



AMPLIACIÓN DE SISTEMAS OPERATIVOS Y REDES

Grado en Ingeniería Informática / Doble Grado

Universidad Complutense de Madrid

TEMA 2.3. Gestión de Procesos

PROFESORES:

Rubén Santiago Montero

Eduardo Huedo Cuesta

Rafael Rodríguez Sánchez

Estructura de un Programa

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

Executable and Linking Format (ELF)

Cabecera ELF
Tabla de Programa
Otra Información
Segmento de Texto (Código ejecutable)
Segmento de Datos (Variables estáticas y globales)
Otra Información

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_386, EM_X86_64,
EM_IA_64, EM_SPARC...

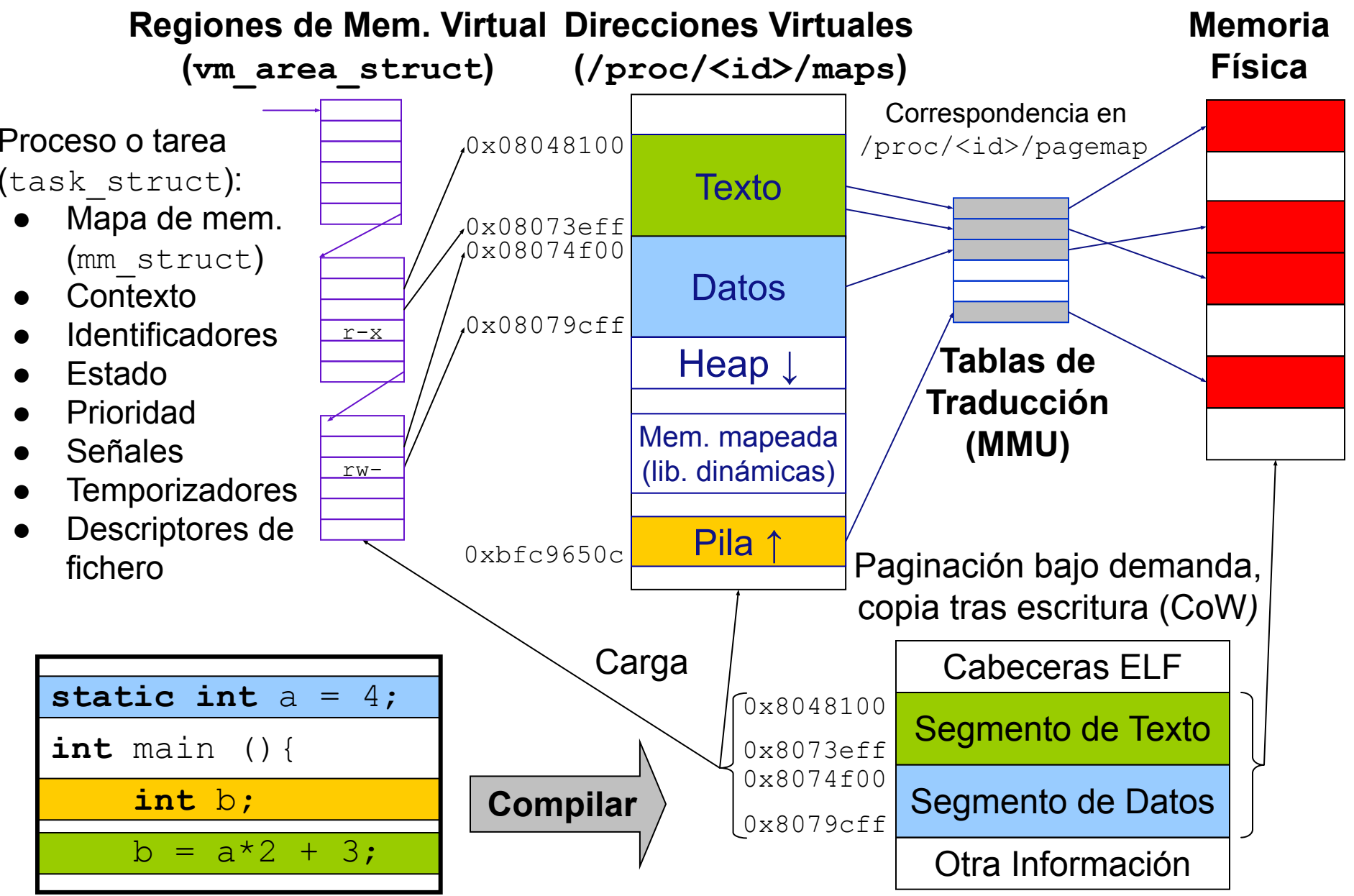
Información para ejecución

```
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
...

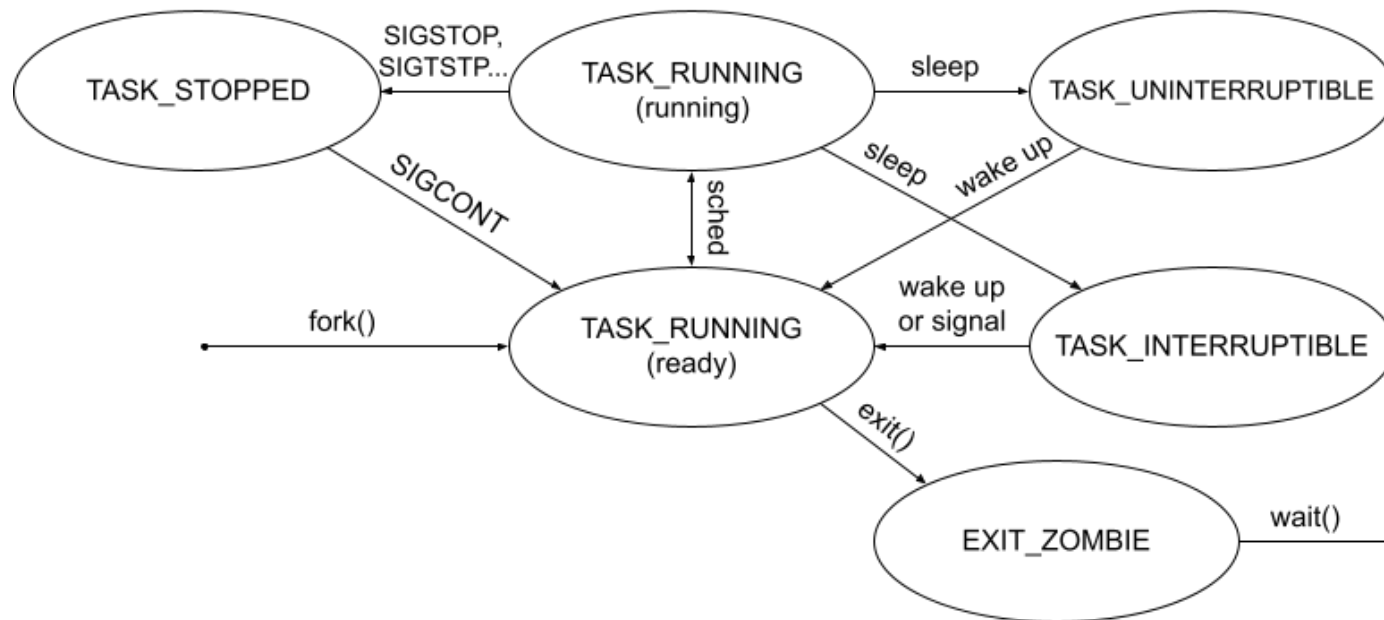
Dirección virtual del segmento

Estructura de un Proceso



Estados de un Proceso

- `TASK_RUNNING`: en ejecución o preparado (en la cola de ejecución), R en `ps`
- `TASK_UNINTERRUPTIBLE`: bloqueado (normalmente por E/S), D en `ps`
- `TASK_INTERRUPTIBLE`: esperando (a que un evento se complete), S en `ps`
- `TASK_STOPPED`: parado (por una señal), T en `ps`
- `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), Z en `ps`

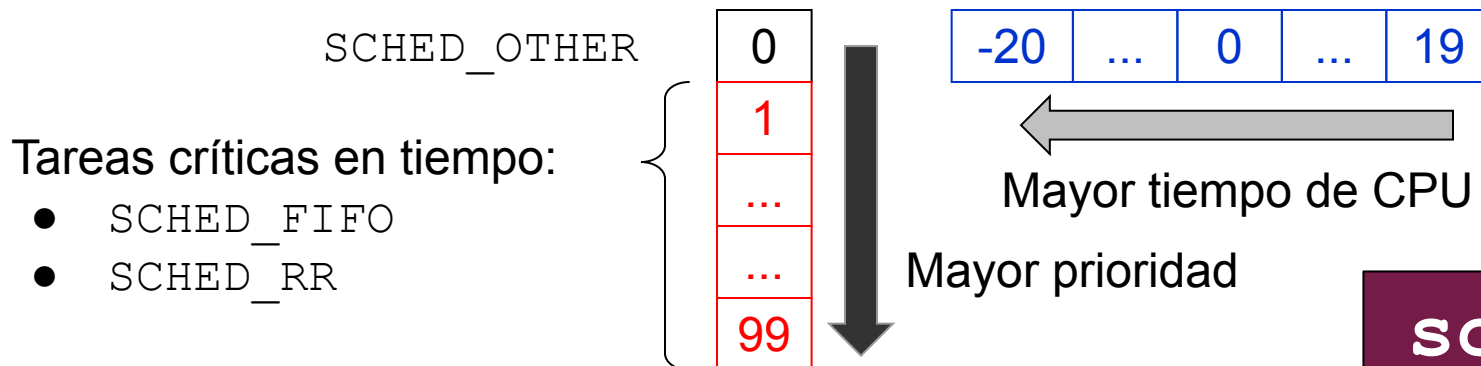


- El comando `ps` muestra la lista de procesos, sus identificadores y sus atributos, incluyendo el estado

ps (1)

Planificador

- Componente del núcleo que determina el orden de ejecución de las tareas en función de su prioridad y de la clase de planificación
 - Es expropiativo (una tarea de mayor prioridad siempre expropiará a otra de menor prioridad en ejecución), y la política de planificación solo determina el orden de ejecución de la lista de tareas preparadas con igual prioridad
- **Políticas de planificación** (ver `/usr/include/bits/sched.h`)
 - **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
 - Una tarea de esta política se ejecutará hasta que se bloquee por E/S, sea expropiada por una tarea con mayor prioridad o ceda la CPU
 - **SCHED_RR**: Como la anterior, pero los tareas con igual prioridad se ejecutan por turnos (*round-robin*) durante un *cuanto* de tiempo máximo



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 (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* o tareas)
 - Todas las llamadas tienen su equivalente `pthread_*`
- Las llamadas `fork()` heredan los atributos de planificación

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 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

- El comando `chrt` (*change real-time*) ofrece acceso a esta funcionalidad

Planificador

<sys/time.h>
<sys/resource.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, un PGID o un UID, respectivamente
 - 0 indica el proceso actual, el grupo de procesos del proceso 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
- Los comandos `nice` y `renice` permiten acceder 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) ;
```

