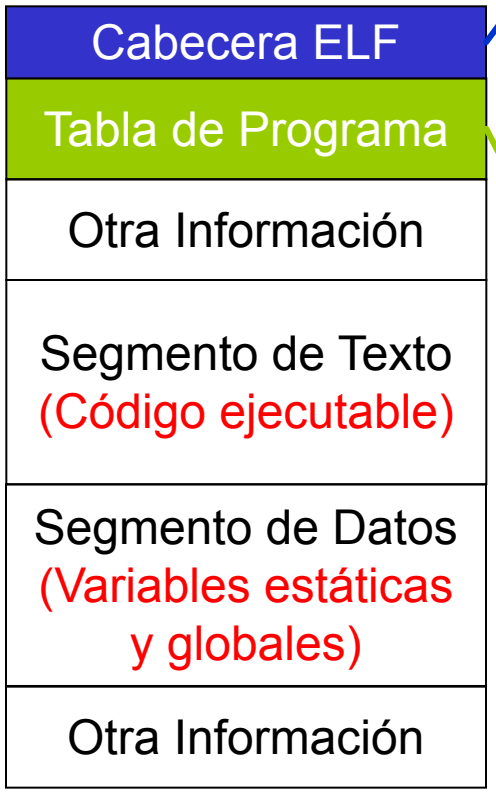
TEMA 2.3. Gestión de Procesos

PROFESORES:
Rubén Santiago Montero
Eduardo Huedo Cuesta
Luis M. Costero Valero

Estructura de un Programa

- Un programa es un conjunto de instrucciones máquina y datos, almacenados en una imagen ejecutable en disco (entidad pasiva)

Executable and Linking Format (ELF)



Organización y atributos

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

ET_REL: Relocatable object
ET_EXEC: Executable
ET_DYN: Shared object
ET_CORE: Core

EM_X86_64, EM_ARM...

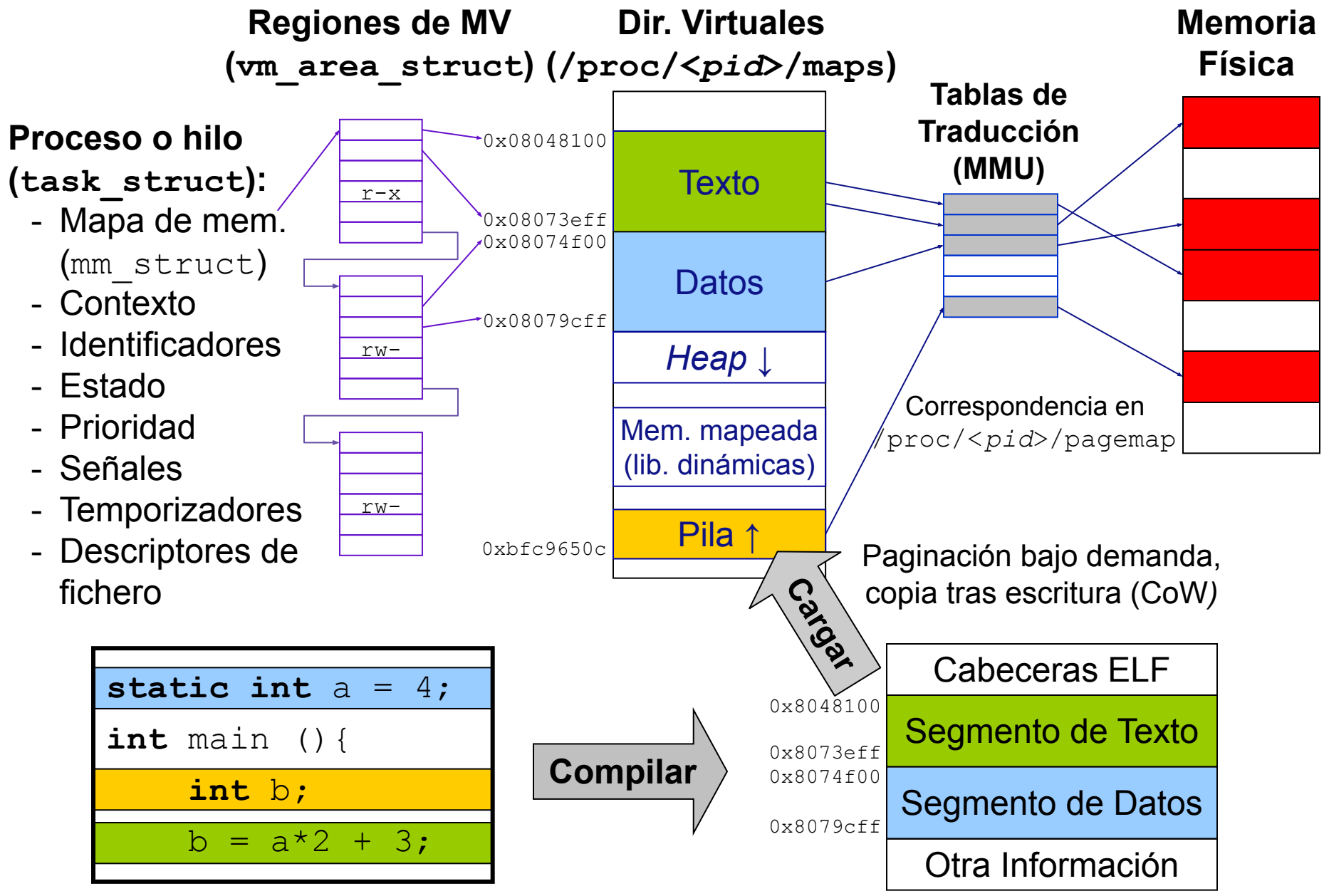
Información para ejecución

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

PT_PHDR: Program header
PT_LOAD: Loadable segment
PT_DYNAMIC: Dynamic linking
...

