

Tema 1: Interfaz de un Sistema Operativo

Ampliación de Sistemas Operativos

Dpto. Ingeniería y Tecnología de Computadores (DITEC)

Universidad de Murcia

Curso 2020/2021

- 1 Introducción [KER:C2] [STE:C1]
- 2 Estándares [KER:C1] [STE:C2]
- 3 Llamadas al sistema y funciones de biblioteca [KER:C3]
- 4 Servicios POSIX y `glibc`
 - Creación y terminación de procesos [KER:C6,24-28]
 - Acceso a sistemas de ficheros [KER:C4-5,13-15,18]
 - Gestión de memoria virtual [KER:C49-50]
 - Gestión de memoria dinámica [KER:C7]
 - Comunicación/sincronización de procesos (IPC) [KER:C20-23,43-44]

1. Introducción [KER:C2] [STE:C1]

- Software que gestiona los recursos (procesos, memoria y E/S)
 - **Kernel**
 - Creación y terminación de procesos
 - Credenciales de procesos: Identificadores y permisos
 - Planificación de procesos: *Preemptive multitasking*
 - Gestión de memoria: *Dynamic memory / Virtual memory*
 - Administración de usuarios y grupos: *Virtual private computer*
 - Acceso a sistema de ficheros: *Virtual File System (VFS)*
 - Comunicación y sincronización de procesos (IPC)
 - Acceso a dispositivos de E/S
 - *Networking*
 - **System Call Application Programming Interface (API)**
 - Conjunto de herramientas software que acompañan al kernel
 - *Shell*, interfaz gráfica, editores, utilidades, etc.

Intérprete de comandos o *shell*

- Procesa las líneas de órdenes tecleadas por el usuario y ejecuta cada comando usando la *System Call API* del Sistema Operativo
 - Comandos internos (**help** comando) y externos (**man** comando)
- Intérpretes de comandos más comunes:
 - *Bourne shell* (**sh**)
 - *C shell* (**csh**)
 - *Korn shell* (**ksh**)
 - ***Bourne again shell* (bash)**
- El *shell* está estandarizado (POSIX.2 ó 1003.2-1992)
 - Tanto **ksh** como **bash** se ajustan a POSIX.2 pero también incluyen algunas extensiones no portables
- El *shell* por defecto depende del sistema operativo
- Para averiguar el *shell* que usamos:

```
1 $ echo $SHELL
/bin/bash
```

2. Estándares [KER:C1] [STE:C2]

- En 1989, ANSI aprobó el primer estándar de C (C89)
- En 1990, ISO/IEC lo adoptó también (C90)
- El estándar de C ha sufrido tres revisiones (C95, C99 y C11)¹
- El estándar de C incluye no sólo la sintaxis y la semántica de C sino también su biblioteca estándar (*The Standard C Library*)
- Referencias:
 - http://en.wikipedia.org/wiki/ANSI_C
 - <http://en.cppreference.com/w/c>
 - Manual gcc 9.3 (Ubuntu 20.04)²
 - <http://man7.org/linux/man-pages/man7/standards.7.html>

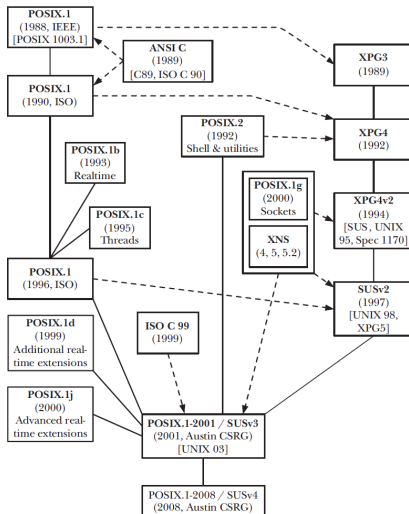
¹ C18, previamente C17, solamente corrige errores de C11.

² En gcc 9.3, `-std=c89` y `-std=c90` equivalen a `-ansi` para C. Por defecto, gcc 9.3 usa gnu18 para C (C18 con extensiones no portables). Para más información, véase `info gcc`.

- De 1969 a 1974 se desarrollaron las cinco primeras versiones de UNIX en AT&T y en 1975 AT&T liberó el código fuente de la sexta
- En los 80, la Universidad de Berkeley liberó varias versiones de BSD (*Berkeley Software Distribution*) basadas en UNIX de AT&T
 - FreeBSD, NetBSD y OpenBSD
- En paralelo, AT&T comercializó UNIX como System V (SVr1 a SVr4)
- En 1989, arrancó el proyecto GNU de la mano de Richard Stallman con el objetivo de crear una implementación libre de UNIX
 - Aplicaciones y utilidades con licencia GPL como **gcc**, **gdb** o **make**
 - Ante la ausencia de un kernel libre maduro, GNU adopta Linux
- Hoy todavía se comercializan versiones de UNIX como Solaris (antes SunOS) basado en SVr4 y Mac OS X basado en Mach/FreeBSD
- En 1994, se libera la versión 1.0 de Linux con licencia GPL
- A día de hoy, el término genérico Linux se refiere a la combinación del kernel con bibliotecas, herramientas e instalación (distribuciones)

Relación entre los estándares C y UNIX

POSIX es una familia de estándares para sistemas operativos de IEEE que regula las interfaces pero no las implementaciones



- ¿Cómo se debería inicializar `a` en aras de la portabilidad?³

```
1  int a[N];  
2  ...  
3  memset(a, 0, N * sizeof(int)); /* void *memset(void *s, int c, size_t n); */  
4  
5  /* o bien: */  
6  bzero(a, N * sizeof(int));      /* void bzero(void *s, size_t n); */
```

³ Compara `man memset` con `man bzero`.

3. Llamadas al sistema y funciones de biblioteca [KER:C3]

¿Qué es una llamada al sistema?

- Las llamadas al sistema o *syscalls* permiten a un proceso de usuario solicitar servicios de diferente naturaleza al Sistema Operativo
- La mayoría están encapsuladas en funciones de biblioteca que suelen tener el mismo nombre que la *syscall* correspondiente
- Cada llamada al sistema:
 - Implica un cambio de modo usuario a modo kernel
 - Se identifica mediante un número que es único
 - Puede requerir un conjunto de argumentos
- Referencias:
 - <http://man7.org/linux/man-pages/man2/intro.2.html>
 - <http://man7.org/linux/man-pages/man2/syscalls.2.html>
 - Sección 2 de `man`, por ejemplo, `man 2 open`⁴

⁴ Compara `man open`, `man 2 open` y `man -a open`.

¿Por qué son importantes las llamadas al sistema?

- Porque son un recurso imprescindible:

```
1  $ strace -c ls
   % time    seconds    usecs/call    calls    errors  syscall
3  -----
   0.00    0.000000         0         7         read
5   0.00    0.000000         0         1         write
   0.00    0.000000         0        11         close
7   0.00    0.000000         0        10         fstat
   0.00    0.000000         0        17         mmap
9   0.00    0.000000         0        12         mprotect
   0.00    0.000000         0         1         munmap
11  0.00    0.000000         0         3         brk
   0.00    0.000000         0         2         rt_sigaction
13  0.00    0.000000         0         1         rt_sigprocmask
   0.00    0.000000         0         2         ioctl
15  0.00    0.000000         0         8         8 access
   0.00    0.000000         0         1         execve
17  0.00    0.000000         0         2         getdents
   0.00    0.000000         0         2         2 statfs
19  0.00    0.000000         0         1         arch_prctl
   0.00    0.000000         0         1         set_tid_address
21  0.00    0.000000         0         9         openat
   0.00    0.000000         0         1         set_robust_list
23  0.00    0.000000         0         1         prlimit64
   -----
25 100.00    0.000000                93        10 total
```

¿Por qué son importantes las llamadas al sistema?

- Porque permiten entender qué hace un programa y cómo lo hace:

```
1 $ strace -e trace=%file wget 2>&1 1>/dev/null | grep open | grep etc
   openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
3   openat(AT_FDCWD, "/etc/wgetrc", O_RDONLY) = 3
```

- Porque determinan la portabilidad de un programa:
 - Todos los ejemplos que incluyen la *feature test macro*⁵ `_POSIX_C_SOURCE 200809L` son portables
- Porque son parte esencial de la seguridad del sistema⁶

⁵ http://man7.org/linux/man-pages/man7/feature_test_macros.7.html

⁶ En general, cada llamada al sistema comprueba que el usuario tiene autorización para realizar las acciones requeridas.

¿Cuál es el coste de una llamada al sistema?

● Código: syscall_vs_funcall.c

```
1  int foo() { return (10); }

3  int main(void)
4  {
5      long i;
6      pid_t ppid;
7      int result;
8      struct timespec start, end;
9      float avg_time_syscall, avg_time_funcall;

11     /* Estimate average latency for system call */
12     clock_gettime(CLOCK_MONOTONIC, &start);
13     for(i = 0; i < N; i++) ppid = getppid();    // *** SYSCALL ***
14     clock_gettime(CLOCK_MONOTONIC, &end);
15     avg_time_syscall = (nanosec(end) - nanosec(start)) / (N * 1.0);

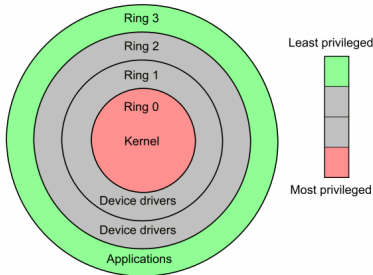
17     /* Estimate average latency for function call */
18     clock_gettime(CLOCK_MONOTONIC, &start);
19     for(i = 0 ; i < N; i++) result = foo();      // *** FUNCTION CALL ***
20     clock_gettime(CLOCK_MONOTONIC, &end);
21     avg_time_funcall = (nanosec(end) - nanosec(start)) / (N * 1.0);

23     printf("Average latency for system call getppid : %f\n", avg_time_syscall);
24     printf("Average latency for function call : %f\n", avg_time_funcall);

25
26     return EXIT_SUCCESS;
27 }
```

Modo usuario *versus* modo kernel [STA.C3.4]

- En modo kernel, el código tiene acceso ilimitado tanto al hardware (instrucciones y registros *privilegiados*) como a la memoria
- En modo usuario, el código sólo tiene acceso restringido a su espacio de direcciones virtuales
- En x86, el registro EFLAGS proporciona cuatro anillos de protección aunque normalmente sólo se usan el 0 (kernel) y el 3 (usuario)



- El *Process Control Block* (PCB) de un proceso contiene:
 - Información sobre el estado del procesador
 - Información de identificación
 - Información de control
- Un cambio de contexto implica:
 - Guardar en el PCB el estado del procesador
 - Actualizar el PCB del proceso interrumpido
 - Mover el PCB del proceso interrumpido a la cola adecuada
 - Seleccionar el nuevo proceso a ejecutar (*scheduling*)
 - Actualizar el PCB del nuevo proceso
 - Actualizar las estructuras de datos de gestión de memoria
 - Restaurar el estado del procesador desde el PCB del nuevo proceso
- Por el contrario, un cambio de modo sólo requiere:
 - Guardar el estado del procesador⁷

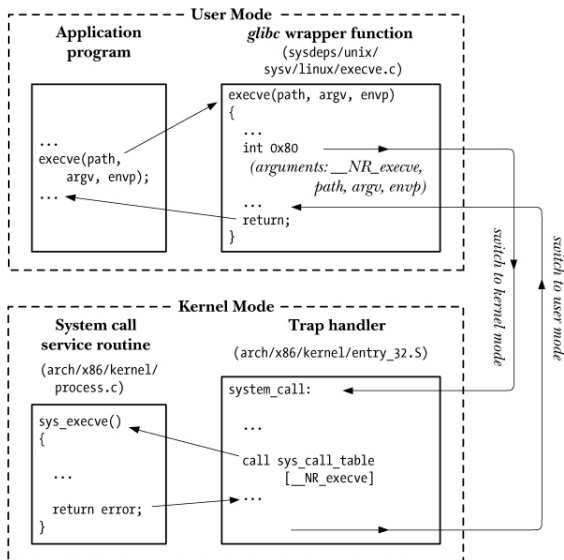
⁷ En Linux, esto no es cierto si se usa *Kernel Page-Table Isolation* (KPTI). En tal caso, cada llamada al sistema requiere un cambio de contexto. Para más detalles, véase https://en.wikipedia.org/wiki/Kernel_page-table_isolation.

