# Introducción a Sistemas Operativos: Procesos

*Francisco J Ballesteros*

## 1. Programas y procesos

A un programa en ejecución se lo denomina *proceso*. El nombre *programa* no se utiliza para referirse a un programa que está ejecutando dado que ambos conceptos (programa y proceso) son diferentes. La diferencia es la misma que hay entre una receta de cocina para hacer galletas y una galleta hecha con la receta. El programa es únicamente una serie de datos (con las instrucciones y datos para ejecutarlo) y no es algo vivo. Por otro lado, un proceso es un programa que está vivo. Tiene una serie de registros, incluyendo un contador de programa, y utiliza una pila para hacer llamadas a procedimiento (y al sistema). Esto significa que un proceso tiene un *flujo de control* que ejecuta una instrucción tras otra, como ya sabes.

La diferencia queda clara si consideras que puedes ejecutar a la vez el mismo programa, varias veces. Por ejemplo, la figura 1 muestra varias ventanas que están ejecutando un shell.

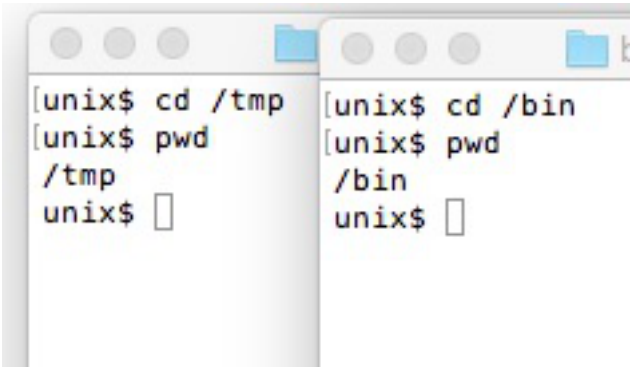


**Figura 1:** *Varias ventanas ejecutando un shell. Hay solo un programa pero tenemos varios procesos.*

En ambos casos ejecutan el mismo programa: el que está en `/bin/sh`. Pero, aunque hay un sólo programa, tenemos varios procesos. Por ejemplo, si cambiamos de directorio en uno de ellos, el otro sigue inalterado. Cada proceso tiene sus propias variables en la memoria. Y, aunque el programa es el mismo, los valores de las variables serán en general diferentes en cada proceso. Esto es obvio si piensas que cada shell estará haciendo una cosa distinta en un momento dado. No obstante, el programa tiene únicamente un conjunto de variables declaradas.

¿Qué es entonces un proceso? Piensa en los programas que has hecho. Elige uno de ellos. Cuando lo ejecutas, comienza a ejecutar *independientemente* del resto de programas en el ordenador. Al programarlo, ¿has tenido que tener en cuenta otros programas como el shell, el sistema de ventanas, el reloj, el navegador web, ...? ¡Por supuesto que no! Haría falta un cerebro del tamaño de la luna para tener todo eso en cuenta. Dado que no existen esos cerebros, el Sistema Operativo se ocupa de darte como abstracción el *proceso*. Gracias a esa abstracción puedes programar y ejecutar un programa olvidando el resto de programas que tienes en el sistema: Los procesos son programas que ejecutan independientemente del resto de programas que están ejecutando en el sistema.

Cada proceso viene con la ilusión de tener su propio procesador. Naturalmente, esto es mentira y es parte de la abstracción. Cuando escribes un programa piensas que la máquina ejecuta una instrucción tras otra. Y piensas que toda la máquina es tuya. Bueno, en realidad, que toda la máquina es del programa que ejecuta.

La implementación de la abstracción *proceso* es responsable de esta fantasía.

Cuando existen varios procesadores o CPUs, varios programas pueden ejecutarse realmente a la vez, o *en paralelo*. Hoy en día, la mayoría de los ordenadores disponen de múltiples procesadores. Pero, en cualquier caso, seguramente ejecutes más programas que procesadores tienes. ¡Al mismo tiempo! Cuenta el número de ventanas que tienes abiertas y piensa que hay al menos un programa ejecutando por cada ventana. Seguro que no tienes tantos procesadores.

Lo que sucede es que el sistema hace los ajustes necesarios para que cada programa pueda ejecutar durante un tiempo. La figura 2 muestra la memoria del sistema con varios procesos ejecutando.