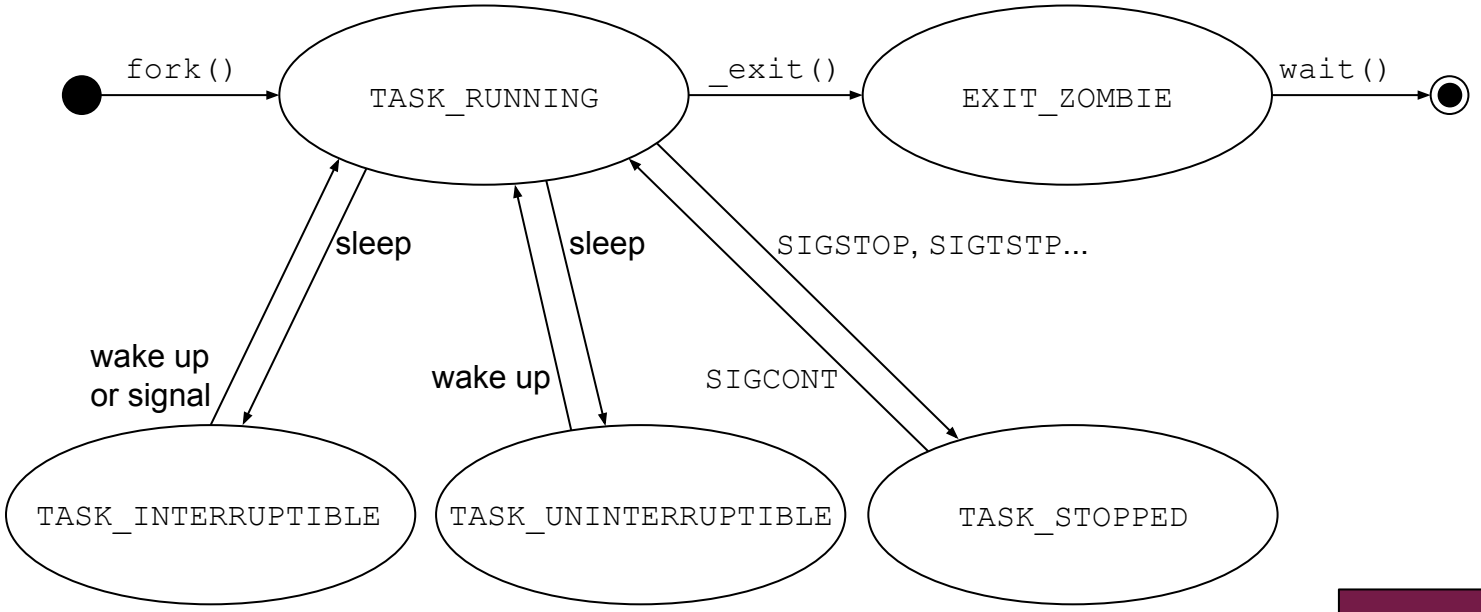
Dirección virtual del segmento

Estructura de un Proceso



Estados de un Proceso

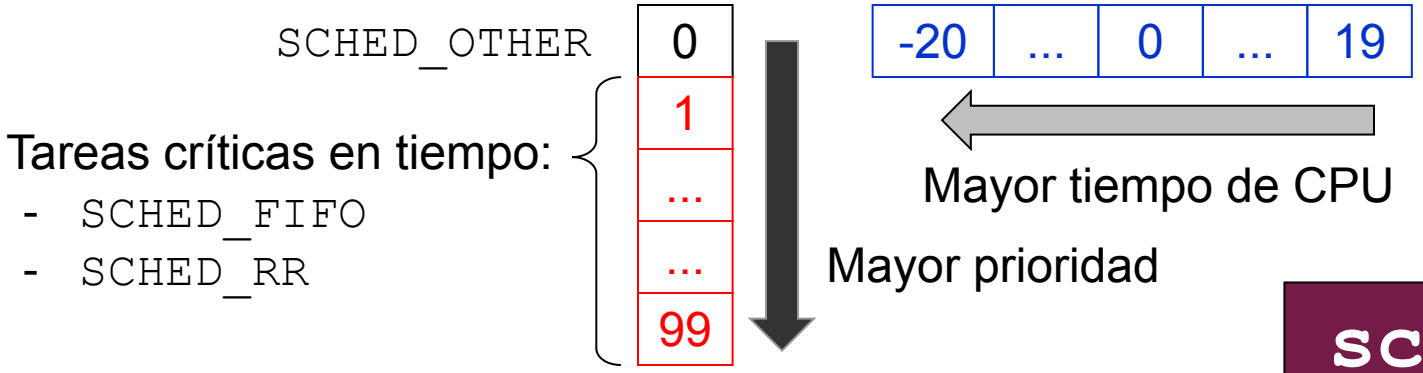
- `ps(1)` muestra la lista de procesos, sus identificadores y sus atributos, incluyendo el estado:
  - `R` (`TASK_RUNNING`): en ejecución o preparado (en la cola de ejecución)
  - `D` (`TASK_UNINTERRUPTIBLE`): bloqueado (normalmente por E/S)
  - `S` (`TASK_INTERRUPTIBLE`): esperando (a que un evento se complete)
  - `T` (`TASK_STOPPED`): parado (por una señal)
  - `Z` (`EXIT_ZOMBIE`): muerto (ya no existe, pero deja su entrada en la tabla de procesos para que el proceso padre recoja su estado de salida)



`ps(1)`

Planificador

- Componente del núcleo que determina el orden y el tiempo de ejecución de las tareas en función de su prioridad y de su política de planificación
  - Es expropiativo (una tarea de mayor prioridad siempre expropiará a otra de menor prioridad en ejecución) y la política de planificación sólo afecta a las tareas preparadas con igual prioridad
- **Políticas de planificación**
  - **`SCHED_OTHER`:** Política estándar de tiempo compartido con prioridad 0, que considera el valor de *nice* (entre -20 y 19, 0 por defecto) para repartir la CPU
  - **`SCHED_FIFO`:** Política de tiempo real FIFO con prioridades entre 1 y 99, donde las tareas se ejecutan hasta que se bloquean por E/S, son expropiadas por una tarea con mayor prioridad o ceden la CPU (`sched_yield(2)`)
  - **`SCHED_RR`:** Como la anterior, pero las tareas con igual prioridad se ejecutan por turnos (*round-robin*) en porciones de tiempo (100 ms por defecto)



`sched(7)`

Planificador

<sched.h>
POSIX

- Consultar y establecer la política de planificación y establecer la prioridad:

```
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 (0 indica el proceso actual)
  - `policy` selecciona la política de planificación (`SCHED_OTHER`, `SCHED_FIFO`, `SCHED_RR`...)
  - `p` establece la nueva prioridad
- Las llamadas afectan realmente al hilo (el planificador maneja hilos o tareas)
  - Las llamadas tienen su equivalente `pthread_*`
- Los procesos heredan los atributos de planificación del proceso padre

Planificador

<sched.h>
POSIX

- Obtener y establecer la prioridad 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 indica 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

- `chrt(1)` (*change real-time*) proporciona acceso a esta funcionalidad

Planificador

- Obtener y establecer el valor de *nice*:

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

```
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, un PGID o un UID, según el valor de `which`
    - 0 indica el proceso actual, el grupo de procesos actual o el UID real del proceso actual, respectivamente
  - `prio` es el nuevo valor de *nice* entre -20 y 19
    - Valores menores representan una mayor porción de CPU