Anatomía de una llamada al sistema o *system call*

- El proceso llama a la función que encapsula la llamada:
 - Copia los argumentos de la pila en registros del procesador
 - Copia el número de llamada al sistema en un registro (**eax**)
 - Ejecuta **int 0x80** que cambia de modo usuario a modo kernel⁸
 - Establecer el valor de **errno** (Véase Manejo de errores)
- En respuesta, el kernel ejecuta la función **system_call()**:
 - Guarda el estado del procesador en la pila del kernel
 - Comprueba la validez del número de llamada
 - Verifica la validez de los argumentos
 - Invoca la rutina de servicio correspondiente (tabla[#syscall]) que realiza el servicio solicitado y devuelve un valor de retorno
 - Restaura el estado del procesador desde la pila del kernel
 - Copia el valor de retorno en la pila del usuario
 - Regresa a la función de biblioteca volviendo a modo usuario

⁸ Los procesadores de AMD e Intel proporcionan mecanismos alternativos al método clásico más rápidos como **sysexit** (32 bits) y **syscall/sysret** (64 bits).

Anatomía de una llamada al sistema o *system call*



Manejo de errores: `errno`, `perror()` y `strerror()`

- Cada llamada al sistema devuelve un valor de retorno (`int`):
 - Éxito: valor de retorno ≥ 0
 - Fallo: valor de retorno < 0
 - Si el valor de retorno es igual a `-EVAL`, la función de biblioteca establece el valor de la variable `errno` a `EVAL` y devuelve `-1`
- El fichero de cabecera `errno.h` contiene todos los posibles valores de retorno y la definición de la variable `errno`
 - <http://man7.org/linux/man-pages/man3/errno.3.html>
- La sección **ERRORS** de la página del manual de cada llamada al sistema contiene los posibles valores de la variable `errno`
- `perror` imprime `msg` seguido del mensaje correspondiente al valor actual de `errno` en `stderr`

```
#include <stdio.h> /* POSIX */
2 void perror(const char *msg);
```

- `strerror` devuelve el mensaje correspondiente al valor de `errno`

```
#include <string.h> /* POSIX */
2 char *strerror(int errnum);
```

Ejemplo de llamada al sistema: open()

● Página del manual

```
$ man 2 open

2
NAME
4  open - open and possibly create a file or device

6  SYNOPSIS
   #include <sys/types.h>
   #include <sys/stat.h>
   #include <fcntl.h>

10
   int open(const char *pathname, int flags);
   int open(const char *pathname, int flags, mode_t mode);

12
DESCRIPTION
14  Given a pathname for a file, open() returns a file descriptor...

16
RETURN VALUE
18  open() returns the new file descriptor, or -1 if an error occurred (in which
   case, errno is set appropriately).

20
ERRORS
22  EACCES...
   ENOSPC...

24
CONFORMING TO
26  SVr4, 4.3BSD, POSIX.1-2001, POSIX.1-2008.
```

Ejemplo de manejo de errores: open()

● Tratamiento selectivo de errores:

```
ret = open(file_name, flags, mode);
2  if( ret == -1 ) {
    if( errno == EACCESS ) {
4      perror("open"); /* fprintf(stderr, "open: %s\n", strerror(errno)); */
        /* Tratamiento de EACCESS */
6    }
    if( errno == ENOSPC ) {
8      perror("open"); /* fprintf(stderr, "open: %s\n", strerror(errno)); */
        /* Tratamiento de ENOSPC */
10   }
    else {
12      /* Tratamiento de otros errores */
    }
14 }
```

● Aborto del programa:

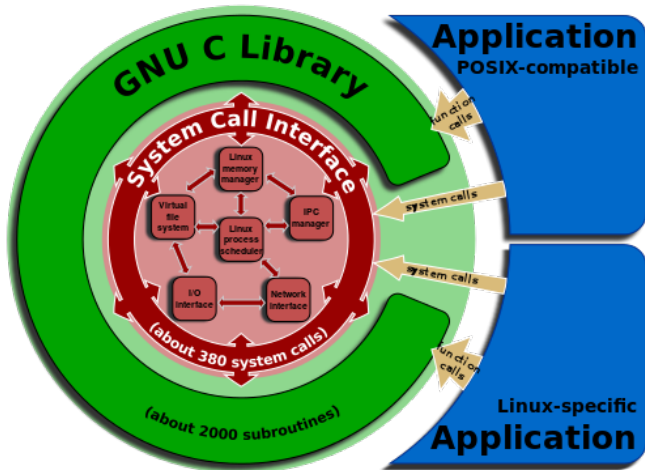
```
ret = open(file_name, flags, mode);
2  if( ret == -1 ) {
    perror("open"); /* fprintf(stderr, "open: %s\n", strerror(errno)); */
4      exit(EXIT_FAILURE);
    }
```

- La especificación de C incluye una biblioteca estándar con macros, variables, tipos y funciones declarados en ficheros de cabecera
 - Por ejemplo, en `stdio.h` se declaran `EOF`, `stdin`, `FILE` y `fopen`
- Implementaciones:
 - *GNU C Library (glibc)*, *BSD libc*, *Microsoft C Run-time Library*,...
- Otros lenguajes incluyen funcionalidad equivalente
 - En C++, el `namespace std` incluye la funcionalidad de la biblioteca estándar de C con ficheros de cabecera similares como `cstdio`
- Referencias:
 - https://en.wikipedia.org/wiki/C_standard_library
 - Sección 3 de `man`, por ejemplo, `man 3 memcp`

- La biblioteca estándar de POSIX extiende la de C:
 - Procesos: `fork()` y `pipe()` se definen en `unistd.h`
 - Threads: API de Pthreads se define en `pthread.h`
 - E/S: `read()`, `write()` y `close()` se definen en `unistd.h`
 - Sockets: API de (Berkeley) sockets se define en `sys/socket.h`
 - ...
- Referencias:
 - https://en.wikipedia.org/wiki/C_POSIX_library
 - https://en.wikipedia.org/wiki/POSIX_Threads

GNU C Library (glibc)

- Implementación de las bibliotecas estándar de C y de POSIX
- Implementación de funciones y llamadas al sistema no portables



- El *shell* y todos los comandos externos usan **glibc**:

```
1 $ ldd /bin/bash | grep libc
   libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (address)
3 $ ldd /bin/ls | grep libc
   libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (address)
```

- Para determinar la versión de **glibc** del sistema:

```
$ /lib/x86_64-linux-gnu/libc.so.6
2 GNU C Library (Ubuntu GLIBC 2.27-3ubuntu1) stable release version 2.27
  ...
4
#include <gnu/libc-version.h>
6 const char *gnu_get_libc_version(void);
   const char *gnu_get_libc_release(void);
```

- Referencia: <https://www.gnu.org/software/libc/>

- Algunas funciones de biblioteca se comportan como llamadas al sistema, por ejemplo `remove()`⁹
 - `perror()` y `strerror()` se pueden usar con normalidad
- Algunas funciones de biblioteca devuelven un valor distinto de -1 en caso de error, por ejemplo `fopen()`, pero usan `errno`¹⁰
 - `perror()` y `strerror()` se pueden usar con normalidad
- Otras funciones de biblioteca no usan `errno`, por ejemplo `memcpy()`¹¹
 - `perror()` y `strerror()` **NO** se deben usar

⁹ Véase la sección RETURN VALUE de man 3 remove.

¹⁰ Véase la sección RETURN VALUE de man 3 fopen.

¹¹ Véase la sección RETURN VALUE de man 3 memcpy.

- Una llamada al sistema es una solicitud al Sistema Operativo por parte de un usuario/programador para que realice un servicio
- La mayoría de las llamadas al sistema están encapsuladas en funciones de biblioteca que suelen tener el mismo nombre
- Cada llamada al sistema implica pasar a modo kernel y, por tanto, es mucho más lenta que una simple llamada a una función
- POSIX estandariza las llamadas al sistema UNIX para hacer posible escribir código que sea portable aunque incluya llamadas al sistema
- **glibc** implementa las bibliotecas estándar de POSIX y de C
 - `man 2 llamada_al_sistema`
 - `man 3 llamada_a_funcion`
 - `man -a llamada`
- Las páginas del manual están disponibles en <http://man7.org>

4. Servicios POSIX y glibc

Creación y terminación de procesos [KER:C6,24-28]

1 Introducción [KER:C2] [STE:C1]

2 Estándares [KER:C1] [STE:C2]

3 Llamadas al sistema y funciones de biblioteca [KER:C3]

4 Servicios POSIX y **glibc**

- Creación y terminación de procesos [KER:C6,24-28]
- Acceso a sistemas de ficheros [KER:C4-5,13-15,18]
- Gestión de memoria virtual [KER:C49-50]
- Gestión de memoria dinámica [KER:C7]
- Comunicación/sincronización de procesos (IPC) [KER:C20-23,43-44]

- Un proceso es una instancia de un programa en ejecución
- Un programa ejecutable es un fichero con información para construir un proceso
 - Formato como, por ejemplo, *Executable and Linking Format* (ELF)¹²
 - Código, dirección de comienzo y datos inicializados¹³
 - Tabla de símbolos (funciones y variables)¹⁴
 - Bibliotecas dinámicas requeridas¹⁵

¹² `file program`

¹³ `size program`

¹⁴ `readelf -s program`

¹⁵ `ldd program`

Servicios POSIX para obtención de IDs

- Cada proceso tiene un identificador único (PID)¹⁶

```
1  #include <unistd.h> /* POSIX */  
    pid_t getpid(void);
```

- `pid_t` es un entero \leq `/proc/sys/kernel/pid_max`

- Cada proceso tiene un proceso padre que lo creó (PPID)¹⁷

```
    #include <unistd.h> /* POSIX */  
2  pid_t getppid(void);
```

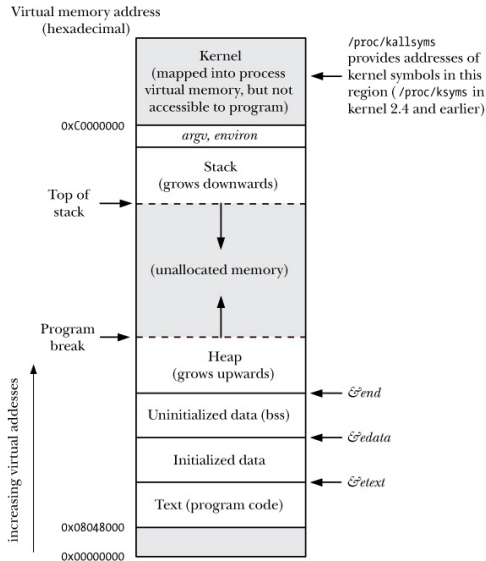
- Árbol de procesos cuya raíz es el proceso *init* con PID 1¹⁸

¹⁶ `cat /proc/$$/status | grep ^Pid:`

¹⁷ `cat /proc/$$/status | grep ^PPid:`

¹⁸ `pstree -s -p $$`

Mapa de memoria de un proceso



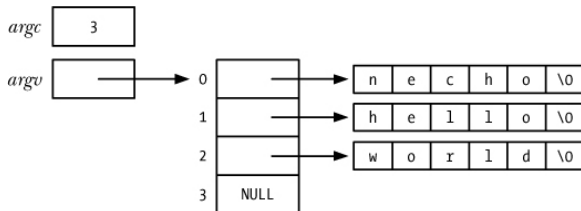
Mapa de memoria de un proceso

¿Dónde reside cada variable del programa? ¿Cuándo se crea y destruye cada una?