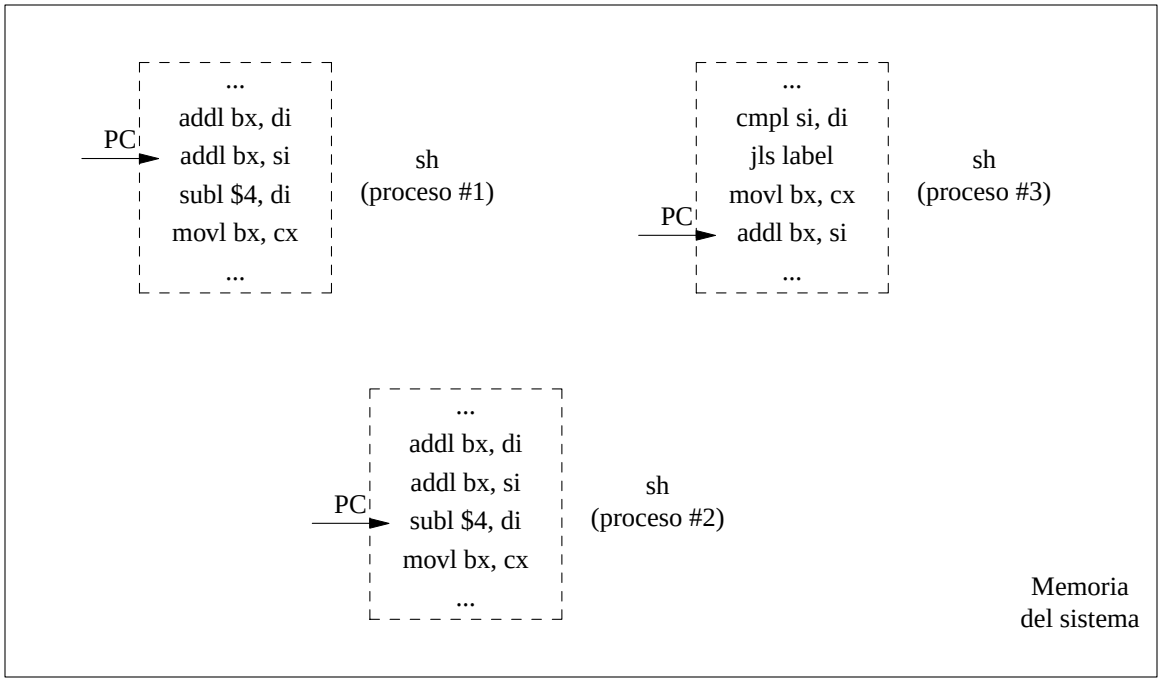


**Figura 2:** *Ejecución concurrente de varios procesos en el mismo sistema.*

Cada uno tiene su propio juego de registros, incluyendo el contador de programa. Además, cada uno tendrá su propia pila. La figura es tan solo una imagen de la memoria en un momento dado. Durante algún tiempo, el proceso 1 que ejecuta `sh` estará ejecutando en el procesador (sin que el Sistema Operativo intervenga mientras tanto). Después, un temporizador hardware arrancado anteriormente por el sistema interrumpirá la ejecución de dicho programa. En ese momento el sistema puede decidir que el proceso 1 ya ha tenido suficiente tiempo de ejecución por el momento, y en tal caso puede *saltar* para continuar ejecutando (por ejemplo), el proceso 2. Este proceso puede estar también ejecutando `sh`, o cualquier otro programa. Pasado cierto tiempo, otra interrupción de reloj llegará y una vez más se interrumpirá la ejecución del proceso que ejecuta. En este caso, el sistema (donde está el código del programa que atiende la interrupción) puede decidir que ya es hora de volver a ejecutar el proceso 1. De ser así, saltará al punto por el que estaba ejecutando dicho proceso la última vez que ejecutó.

Todo esto sucede detrás del telón. El sistema sabe que hay tan solo un flujo de control por cada procesador, y salta de un programa a otro utilizando dicho flujo. Pero, para los usuarios del sistema, lo que importa es que cada proceso parece estar ejecutando de forma independiente del resto. ¡Como si tuviera un procesador dedicado para el solo!