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

<unistd.h>
POSIX+SV+BSD

```
int nice(int inc);
```

- `nice(1)` y `renice(1)` proporcionan acceso a esta funcionalidad

Identificadores de un Proceso

- Cada proceso tiene un identificador único (*Process ID*, PID) y, además, registra el proceso que lo creó (*Parent Process ID*, PPID)
- Obtener los identificadores de un proceso:

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

  - Estas funciones nunca fallan

<unistd.h>
SV+BSD+POSIX

credentials(7)

Identificadores de un Proceso: Grupos

- Los procesos pertenecen a un grupo de procesos (o trabajo), con un PGID (*Process Group ID*) igual al PID del proceso líder del grupo
- Su principal uso es la distribución de señales
  - Los procesos hijos heredan el grupo de procesos del padre
  - Los procesos en una tubería de la *shell* están en el mismo grupo de procesos
- Obtener o establecer el identificador del grupo de procesos:

`<unistd.h>`
POSIX

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

  - Si `pid` es 0, se refiere al proceso que hace la llamada
  - Si `pgid` es 0, se usa como PGID el PID del proceso indicado en `pid`

`credentials(7)`

Identificadores de un Proceso: Sesiones

- Los grupos de procesos se agrupan en sesiones, con un SID (*Session ID*) igual al PGID del grupo de procesos líder de sesión
  - Un proceso puede crear una sesión si no es el líder de un grupo de procesos (para asegurarse suele hacer `fork(2)` primero), convirtiéndose en el líder de la sesión y de un nuevo grupo de procesos en la sesión
- Las sesiones y los grupos de procesos soportan el **control de trabajos** de la *shell*
  - Todos los procesos en una sesión comparten un terminal de control
  - Un grupo es el *trabajo en primer plano*, el resto son *trabajos en segundo plano*
    - El trabajo en primer plano recibe las señales generadas mediante teclado (ej. Ctrl+C → `SIGINT`) y puede leer del terminal
    - Si un trabajo en segundo plano intenta leer del terminal, es suspendido (con la señal `SIGTTIN`)
  - Al cerrar el terminal, se envía `SIGHUP` a todos los procesos de la sesión
- Obtener el identificador de sesión:

```
pid_t getsid(pid_t pid);
```

`<unistd.h>`
SV+POSIX

- Crear una sesión (y grupo):

```
pid_t setsid(void);
```

- `setsid(1)` ejecuta un programa en una nueva sesión

Directorio de Trabajo

- Es el directorio usado para resolver toda **ruta relativa** en el proceso
- Obtener la **ruta absoluta** del directorio de trabajo:

<unistd.h>
POSIX

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

  - La ruta se copia en `buffer` de tamaño `size`
  - Si el tamaño de la ruta, incluyendo el carácter `'\0'` de fin de cadena, excede `size` bytes, la función devuelve `NULL` y establece **errno=ERANGE**

- Cambiar el directorio de trabajo de un proceso:

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

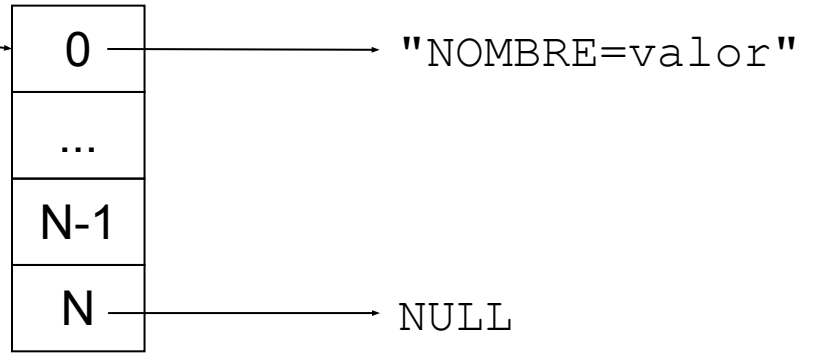
- `pwd(1)` y los comandos de la *shell* `pwd` y `cd` proporcionan acceso a esta funcionalidad

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(1)` o la *shell*
- El entorno es un conjunto de cadenas de caracteres en la forma “NOMBRE=valor”
  - Por convenio, las variables de entorno están en mayúsculas

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

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



- Obtener, establecer o eliminar variables de entorno:

<stdlib.h>
SV+BSD+POSIX

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

- `env(1)` y el comando de la *shell* `export` proporcionan acceso a esta funcionalidad

Creación de Procesos

<unistd.h>
SV+POSIX+BSD

- Crear un proceso hijo:

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

  - Devuelve:
    - 0 al proceso hijo
    - El PID del proceso hijo al padre
    - -1 al proceso padre en caso de fallo, no se crea el proceso hijo y se establece `errno`
  - El nuevo proceso ejecuta el mismo código que el proceso padre y recibe una copia de los descriptores de los ficheros abiertos por el padre

Creación de Procesos

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

    pid = fork();

    switch (pid) {
    case -1:
        perror("fork");
        exit(1);
    case 0:
        printf("Child: %i (parent: %i)\n", getpid(), getppid());
        break;
    default:
        printf("Parent: %i (child: %i)\n", getpid(), pid);
        break;
    }

    return 0;
}
```

Finalización de un Proceso

- Un proceso puede finalizar por dos motivos:
  - Voluntariamente, llamando a `_exit(2)` (o ejecutando **`return`** desde `main()`)
  - Al recibir una señal (hay múltiples causas)
- Terminar el proceso:

`<unistd.h>`
SV+POSIX+BSD

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

  - `status` es el estado de salida, que debe ser un número menor que 255
    - Por convenio, 0 (`EXIT_SUCCESS`) indica éxito y 1 (`EXIT_FAILURE`), error
    - No devolver nunca `errno` ni -1
    - Accesible en la *shell* vía `$?` o en el proceso padre vía `wait(2)`
  - Los descriptores de fichero abiertos por el proceso se cierran
  - Los hijos del proceso quedan huérfanos y son adoptados por un proceso antecesor (`init(1)` o `systemd(1)`)
  - El proceso padre recibe la señal `SIGCHLD`
- La función `exit(3)` llama a las funciones registradas con `atexit(3)` y `on_exit(3)`, vacía (*flush*) y cierra los *streams* abiertos de `stdio(3)`, elimina los ficheros temporales creados por `tmpfile(3)`, y acaba llamando a `_exit(2)`

Finalización de un Proceso

- Si un proceso termina y su padre no le espera, se queda en estado zombi
- Esperar a que un proceso hijo cambie de estado:

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

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

  - `pid` especifica a qué procesos hijos esperar:
    - > 0: El proceso cuyo PID es `pid`
    - 0: Un proceso de su grupo
    - -1: Cualquier proceso (igual que `wait(2)`)
    - < -1: Un proceso del grupo cuyo PGID es `-pid`
  - `options` es una OR bit a bit de las siguientes opciones:
    - `WNOHANG`: retorna sin esperar si no hay hijos que hayan terminado
    - `WUNTRACED`: retorna si un proceso hijo ha sido detenido
    - `WCONTINUED`: retorna si un proceso hijo detenido ha sido reanudado
  - `wstatus` contiene información de estado, que puede consultarse con macros:
    - `WIFEXITED(s)` indica si el hijo terminó normalmente con `_exit(2)` y, en ese caso, `WEXITSTATUS(s)` devuelve el estado de salida
    - `WIFSIGNALED(s)` indica si el hijo terminó al recibir una señal y, en ese caso, `WTERMSIG(s)` devuelve el número de la señal recibida
  - Devuelve el PID del hijo terminado o -1 en caso de error

Ejecución de Programas

<unistd.h>
POSIX+SV+BSD

- Ejecutar un programa:

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

  - Reemplaza la imagen del proceso actual por una nueva
  - `path` es la ruta a un ejecutable ELF o un guión que empiece por