<unistd.h>
SV+BSD+POSIX

credentials (7)

Identificadores de un Proceso: Grupos

- Los procesos pertenecen a un grupo de procesos, con un PGID (Process Group ID) igual al PID del proceso líder del grupo
 - Su principal uso es la distribución de señales
- Obtener o establecer el identificador del grupo de procesos:
 - Si `pid` es 0, se refiere al proceso que hace la llamada
 - Si `pgid` es 0, se usa el PID del proceso indicado en `pid`

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

<unistd.h>

POSIX

Identificadores de un Proceso: Sesiones

- Los grupos de procesos se pueden agrupar en sesiones, con un SID (Session ID)
 - Se usan para gestionar el acceso al sistema
- Un proceso de *login* crea una sesión y abre un terminal de control
 - Todos los procesos y grupos de procesos del usuario pertenecen a esa sesión y comparten el terminal
 - En la desconexión, se envía la señal `SIGHUP` a todos los procesos de la sesión
- Un proceso puede crear una sesión si no es el líder de un grupo de procesos
 - Para asegurarse de que no es el líder, suele hacer `fork(2)` primero
 - El proceso es el líder de la sesión (su SID se establece a su PID) y de un nuevo grupo de procesos en la sesión (su PGID se establece a su PID)
 - Inicialmente, la sesión no tiene terminal de control

- Obtener el identificador de sesión:

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

<code><unistd.h></code>

SV+POSIX

- Crear una nueva sesión y un nuevo grupo de procesos:

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

- El comando `setsid` permite crear una nueva sesión (y un nuevo grupo asociado)

Directorio de Trabajo

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

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

<unistd.h>

POSIX

- 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);
```

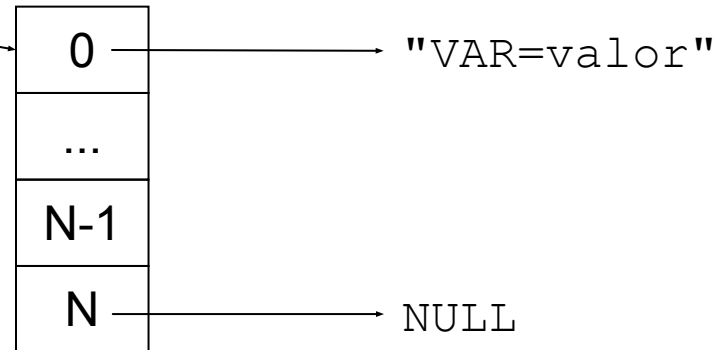
- El comando `pwd` y los comandos internos de la *shell* `pwd` y `cd` proporcionan acceso a esta funcionalidad

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` 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
- ...



- Obtener, establecer o eliminar variables de entorno:

```
char *getenv(const char *name);
```

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

```
int unsetenv(const char *name);
```

`<stdlib.h>`

SV+BSD+POSIX

- El comando interno de la *shell* `export` y el comando `env` proporcionan acceso a esta funcionalidad

Creación de Procesos

<unistd.h>

SV+POSIX+BSD

- Crear un proceso hijo:

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

- La función devuelve:
 - 0: Ejecutando el hijo
 - >0: Ejecutando el padre, el valor es el PID del hijo
 - -1: Fallo
- 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

```
int main() {
    pid_t pid;

    pid = fork();

    switch (pid) {
        case -1:
            perror("fork");
            exit(1);

        case 0:
            printf("Hijo %i (padre: %i)\n", getpid(), getppid());
            break;

        default:
            printf("Padre %i (hijo: %i)\n", getpid(), pid);
            break;
    }

    return 0;
}
```

Finalización de un Proceso

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

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

 - `status` es el estado de salida, que debe ser un número menor que 255
 - Por convenio, 0 (`EXIT_SUCCESS`) significa éxito y 1 (`EXIT_FAILURE`) significa error
 - Nunca devolver `errno` ni -1
 - Accesible en la *shell* vía `$?` o en el proceso padre vía `wait(2)`
 - Cualquier descriptor de fichero abierto por el proceso se cierra
 - Cualquier hijo del proceso se vuelve huérfano y es heredado por el proceso 1 (`init` o `systemd`)
 - El proceso padre recibe una señal `SIGCHLD`

Finalización de un Proceso

`<sys/types.h>`

`<sys/wait.h>`

SV+POSIX+BSD

- Esperar la finalización (o cambio de estado) de un proceso hijo:

```
pid_t wait(int *status);
```

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

- `pid` especifica a qué procesos hijo esperar:
 - `> 0` indica uno con identificador `pid`
 - `0` indica uno del grupo de procesos del padre
 - `-1` indica cualquiera (igual que `wait()`)
 - `< -1` indica el que su PGID es `-pid`
- `options` es una OR de las siguientes opciones:
 - `WNOHANG`: retorna sin esperar si no hay hijos que hayan terminado
 - `WUNTRACED`: retorna si el proceso ha sido detenido
 - `WCONTINUED`: retorna si un hijo detenido ha sido reanudado
- `status` contiene información de estado, que puede consultarse con macros:
 - `WIFEXITED(s)` indica si el hijo terminó normalmente vía `exit()` 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
- Un proceso que termina pero no ha sido esperado se convierte en zombi

Ejecución de Programas

<unistd.h>

POSIX

- Ejecutar un programa:

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

- Sustituye la imagen del proceso actual por una nueva
- El primer elemento de `argv` debe ser el nombre de fichero del programa (ejecutable binario o *script*) y el último ha de ser `NULL`

- Las siguientes funciones usan la llamada `execve(2)`:

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

	Ruta absoluta	Ruta relativa	Nuevo entorno
Lista de argumentos	<code>execl()</code>	<code>execlp()</code>	<code>execl_e()</code>
Vector de argumentos	<code>execv()</code>	<code>execvp()</code>	<code>execve()</code>

Ejecución de Programas

<stdlib.h>

ANSI C+POSIX

- Ejecutar un comando de la *shell*:

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

- Usa `fork(2)` para crear un proceso hijo que ejecute el comando de la *shell* especificado en `command` usando `execl(3)` como:

```
execl("/bin/sh", "sh", "-c", command, (char *) 0);
```

- La llamada retorna cuando termina la ejecución del comando (salvo si se ejecuta en segundo plano)
- Devuelve el código de finalización del comando, obtenido con `waitpid(2)`, o -1 en caso de error

Límites de Recursos

<sys/time.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;    /* Valor máximo */
};
```

- resource puede ser
 - RLIMIT_CPU: Max. tiempo de CPU (segundos)
 - RLIMIT_FSIZE: Max. tamaño de fichero (bytes)
 - RLIMIT_DATA: Max. tamaño del heap (bytes)
 - RLIMIT_STACK: Max. tamaño de pila (bytes)
 - RLIMIT_CORE: Max. tamaño de fichero core (bytes)
 - RLIMIT_NPROC: Max. número de procesos
 - RLIMIT_NOFILE: Max. número de descriptores de fichero
- rlim especifica el límite (el valor RLIM_INFINITY indica ilimitado)
- El comando interno de la *shell* `ulimit` proporciona acceso a esta llamada