● Código: mem_map.c

```
1  #define _POSIX_C_SOURCE 200809L
2  #include <stdio.h>
3  #include <stdlib.h>
4
5  int numbers[] = {1, 2, 3, 4, 5};
6  char gbuf[1024];
7
8  int func(int param1)
9  {
10     static int x = 1;
11
12     return param1 + x++;
13 }
14
15 int main(void)
16 {
17     int var1 = 1;
18     int var2, var3;
19     char buf[1024];
20     char* p = malloc(1024);
21
22     var2 = func(var1);
23     var3 = func(var2);
24
25     printf("%d %d %d\n", var1, var2, var3);
26
27     return EXIT_SUCCESS;
28 }
```

Parámetros de línea de comandos de un proceso



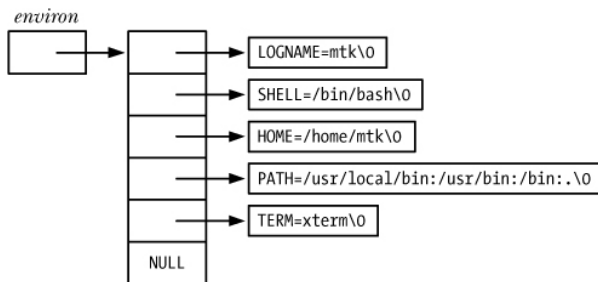
● Código: `argv.c`

```
1  int main(int argc, char* argv[])
2  {
3      int i;
4      for (i = 0; i < argc; i++)
5          printf("argv[%d] = %s\n", i, argv[i]);
6      return EXIT_SUCCESS;
7  }
```

- Los parámetros se almacenan en la memoria del proceso (`argv`)¹⁹
- Se pueden procesar con la función de biblioteca `getopt` (POSIX)

¹⁹ `cat /proc/$$/cmdline`

Variables de entorno de un proceso



• Como variable: env1.c

```
1  extern char **environ;
2  int main(int argc, char* argv[])
3  {
4      char **ep;
5      for (ep = environ; *ep != NULL; ep++)
6          printf("%s\n", *ep);
7      return EXIT_SUCCESS;
8  }
```

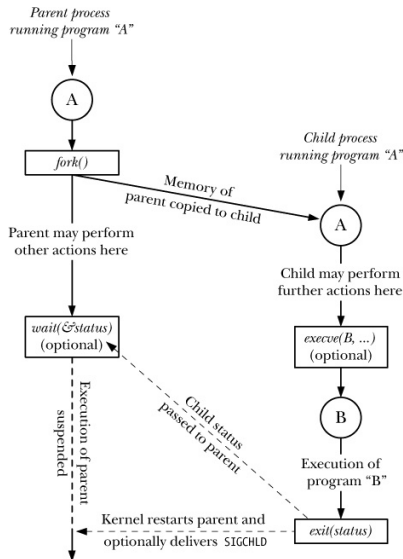
• Como parámetro: env2.c

```
1  int main(int argc, char* argv[],
2          char* environ[])
3  {
4      char **ep;
5      for (ep = environ; *ep != NULL; ep++)
6          printf("%s\n", *ep);
7      return EXIT_SUCCESS;
8  }
```

Variables de entorno de un proceso

- Cuando se crea un proceso, el proceso hijo hereda una copia del entorno del proceso padre
- El entorno se almacena en la memoria del proceso (*environ*)
- Modificación del entorno desde el *shell*:
 - `export`, `env`, `unset`, `printenv`
- Modificación del entorno de un programa desde el *shell*:
 - `env [OPTION]... [-] [NAME=VALUE]... ./program`
 - `NAME=VALUE ./program`
- Modificación del entorno desde un proceso:
 - `getenv`, `putenv`, `setenv`, `unsetenv`
- Referencias:
 - `man 7 environ`

Servicios POSIX para la creación de procesos



- La llamada al sistema **fork()** crea un nuevo proceso *hijo* que es una copia casi exacta del proceso *padre* (el que llama a **fork()**)

```
#include <unistd.h> /* POSIX */  
2 pid_t fork(void);
```

- pid_t** es el PID del hijo en el padre y 0 en el hijo
- El proceso padre y el proceso hijo comparten la misma copia del código del programa en modo sólo lectura
- El proceso hijo obtiene una copia exacta de los datos, el *heap* y la pila del proceso padre
- El proceso hijo recibe un **duplicado** de todos los descriptores de fichero del proceso padre
- Tanto el proceso padre como el proceso hijo podrían ocupar la CPU
 - Desde el kernel 2.6.32, el proceso padre se ejecuta primero²⁰

²⁰ `cat /proc/sys/kernel/sched_child_runs_first`

fork(): Proceso padre y proceso hijo

● Código: fork.c

```
1  int main(void)
2  {
3      pid_t pid; /* Used in parent to record PID of child */
4
5      switch (pid = fork()) {
6          case -1: /* fork() failed */
7              /* Handle error */
8              break;
9          case 0: /* Child comes here after successful fork() */
10             /* Perform actions specific to child */
11             printf("Child with PID %d created by parent with PID %d\n",
12                  getpid(), getppid());
13             break;
14          default: /* Parent comes here after successful fork() */
15             /* Perform actions specific to parent */
16             printf("Parent with PID %d forked child with PID %d\n",
17                  getpid(), pid);
18             break;
19      }
20
21      return EXIT_SUCCESS;
22 }
```


- ¿Cuántos procesos resultarían de la ejecución del siguiente fragmento de código (asumiendo que `fork()` nunca falla)?

```
fork();  
2  if (fork()) fork();  
    fork();
```

fork(): Memoria virtual

● Código: fork_memory.c

```
1  int main(void)
2  {
3      char contents[] = { 'A', 'S', 'O', '\0' };
4      pid_t pid; /* Used in parent to record PID of child */
5
6      switch (pid = fork()) {
7          case -1: /* fork() failed */
8              /* Handle error */
9              break;
10         case 0: /* Child comes here after successful fork() */
11             /* Perform actions specific to child */
12             contents[0] = 'I';
13             printf("Contents in child: %s\n", contents);
14             break;
15         default: /* Parent comes here after successful fork() */
16             /* Perform actions specific to parent */
17             wait(NULL); /* Wait for child to finish */
18             printf("Contents in parent: %s\n", contents);
19             break;
20     }
21
22     return EXIT_SUCCESS;
23 }
```

Servicios POSIX para esperar a un proceso hijo

- La llamada al sistema `wait()` espera a que uno de los procesos *hijo* del proceso que la invoca termine

```
1 #include <sys/wait.h> /* POSIX */  
   pid_t wait(int *status);
```

- `wait()` se bloquea hasta que un proceso hijo haya terminado
 - Si alguno ya lo ha hecho, no se bloquea
- `pid_t` es el PID del proceso hijo que ha finalizado
 - Si es -1, no hay más procesos hijo y se establece `errno` a `ECHILD`
- `status` devuelve el código de retorno de dicho proceso hijo
- La llamada al sistema `wait()` tiene algunas limitaciones:
 - Si un proceso padre crea varios procesos hijo, no es posible esperar a que uno de ellos en concreto termine
 - Si ningún proceso hijo ha finalizado, tras una llamada a `wait()` el proceso padre se bloquea
 - Con `wait()` sólo es posible averiguar cuándo un proceso hijo ha terminado, no se puede saber cuándo para/reanuda su ejecución

Servicios POSIX para esperar a un proceso hijo

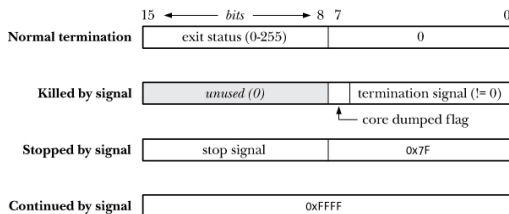
- `waitpid()` subsana las limitaciones de `wait()`

```
#include <sys/wait.h> /* POSIX */  
2 pid_t waitpid(pid_t pid, int *status, int options);
```

- Si `pid > 0`, espera a que termine el proceso hijo con PID `pid`
- Si `pid = -1`, espera a que termine cualquier hijo, es decir, `waitpid(-1, &status, 0)` equivale a `wait(&status)`
- El parámetro `options` es una máscara de bits:
 - `WUNTRACED`: Informa acerca de hijos cuando se detiene su ejecución
 - `WCONTINUED`: Informa acerca de hijos cuando se reanuda su ejecución
 - `WNOHANG`: Llamada no bloqueante de manera que si no hay ningún proceso hijo que cumpla las condiciones especificadas, `waitpid()` devuelve 0 y se establece `errno` a `ECHILD`

Servicios POSIX para esperar a un proceso hijo

- El valor **status** devuelto por **wait()** y **waitpid()** permite distinguir los siguientes sucesos²¹:
 - El proceso hijo terminó llamando a **_exit()** o **exit()**²²
 - El proceso hijo terminó por la entrega de una señal no tratada
 - El proceso hijo detuvo su ejecución por entrega de señal (**WUNTRACED**)
 - El proceso hijo siguió su ejecución por entrega de señal (**WCONTINUED**)



- El parámetro **status** se debe procesar siempre con las macros **WIFEXITED** (**WEXITSTATUS**), **WIFSIGNALED** (**WTERMSIG**), **WIFSTOPPED** y **WIFCONTINUED** porque la codificación depende de la implementación

²¹ true ; echo \$? ; false ; echo \$? ; sleep 60 [CTRL+C] ; echo \$?

²² Comparaman `exit` y `_exit`.

Servicios POSIX para esperar a un proceso hijo

● Código: waitpid.c

```
1  int main(void)
2  {
3      pid_t pid; /* Used in parent to record PID of child */
4      int status; /* Used in parent to record status of child */
5
6      switch (pid = fork()) {
7          case -1: /* fork() failed */
8              /* Handle error */
9              break;
10         case 0: /* Child comes here after successful fork() */
11             /* Perform actions specific to child */
12             printf("Child with PID %d created by parent with PID %d\n",
13                    getpid(), getppid());
14             exit(EXIT_SUCCESS);
15             break;
16         default: /* Parent comes here after successful fork() */
17             /* Perform actions specific to parent */
18             printf("Parent with PID %d forked child with PID %d\n",
19                    getpid(), pid);
20             while(waitpid(pid, &status, WNOHANG) == 0)
21                 sleep(1);
22             if (WIFEXITED(status))
23                 printf("Child exited, status=%d\n", WEXITSTATUS(status));
24             break;
25     }
26
27     return EXIT_SUCCESS;
28 }
```

- El tiempo de vida de un proceso padre no tiene por qué coincidir con el de sus procesos hijo
 - ¿Qué sucede si un proceso padre termina sin llamar a `wait()`?
 - El proceso huérfano es adoptado por el proceso `init` cuyo PID es 1
 - Cuando el proceso huérfano acaba, `init` llama a `wait()`
 - ¿Qué pasa si un hijo termina antes de que su padre llame a `wait()`?
 - El proceso hijo se convierte en un proceso *zombie* y el kernel libera sus recursos pero mantiene una entrada en la tabla de procesos hasta que el proceso padre llame a `wait()`
- Cada vez que un proceso hijo termina, se envía una señal `SIGCHLD` al proceso padre que se podría usar para llamar a `wait()`

- ¿Qué crearía el siguiente fragmento de código?:

```
1  if (! fork()) sleep(60); exit(0);
```

- Un proceso *zombie*.
- Un proceso huérfano.

¿Qué sucedería si eliminásemos el '!'?

- ¿Qué sucedería si un proceso tratase de crear procesos hijo de manera ininterrumpida pero no llamase nunca a `wait()`?

Servicios POSIX para ejecución de programas

- La llamada al sistema `execve()` carga un nuevo programa en la memoria del proceso y sustituye los datos, el *heap* y la pila

```
1  #include <unistd.h> /* POSIX */  
   int execve(const char *pathname, char *const argv[], char *const envp[]);
```

- `pathname` es la ruta absoluta o relativa del nuevo programa
- `argv` contiene los argumentos para el nuevo programa
- `envp` contiene el entorno para el nuevo programa
- Nunca regresa en caso de éxito y siempre devuelve -1 (fallo)
- El PID del proceso que llama a `execve()` no cambia

- Código: `execve.c` 

```
   int main(int argc, char* argv[])  
2  {  
       const char epath[] = "/bin/ls";  
4     char *eargv[] = {" /bin/ls", "-l", NULL};  
  
6     int ret = execve(epath, eargv, NULL);  
       assert(ret == -1);  
7     printf("execve failed\n");  
8     }  
}
```

Servicios POSIX para ejecución de programas

- Tras una llamada a `execve()`, el nuevo programa hereda todos los descriptores de fichero abiertos
- Las funciones `execle()`, `execlp()`, `execvp()`, `execv()` y `execl()` proporcionan una interfaz distinta pero todas se basan en `execve()`

Función	Fichero	Argumentos	Entorno
<code>execve</code>	Ruta absoluta	Array	<code>envp</code>
<code>execle</code>	Ruta absoluta	Lista	<code>envp</code>
<code>execlp</code>	Fichero + <code>PATH</code>	Lista	Entorno heredado
<code>execvp</code>	Fichero + <code>PATH</code>	Array	Entorno heredado
<code>execv</code>	Ruta absoluta	Array	Entorno heredado
<code>execl</code>	Ruta absoluta	Lista	Entorno heredado

● Código: `execlp.c`

```
1  int main(int argc, char* argv[])
2  {
3      const char ebin[] = "ls";
4
5      int ret = execlp(ebin, "ls", "-l", NULL);
6      assert(ret == -1);
7      printf("execlp failed\n");
8  }
```