```
#!interpreter [optional-arg]
```

  - El primer elemento de `argv` es el nombre del programa y el último es `NULL`

- Las familia de funciones `exec(3)` usa `execve(2)`:

```
int execl(const char *path, const char *a0, ...);
int execlp(const char *file, 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 *file, char *const argv[]);
```

Argumentos	Ruta abs./rel.	Busca en `PATH`	Nuevo entorno
Lista	`execl()`	`execlp()`	`execle()`
Vector	`execv()`	`execvp()`	`execve()`

Ejecución de Programas

<stdlib.h>
ANSI C+POSIX

- Ejecutar un comando de la *shell*:

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

  - Crea un proceso hijo con `fork(2)`
  - El proceso hijo ejecuta el comando especificado en `command` usando `execl(3)`:

```
execl("/bin/sh", "sh", "-c", command, (char *) NULL);
```

  - El proceso padre espera al hijo con `waitpid(2)`, obteniendo su estado de finalización
  - Devuelve el estado de finalización del proceso hijo, 127 si no se pudo ejecutar la *shell* o -1 en caso de error (y establece `errno`)

Límites de Recursos

<sys/time.h>
<sys/resource.h>

SV+BSD+POSIX

- 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;    /* Valor máximo */
};
```

  - `resource` indica el recurso limitado y puede ser, por ejemplo:
    - `RLIMIT_CPU`: Tiempo de CPU (segundos)
    - `RLIMIT_FSIZE`: Tamaño de fichero (bytes)
    - `RLIMIT_DATA`: Tamaño del segmento de datos (bytes)
    - `RLIMIT_STACK`: Tamaño de la pila (bytes)
    - `RLIMIT_CORE`: Tamaño de fichero de volcado de memoria (bytes)
    - `RLIMIT_NPROC`: Número de procesos
    - `RLIMIT_NOFILE`: Número de descriptores de fichero
  - `rlim` especifica el límite (el valor `RLIM_INFINITY` indica ilimitado)
- El comando de la *shell* `ulimit` proporciona acceso a esta funcionalidad

Uso de Recursos

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

- Obtener el uso de recursos:

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

struct rusage {
    struct timeval ru_utime;/* t. CPU en modo usuario */
    struct timeval ru_stime;/* t. CPU en modo sistema */
    long   ru_maxrss;       /* RSS máximo */
    long   ru_minflt;       /* páginas reclamadas */
    long   ru_majflt;       /* fallos de página */
    long   ru_inblock;      /* ops. de entrada de bloques */
    long   ru_oublock;      /* ops. de salida de bloques */
    ...                     /* ver man getrusage */
}
```

  - `who` puede ser
    - `RUSAGE_SELF`: el proceso (todos sus hilos)
    - `RUSAGE_CHILDREN`: los hijos del proceso terminados y esperados
    - `RUSAGE_THREAD`: el hilo

- `time(1)` con la opción `-v` proporciona esta información (`time` también es una palabra reservada de la *shell*)

Señales

Señales

- Las señales son **interrupciones software**, que informan a un proceso de la ocurrencia de un evento de forma **asíncrona**
  - Las genera un proceso o el núcleo del sistema
- Las opciones en la ocurrencia de un evento son:
  - Bloquear la señal
  - Ignorar la señal
  - Realizar la acción por defecto asociada a la señal, que en general consiste en terminar la ejecución del proceso
  - Capturar la señal con un manejador, que es una función definida por el programador que se invoca automáticamente al recibir la señal
- 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 (**F**, terminar proceso)
- **SIGINT**: Interrupción. Se puede generar con `Ctrl+C` (**F**)
- **SIGQUIT**: Finalización. Se puede generar con `Ctrl+\` (**F** y **C**, volcado de mem.)
- **SIGSTOP**: Parar proceso. No se puede capturar, bloquear ni ignorar (**P**, parar)
- **SIGTSTP**: Parar proceso. Se puede generar con `Ctrl+Z` (**P**)
- **SIGCONT**: Reanudar proceso parado (continuar)
- **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 se puede capturar, bloquear ni ignorar (**F**)
- **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**)
- **SIGALRM**: Despertador, contador a 0 (**F**)
- **SIGTERM**: Terminar proceso (**F**)
- **SIGUSR1** y **SIGUSR2**: Señales de usuario (**F**)
- **SIGCHLD**: Terminación del proceso hijo (**I**, ignorar)
- **SIGURG**: Recepción de datos urgentes en socket (**I**)

`signal(7)`

Señales: Envío

- Enviar una señal a un proceso:

<signal.h>
SV+BSD+POSIX

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

  - `pid` indica a qué procesos se enviará la señal:
    - >0: Al proceso con ese PID
    - 0: A todos los procesos de su grupo
    - -1: A todos los procesos (de mayor a menor), excepto el propio proceso y el de PID 1 (`init(1)` o `systemd(1)`)
    - <-1: A todos los procesos del grupo cuyo PGID es `-pid`
  - `signal` es la señal que se enviará (si es 0, se simula el envío)
- `kill(1)` y el comando de la *shell* `kill` proporcionan acceso a esta funcionalidad
- Funciones equivalentes:

<signal.h>
ANSI-C+POSIX

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

<stdlib.h>
SV+BSD+POSIX

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

Señales: Ejemplo de Envío

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

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

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

Señales: Conjuntos de Señales

- Las señales se agrupan en conjuntos de señales POSIX para definir máscaras de señales
  - Tipo opaco `sigset_t` que depende del sistema
  - Implementado como mapa de bits
- Operaciones con conjuntos de señales POSIX:

`<signal.h>`
POSIX

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

  - `sigemptyset()` inicializa un conjunto sin señales
  - `sigfillset()` inicializa un conjunto con todas las señales
  - `sigaddset()` añade una señal a un conjunto
  - `sigdelset()` elimina una señal de un conjunto
  - `sigismember()` comprueba si una señal pertenece a un conjunto

`sigsetops(3)`

Señales: Bloqueo

<signal.h>
POSIX

- La máscara de señales es el conjunto de señales bloqueadas (por ejemplo, para proteger regiones de código)
- Consultar y establecer la máscara de señales:

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

  - `how` define el comportamiento:
    - `SIG_BLOCK`: Añade el conjunto `set` a la máscara de señales
    - `SIG_UNBLOCK`: Elimina el conjunto `set` de la máscara de señales (puede desbloquearse una señal que no estuviera bloqueada)
    - `SIG_SETMASK`: Establece el conjunto `set` como máscara de señales
  - Devuelve en el conjunto `oset` la máscara de señales anterior (si no es `NULL`)
- Consultar señales pendientes:

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

  - Devuelve en `set` el conjunto de señales pendientes, es decir, recibidas estando bloqueadas
  - Usar `sigismember(3)` para determinar la señal y `sigprocmask(2)` para desbloquearla y tratarla

Señales: Ejemplo de Bloqueo

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

int main() {
   sigset_t blk_set;

   sigemptyset(&blk_set);
   sigaddset(&blk_set, SIGINT);          /* Ctrl+C */
   sigaddset(&blk_set, SIGQUIT);         /* Ctrl+\ */
   sigaddset(&blk_set, SIGTSTP);         /* Ctrl+Z */

   sigprocmask(SIG_BLOCK, &blk_set, NULL);

   /* Critical code */

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

Señales: Captura

- Es posible modificar la acción por defecto realizada por un proceso al recibir una señal definiendo una función manejadora de la señal (*handler*)
- Obtener y establecer la acción asociada a una señal:

<signal.h>
POSIX+SV