Como todos los procesos aparentan ejecutar de forma simultánea, decimos que ejecutan *concurrentemente*. O dicho de otro modo, decimos que son procesos concurrentes. En algunos casos, realmente ejecutan a la vez (cada uno en un procesador). De ser así, decimos que ejecutan *paralelamente*. En la mayoría de los

casos compartirán procesador (un tiempo cada uno) y diremos que son *pseudo-paralelos*. Pero, en la práctica, da igual si ejecutan de un modo u otro y simplemente los consideramos concurrentes. A la hora de programar y de utilizar el sistema esto es todo lo que importa.

En este tema vamos a explorar el proceso que obtenemos cuando ejecutamos un programa. Pero antes de hacer esto es importante ver qué hay en un programa y qué hay en un proceso, cosa que haremos a continuación.

## 2. Planificación y estados de planificación

Supongamos que hay sólo un procesador. De haber varios, lo que vamos a decir se aplica a cada uno de ellos sin más complicación. Cada proceso aparenta ejecutar de forma independiente, con su flujo de control propio, aunque el procesador sólo implementa un flujo de control.

Lo que sucede es que cuando un proceso entra al kernel, ya sea por una interrupción procedente del hardware o por que hace una llamada al sistema, el sistema tiene almacenado el estado del proceso que estaba ejecutando (normalmente en la pila en que se ha salvado el estado en la interrupción o al entrar al sistema). Gracias a esto, el sistema puede decidir que hay que saltar hacia otro proceso y puede *retornar* de la interrupción o llamada al sistema utilizando el estado que se salvó anteriormente para *otro* proceso distinto. Por ejemplo, se puede cambiar la pila por la de otro proceso de tal forma que cuando se retorne sea hacia ese otro proceso. De este modo, a cada proceso se le da un tiempo de procesador y, después, el sistema salta a otro distinto. A esta cantidad de tiempo se la denomina *cuanto*, y puede ser del orden de 100ms (lo que es una enormidad de tiempo para la máquina).

A transferir el control de un proceso a otro, salvando el estado del antiguo y recargando el estado del nuevo, lo denominamos *cambio de contexto*. Esto es obvio si pensamos que cambiamos el *contexto* (registros, pila, etc.) de un proceso a otro. Es de notar que es el kernel el que hace estos cambios de contexto. ¡Tu nunca incluyes saltos en tus programas para que otros programas ejecuten!

La parte del kernel responsable de decidir qué proceso es el siguiente que ejecuta en un cambio de contexto se denomina *planificador*, o *scheduler*. Es fácil si pensamos en que planifica la ejecución de los procesos. A las decisiones del planificador las conocemos como *planificación*, o *scheduling*. ¡Sorprendente! En la mayoría de sistemas el kernel puede sacar del procesador a un proceso incluso si este no hace llamadas al sistema (por ejemplo, mientras ejecuta un bucle). Se suelen utilizar las interrupciones de reloj a tal efecto. En este tipo de planificación, decimos que tenemos un planificador expulsivo (o, en Inglés, *preemptive*). Cuando el planificador no utiliza interrupciones para expulsar procesos decimos que tenemos un planificador *cooperativo*. Pero, en este último caso, un programa que entre en un bucle infinito sin llamar al sistema puede convertir la máquina en un pisapapeles.

Con un único procesador, sólo un proceso puede ejecutar en cada momento. El resto en general pueden estar *listos para ejecutar* (*ready*) pero no estarán ejecutando. Ya conoces dos estados de planificación: *listo* y *ejecutando*. Un proceso ejecutando pasa a estar listo para ejecutar si lo expulsan del procesador cuando su tiempo termina. En ese momento, el kernel ha de elegir a un proceso de entre los que están listos para ejecutar y ha de ponerlo a ejecutar.

Los estados de planificación son tan sólo constantes que define el sistema para guardarlas en los *records* o *structs* con los que implementa los procesos y saber así si puede elegir un proceso para ejecutarlo o no. La figura 3 muestra los estados de planificación de un proceso.