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: por el proceso (todos sus *threads*)
- RUSAGE_CHILDREN: por todos los hijos del proceso
- RUSAGE_THREAD: por el *thread*

- El comando `time -v` proporciona esta información (`time` es también una palabra reservada de la *shell*)



AMPLIACIÓN DE SISTEMAS OPERATIVOS Y REDES

Grado en Ingeniería Informática / Doble Grado

Universidad Complutense de Madrid

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 o 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 o 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**, **SIGUSR2**: Señales de usuario (**F**)
- **SIGCHLD**: Terminación del proceso hijo (**I**, ignorar)
- **SIGURG**: Recepción de datos urgentes en socket (**I**)

signal(7)

Señales: Envío

<signal.h>

SV+BSD+POSIX

- Enviar una señal a un proceso:

```
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 el 1
 - <-1: Se envía a todos los procesos del grupo con PGID igual a `-pid`
- `signal` es la señal que se enviará (si es 0, se simula el envío)

- El comando `kill`, que también es un comando interno de la *shell*, proporciona acceso a esta llamada

- Llamadas equivalentes:

```
int raise(int signal);
```

```
int abort(void);
```

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

<signal.h>

ANSI-C

<stdlib.h>

SV+BSD+POSIX

Señales: Ejemplo de Envío

```
#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 (ver `/usr/include/bits/sigset.h`)

- Operaciones con conjuntos de señales POSIX:

<code><signal.h></code>
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 como vacío, excluyendo todas las señales
- `sigfillset()` inicializa un conjunto como lleno, incluyendo 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

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 las señales bloqueadas:

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

- `how` define el comportamiento:

- `SIG_BLOCK`: Añade el conjunto `set` al conjunto de señales actualmente bloqueadas (“OR”)
- `SIG_UNBLOCK`: Elimina el conjunto `set` del conjunto de señales bloqueadas (puede desbloquearse una señal que no estuviera bloqueada)
- `SIG_SETMASK`: Reemplaza el conjunto de señales actuales por `set`

- `oset` almacena el conjunto previo de señales bloqueadas (distinto de `NULL`)

- Comprobar señales pendientes:

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

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

Señales: Ejemplo de Bloqueo

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

int main() {
    sigset_t blk_set;

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

    sigprocmask(SIG_BLOCK, &blk_set, NULL);

    /* Código protegido */

    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:

```
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 el nuevo manejador para la señal (puede ser `NULL`)
- `oldact` almacenará el antiguo controlador de la señal (puede ser `NULL`)

<signal.h>
POSIX

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 el conjunto de señales que serán bloqueadas durante el tratamiento de la señal
 - Además, por defecto se bloquea la señal en cuestión
 - `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 para tratar la señal con argumentos adicionales (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`
 - No usar **funciones no reentrantes**, como `malloc`, `free` 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 algún *flag* y salir
- Tener en cuenta siempre que las señales son **asíncronas**

Señales: Espera

<signal.h>

POSIX

- 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, restaurando la **máscara original**
 - Siempre devuelve -1 y, normalmente, establece `errno` a `EINTR`
- Alternativamente, suspender un proceso de forma más sencilla:

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

<unistd.h>

POSIX

- Suspenden la ejecución durante los segundos especificados o indefinidamente, respectivamente, o bien, 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 `sleep(3)` o cualquier otra función que use el mismo temporizador, como `setitimer(2)`
- No se heredan con `fork(2)`, pero sí se mantienen tras `execve(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 que queda */
}
```

- `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`



AMPLIACIÓN DE SISTEMAS OPERATIVOS Y REDES

Grado en Ingeniería Informática / Doble Grado

Universidad Complutense de Madrid

Comunicación entre Procesos. Tuberías

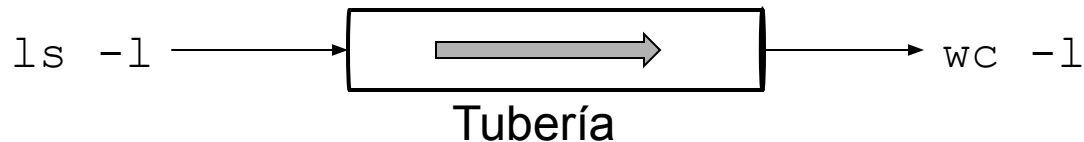
Introducción

- Mecanismos de sincronización:
 - Mismo sistema
 - Señales (Tema 2.3)
 - Ficheros con cerrojos (Tema 2.2)
 - Mutex y variables de condición (solo para threads 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)