```
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 (excepto `SIGKILL` y `SIGSTOP`)
  - `act` contiene la nueva acción para la señal (puede ser `NULL`)
  - Devuelve en `oldact` la antigua acción de la señal (si no es `NULL`)

Señales: Captura

- Campos de la estructura `sigaction`:
  - `sa_handler` es el nuevo manejador para la señal. Su valor puede ser:
    - `SIG_DFL` para la acción por defecto
    - `SIG_IGN` para ignorar la señal
    - Un puntero a una función:

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

  - `sa_mask` es la máscara de señales a aplicar durante el tratamiento de la señal
    - Por defecto, también se bloquea la señal que se está tratando
  - `sa_flags` modifica el comportamiento del proceso de gestión de la señal:
    - `SA_NODEFER` no bloquea la señal que se está tratando
    - `SA_RESTART` reinicia ciertas llamadas al sistema interrumpidas para compatibilidad con BSD (en otro caso, fallan con `errno=EINTR`)
    - `SA_RESETHAND` restaura el manejador por defecto tras tratar la señal
    - `SA_SIGINFO` usa una función con argumentos adicionales para tratar la señal (campo `sa_sigaction`)

Señales: Captura

- La ejecución del proceso se interrumpe y se llama al manejador
  - Cuando el manejador termina, se restaura la ejecución en el punto donde se produjo la señal
- Hay que tomar algunas precauciones en el manejador:
  - Declarar las variables globales como **`volatile`**, para indicar al compilador que su valor puede cambiar de forma inespera
  - No usar **funciones no reentrantes**, como `malloc(3)`, `free(3)` o funciones de la biblioteca `stdio`
  - Guardar y restaurar el valor de `errno` si llama a alguna función que pueda modificarlo
- Como regla general, hacer lo menos posible en el manejador
  - Normalmente, fijar una variable de condición y salir
- Tener en cuenta siempre que las señales son **asíncronas**

Señales: Espera

<signal.h>
POSIX

- Esperar una señal:

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

  - La máscara de señales se sustituye temporalmente por `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, restaurando la **máscara original**
  - Siempre devuelve -1 y, normalmente, establece `errno` a `EINTR`

<unistd.h>
POSIX

- Esperar una señal de forma más sencilla:

```
unsigned int sleep(unsigned int seconds);
int pause(void);
```

  - Suspenden la ejecución durante los segundos especificados o indefinidamente, respectivamente, o hasta recibir una señal que deba ser tratada

Señales: Alarmas y Temporizadores

<unistd.h>
SV+BSD+POSIX

- Fijar una alarma:

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

  - Se programa el temporizador `ITIMER_REAL` para generar una señal `SIGALRM` en `secs` segundos (si es cero, no se planifica ninguna nueva alarma)
    - Cualquier alarma programada previamente se cancela
    - Debe instalarse antes un manejador
  - Devuelve el valor de segundos restantes para que se produzca el final de la cuenta (0 si no hay ninguna fijada)
  - No mezclar con otras funciones que usen el mismo temporizador o señal, como `setitimer(2)` (o `sleep(3)` en algunos sistemas)
  - Las alarmas se mantienen tras `execve(2)`, pero no se heredan con `fork(2)`