En algunos casos un proceso leerá de teclado o de una conexión de red o cualquier otro dispositivo. Cuando esto ocurre, el proceso habrá de esperar a que el usuario pulse una tecla, a que lleguen datos, o a que suceda alguna otra cosa. El proceso podría esperar entrando en un bucle, pero eso sería un desperdicio de procesador. En lugar de eso, cuando un proceso llama al sistema y el kernel ve que no puede continuar, se le

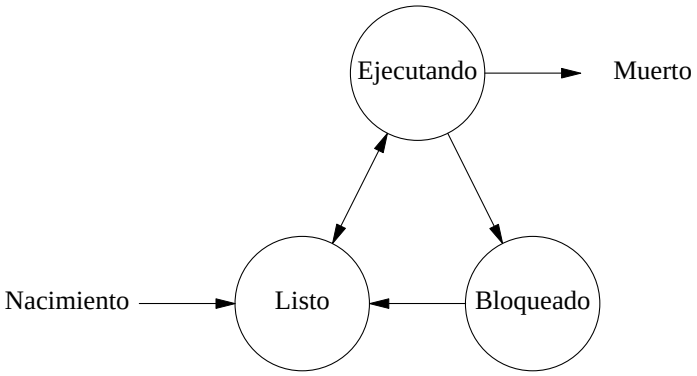


**Figura 3:** *Estados de planificación de un proceso.*

marca como *bloqueado* (otro estado de planificación) para que el kernel no lo ponga a ejecutar.

Los dispositivos en entrada/salida son tan lentos en comparación con el procesador que es posible ejecutar multitud de cosas mientras un proceso espera a que su entrada/salida termine. A hacer tal cosa se la denomina *multiprogramación*.

Pasado cierto tiempo, un proceso bloqueado volverá a marcarse como listo para ejecutar si el evento que estaba esperando el proceso sucede. Por ejemplo, si el proceso llamó al kernel para leer de teclado y, debido a eso, se bloqueó, en el momento en que escribimos algo en el teclado y el kernel puede dárselo el proceso, dicho proceso pasará a estar listo para ejecutar. La siguiente vez que el planificador ejecute, el proceso es un candidato a ejecutar.

Otra forma de dibujar el estado de los procesos a lo largo del tiempo es utilizar un *diagrama de planificación*. Se trata de dibujar una línea para cada proceso de tal forma que indicamos en qué estado está en cada momento (cambiando la forma o el color de la línea). Naturalmente, si tenemos un procesador, sólo podrá existir un proceso ejecutando en cada momento. Y además, ya sabes que en realidad el salto de un proceso a otro sucede utilizando el mecanismo que hemos descrito antes. Un ejemplo de diagrama de planificación podría ser el que puede verse en la figura 4. En dicha figura, hay dos procesos ejecutando concurrentemente en un sólo procesador. Uno ejecuta `sh` y el otro `ls`.

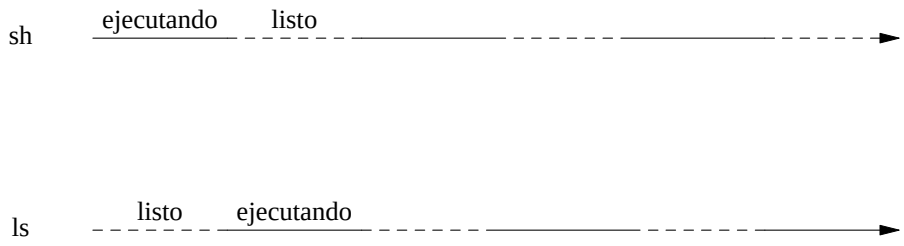


**Figura 4:** *Diagrama de planificación. El tiempo fluye hacia la derecha en la figura.*

### 3. Carga de programas

Vimos que cuando un programa se compila y se enlaza obtenemos un fichero ejecutable. Este fichero contiene toda la información necesaria para que el Sistema Operativo pueda ponerlo a ejecutar (convertirlo en un proceso). Hay diferentes partes del binario que contienen tipos diferentes de información (código, datos, etc.) y se denominan *secciones* del ejecutable. Además, el ejecutable suele comenzar con una estructura de datos denominada cabecera que incluye información sobre qué secciones están presentes en el ejecutable.

Concretamente, dónde empiezan en el fichero, de qué tipo son y qué tamaño tienen.

Una sección del fichero contiene el texto del programa (el código). Las variables globales inicializadas están en otra sección. O, con más precisión, los bytes que corresponden a los datos inicializados están en dicha sección. En realidad el sistema no sabe *nada* respecto a qué variables hay en tu programa o qué significan dichas variables. Simplemente se guardan en el ejecutable los bytes que, una vez cargados en la memoria, dan el valor inicial a las variables que utilizas. Para variables sin inicializar, sólo es preciso guardar en la cabecera del fichero ejecutable qué tamaño en bytes es preciso para dichas variables (y a partir de qué dirección estarán cargadas en la memoria). Dado que no tienen valor inicial, o mejor dicho, dado que su valor inicial es cero (todos los bytes a cero), no es preciso guardar los ceros en el ejecutable. Con indicar el tamaño de dicha sección de memoria basta.

