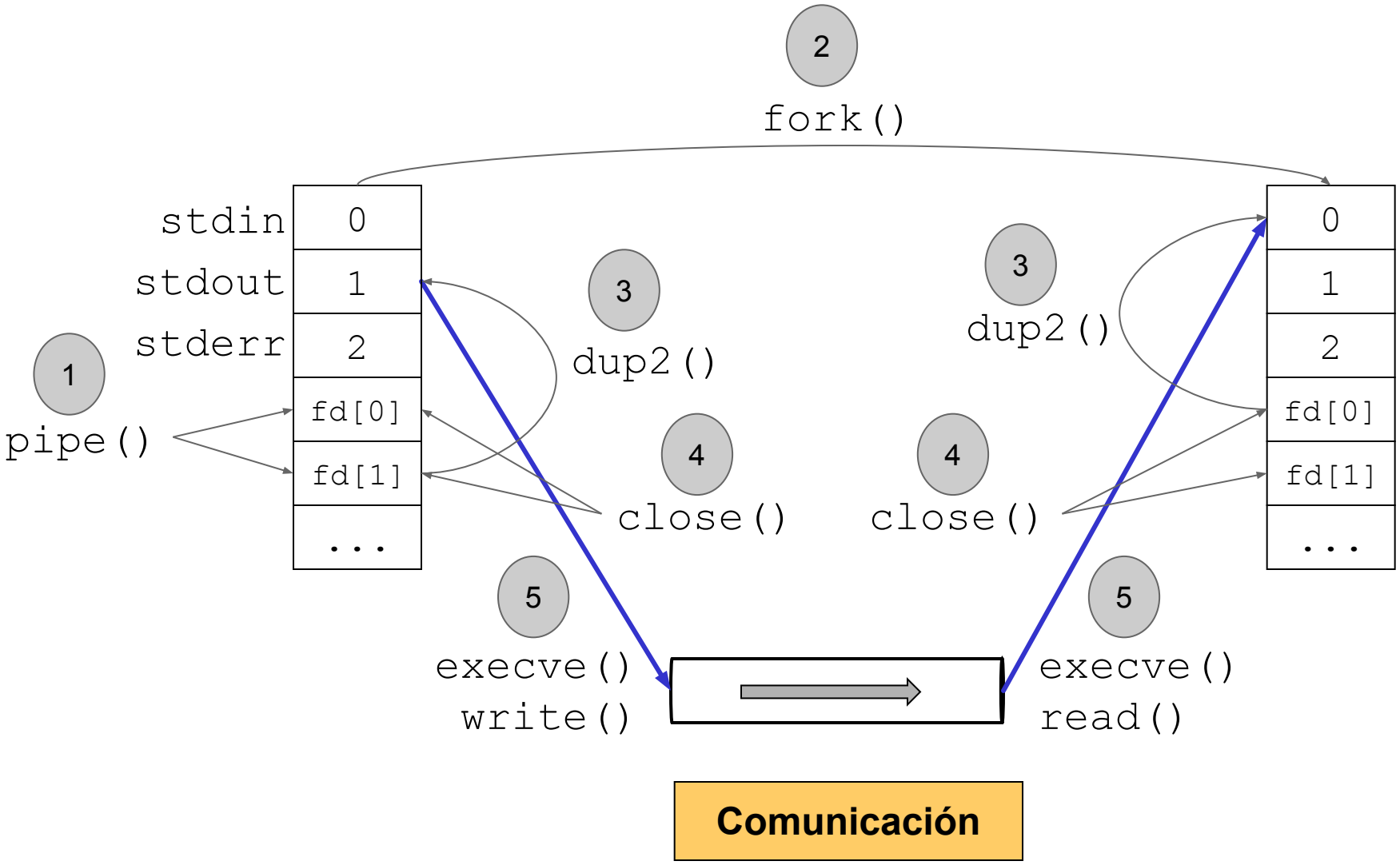
Tuberías sin Nombre



Tuberías sin Nombre



Tuberías sin Nombre



Tuberías con Nombre

- La comunicación mediante **tuberías sin nombre** se realiza únicamente entre **procesos con relación de parentesco**
- **Tubería con nombre**: Tipo especial de fichero con la misma funcionalidad que las tuberías sin nombre, salvo que se accede como parte del sistema de ficheros
  - La entrada en el sistema de ficheros solo sirve para que los procesos abran la tubería con `open()` usando un nombre
  - El núcleo realiza la sincronización y almacena los datos internamente, sin escribirlos en el sistema de ficheros
  - Varios procesos pueden abrir la tubería para lectura o escritura, y ambos extremos (de lectura y de escritura) deben abrirse antes de poder intercambiar datos, ya que la apertura se bloquea hasta que se abre el otro extremo
    - Al abrir para lectura, esto se puede modificar con el *flag* `O_NONBLOCK`

Operación E/S	**Condición**	**Resultado**
Lectura	FIFO vacío, con escritor	Se bloquea
Escritura	FIFO vacío, con lector	Se bloquea
Lectura	FIFO vacía, sin escritor	Devuelve 0 (EOF)
Escritura	Sin lector	Recibe `SIGPIPE`

Tuberías con Nombre

```
<sys/types.h>
<sys/stat.h>
<fcntl.h>
<unistd.h>
```

SV+BSD

- Crear ficheros especiales:

```
int mknod(const char *filename,
          mode_t mode, dev_t dev);
```

  - `filename` es el nombre del fichero (archivo, dispositivo, tubería) que se creará
  - `mode` especifica los permisos y el tipo de archivo que se creará. Su valor se fijará mediante OR lógica de permisos (*umask*). El tipo ha de ser:
    - `S_IFREG`: Archivo regular
    - `S_IFCHR`: Dispositivo de caracteres (`dev` = *major,minor*)
    - `S_IFBLK`: Dispositivo de bloques (`dev` = *major,minor*)
    - `S_IFIFO`: Tubería con nombre