Señales: Alarmas y Temporizadores

<sys/time.h>
SV+BSD

- Consultar o fijar alarmas asociadas a otros temporizadores:

```
int getitimer(int which,
        struct itimerval *curr_value);
int setitimer(int which, struct itimerval *new_value,
        struct itimerval *old_value);

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

  - `which` selecciona el temporizador:
    - `ITIMER_REAL`: Tiempo real (*wall-clock*), genera `SIGALRM`
    - `ITIMER_VIRTUAL`: Tiempo de CPU en modo usuario, genera `SIGVTALRM`
    - `ITIMER_PROF`: Tiempo de CPU total (es decir, en modo usuario y sistema), genera `SIGPROF`

Comunicación entre Procesos. Tuberías

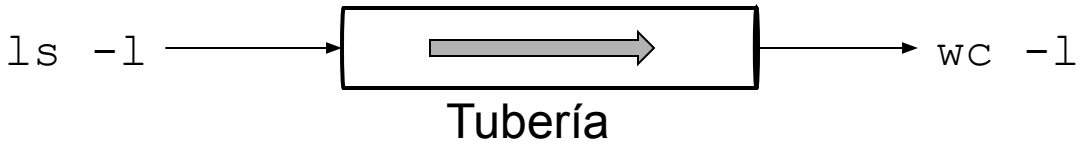
Introducción

- Mecanismos de sincronización:
  - Mismo sistema
    - Señales (Tema 2.3)
    - Ficheros con cerrojos (Tema 2.2)
    - Mutex y variables de condición (solo para hilos de un proceso)
    - Semáforos (System V IPC)
    - Colas de mensajes (System V IPC)
  - Distintos sistemas
    - Basados en sockets (Tema 2.4)
- Compartición de datos entre procesos:
  - Mismo sistema
    - Memoria compartida (System V IPC)
    - Tuberías sin nombre o *pipes* (Tema 2.3)
    - Tuberías con nombre o FIFOs (Tema 2.3)
    - Colas de mensajes (System V IPC)
    - Basados en ficheros (Tema 2.2)
  - Distintos sistemas
    - Basados en sockets (Tema 2.4)

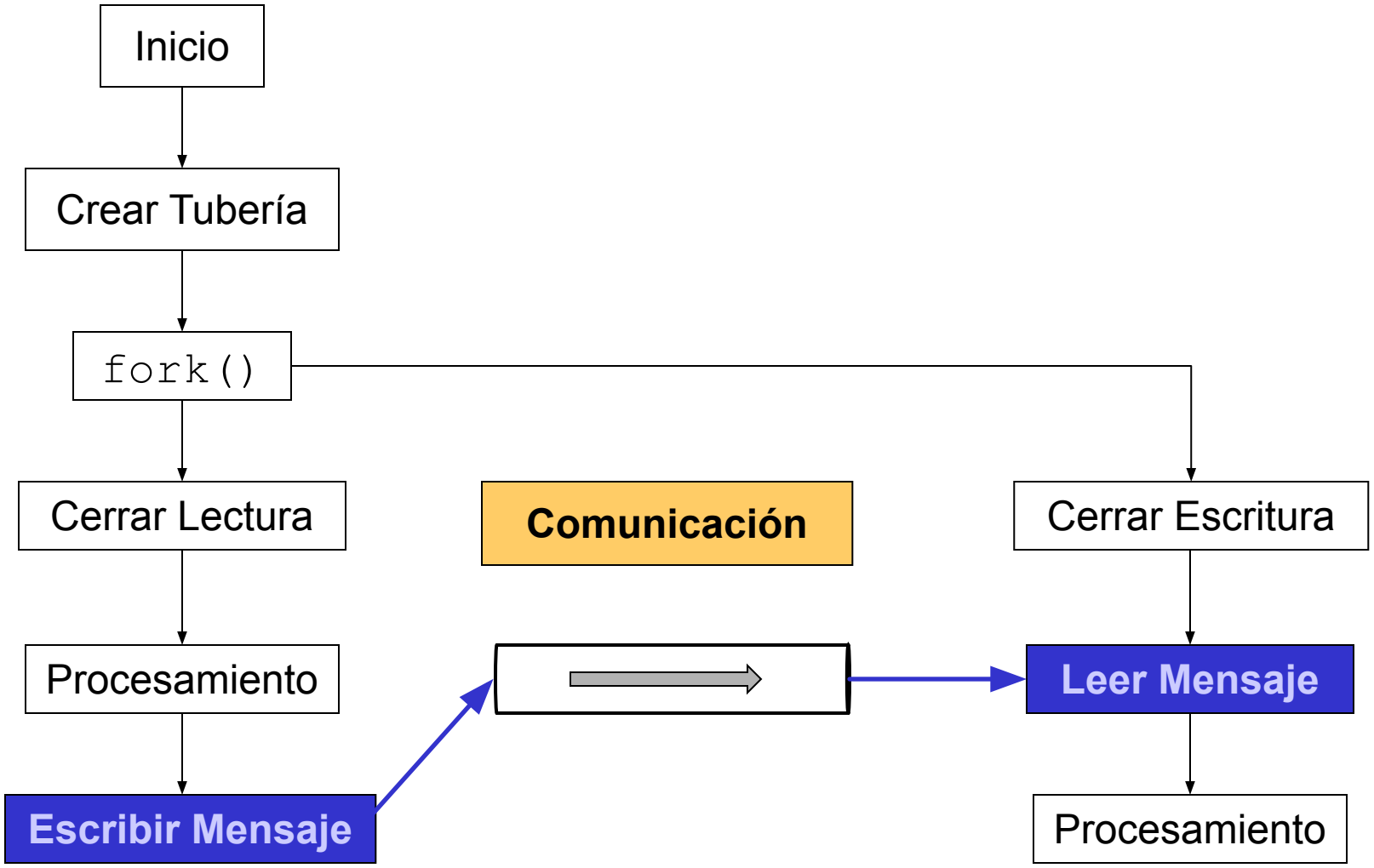
Tuberías sin Nombre

- Proporcionan un canal de comunicación unidireccional entre procesos
- El sistema las **trata** a todos los efectos **como ficheros:**
  - Inodo
  - 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**

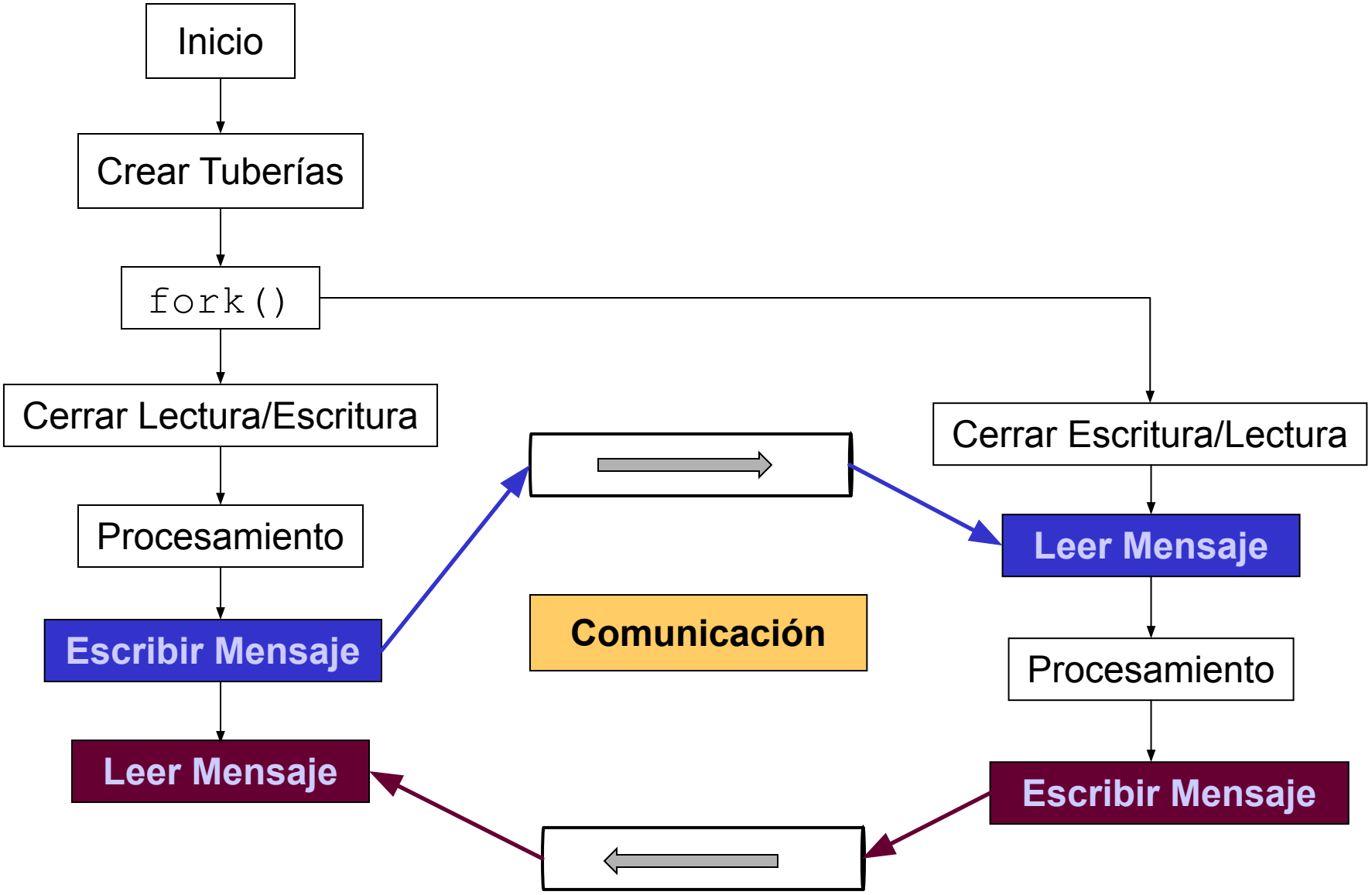
```
$ ls -l | wc -l
```



Tuberías sin Nombre: Unidireccional



Tuberías sin Nombre: Bidireccional



Tuberías sin Nombre

<unistd.h>

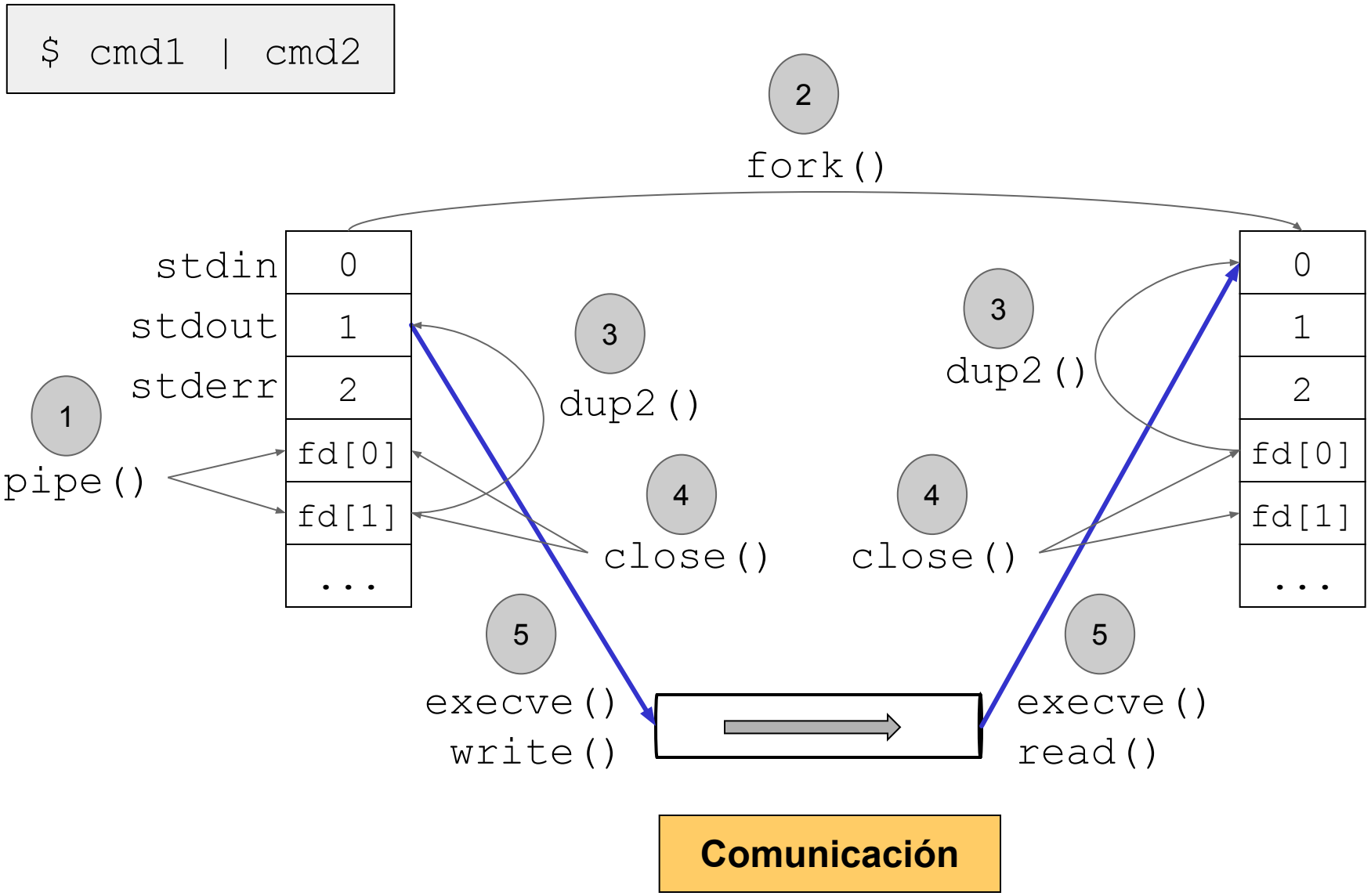
SV+BSD+POSIX

- Crear una tubería:

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

  - Devuelve en `fd[0]` el extremo de lectura:
    - Si la tubería está vacía, `read(2)` se bloqueará hasta que haya datos disponibles
    - Si todos los **descriptores de escritura se han cerrado**, `read(2)` devolverá cero, indicando el fin de fichero