Habitualmente, el ejecutable contiene también una tabla de símbolos con información para depuradores y programas como *nm(1)* que indica los nombres de los símbolos en el fuente, y los nombres de los ficheros fuente y números de línea. Esta información no es para el sistema operativo, es para ti y para el depurador. Para el sistema, tu programa no tiene ningún significado en particular. Sólo tu código sabe el significado de los bytes en los datos que manipula (si son enteros que hay que sumar, o qué hay que hacer con ellos).

Ya vimos como utilizar *nm* para mostrar la información de ficheros objeto y ejecutables. Consideremos ahora el programa del siguiente listado.

**`[global.c]`**:

```
#include <stdlib.h>

char global[1 * 1024 * 1024];
int global2;
int init1 = 4;
int init2 = 3;
static int stint1;
static int stint2 = 3;

int
main(int argc, char*argv[])
{
        exit(0);
}
```

Este programa tiene declaradas varias variables globales. Por un lado, `global` y `global2` son variables globales sin inicializar. Igualmente, `stint1` es una global sin inicializar, aunque declarada *static* (con lo que no se exporta desde el fichero objeto y sólo puede usarse desde el fichero que estamos compilando). Además, tenemos variables globales llamadas `init1`, `init2` y `stinit2` que están inicializadas. Si utilizamos *nm* en el ejecutable resultante de compilar y enlazar este programa vemos lo siguiente:

```
unix$ cc global.c
unix$ nm -n a.out
00000a30 T __start
00000a30 T _start
00000c30 T atexit
00000ca4 T main
...
00201008 D __progname
00201010 D __dso_handle
00201020 D init1
00201024 D init2
00201028 d stinit2
...
00401320 A __bss_start
00401320 A _edata
00401360 b stint1
00401380 B _dl_skipnum
004014a0 B global2
004014c0 B global
005014c0 A _end
crun%
```

El comando *nm* nos informa de las direcciones en que podremos encontrar, una vez cargado en la memoria, cada uno de los símbolos del ejecutable. Como puedes vers, el código del texto del programa (las instrucciones) estarán a partir de la dirección `a30` de memoria. Esos símbolos se muestran con la letra "`T`" en la salida de *nm* para indicar que son de texto.

Las variables `init1`, `init2`, y `stinit2` estarán a partir de la dirección `201020` (en hexadecimal). Puedes ver como cuatro bytes después de `init1`, en la dirección `201024`, estará `init2`. Dado que `init1` es un *int* y que en este sistema (y con este compilador) ocupa 4 bytes, eso tiene sentido. Todas estos símbolos se muestran con la letra "`D`" (o "`d`"), indicado que corresponden a datos inicializados. Las letras que muestra *nm* son mayúsculas cuando los símbolos se exportan desde el fichero objeto (pueden usarse desde otros ficheros objeto dentro del mismo ejecutable) y minúscula en caso contrario. Fíjate en `init2` y en `stinit2` en la salida de *nm* y en el código fuente. ¿Ves la diferencia?

Las variables `global2` y `global` se muestran con la letra "`B`", indicando que corresponden a variables sin inicializar. Para estas variables, el valor inicial será cero (todos los bytes a cero). Puedes ver cómo la variable `global` está en la dirección `004014c0` y, un Mbyte después, en la dirección `005014c0` está la siguiente variable.

El código ocupará una zona de memoria contigua y tendrá permisos para permitir la ejecución de instrucciones. Habitualmente, no tendrá permisos para permitir la escritura de esa zona de memoria. El contenido de esta memoria se inicializa leyendo los bytes del fichero ejecutable que corresponden al texto del programa. Los datos inicializados ocuparán otra zona de memoria contigua, con permisos para leer y escribir dicha memoria. Esta memoria se inicializará leyendo del fichero ejecutable los bytes que corresponden al valor inicial de esa zona de memoria. Los datos sin inicializar ocuparán otra zona, inicializada a cero.