- El siguiente comando proporciona acceso a esta funcionalidad:

```
mknod [-m permisos] nombre tipo
```

  - `nombre` es el nombre del fichero que se creará
  - `tipo` es el tipo del archivo puede ser
    - `b`: Dispositivo de bloques
    - `c`: Dispositivo de caracteres
    - **`p`:** FIFO

Tuberías con Nombre

<sys/types.h>
<sys/stat.h>

POSIX

- Crear tuberías con nombre:

```
int mkfifo(const char *filename,
           mode_t mode);
```

  - `filename` es el nombre de la tubería que se creará
  - `mode` determina los permisos con que se creará la tubería. Estos permisos se ven modificados por el *umask* del proceso

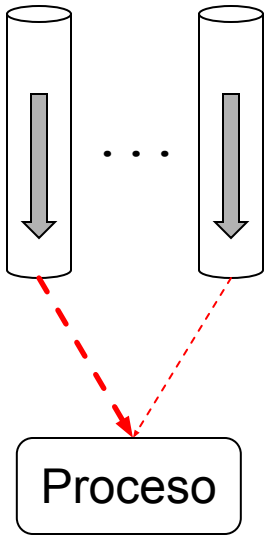
- El siguiente comando proporciona acceso a esta funcionalidad:

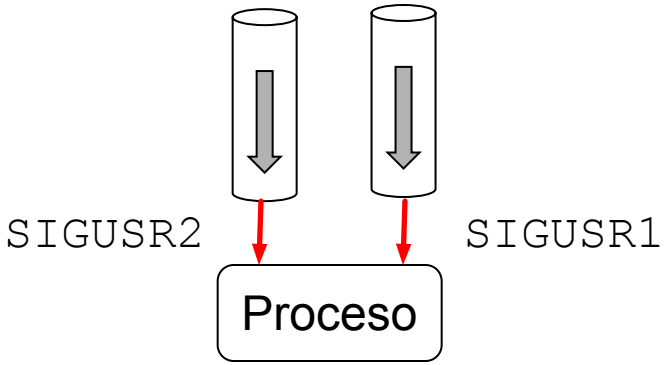
```
mkfifo [-m permisos] nombre
```

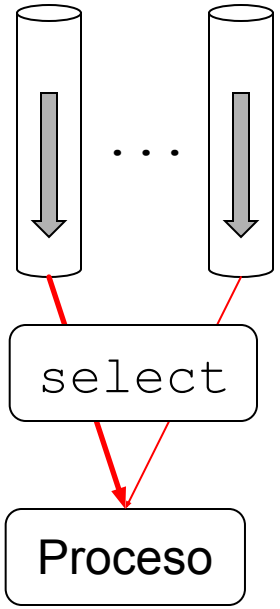
  - `nombre` es el nombre del fichero que se creará

Sincronización de E/S

- Cuando un proceso gestiona varios canales de E/S, debe seleccionar los que están listos en cada momento para realizar la operación
- Alternativas:
  - E/S no bloqueante (encuesta)
  - E/S conducida por eventos (notificación asíncrona)
  - Multiplexación de E/S síncrona







Multiplexación de E/S Síncrona

- Consiste en seleccionar en cada momento del descriptor de fichero que esté listo para realizar la operación de entrada/salida, permitiendo realizarla de forma **síncrona**

<sys/select.h>
POSIX+BSD

- Seleccionar un descriptor de fichero preparado:

```
int select(int n, fd_set *Rset,
           fd_set *Wset, fd_set *Eset, struct timeval *tout);
```

  - `n` es el mayor de los descriptores en cualquiera de los tres conjuntos, más 1
  - `Rset` es el conjunto de descriptores de lectura, se comprobará si hay datos disponibles
  - `Wset` es el conjunto de descriptores de escritura, se comprobará si es posible escribir de forma inmediata
  - `Eset` es el conjunto de descriptores de excepción, se comprobará si hay alguna condición especial
  - `tout` es el tiempo máximo en el que retornará la función. Si es 0, retorna inmediatamente. Si es NULL, se bloquea hasta que se produce un cambio

Multiplexación de E/S Síncrona

- Macros para la manipular los conjuntos:

  `void FD_ZERO(fd_set *set)`: Inicializa `set` como conjunto vacío
  `void FD_SET(int fd, fd_set *set)`: Añade `fd` a `set`
  `void FD_CLR(int fd, fd_set *set)`: Elimina `fd` de `set`
  `int FD_ISSET(int fd, fd_set *set)`: Comprueba si `fd` está en `set`

- `select()` devuelve el número de descriptores que han experimentado un cambio de estado y almacena los descriptores en los que se ha producido algún cambio de condición en los conjuntos `Wset`, `Rset` y `Eset`, según corresponda
  - En caso de que expire el tiempo máximo, devuelve 0

- Si se produce un error, los conjuntos no se modifican y `tout` queda indeterminado
  - `EBADF`: Descriptor no válido en alguno de los tres conjuntos
  - `EINTR`: Señal no bloqueada recibida
  - `EINVAL`: Valor de `n` negativo

Multiplexación de E/S Síncrona

```
...
fd_set conjunto;

FD_ZERO(&conjunto);
FD_SET(0, &conjunto);

timeout.tv_sec = 2;
timeout.tv_usec = 0;

cambios = select(1, &conjunto, NULL, NULL, &timeout);

if (cambios == -1)
    perror("select()");
else if (cambios) {
    read(0, buffer, 80);
    printf("Datos nuevos: %s\n", buffer);
} else {
    printf("Ningún dato nuevo en 2 seg.\n");
}
...
```