  - Devuelve en `fd[1]` el extremo de escritura:
    - Si la tubería se llena, `write(2)` se bloqueará hasta que se lean suficientes datos para que se pueda completar la escritura
    - Si todos los **descriptores de lectura se han cerrado**, `write(2)` enviará la señal `SIGPIPE` al proceso y, si se ignora la señal, fallará con `errno=EPIPE`
  - Es importante cerrar los descriptores que no se usen para que se indique el fin de fichero o se envíe `SIGPIPE`/`EPIPE` cuando sea necesario

Tuberías sin Nombre: Ejemplo



Tuberías con Nombre

- La comunicación mediante **tuberías sin nombre** se puede realizar únicamente entre **procesos con relación de parentesco**
- Una **tubería con nombre**, o **fichero especial FIFO**, es similar a una tubería sin nombre, salvo que se accede a ella mediante el sistema de ficheros
  - La entrada en el sistema de ficheros solo sirve para que los procesos abran la misma tubería con `open(2)` usando un nombre
  - El núcleo realiza la sincronización y almacena los datos internamente, sin escribirlos en el sistema de ficheros
  - El extremo de lectura se abre con `O_RDONLY` y el de escritura, con `O_WRONLY`
  - Varios procesos pueden abrir la tubería para lectura o escritura
- Deben abrirse ambos extremos antes de poder intercambiar datos
  - Normalmente, la apertura de un extremo se bloquea hasta que se abre el otro extremo
  - En modo no bloqueante (opción `O_NONBLOCK`), la apertura para lectura no se bloqueará aunque la tubería no esté abierta para escritura

Tuberías con Nombre

<sys/stat.h>
SV+BSD

- Crear ficheros especiales:

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

  - `mode` especifica el tipo de fichero a crear y sus permisos (modificados por *umask*), combinados con una OR bit a bit
  - El tipo puede de ser:
    - `S_IFREG`: Fichero regular
    - `S_IFCHR`: Dispositivo de caracteres (`dev` = *major,minor*)
    - `S_IFBLK`: Dispositivo de bloques (`dev` = *major,minor*)
    - **`S_IFIFO`**: Tubería con nombre
    - `S_IFSOCK`: Socket UNIX

- `mknod(1)` proporciona acceso a esta funcionalidad:

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

  - `tipo` 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);
```