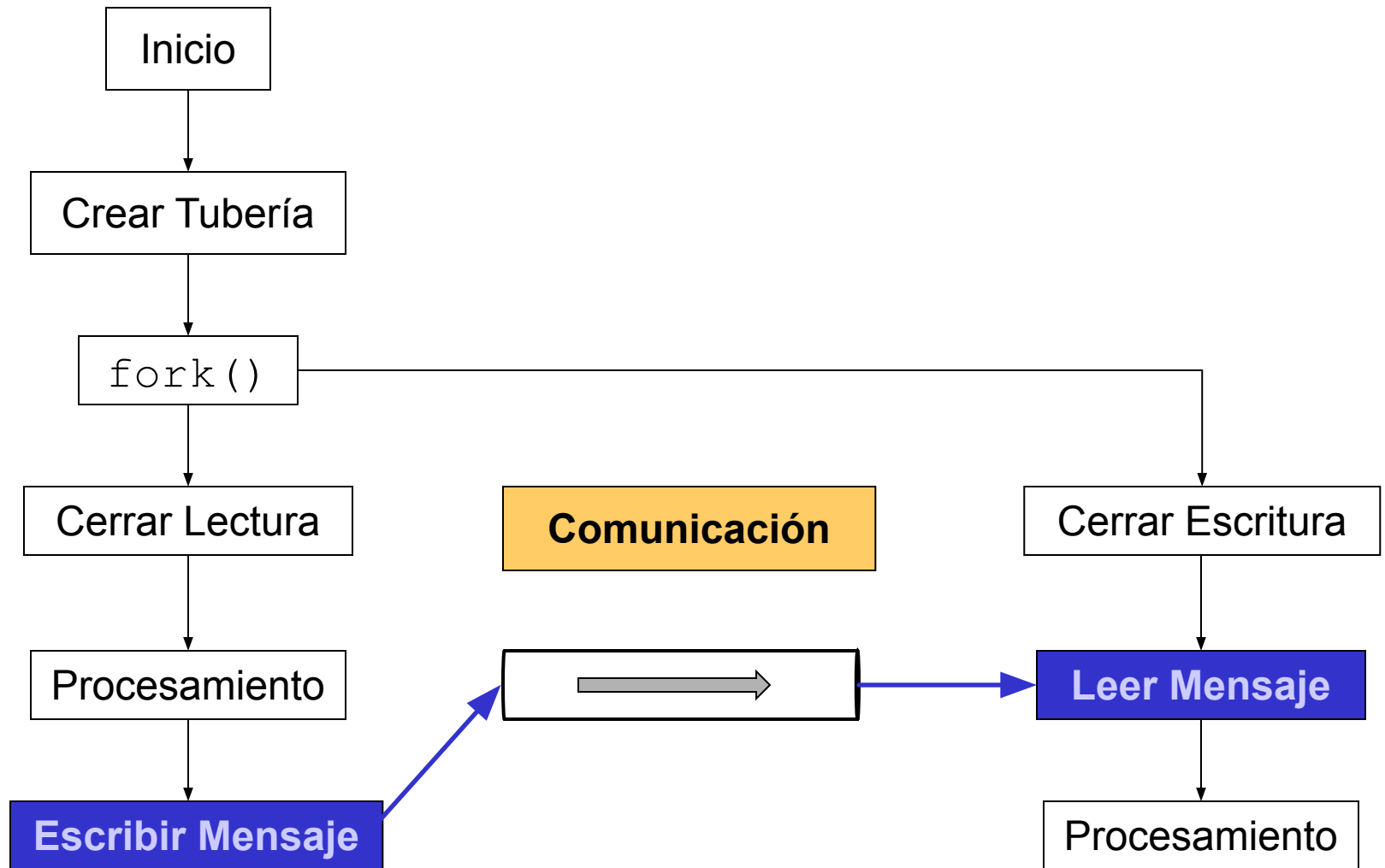
Tuberías sin Nombre

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

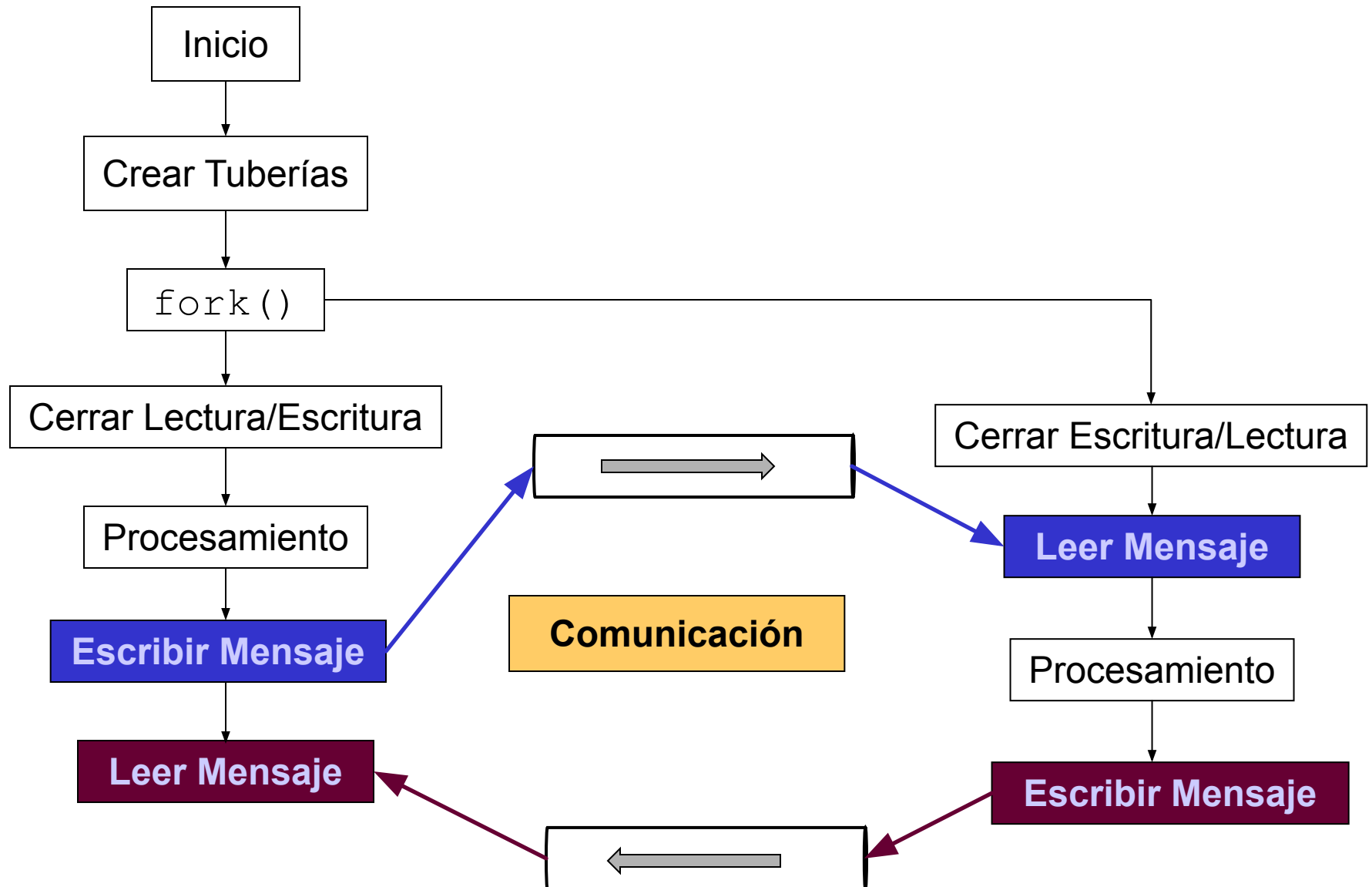
```
$ ls -l | wc -l
```



Tuberías sin Nombre: Unidireccional



Tuberías sin Nombre: Bidireccional



Tuberías sin Nombre

<unistd.h>

SV+BSD+POSIX

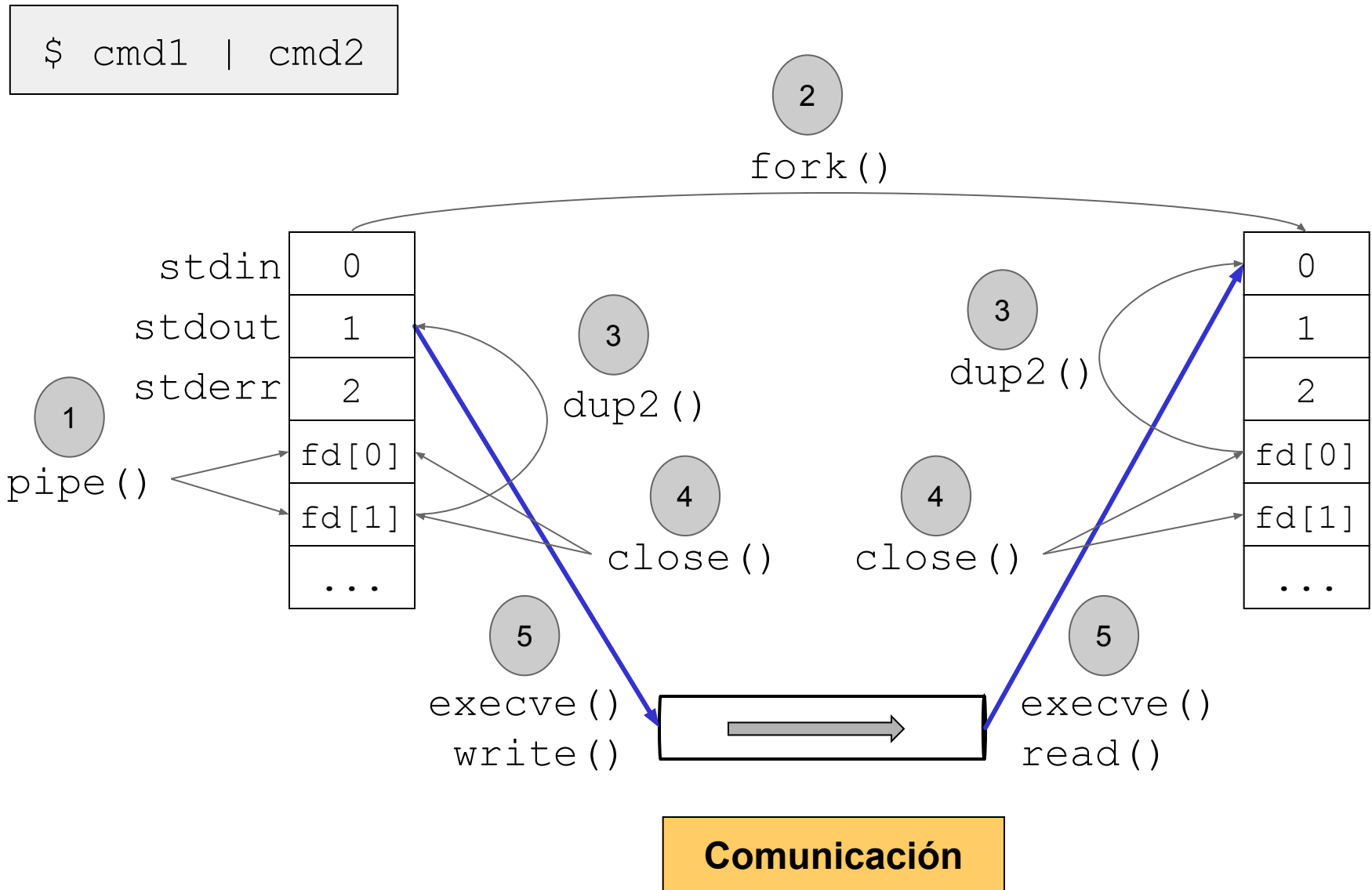
- Crear una tubería:

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

Extremo de escritura: `fd[1]`  Extremo de lectura: `fd[0]`

- Si la tubería está vacía, `read(2)` se bloqueará hasta que haya datos disponibles
- 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 escritura se han cerrado**, `read(2)` devolverá cero, indicando el fin de fichero
- 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 `EPIPE`
- Los descriptores que no sean necesarios deben cerrarse para asegurarse de que se notifica el fin de fichero o `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 como