Intérpretes

- La mayor parte de los kernels UNIX permiten ejecutar *scripts* como si fueran binarios si el fichero es ejecutable y comienza por **#!**
 - Al encontrar la secuencia **#!** al comienzo del fichero, `execve()` ejecuta el intérprete con la siguiente línea de comandos:

Script file (located at *script-path*)

`#! interpreter-path optional-arg`

execve() call within program
execve(script-path, argv, envp)

*(excludes
argv(0))*

`interpreter-path optional-arg script-path arg...`

Argument list given to interpreter

● Código: `interpreter.c`

```
1  int main(int argc, char* argv[])
2  {
3      const char epath[] = "script.sh";
4      char *eargv[] = {"script.sh", NULL};
5
6      int ret = execve(epath, eargv, NULL);
7      assert(ret == -1);
8      printf("execve failed\n");
9
10     return EXIT_SUCCESS;
11 }
```

● Script: `script.sh`

```
1  #! /bin/sh
   echo ASO
```

Servicios POSIX para terminación de procesos

- Un proceso puede terminar con `return status;` en `main()` o con `exit(status);` en cualquier parte del programa

```
#include <stdlib.h> /* C99 y POSIX */
2 void exit(int status);
```

- `return status` es equivalente a `exit(status)`
- No hay reglas sobre los valores de `status` pero por convenio:
 - `EXIT_SUCCESS (0)` y `EXIT_FAILURE (1)`
- Una llamada a `exit()`:
 - Ejecuta los *exit handlers* registrados con `atexit()` o con `on_exit()` en orden inverso de registro
 - Ejecuta `fflush(NULL)`
 - Ejecuta la llamada al sistema `_exit()`

```
#include <unistd.h> /* C99 y POSIX */
2 void _exit(int status); /* 0 <= status <= 255 */
```

- La terminación de un proceso implica el cierre de sus descriptores de fichero y de directorio

- La llamada al sistema **fork()** crea un proceso *hijo* que es una copia casi exacta del proceso *padre*
- La llamada al sistema **wait()** espera a que uno de los procesos *hijo* del proceso que la invoca termine
- La llamada al sistema **waitpid()** subsana limitaciones de **wait()**
- El tiempo de vida de un proceso padre no tiene por qué coincidir con el de sus procesos hijo: Procesos huérfanos y *zombies*
- La llamada al sistema **execve()** carga un nuevo programa en la memoria del proceso y sustituye los datos, el *heap* y la pila

1 Introducción [KER:C2] [STE:C1]

2 Estándares [KER:C1] [STE:C2]

3 Llamadas al sistema y funciones de biblioteca [KER:C3]

4 Servicios POSIX y **glibc**

- Creación y terminación de procesos [KER:C6,24-28]
- Acceso a sistemas de ficheros [KER:C4-5,13-15,18]
- Gestión de memoria virtual [KER:C49-50]
- Gestión de memoria dinámica [KER:C7]
- Comunicación/sincronización de procesos (IPC) [KER:C20-23,43-44]

El modelo de E/S universal

- Un descriptor de fichero es un entero positivo utilizado por las llamadas al sistema para acceder a todo tipo de ficheros abiertos, incluyendo ficheros regulares, *pipes*, FIFOs, terminales y dispositivos
- Por convención, hay tres descriptors de fichero abiertos por el *shell* para cada programa antes de ejecutarlo: `STDIN_FILENO` (0), `STDOUT_FILENO` (1) y `STDERR_FILENO` (2)²³
- La biblioteca estándar de C usa *streams* en lugar de descriptors
- Los *streams* `stdin`, `stdout` y `stderr` son equivalentes a `STDIN_FILENO`, `STDOUT_FILENO` y `STDERR_FILENO`²⁴

File descriptor	Purpose	POSIX name	stdio stream
0	standard input	STDIN_FILENO	<i>stdin</i>
1	standard output	STDOUT_FILENO	<i>stdout</i>
2	standard error	STDERR_FILENO	<i>stderr</i>

²³ `ls /proc/$$/fdinfo`

²⁴ La llamada al sistema `write(STDOUT_FILENO, ...)` es similar a `fprintf(stdout, ...) (printf(...))`.

El modelo de E/S universal

- Las llamadas `open()`, `read()`, `write()` y `close()` se pueden usar para realizar operaciones de E/S sobre cualquier tipo de fichero

```
#include <sys/types.h> /* POSIX */
2  #include <sys/stat.h>
   #include <fcntl.h>
4  int open(const char *pathname, int flags, mode_t mode);

6  #include <unistd.h> /* POSIX */
   ssize_t read(int fd, void *buf, size_t count);
8  ssize_t write(int fd, const void *buf, size_t count);
   int close(int fd);
```

- `open()` siempre devuelve el descriptor de fichero no usado más bajo
- En `read()/write()`, `count` es el número de bytes a leer/escribir

syscall	Valor devuelto: rc			
	rc == count	rc < count	rc == 0	rc == -1
read()	Lectura total	Lectura parcial	EOF	¡Error!
write()	Escritura total	Escritura parcial	No escritura	¡Error!

- Cuando un proceso termina, todos sus descriptors se cierran

El modelo de E/S universal

- *Flags* de modo de acceso (sólo lectura), *flags* de creación (no se pueden consultar ni modificar) y *flags* de estado (lectura y escritura)

Flag	Purpose	SUS?
O_RDONLY	Open for reading only	v3
O_WRONLY	Open for writing only	v3
O_RDWR	Open for reading and writing	v3
O_CLOEXEC	Set the close-on-exec flag (since Linux 2.6.23)	v4
O_CREAT	Create file if it doesn't already exist	v3
O_DIRECT	File I/O bypasses buffer cache	
O_DIRECTORY	Fail if <i>pathname</i> is not a directory	v4
O_EXCL	With O_CREAT: create file exclusively	v3
O_LARGEFILE	Used on 32-bit systems to open large files	
O_NOATIME	Don't update file last access time on <i>read()</i> (since Linux 2.6.8)	
O_NOCTTY	Don't let <i>pathname</i> become the controlling terminal	v3
O_NOFOLLOW	Don't dereference symbolic links	v4
O_TRUNC	Truncate existing file to zero length	v3
O_APPEND	Writes are always appended to end of file	v3
O_ASYNC	Generate a signal when I/O is possible	
O_DSYNC	Provide synchronized I/O data integrity (since Linux 2.6.33)	v3
O_NONBLOCK	Open in nonblocking mode	v3
O_SYNC	Make file writes synchronous	v3

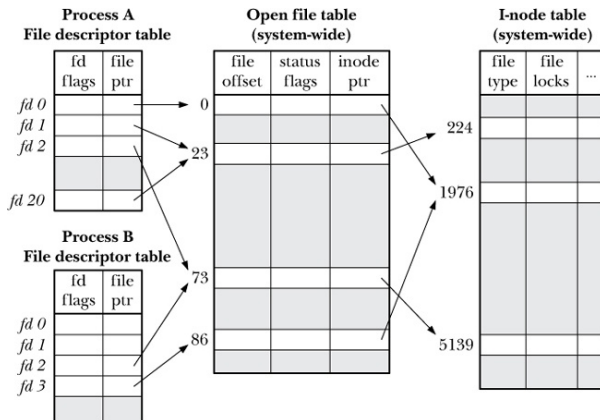
Ejemplo: cat fichero

● Código: cat.c

```
1      fd = open(argv[1], O_RDONLY);
2      if (fd == -1) {
3          perror("open");
4          exit(EXIT_FAILURE);
5      }
6
7      while((num_read = read(fd, buf, BUF_SIZE)) > 0) {
8          num_written = write(STDOUT_FILENO, buf, num_read);
9          if (num_written == -1) {
10             perror("write");
11             exit(EXIT_FAILURE);
12         }
13         /* Escrituras parciales no tratadas */
14         assert(num_written == num_read);
15     }
16
17     if (num_read == -1) {
18         perror("read");
19         exit(EXIT_FAILURE);
20     }
21
22     if (close(fd) == -1) {
23         perror("close");
24         exit(EXIT_FAILURE);
25     }
26
27     exit(EXIT_SUCCESS);
28 }
```

Servicios POSIX para E/S con ficheros

- Relaciones entre descriptores, ficheros abiertos y nodos-i:²⁵



- Si el *offset* del fichero no estuviese en la tabla de ficheros abiertos, ¿qué pasaría al ejecutar la línea de órdenes (`ls ; df`) > `f1`?


²⁵ Los descriptores de fichero 0, 1 y 2 apuntan, por defecto, a un fichero especial de caracteres (por ejemplo, `/dev/tty1`).

- Duplicación de descriptores de fichero:

```
#include <unistd.h> /* POSIX */  
2 int dup(int oldfd); int dup2(int oldfd, int newfd);
```

- **dup()** devuelve una copia de **oldfd** usando el descriptor disponible más bajo mientras que **dup2()** crea una copia de **oldfd** en **newfd** cerrando **newfd** si fuese necesario

Caso práctico: Descriptores de fichero

- Dibuja las tablas de descriptores de ficheros, de ficheros abiertos y de nodos-i tras la tercera llamada a `write()`: `fdt_ofst_int.c` 

```
1  int main(void)
2  {
3      pid_t pid; /* Used in parent to record PID of child */
4      int fd_s1, fd_s2, fd_us;
5      fd_s1 = open("shared", O_WRONLY | O_CREAT | O_TRUNC, S_IRWXU);
6      write(fd_s1, "ESO", 3);
7      switch (pid = fork()) {
8          case -1: /* fork() failed */
9              /* Handle error */
10             break;
11          case 0: /* Child comes here after successful fork() */
12              /* Perform actions specific to child */
13              fd_s2 = dup(fd_s1);
14              fd_us = open("shared", O_WRONLY | O_CREAT | O_TRUNC, S_IRWXU);
15              write(fd_s2, "ISO", 3);
16              write(fd_us, "ASO", 3);
17              close(fd_s2);
18              close(fd_us);
19              break;
20          default: /* Parent comes here after successful fork() */
21              /* Perform actions specific to parent */
22              wait(NULL);
23              close(fd_s1);
24              break;
25      }
26
27      return EXIT_SUCCESS;
28 }
```

Caso práctico: Descriptores de fichero

```
int main(void)
```

Tabla: Padre

fd	fptr
0	0x12350
1	0x12360
2	0x12370

Tabla: OFT

index	offset	i-node
-	-	-
0x12350	-	134
0x12360	-	134
0x12370	-	134

Tabla: INT

i-node	file	filetype
134	/dev/tty	caracteres

```
fd_s1 = open("shared", . . . )26
```

Tabla: Padre

fd	fptr
0	0x12350
1	0x12360
2	0x12370
3	0x12380

Tabla: OFT

index	offset	i-node
-	-	-
0x12350	-	134
0x12360	-	134
0x12370	-	134
0x12380	0	150

Tabla: INT

i-node	file	filetype
134	/dev/tty	caracteres
...
150	shared	regular

²⁶ fd_s1 vale 3.

Caso práctico: Descriptores de fichero

```
write(fd_s1, "ES0", 3)27
```

Tabla: Padre

fd	fptr
0	0x12350
1	0x12360
2	0x12370
3	0x12380

Tabla: OFT

index	offset	i-node
-	-	-
0x12350	-	134
0x12360	-	134
0x12370	-	134
0x12380	3	150

Tabla: INT

i-node	file	filetype
134	/dev/tty	caracteres
...
150	shared	regular

```
pid = fork()
```

Tabla: Padre

fd	fptr
0	0x12350
1	0x12360
2	0x12370
3	0x12380

Tabla: Hijo

fd	fptr
0	0x12350
1	0x12360
2	0x12370
3	0x12380

Tabla: OFT

index	offset	i-node
-	-	-
0x12350	-	134
0x12360	-	134
0x12370	-	134
0x12380	3	150

Tabla: INT

i-node	file	filetype
134	/dev/tty	caracteres
...
150	shared	regular

²⁷ fd_s1 vale 3.

Caso práctico: Descriptores de fichero

```
fd_sd2 = dup(fd_s1)28
```

Tabla: Padre

fd	fptr
0	0x12350
1	0x12360
2	0x12370
3	0x12380

Tabla: Hijo

fd	fptr
0	0x12350
1	0x12360
2	0x12370
3	0x12380
4	0x12380

Tabla: OFT

index	offset	i-node
-	-	-
0x12350	-	134
0x12360	-	134
0x12370	-	134
0x12380	3	150

Tabla: INT

i-node	file	filetype
134	/dev/tty	caracteres
...
150	shared	regular

```
fd_us = open("shared", . . . )29
```

Tabla: Padre

fd	fptr
0	0x12350
1	0x12360
2	0x12370
3	0x12380

Tabla: Hijo

fd	fptr
0	0x12350
1	0x12360
2	0x12370
3	0x12380
4	0x12380
5	0x12390

Tabla: OFT

index	offset	i-node
-	-	-
0x12350	-	134
0x12360	-	134
0x12370	-	134
0x12380	3	150
0x12390	0	150

Tabla: INT

i-node	file	filetype
134	/dev/tty	caracteres
...
150	shared	regular

²⁸ fd_s2 vale 4.