Cada una de estas zonas es contigua, tiene una dirección de comienzo y un tamaño, y la memoria que ocupa tiene las mismas propiedades (permisos, cómo se inicializa, etc.). Es precisamente a eso a lo que se denomina *segmento*. Cuando el program ejecute, tendrá un segmento de texto con el código, otro de datos con variables inicializadas, otro denominado *BSS* con datos sin inicializar (que parten con los bytes a cero) y otro de pila. Véase la figura 5. El nombre BSS viene de una instrucción de una máquina que ya no existe que se utilizaba para inicializar memoria a cero (y son las iniciales de *Base Stack Segment*, aunque hoy día no tiene que ver con la pila).

Cuando el sistema ejecute el programa contenido en este fichero ejecutable, cargará en la memoria el código procedente del mismo, y los valores iniciales para variables inicializadas. Igualmente, la memoria que ocupen las variables sin inicializar se inicializará a cero y se creará una pila para que puedan hacerse llamadas a procedimiento. El trozo de código del kernel que se ocupa de todo esto se denomina cargador.

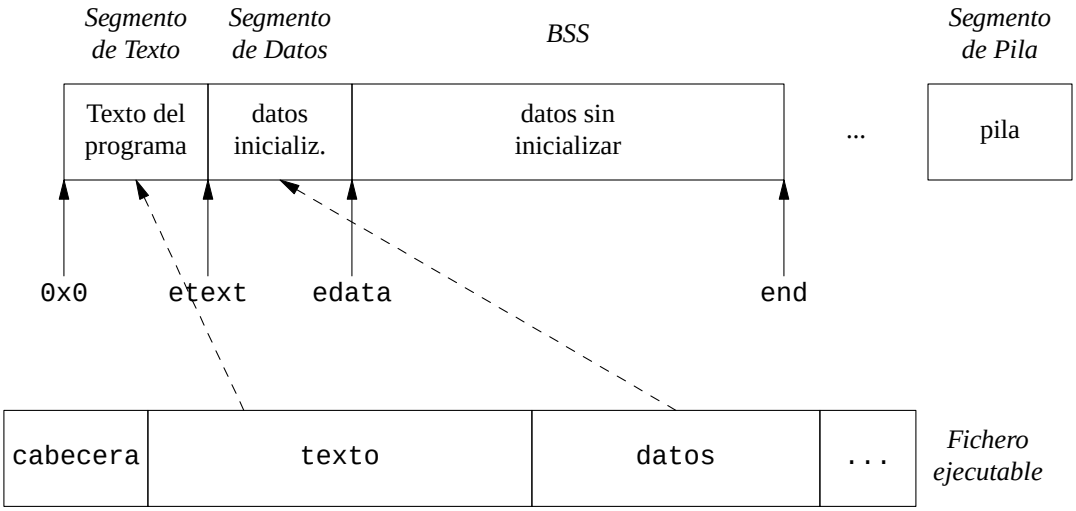


**Figura 5:** *El cargador se ocupa de cargar los segmentos de un proceso con datos procedentes del ejecutable.*

Pero es importante que comprendas que el sistema *no sabe nada* sobre tu programa. Ni siquiera sabe ni qué funciones has definido ni qué variables utilizas. ¡Ni le importa! Si utilizamos el comando *strip(1)* para eliminar la información de la tabla de símbolos del ejecutable, podremos verlo.

```
unix$ ls -l a.out
-rwxr-xr-x  1 nemo  wheel  8729 May 23 14:59 a.out*
unix$ strip a.out
unix$ ls -l a.out
-rwxr-xr-x  1 nemo  wheel  6776 May 23 15:26 a.out*

unix$ nm -n a.out
00000000 W __cxa_atexit
00000cc0 T __fini
00201008 D __data_start
00201008 D __progname
003012a0 D __got_start
00301320 D __got_end
00401320 A __bss_start
00401320 A _edata
005014c0 A _end
```

Tras ejecutar *strip*, *nm* no puede obtener información alguna respecto que qué variables globales o qué funciones están contenidas en el ejecutable. Como hemos compilado con enlace dinámico aún quedan algunos símbolos que son necesarios para que el enlazado dinámico funcione, pero eso es todo. ¡Nadie puede saber que había algo llamado `global` en tu código!

El cargador se limitará a cargar los bytes de texto hacia el segmento de texto y los bytes de datos inicializados hacia el segmento de datos. Lo que signifiquen esos bytes es cosa tuya. En cuanto al BSS, el ejecutable indica en la cabecera cuantos bytes son necesarios, pero naturalmente no guardamos 1Mbyte de ceros en el ejecutable para nuestra variable `global`. Simplemente se indica en el ejecutable en número total de bytes a cero y la dirección en que deben comenzar.