  - `mode` especifica los permisos (modificados por *umask*) con que se creará la tubería

- `mkfifo(1)` proporciona acceso a esta funcionalidad:

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

Sincronización de E/S

- Cuando un proceso gestiona varios canales de E/S (tubería, socket o terminal), no debe bloquearse indefinidamente en uno de ellos mientras otros están listos para realizar una operación
- Alternativas:
  - E/S no bloqueante: Opción `O_NONBLOCK`
    - En lugar de bloquearse, la llamada falla con `errno=EAGAIN`
    - Es como la E/S por encuesta y consume tiempo de CPU innecesariamente, ya que el proceso nunca se bloquea, incluso si ningún descriptor está listo
  - E/S guiada por eventos: Opción `O_ASYNC`
    - El proceso recibe una señal (por defecto, `SIGIO`) cuando el descriptor está preparado para realizar la operación
    - La gestión de señales asíncronas modifica la lógica del programa
  - **Multiplexación de E/S síncrona**: `select(2)`, `poll(2)` y `epoll(7)`
    - El proceso monitoriza varios descriptores a la vez, esperando a que uno o varios estén listos para realizar una operación de E/S determinada de forma **síncrona**

Multiplexación de E/S Síncrona

<sys/select.h>
POSIX+BSD

- Seleccionar descriptores de fichero preparados:

```
int select(int nfds, fd_set *readfds,
           fd_set *writefds, fd_set *exceptfds,
           struct timeval *timeout);
```

  - `readfds` es el conjunto de descriptores de lectura
  - `writefds` es el conjunto de descriptores de escritura
  - `exceptfds` es el conjunto de descriptores de condiciones excepcionales
    - Por ejemplo, que haya datos urgentes (*out-of-band*) en un socket TCP
  - `nfds` es el mayor de los descriptores en los tres conjuntos, más 1
  - `timeout` es el tiempo máximo en el que retornará la función
    - Si es `{0, 0}`, retorna inmediatamente
    - Si es `NULL`, se bloquea hasta que se produce un cambio
  - Devuelve el número de descriptores listos o 0 si expira el tiempo máximo
    - Los conjuntos se modifican para indicar qué descriptores están listos para cada operación
  - Si se produce un error, los conjuntos no se modifican y `timeout` queda indeterminado

Multiplexación de E/S Síncrona

- Macros para la manipular los conjuntos (`select(2)`):

```
void FD_ZERO(fd_set *set);
void FD_SET(int fd, fd_set *set);
void FD_CLR(int fd, fd_set *set);
int FD_ISSET(int fd, fd_set *set);
```

  - `FD_ZERO()` inicializa un conjunto como conjunto vacío
  - `FD_SET()` añade un descriptor a un conjunto
  - `FD_CLR()` elimina un descriptor de un conjunto
  - `FD_ISSET()` comprueba si un descriptor está en un conjunto, lo cual es útil después de que la llamada retorna

Multiplexación de E/S Síncrona

```c
#include <...>

int main (void) {
    fd_set set;
    struct timeval timeout;
    char buffer[80];

    while (1) {
        FD_ZERO(&set);
        FD_SET(0, &set);

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

        int changes = select(1, &set, NULL, NULL, &timeout);

        if (changes == -1) {
            perror("select()");
        } else if (changes) {
            if (FD_ISSET(0, &set)) {
                int bytes = read(0, buffer, 80);
                buffer[bytes-1] = '\0';
                printf("New data: %s.\n", buffer);
            }
        } else {
            printf("No new data in two seconds.\n");
        }
    }
}
```

Ejemplos de Preguntas Teóricas

¿En qué política de planificación se usa el valor de *nice*?

- ☐ SCHED_NICE.
- ☐ SCHED_OTHER.
- ☐ SCHED_RR.

¿En qué se diferencia una tubería sin nombre y una tubería con nombre?

- ☐ En la forma de crearse.
- ☐ En el patrón de comunicación.
- ☐ En que la comunicación en una se realiza en memoria y en otra, a través del sistema de ficheros.

¿Para qué sirve la opción SA_RESTART al instalar un manejador de señal?

- ☐ Para reiniciar el manejador de señal por defecto tras tratar la señal.
- ☐ Para reiniciar la señal una vez tratada.
- ☐ Para reiniciar la llamada al sistema interrumpida.