²⁹ fd_us vale 5.

Caso práctico: Descriptores de fichero

`write(fd_s2, "ISO", 3)`³⁰

Tabla: Padre

fd	fptr
0	0x12350
1	0x12360
2	0x12370
3	0x12380

Tabla: Hijo

fd	fptr
0	0x12350
1	0x12360
2	0x12370
3	0x12380
4	0x12380

Tabla: OFT

index	offset	i-node
-	-	-
0x12350	-	134
0x12360	-	134
0x12370	-	134
0x12380	6	150
0x12390	0	150

Tabla: INT

i-node	file	filetype
134	/dev/tty	caracteres
...
150	shared	regular

`write(fd_us, "ASO", 3)`³¹

Tabla: Padre

fd	fptr
0	0x12350
1	0x12360
2	0x12370
3	0x12380

Tabla: Hijo

fd	fptr
0	0x12350
1	0x12360
2	0x12370
3	0x12380
4	0x12380
5	0x12390

Tabla: OFT

index	offset	i-node
-	-	-
0x12350	-	134
0x12360	-	134
0x12370	-	134
0x12380	6	150
0x12390	3	150

Tabla: INT

i-node	file	filetype
134	/dev/tty	caracteres
...
150	shared	regular

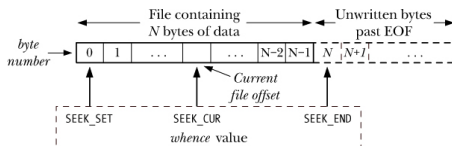
³⁰ `fd_s2` vale 4.

³¹ `fd_us` vale 5.

Servicios POSIX para E/S con ficheros

- Cada fichero abierto tiene asociado un *offset* en el que comenzará la próxima operación `read()` o `write()`
- El *offset* apunta al principio del fichero cuando se abre y aumenta después de cada operación `read()` o `write()` en función de `count`
- La llamada `lseek()` permite modificar el *offset* de un fichero abierto

```
1 #include <unistd.h> /* POSIX */  
   off_t lseek(int fd, off_t offset, int whence);
```



- `lseek(fd, 0, SEEK_CUR)` recupera el *offset* actual sin modificarlo
- Si se lee más allá de EOF, `read()` devuelve cero, pero si se escribe más allá de EOF, se crea un *agujero* que no ocupa espacio en disco

● Código: lseek.c

```
1  int main()
2  {
3      int fd;
4      char c = 'a';
5
6      if (-1 == (fd = open("test", O_RDWR | O_CREAT | O_TRUNC, S_IRUSR | S_IWUSR)))
7      {
8          perror("open()");
9          exit(EXIT_FAILURE);
10     }
11     /* Seek at 1,000,000 bytes from the beginning */
12     if (-1 == lseek (fd, 1000000, SEEK_SET))
13     {
14         perror("lseek()");
15         exit(EXIT_FAILURE);
16     }
17     /* Write one byte */
18     if (-1 == write (fd, &c, 1))
19     {
20         perror("write()");
21         exit(EXIT_FAILURE);
22     }
23     if (-1 == close(fd))
24     {
25         perror("close()");
26         exit(EXIT_FAILURE);
27     }
28
29     exit(EXIT_SUCCESS);
30 }
```

- Consulta de los atributos de un fichero o directorio³²:

```
1  #include <sys/stat.h> /* POSIX */
   int stat(const char *pathname, struct stat *statbuf);
3  int lstat(const char *pathname, struct stat *statbuf);
   int fstat(int fd, struct stat *statbuf);
5
   struct stat {
7     dev_t st_dev;           /* IDs of device on which file resides */
     ino_t st_ino;           /* I-node number of file */
9     mode_t st_mode;        /* File type and permissions */
     nlink_t st_nlink;      /* Number of (hard) links to file */
11    uid_t st_uid;          /* User ID of file owner */
     gid_t st_gid;          /* Group ID of file owner */
13    dev_t st_rdev;         /* IDs for device special files */
     off_t st_size;         /* Total file size (bytes) */
15    blksize_t st_blksize;   /* Optimal block size for I/O (bytes) */
     blkcnt_t st_blocks;    /* Number of (512B) blocks allocated */
17    time_t st_atime;        /* Time of last file access */
     time_t st_mtime;       /* Time of last file modification */
19    time_t st_ctime;        /* Time of last status change */
   };
```

- **stat()**, **lstat()** y **fstat()** proporcionan información sobre un fichero obtenida en su mayor parte del nodo-i correspondiente

³² stat test

- Creación y eliminación de directorios:

```
#include <sys/stat.h> /* POSIX */
2 int mkdir(const char *pathname, mode_t mode);
#include <unistd.h>
4 int rmdir(const char *pathname);
```

- Dada una ruta absoluta o relativa, `mkdir()` y `rmdir()` crean y eliminan un directorio, respectivamente

- Proceso de directorios:

```
#include <dirent.h> /* POSIX */
2 DIR *opendir(const char *dirpath);
struct dirent *readdir(DIR *dirp);
4 struct dirent {
    ino_t d_ino; /* File i-node number */
    char d_name[]; /* Null-terminated name of file */
};
8 void rewinddir(DIR *dirp);
int closedir(DIR *dirp);
```

- `opendir()` *abre* un directorio, `readdir()` devuelve una entrada del directorio, `rewinddir()` *rebobina* el directorio y `closedir()` lo *cierra*

Servicios POSIX para E/S con directorios

● Código: list_files.c

```
1  void list_files(const char *dirpath)
2  {
3      DIR *dirp;
4      struct dirent *dp;
5
6      dirp = opendir(dirpath);
7      if (dirp == NULL) {
8          fprintf(stderr, "opendir failed on '%s'", dirpath);
9          return;
10     }
11     for (;;) {
12         errno = 0; /* To distinguish error from end-of-directory */
13         dp = readdir(dirp);
14         if (dp == NULL)
15             break;
16
17         if (strcmp(dp->d_name, ".") == 0 || strcmp(dp->d_name, "..") == 0)
18             continue; /* Skip . and .. */
19
20         printf("%s\n", dp->d_name);
21     }
22     if (errno != 0) {
23         perror("readdir");
24         exit(EXIT_FAILURE);
25     }
26     if (closedir(dirp) == -1) {
27         perror("closedir");
28         exit(EXIT_FAILURE);
29     }
30 }
```

- Renombrado de ficheros o directorios:

```
#include <stdio.h> /* POSIX */  
2 int rename(const char *oldpath, const char *newpath);
```

- La llamada **rename()** renombra un fichero o un directorio, o bien lo mueve a otro directorio dentro del mismo sistema de ficheros
- **rename()** no mueve ni copia bloques de datos del fichero, no actualiza su contador de enlaces y no afecta a los procesos que tengan descriptores de fichero abiertos para ese fichero

- Eliminación de ficheros o directorios:

```
#include <stdio.h> /* POSIX */  
2 int remove(const char *pathname);
```

- **remove()** llama a **unlink()** si **pathname** es un fichero y a **rmdir()** si es un directorio, es decir, borra un fichero o un directorio

- Obtención del directorio de trabajo de un proceso:

```
#include <unistd.h> /* POSIX */  
2 char *getcwd(char *cwdbuf, size_t size);
```

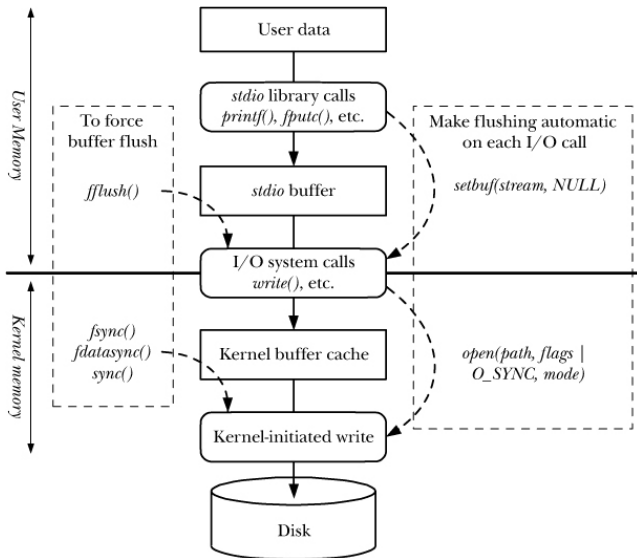
- `getcwd()` devuelve el directorio de trabajo actual de un proceso que determina el punto inicial para la resolución de rutas relativas

- Cambio del directorio de trabajo de un proceso:

```
#include <unistd.h> /* POSIX */  
2 int chdir(const char *pathname);
```

- `chdir()` cambia el directorio de trabajo actual de un proceso

Buffers de E/S: *Standard IO Library (stdio)* y Kernel



Standard IO Library (stdio) buffer

- Algunas funciones de **stdio**, agrupan los datos leídos/escritos en *buffers* de memoria para reducir el número de llamadas al sistema
 - fopen()**, **fread()**, **fwrite()**, **fscanf()**, **fprintf()**, **fclose()**

```
#include <stdio.h> /* C99 y POSIX */
2 int fflush(FILE *stream);
```

- Por defecto, la estrategia para cada **stream** es diferente:³³
 - stdin** y **stdout**: *line buffering*
 - stderr**: sin *buffering*
 - Otros *streams* de salida (ficheros): *block buffering*
- fflush()** obliga a escribir el *buffer* de **stream** con **write()**
 - fflush(NULL)** escribe el *buffer* de todos los *streams* de **stdio**
 - Si se cierra un **stream**, automáticamente se ejecuta **fflush(stream)**

- Código: **noprint.c** 

```
int main(int argc, char* argv[])
2 {
    printf("Depurando...");
4     raise(SIGSEGV);
    return EXIT_SUCCESS;
6 }
```

³³ Se puede cambiar con las funciones de la biblioteca estándar de C **setbuf()** y **setvbuf()**.

Ejemplo: fcat fichero

● Código: fcat.c

```
1      f = fopen(argv[1], "r");
2      if (f == NULL) {
3          perror("open");
4          exit(EXIT_FAILURE);
5      }
6
7      while((num_read = fread(buf, sizeof(char), BUF_SIZE, f)) > 0) {
8          num_written = fwrite(buf, sizeof(char), num_read, stdout);
9          if (ferror(f)) {
10             fprintf(stderr, "fwrite error");
11             exit(EXIT_FAILURE);
12         }
13         /* Escrituras parciales no tratadas */
14         assert(num_written == num_read);
15     }
16
17     if (ferror(f)) {
18         fprintf(stderr, "fread error");
19         exit(EXIT_FAILURE);
20     }
21
22     if (fflush(f) == EOF) {
23         perror("fflush");
24         exit(EXIT_FAILURE);
25     }
26
27     if (fclose(f) == EOF) {
28         fprintf(stderr, "fclose error");
29         exit(EXIT_FAILURE);
30     }
```

Kernel *buffer cache*

- Las llamadas `read()` y `write()` NO operan directamente sobre el disco sino que sólo copian datos desde o hacia kernel *buffer cache*
 - `write()` no necesita esperar a que la operación termine
 - `read()` lee de *buffer cache* hasta que se produce un fallo³⁴
 - Se reduce la latencia de `read()` y `write()` y los accesos a disco
 - Desde el kernel 2.4, *buffer cache* = *page cache*
- Las llamadas `fsync()`, `fdatasync()` y `sync()` permiten sincronizar el contenido de *buffer cache* con el disco

```
1  #include <unistd.h>      /* POSIX */
   int fdatasync(int fd); /* Synchronized IO data integrity completion */
3  int fsync(int fd);      /* Synchronized IO file integrity completion */
   void sync(void);        /* System-wide fsync() */
```

- El kernel sincroniza el contenido de *buffer cache* con el disco a intervalos regulares en ausencia de sincronizaciones explícitas
- El *flag* `O_SYNC` de `open()` hace que todas las operaciones de lectura y escritura sean síncronas (`open()` + `fsync()`)
 - ¿Por qué no se habilita por defecto?