Volvamos a pensar en los segmentos de un proceso. En la figura 5 puedes ver que el enlazador suele dejar en el ejecutable un símbolo llamado `etext` cuya dirección es el final de texto, otro `edata` cuya dirección es el final de los datos inicializados y otro `end` cuya dirección es el final del BSS. Con estos símbolos puedes saber qué direcciones utilizan tus principales segmentos. El comando *size(1)* muestra el tamaño que tendrán los segmentos de un programa cuando lo ejecutes:

```
unix$ size a.out
text    data    bss     dec     hex
2845    500     1048992 1052337 100eb1
```

Para nuestro programa, el BSS tendrá `1048992` bytes.

Los segmentos son parte de la abstracción que te da UNIX para que tengas procesos. Como habrás observado, aunque ejecutemos muchos procesos, todos ellos piensan que tienen la memoria para ellos solos. Por ejemplo, en nuestro programa la variable `global` estaría en la dirección `004014c0`. Si lo ejecutamos varias veces de forma simultánea, cada uno de los procesos tendrá su propia versión y pensará que es el único en el mundo. En todos ellos, `global` estará en la dirección `004014c0`.

Simplemente, la memoria del proceso es memoria virtual. Esta memoria no existe en realidad. Por eso se la denomina virtual. El sistema utiliza el hardware de traducción de direcciones (la unidad de gestión de memoria o MMU) para hacer que las direcciones que utiliza el procesador se cambien por otras *al vuelo* antes de que el hardware de memoria las vea. Cada proceso utilizará zonas distintas de memoria física y, gracias a la traducción de direcciones, la dirección `004014c0` de cada proceso se cambia por la dirección de memoria física que dicho proceso utilice. Pero todos ejecutan instrucciones que acceden a `global` a partir de la dirección `004014c0`.

Las direcciones de memoria virtual se traducen a direcciones de memoria física empleando una tabla (que inicializa para cada proceso el sistema operativo) llamada *tabla de páginas*. Esta tabla traduce de 4 en 4 Kbytes (o un tamaño similar) las direcciones. A cada trozo de 4 Kbytes de memoria virtual lo llamamos página y al trozo correspondiente de memoria física lo denominamos marco de página.

La primera página de memoria suele dejarse sin traducción, lo que hace que sea imposible utilizar sus direcciones sin ocasionar un error (similar al que se produciría si divides por cero). Este error o excepción se atiende de forma similar a una interrupción y, habitualmente, el sistema operativo detiene la ejecución del programa que lo ha causado. Por eso no puedes atravesar un puntero a *nil*. La dirección de memoria cero es inválida, y *nil* es el valor cero utilizado como dirección de memoria.

Todo esto es importante puesto que tiene impacto en tus programas. Habitualmente el sistema carga tu código y datos, y te asigna memoria física, conforme utilizas direcciones de memoria virtual en tus segmentos. Inicialmente no habrá traducciones a memoria física para casi nada en tu proceso y, conforme el procesador utilice direcciones, el sistema irá cargando en demanda el resto. A esto se lo denomina *paginación en demanda*.

La paginación en demanda es fácil de implementar puesto que el sistema sabe qué segmentos tiene el proceso y qué direcciones usan estos. Además, sabe cómo inicializar la memoria de dichos segmentos (leyendo valores iniciales del fichero ejecutable o inicializándola a cero, por ejemplo). Inicialmente, un segmento puede tener sus páginas sin traducción a (marcos de página en) memoria física. Cuando el procesador utiliza una dirección en dichas páginas, el hardware eleva una excepción y el manejador instalado por el sistema operativo la atiende. Dicho manejador comprueba que en teoría el proceso debería poder utilizar dicha página de memoria y asigna un marco de página (poniendo una traducción en la tabla de páginas) inicializado según corresponda. Cuando el manejador retorna, el programa de usuario continúa su ejecución como si nunca hubiese sucedido el fallo de página.