parte del 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 (*flag* `O_NONBLOCK`), la apertura para lectura no se bloqueará aunque la tubería no esté abierta para escritura

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 (fichero, dispositivo, tubería) que se creará
- `mode` especifica los permisos (modificados por *umask*) y el tipo de fichero que se creará como OR lógica. El tipo ha 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

- El comando `mknod` 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);
```

- `filename` es el nombre de la tubería que se creará
- `mode` son los permisos (modificados por *umask*) con que se creará la tubería

- El comando `mkfifo` 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:** Funciones `select()`, `poll()` y `epoll()`
 - El proceso monitoriza varios descriptors 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 descriptors de fichero preparados:

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

- `readfds` es el conjunto de descriptors de lectura
- `writefds` es el conjunto de descriptors de escritura
- `exceptfds` es el conjunto de descriptors de condiciones excepcionales
 - Por ejemplo, que haya datos urgentes (*out-of-band*) en un socket TCP
- `nfd` es el mayor de los descriptors 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 descriptors listos o 0 si expira el tiempo máximo
 - Los conjuntos se modifican para indicar qué descriptors 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:

```
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 `select()` retorne

Multiplexación de E/S Síncrona

```
...
fd_set rfds;

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

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

cambios = select(1, &rfds, 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");
}

...
```

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.