³⁴ En caso de acceso secuencial, el kernel podría realizar operaciones de *prefetching*.

Ejemplo: cat fichero

● Código: cat_sync.c

```
1      fd = open(argv[1], O_RDONLY);
2      if (fd == -1) {
3          perror("open");
4          exit(EXIT_FAILURE);
5      }
6
7      while((num_read = read(fd, buf, BUF_SIZE)) > 0) {
8          num_written = write(STDOUT_FILENO, buf, num_read);
9          if (num_written == -1) {
10             perror("write");
11             exit(EXIT_FAILURE);
12         }
13         /* Escrituras parciales no tratadas */
14         assert(num_written == num_read);
15     }
16
17     if (num_read == -1) {
18         perror("read");
19         exit(EXIT_FAILURE);
20     }
21
22     if (fsync(fd) == -1) {
23         perror("fsync");
24         exit(EXIT_FAILURE);
25     }
26
27     if (close(fd) == -1) {
28         perror("close");
29         exit(EXIT_FAILURE);
30     }
```

- Las llamadas **open()**, **read()**, **write()** y **close()** se pueden usar para realizar operaciones de E/S sobre cualquier tipo de fichero
- Dado un nodo-i, puede haber varias entradas en la tabla de ficheros abiertos que apunten a él y, a su vez, también pueden existir varios descriptores de fichero para la misma entrada en dicha tabla
- Algunas funciones de **stdio**, agrupan los datos leídos/escritos en *buffers* de memoria para reducir el número de llamadas al sistema
- Las llamadas **read()** y **write()** NO operan directamente sobre el disco sino que sólo copian datos desde o hacia kernel *buffer cache*

Gestión de memoria virtual [KER:C49-50]

1 Introducción [KER:C2] [STE:C1]

2 Estándares [KER:C1] [STE:C2]

3 Llamadas al sistema y funciones de biblioteca [KER:C3]

4 Servicios POSIX y **glibc**

- Creación y terminación de procesos [KER:C6,24-28]
- Acceso a sistemas de ficheros [KER:C4-5,13-15,18]
- Gestión de memoria virtual [KER:C49-50]
- Gestión de memoria dinámica [KER:C7]
- Comunicación/sincronización de procesos (IPC) [KER:C20-23,43-44]

- Protección y bloqueo de páginas de memoria virtual:

```
1  #include <sys/mman.h> /* POSIX */
   int mprotect(void *addr, size_t length, int prot);
3  int mlock(const void *addr, size_t len);
   int munlock(const void *addr, size_t len);
5  int mlockall(int flags);
   int munlockall(void);
```

- `mprotect()` permite proteger páginas contra lectura, escritura y/o ejecución (`PROT_NONE` o `PROT_READ`, `PROT_WRITE` y/o `PROT_EXEC`)
- Si un proceso intenta acceder a una página violando su tipo de protección, recibe una señal `SIGSEGV` del kernel
- `mlock()` bloquea una o más páginas del proceso en memoria (no pueden ser expulsadas a la zona de intercambio de disco)
- `mlockall()` bloquea todas las páginas del proceso
- Los bloqueos de memoria no son heredados por los procesos hijo creados con `fork()` ni se preservan tras una llamada a `execve()`
- ¿Qué ventajas e inconvenientes tiene bloquear páginas en memoria?

Servicios POSIX para gestión de memoria virtual

● Código: mprotect.c

```
1  int main(int argc, char *argv[])
2  {
3      int pagesize = sysconf(_SC_PAGE_SIZE);
4      if (pagesize == -1) {
5          perror("pagesize");
6          exit(EXIT_FAILURE);
7      }
8
9      /* Allocate a buffer aligned on a page boundary;
10       initial protection is PROT_READ | PROT_WRITE */
11
12     void *buffer;
13     if (posix_memalign(&buffer, pagesize, pagesize) == -1) {
14         perror("posix_memalign");
15         exit(EXIT_FAILURE);
16     }
17
18     memset(buffer, 0, pagesize);
19     printf("Before mprotect call...\n");
20
21     if (mprotect(buffer, pagesize, PROT_NONE) == -1) {
22         perror("mprotect");
23         exit(EXIT_FAILURE);
24     }
25
26     printf("After mprotect call...\n");
27     memset(buffer, 0, pagesize);
28
29     exit(EXIT_SUCCESS);
30 }
```

Servicios POSIX para gestión de mapeos de memoria

- **mmap()** permite mapear un fichero en el espacio de direcciones virtuales de un proceso³⁵

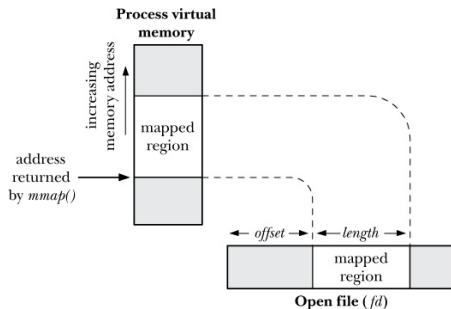
```
1  #include <sys/mman.h> /* POSIX */  
   void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset);  
3  int     munmap(void *addr, size_t length);
```

- **prot** es una máscara de bits idéntica a **prot** de **mprotect()**
- **flags** es una máscara de bits con **MAP_PRIVATE** o **MAP_SHARED**³⁶
 - **MAP_PRIVATE**: Las modificaciones no son visibles para otros procesos ni se escriben a disco
 - **MAP_SHARED**: Las modificaciones son visibles para otros procesos y se escriben a disco
- Los mapeos de memoria son heredados por los procesos hijo creados con **fork()** pero no se preservan tras una llamada a **execve()**

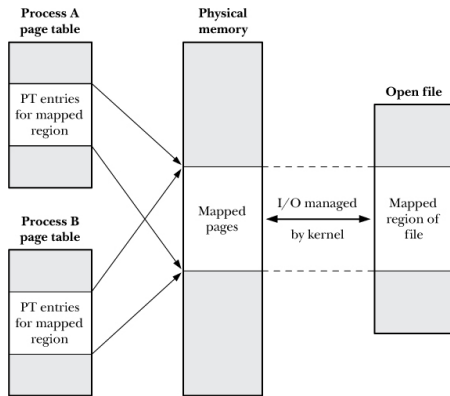
³⁵ cat /proc/\$\$/maps

³⁶ Existen otros tipos de mapeo con usos diferentes.

Ficheros mapeados en memoria



Fichero mapeado en memoria privado o compartido



Fichero mapeado en memoria compartido

- Ficheros mapeados en memoria privados

- Permiten que múltiples procesos compartan el mismo segmento de código y el mismo segmento de datos de un programa o de una biblioteca compartida (*dynamic linker*)
- Se usan para inicializar páginas de memoria virtual con un fichero

```
1 $ ldd /bin/ls | grep libc
   libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007fa1a487a000)
3 $ ldd /bin/ps | grep libc
   libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007fac9cd34000)
```

- Ficheros mapeados en memoria compartidos

- E/S mapeada en memoria (*memory-mapped IO*)
 - Simplifica la lógica de la aplicación eliminando `read()` y `write()`
 - Mejora el rendimiento en algunos casos al no requerir el uso de los buffers en el espacio de memoria del kernel ni las llamadas al sistema
- Comunicación rápida entre procesos (*fast IPC*)

- Las bibliotecas agrupan funciones usadas por múltiples programas
- Cada programa incluye una lista de sus dependencias dinámicas (bibliotecas compartidas) en el fichero binario (`ldd ./programa`)
- Cuando un programa se ejecuta, el *dynamic linker* satisface las dependencias dinámicas buscando las bibliotecas compartidas en un conjunto de directorios predeterminados (p.e., `/lib` y `/usr/lib`)³⁷
 - En Ubuntu 18.04, el *dynamic linker* corresponde a la biblioteca `ld-linux-x86-64.so.2 (/lib/x86_64-linux-gnu/ld-2.27.so)`
 - Cada biblioteca compartida se mapea en el espacio de direcciones virtuales del proceso cargándola en memoria física si no lo está ya
 - El código se tiene que compilar como *Position Independent Code* (PIC) porque la ubicación del código de una biblioteca en el espacio de direcciones virtuales de un proceso se desconoce de antemano
 - Si un símbolo (p.e., una función) está definido en varias bibliotecas compartidas, por defecto, el proceso usa el que encuentra primero

³⁷ Para más detalles, véase <http://man7.org/linux/man-pages/man8/ld.so.8.html>

● Código: mmap.c 38

```
1      fd = open(argv[1], O_RDWR);
2      if (fd == -1) {
3          perror("open");
4          exit(EXIT_FAILURE);
5      }
6      addr = mmap(NULL, ARRAY_SIZE_IN_FILE * sizeof(char),
7                  PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
8      if (addr == MAP_FAILED) {
9          perror("mmap");
10         exit(EXIT_FAILURE);
11     }
12     if (close(fd) == -1) {
13         perror("close");
14         exit(EXIT_FAILURE);
15     }
16     letter = (char*) addr;
17     for(int i = 0; i < ARRAY_SIZE_IN_FILE; i++) {
18         printf("Current letter=%c\n", (*letter)++); letter++;
19     }
20     if (munmap(addr, sizeof(long)) == -1) {
21         perror("munmap");
22         exit(EXIT_FAILURE);
23     }
```

38 echo -n aaaaaaaaaa > letters ; ./mmap letters ; hexdump letters ; ./mmap letters ; hexdump letter

- **mprotect()** protege páginas contra lectura, escritura y/o ejecución
- La llamada al sistema **mmap()** crea un mapeo de memoria en el espacio de direcciones virtuales del proceso privado o compartido
- Los ficheros mapeados en memoria privados permiten a varios procesos compartir una biblioteca
- Los ficheros mapeados en memoria compartidos permiten a varios procesos mapear un fichero, o parte de él, en sus espacios de direcciones

Gestión de memoria dinámica [KER:C7]

1 Introducción [KER:C2] [STE:C1]

2 Estándares [KER:C1] [STE:C2]

3 Llamadas al sistema y funciones de biblioteca [KER:C3]

4 Servicios POSIX y **glibc**

- Creación y terminación de procesos [KER:C6,24-28]
- Acceso a sistemas de ficheros [KER:C4-5,13-15,18]
- Gestión de memoria virtual [KER:C49-50]
- Gestión de memoria dinámica [KER:C7]
- Comunicación/sincronización de procesos (IPC) [KER:C20-23,43-44]

- La gestión de memoria una vez el programa se ha cargado se realiza de forma dinámica
- Reserva de memoria en el *heap* (según el mapa de memoria, el *heap* comienza a continuación del programa cargado):
 - El límite actual del *heap* se denomina *program break*
 - Inicialmente, el *program break* es igual a la dirección `&end`
 - Las llamadas al sistema `brk()` y `sbrk()` permiten manipularlo

```
#include <unistd.h> /* POSIX legacy */  
2 int brk(void *end_data_segment);  
void *sbrk(intptr_t increment);
```

- `brk()` modifica el valor absoluto del *program break*
- `sbrk()` devuelve el valor del *program break* y lo incrementa o disminuye, es decir, `sbrk(0)` simplemente devuelve su valor actual sin cambiarlo
- Ambas llamadas existen en Linux pero han sido eliminadas de POSIX

- Reserva de memoria en el *heap*:

- Las funciones `malloc()`/`free()` reservan/liberan memoria del *heap*:

```
1  #include <stdlib.h>
   void *malloc(size_t size); /* C99 y POSIX */
3  void *calloc(size_t numitems, size_t size); /* C99 y POSIX */
   int  posix_memalign(void **memptr,
5                      size_t alignment, size_t size); /* POSIX */
   void *aligned_alloc(size_t alignment, size_t size); /* C11 */
7  void *realloc(void *ptr, size_t size); /* C99 y POSIX */
   void free(void *ptr); /* C99 y POSIX */
```

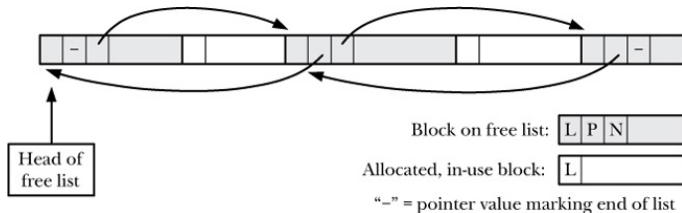
- `malloc()` devuelve `void *` que se puede asignar a cualquier puntero
- `calloc()` es similar a `malloc()` pero inicializa el bloque a 0
- `posix_memalign()` es similar a `malloc()` pero el bloque está alineado a un múltiplo de `alignment` que debe ser potencia de 2
- En `posix_memalign/aligned_alloc`, `size` es múltiplo de `alignment`
- `realloc()` redimensiona el tamaño del bloque a `size`
- `free()` libera un bloque previamente reservado con `malloc()`, `calloc()`, `posix_memalign()`, `aligned_alloc()` o `realloc()`

To free() or not to free()?

- Cuando un proceso termina, toda su memoria se devuelve al sistema
- Si un proceso reserva memoria y la utiliza hasta que termina, resulta habitual dejar que la misma se devuelva al sistema automáticamente
- Aunque esta estrategia puede ser aceptable en algunos casos, existen buenas razones para liberar explícitamente la memoria:
 - Código fuente mucho más legible y fácil de mantener o extender
 - El código se puede usar como parte de otro programa (no sólo como un programa separado que depende del S.O. para limpiar su memoria)
 - Las herramientas de depuración identificarán las regiones de memoria reservadas pero no liberadas como fugas de memoria (*memory leaks*)

Implementación de `malloc()` y `free()`

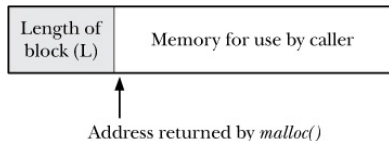
- Internamente, `malloc()` y `free()` mantienen lista de bloques libres



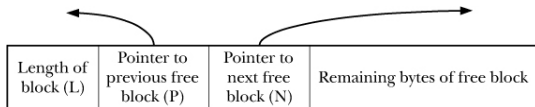
- Todos los bloques restantes que están en el *heap* y que no están en la lista de bloques libres, están ocupados o no han sido liberados
- `malloc()` explora la lista buscando un bloque de tamaño \geq `size`
 - Si el bloque es más grande que `size`, `malloc()` lo fragmenta
 - Si no hay ningún bloque de este tamaño, `malloc()` llama a `sbrk()`

Implementación de `malloc()` y `free()`

- `malloc()` reserva memoria adicional en cada bloque para su tamaño



- `free()` utiliza el tamaño para insertar el bloque de nuevo en la lista
- Además, `free()` usa el propio bloque para almacenar *Prev* y *Next*



Implementación de `malloc()` y `free()`

Reglas básicas para usar `malloc()` y `free()` correctamente:

- No modificar ningún byte fuera del rango reservado con `malloc()`
- No modificar el puntero entre las llamadas a `malloc()` y `free()`
- No llamar a `free()` con un bloque no reservado con `malloc()`
- No liberar el mismo bloque más de una vez con `free()`
- Garantizar que cualquier bloque no usado es liberado con `free()`
 - Si un proceso reserva repetidamente bloques y no los libera, antes o después el *heap* alcanzará su tamaño máximo (*memory exhaustion*)

Implementación de malloc() y free()

● Código: sbrk.c

```
2  /*
4  +-----+-----+-----+-----+
   | block 0   | block 1   | ...       | block 999   |
   +-----+-----+-----+-----+
                                           <--- brk

6  After free:
   +-----+-----+-----+-----+
   | (free)    | (free)    | ...       | block 999   |
   +-----+-----+-----+-----+
                                           <--- brk

8  */

10
12
14  int main()
15  {
16      char* blocks[NBLOCKS];
17
18      printf("Initial program break:          %10p\n", sbrk(0));
19
20      printf("Allocating %d blocks of 20,000 bytes\n",NBLOCKS);
21      for (int i = 0; i < NBLOCKS; ++i)
22          blocks[i] = malloc(20000);
23      printf("After malloc(), program break is: %10p\n", sbrk(0));
24
25      printf("Freeing blocks (all but last)\n");
26      for (int i = 0; i < NBLOCKS - 1; ++i)
27          free (blocks[i]);
28      printf("After free(), program break is:   %10p\n", sbrk(0));
29
30      exit(EXIT_SUCCESS);
31  }
```

Caso práctico: malloc() y free()

- ¿Qué sucedería si ejecutásemos el siguiente código? ¿Por qué?

```
1  char *path = (char*) malloc(PATH_MAX * sizeof(char));  
  
3  if ( ! getcwd(path, PATH_MAX) )  
4  {  
5      perror("get_cmd: getcwd");  
6      free(path);  
7      exit(EXIT_FAILURE);  
8  }  
  
9  
10 path = basename(path);  
11 . . .  
   free(path);
```

- ¿Por qué **free()** no especifica el tamaño del bloque liberado?

- ¿Qué errores hay en este programa?: mem_leak.c  39

```
1  #define _POSIX_C_SOURCE 200809L
2  #include <stdlib.h>
3
4  #define BUFSIZE 10
5
6  void f(void)
7  {
8      int* x = malloc(BUFSIZE * sizeof(int));
9      x[BUFSIZE] = 0;
10 }
11
12 int main(void)
13 {
14     f();
15     return 0;
16 }
```

³⁹ valgrind --leak-check=full ./mem_leak

Implementación de `realloc()`

- `realloc()` se usa para aumentar el tamaño del bloque
 - Si el bloque está en la mitad del *heap*, `realloc()` intenta fusionar el bloque actual con los siguientes bloques libres, si los hay, hasta alcanzar el tamaño mínimo requerido por el parámetro **size**
 - Si no fuese posible, `realloc()` reserva un nuevo bloque y **copia** todos los datos desde el bloque antiguo hasta el bloque nuevo
 - Si el bloque está al final del *heap*, `realloc()` llama a `sbrk()`

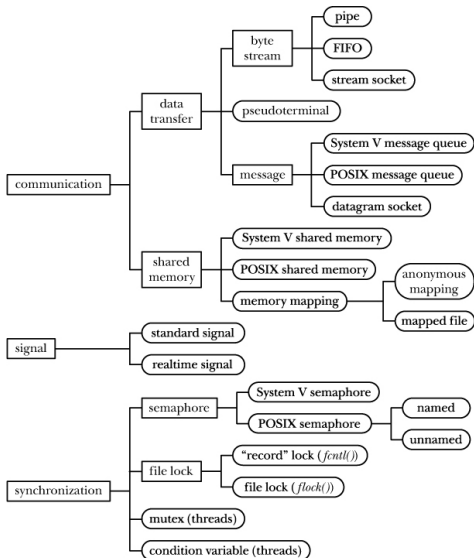
- Las funciones `malloc()`/`free()` reservan/liberan memoria del *heap*
- Por cada llamada a `malloc()` debe haber otra llamada a `free()`
- Entre la llamada a `malloc()` y `free()` no se debe modificar el puntero
- Si un proceso reserva repetidamente bloques y no los libera, antes o después el *heap* alcanzará su tamaño máximo (*memory exhaustion*)

Comunicación/sincronización de procesos (IPC)

[KER:C20-23,43-44]

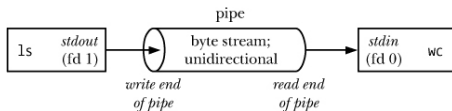
- 1 Introducción [KER:C2] [STE:C1]
- 2 Estándares [KER:C1] [STE:C2]
- 3 Llamadas al sistema y funciones de biblioteca [KER:C3]
- 4 Servicios POSIX y **glibc**
 - Creación y terminación de procesos [KER:C6,24-28]
 - Acceso a sistemas de ficheros [KER:C4-5,13-15,18]
 - Gestión de memoria virtual [KER:C49-50]
 - Gestión de memoria dinámica [KER:C7]
 - Comunicación/sincronización de procesos (IPC) [KER:C20-23,43-44]

Taxonomía de servicios para IPC

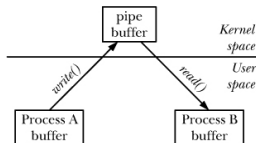


Tuberías

- Una tubería o *pipe* es un flujo de bytes unidireccional, que se entregan en el mismo orden de envío, entre dos procesos



- Una tubería es un *buffer* de capacidad limitada en el espacio de direcciones del kernel⁴⁰



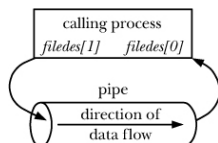
- Una lectura con **read()** de una tubería *vacía* se bloquea (si se cierra el extremo izquierdo, **read()** devuelve cero –EOF–)
- Una escritura con **write()** a una tubería *llena* se bloquea

⁴⁰ `cat /proc/sys/fs/pipe-max-size`

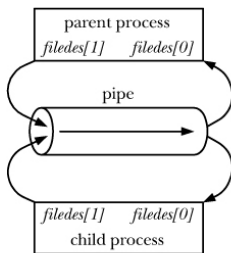
Servicios POSIX para tuberías

- Una llamada a `pipe()` devuelve dos descriptors de fichero, uno para lectura (`filedes[0]`) y otro para escritura (`filedes[1]`)

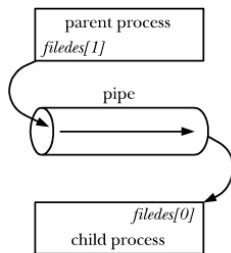
```
#include <unistd.h> /* POSIX */
2 int filedes[2];
if (pipe(filedes) == -1) {
4     perror("pipe");
    exit(EXIT_FAILURE);
6 }
```



- Una tubería puede utilizarse para comunicación/sincronización entre procesos *relacionados* (la tubería fue creada por un ancestro común)



a) After `fork()`



b) After closing unused descriptors

● Código: ls_pipe.c

```
2      switch (fork()) {
3          case -1:
4              perror("fork 1");
5              exit(EXIT_FAILURE);
6              break;
7          case 0: /* First child: exec 'ls' to write to pipe */
8              if (close(filedes[0]) == -1) { /* Read end is unused */
9                  perror("close 1");
10                 exit(EXIT_FAILURE);
11             }
12             /* dup2 stdout on write end of pipe; close duplicated descriptor */
13             if (filedes[1] != STDOUT_FILENO) { /* Defensive check */
14                 if (dup2(filedes[1], STDOUT_FILENO) == -1) {
15                     perror("dup2 1");
16                     exit(EXIT_FAILURE);
17                 }
18                 if (close(filedes[1]) == -1) {
19                     perror("close 2");
20                     exit(EXIT_FAILURE);
21                 }
22             }
23             execlp("ls", "ls", (char *) NULL);
24             perror("execlp ls");
25             break;
26         default: /* Parent falls through */
27             break;
28     }
```


● Código (cont.):

```
2      switch (fork()) {
3          case -1:
4              perror("fork 2");
5              exit(EXIT_FAILURE);
6              break;
7          case 0: /* Second child: exec 'wc' to read from pipe */
8              if (close(filedes[1]) == -1) { /* Write end is unused */
9                  perror("close 3");
10                 exit(EXIT_FAILURE);
11             }
12             /* dup2 stdin on read end of pipe; close duplicated descriptor */
13             if (filedes[0] != STDIN_FILENO) { /* Defensive check */
14                 if (dup2(filedes[0], STDIN_FILENO) == -1) {
15                     perror("dup2 2");
16                     exit(EXIT_FAILURE);
17                 }
18                 if (close(filedes[0]) == -1) {
19                     perror("close 4");
20                     exit(EXIT_FAILURE);
21                 }
22             }
23             execlp("wc", "wc", "-l", (char *) NULL);
24             perror("execlp wc");
25             break;
26         default: /* Parent falls through */
27             break;
28     }
```

- Una señal es una notificación a un proceso ante un evento⁴¹
 - Excepción hardware: Dirección de memoria inválida (SIGSEGV)
 - Evento software: Un proceso hijo ha terminado (SIGCHLD)
 - Notificación de E/S: Interrupción con CTRL+C (SIGINT)
- Un proceso puede recibir una señal enviada por otro o por el kernel
- Una señal es un entero ≥ 1 definida en `signal.h` como SIGXXXX
- Desde que una señal se genera hasta que se entrega está *pendiente*
- Una señal pendiente se entrega al proceso en la siguiente transición de modo kernel a modo usuario, es decir, cuando se completa una llamada al sistema o cuando se reanuda su ejecución (*timeslice*)
- Un proceso puede bloquear la entrega de una señal de manera indefinida mediante una *máscara de señales* que la deja *bloqueada*
- Referencia: <http://man7.org/linux/man-pages/man7/signal.7.html>

⁴¹ `cat /proc/$$/status | grep ^Signal: ⇒ SigPnd (per-thread pending signals), ShdPnd (process-wide pending signals), SigBlk (blocked signals), SigIgn (ignored signals), and SigCgt (caught signals)`

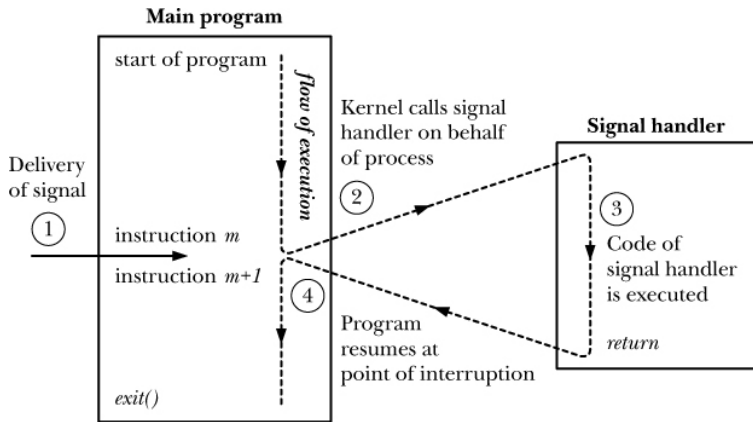
- Cuando un proceso recibe una señal, ejecuta la acción por defecto, que puede ser una de las siguientes:
 - La señal es ignorada
 - El proceso es matado (*killed*)
 - Se genera un fichero *core* y el proceso es matado
 - El proceso se detiene (*stopped*)
 - El proceso reanuda su ejecución (*resumed*)
- Un proceso puede cambiar la acción por defecto:
 - Ignorando la señal
 - Ejecutando un manejador de señales (*signal handler*)
 - Restaurando la acción por defecto

Señales

Acciones por defecto

Name	Signal number	Description	SUSv3	Default
SIGABRT	6	Abort process	•	core
SIGALRM	14	Real-time timer expired	•	term
SIGBUS	7 (SA=10)	Memory access error	•	core
SIGCHLD	17 (SA=20, MP=18)	Child terminated or stopped	•	ignore
SIGCONT	18 (SA=19, M=25, P=26)	Continue if stopped	•	cont
SIGEMT	undef (SAMP=7)	Hardware fault	•	term
SIGFPE	8	Arithmetic exception	•	core
SIGHUP	1	Hangup	•	term
SIGILL	4	Illegal instruction	•	core
SIGINT	2	Terminal interrupt	•	term
SIGIO / SIGPOLL	29 (SA=23, MP=22)	I/O possible	•	term
SIGKILL	9	Sure kill	•	term
SIGPIPE	13	Broken pipe	•	term
SIGPROF	27 (M=29, P=21)	Profiling timer expired	•	term
SIGPWR	30 (SA=29, MP=19)	Power about to fail	•	term
SIGQUIT	3	Terminal quit	•	core
SIGSEGV	11	Invalid memory reference	•	core
SIGSTKFLT	16 (SAM=undef, P=36)	Stack fault on coprocessor	•	term
SIGSTOP	19 (SA=17, M=23, P=24)	Sure stop	•	stop
SIGSYS	31 (SAMP=12)	Invalid system call	•	core
SIGTERM	15	Terminate process	•	term
SIGTRAP	5	Trace/breakpoint trap	•	core
SIGTSTP	20 (SA=13, M=24, P=25)	Terminal stop	•	stop
SIGTTIN	21 (M=26, P=27)	Terminal read from BG	•	stop
SIGTTOU	22 (M=27, P=28)	Terminal write from BG	•	stop
SIGURG	23 (SA=16, M=21, P=29)	Urgent data on socket	•	ignore
SIGUSR1	10 (SA=30, MP=16)	User-defined signal 1	•	term
SIGUSR2	12 (SA=31, MP=17)	User-defined signal 2	•	term
SIGVTALRM	26 (M=23, P=20)	Virtual timer expired	•	term
SIGWINCH	28 (M=20, P=23)	Terminal window size change	•	ignore
SIGXCPU	24 (M=30, P=33)	CPU time limit exceeded	•	core
SIGXFSZ	25 (M=31, P=34)	File size limit exceeded	•	core

- Un manejador de señales es una función definida por el usuario que realiza acciones apropiadas en respuesta a una señal concreta⁴²



⁴² La interrupción del *timer* puede reanudar la ejecución del kernel en cualquier momento. Por tanto, no es posible predecir cuando entrará en acción un manejador de señales cuando el proceso que lo instaló recibe la señal que provoca su ejecución.

- No todas las llamadas al sistema y funciones de biblioteca se pueden usar de forma *segura* en un manejador de señales⁴³
 - Una función es **reentrante** si puede ser ejecutada de forma *segura* por múltiples *threads* dentro del mismo proceso de manera simultánea
 - Una función es **no reentrante** si modifica estructuras de datos globales (`strtok()`), devuelve información en memoria global (`getpwuid()`) o usa estructuras de datos globales internamente (`printf()`)
 - Si un manejador de señales actualiza una estructura de datos global, dicho manejador no es reentrante con respecto al programa principal
- En general, una llamada al sistema o función de biblioteca es *segura* (*async-signal-safe*) si es reentrante o no puede ser interrumpida
- Referencia: <http://man7.org/linux/man-pages/man7/signal-safety.7.html>
<http://man7.org/linux/man-pages/man7/attributes.7.html>

⁴³ Para saber si una llamada es segura, véase la sección ATTRIBUTES de la página del manual correspondiente.

● Consulta y modificación de la acción por defecto:

```
#include <signal.h> /* POSIX */
2  int sigaction(int sig, const struct sigaction *act, struct sigaction *oldact);

4  struct sigaction {
    void (*sa_handler)(int); /* Address of handler */
6  void (*sa_sigaction)(int, siginfo_t *, void *); /* Address of handler */
    sigset_t sa_mask; /* Signals blocked during handler invocation */
8  int sa_flags; /* Flags controlling handler invocation */
    void (*sa_restorer)(void); /* Not for application use */
10 };
```

- **sig** es la señal para la cual se realiza la consulta o modificación
- **sa_handler** se refiere al manejador de la señal o a las constantes **SIG_IGN** (ignorar la señal) o **SIG_DFL** (restaurar la acción por defecto)
- **sa_mask** define qué señales se bloquearán durante la ejecución del manejador de la señal además de las ya bloqueadas en la máscara de señales del proceso con **sigprocmask()** (incluida la señal **sig**)
- **sa_flags** es una máscara de bits con opciones que afectan a la ejecución del manejador de la señal, por ejemplo, **SA_RESETHAND** restablece la acción por defecto cuando se ejecuta el manejador y **SA_SIGINFO** permite al manejador obtener información adicional en **siginfo_t** como, por ejemplo, el PID del proceso que envía la señal

- Consulta y modificación de la máscara de bloqueo de señales:

```
#include <signal.h> /* POSIX */  
2  int sigprocmask(int how, const sigset_t *set, sigset_t *oldset);
```

- Si `set` es `NULL`, `sigprocmask()` devuelve la máscara de señales del proceso en `oldset`
 - En caso contrario, `sigprocmask()` modifica la máscara de señales del proceso en función de `how`:
 - `SIG_BLOCK` añade `set` a la máscara de señales del proceso
 - `SIG_UNBLOCK` elimina `set` de la máscara de señales del proceso
 - `SIG_SETMASK` establece la máscara de señales del proceso a `set`
- `sigset_t` es un conjunto de señales gestionado con `sigemptyset()`, `sigfillset()`, `sigaddset()`, `sigdelset()` y `sigismember()`
- `SIGKILL` y `SIGSTOP` no se pueden capturar, bloquear o ignorar

- Consulta de la máscara de señales pendientes de entrega:

```
#include <signal.h> /* POSIX */  
2  int sigpending(sigset_t *set);
```

- **sigpending()** devuelve el conjunto de señales pendientes para el proceso que realiza la llamada que se consulta con **sigismember()**
- El conjunto de señales pendientes es sólo una máscara que indica si una señal se ha recibido o no, es decir, si una señal se recibe varias veces mientras está bloqueada, quedará registrada en el conjunto de señales pendientes y sólo se entregará una vez que se desbloquee

- Envío y recepción de señales:

```
#include <signal.h> /* POSIX */
2  int kill(pid_t pid, int sig);
   int raise(int sig);
4  #include <stdlib.h> /* POSIX */
   void abort(void);
6  #include <unistd.h> /* POSIX */
   int pause(void);
```

- **kill()** entrega la señal **sig** a uno o más procesos:
 - Si **pid** > 0, se entrega la señal al proceso con PID **pid**
 - Si **pid** es -1, se entrega la señal a todos los procesos a los que el proceso que realiza la llamada pueda enviar una señal excepto a **init**
 - Si **sig** es 0, no se envía ninguna señal pero se realiza la comprobación de errores que permite determinar la existencia o no de un proceso
- **raise()** entrega la señal **sig** al proceso que ejecuta la llamada
- **abort()** desbloquea la señal **SIGABRT** y/o restaura su acción por defecto para poder entregarla al proceso que hace la llamada
- **pause()** suspende la ejecución del proceso hasta que se recibe una señal que lo mata o causa la ejecución de un manejador de señal

● Código: signal.c

```
1  static void signal_handler(int sig)
2  {
3      /* SIGINT (^C): increase the counter */
4      if (sig == SIGINT) {
5          count++;
6          /* Resume execution at point of interruption */
7          return;
8      }
9
10     /* SIGQUIT (^\\): terminate the process */
11     /* exit is not an async-signal-safe call */
12     _exit(EXIT_SUCCESS);
13 }
```

● Código (cont.):

```
1  int main(int argc, char *argv[])
2  {
3      /* Block signal SIGSEGV */
4      sigset_t blocked_signals;
5      sigemptyset(&blocked_signals);
6      sigaddset(&blocked_signals, SIGSEGV);
7      if (sigprocmask(SIG_BLOCK, &blocked_signals, NULL) == -1) {
8          perror("sigprocmask");
9          exit(EXIT_FAILURE);
10     }
11
12     /* Establish same handler for SIGINT and SIGQUIT */
13     struct sigaction sa;
14     memset(&sa, 0, sizeof(sa)); /* SIGSEGV!!! */
15     sa.sa_handler = signal_handler;
16     sigemptyset(&sa.sa_mask);
17     if (sigaction(SIGINT, &sa, NULL) == -1) {
18         perror("sigaction 1");
19         exit(EXIT_FAILURE);
20     }
21     if (sigaction(SIGQUIT, &sa, NULL) == -1) {
22         perror("sigaction 2");
23         exit(EXIT_FAILURE);
24     }
25 }
```

- Código (cont.):

```
    sigset_t pending_signals;
2    sigemptyset(&pending_signals);
    while(1)      /* Loop forever, waiting for signals */
4    {
        pause(); /* Block until a signal is caught */
6        printf("Caught SIGINT %d times before SIGQUIT\n", count);
        if (sigpending(&pending_signals) == -1) {
8            perror("sigpending");
            exit(EXIT_FAILURE);
10        } else {
            if (sigismember(&pending_signals, SIGSEGV))
12                printf("Pending SIGSEGV\n");
            }
14    }

16    /* The program will never get here! */
    return EXIT_SUCCESS;
18 }
```

Servicios POSIX para señales

Terminal 1	Terminal 2
./signal	
	ps aux grep signal ⇒ PID
¿?	kill -s SIGSEGV PID

Terminal 1	Terminal 2
./signal	
	ps aux grep signal ⇒ PID
¿?	kill -s SIGINT PID
¿?	kill -s SIGINT PID

Terminal 1	Terminal 2
./signal	
	ps aux grep signal ⇒ PID
CTRL+Z	
¿?	kill -s SIGINT PID
¿?	kill -s SIGINT PID
fg	
¿?	kill -s SIGINT PID

- Si un proceso ignora una señal S, cuya acción por defecto es TERM, recibe la señal N veces y, a continuación, restaura la acción por defecto para S, ¿cómo se verá afectada la ejecución del proceso?
- Si un proceso bloquea una señal S, recibe la señal N veces y, a continuación, la desbloquea, ¿cuántas veces se ejecutaría el manejador de dicha señal?

- Una tubería puede utilizarse para comunicación/sincronización entre procesos *relacionados* (la tubería fue creada por un ancestro común)
- Una señal es una notificación asíncrona que se entrega a un proceso en respuesta a un determinado evento hardware o software
- Un manejador de señales es una función definida por el usuario que realiza acciones apropiadas en respuesta a una señal concreta
- No todas las llamadas al sistema y funciones de biblioteca se pueden usar de forma *segura* en un manejador de señales