Por ejemplo, `global` está inicializado a cero porque las páginas en que reside serán inicializadas a cero por el sistema cuando las asigne. Y será así dado que dichas páginas forman parte del BSS. Si tu proceso

lee una posición de `global` por primera vez, es muy posible que al proceso se le asignen 4Kbytes de memoria física para la página a que accede, inicializados a cero. Y por eso tu proceso creerá que `global` está inicializado a cero. Antes de utilizar el array, el MByte es sólo memoria virtual. ¡No consume memoria física! Eso si, si se te ocurre la "buena" idea de utilizar un bucle para inicializar tu array a cero, tu proceso accederá a todas esas direcciones y el sistema le asignará la memoria correspondiente. La memoria virtual es gratis, la memoria física no. Es muy habitual utilizar arrays de gran tamaño sabiendo que sólo van a utilizarse unas pocas posiciones en tiempo de ejecución, por ejemplo para grandes tablas hash y para otros propósitos en algunos run-times de lenguajes de programación. ¿Comprendes por qué puede hacerse esto?

Sucede lo mismo con la pila hoy día. Hace tiempo, el segmento de pila solía crecer a medida que el proceso utilizaba más espacio en la misma. Hoy día, el segmento de pila suele tener un tamaño prefijado y no crece. Dado que la memoria que usa es *virtual*, inicialmente dicha memoria es gratis: no tiene asignada memoria física. Naturalmente, esto es así hasta que dicha memoria se usa a medida que crece la pila.

## 4. Nacimiento

A los subprogramas se los llama y retornan, pero si consideramos un programa que queremos ejecutar en un proceso, no hay *llamadas* a programas ni dichos programas *retornan*. Un proceso simplemente termina cuando le pide al sistema terminar o cuando se comporta mal (por ej., intenta leer una dirección de memoria que no está en ningún segmento) y el sistema lo mata. No obstante, como sabes, cuando ejecutamos comandos en el shell podemos indicar argumentos para que sus programas los procesen y para controlar lo que hacen.

Cuando el shell el pide al sistema que ejecute un programa, una vez que dicho programa está cargado en memoria, el sistema suministra un flujo de control para que ejecute. En realidad sabes que lo habitual es que *no* se cargue todo el programa en memoria y, en cambio, se pagina en demanda aquellas páginas de memoria a las que accede el programa conforme ejecuta. Pues bien, el flujo de control supone valores para los registros del procesador, inicializados para que comiencen la ejecución del programa que se desea ejecutar e incluyendo un contador de programa y un puntero de pila. La pila inicialmente estará prácticamente vacía salvo por los argumentos del programa principal. Cuando compilamos un programa en C, el enlazador se ocupa de indicar en el ejecutable que la dirección de `main` es la dirección en la que hay que empezar a ejecutar código. Es por eso que los programas en C comienzan ejecutando `main`. Los argumentos de `main` son un array de strings (en `argv`) y el número de strings que hay en dicho array (en `argc`).

Teniendo esto en cuenta, el siguiente programa escribe sus argumentos indicando antes de cada uno el índice en `argv`:

```
#include <stdlib.h>
#include <stdio.h>
int
main(int argc, char* argv[])
{
    int i;

    for(i = 0; i < argc; i++) {
        printf("%d\t%s\n", i, argv[i]);
    }
    exit(0);
}
```

Si lo guardamos en el fichero `eco.c` y lo compilamos y lo ejecutamos podemos ver qué argumentos recibe el programa para una línea de comandos dada:

```
unix$ cc eco.c
unix$ ./eco un programa sencillo
0:    ./eco
1:    un
2:    programa
3:    sencillo
unix$
```

Como verás, ¡el primer "argumento" en `argv` es en realidad el nombre del programa! Concretamente, es el nombre del programa tal y como lo hemos escrito en la línea de comandos del shell. Recuerda que "`./eco`" es un camino o path relativo y que "`.`" es el directorio actual. Así pues, `argv[0]` contiene el nombre del programa. Los siguientes strings en `argv` son los argumentos que hemos dado a `eco` en la línea de comandos. Es muy útil emplear `argv[0]` en mensajes de error para identificar ante el usuario al programa que tiene problemas.

Lo que ha sucedido es que el shell ha leído la línea de comandos que hemos escrito, y ha utilizado la primera palabra para localizar el fichero que contiene el programa que queremos ejecutar. Después, le ha pedido a UNIX que lo ejecute y que haga que `main` en dicho programa reciba una copia del array de strings que corresponde a la línea de comandos.

Ya conocemos *echo(1)*. Recuerda que dicho comando acepta la opción "`-n`" para suprimir la impresión del fin de línea tras imprimir sus argumentos. Podemos cambiar el programa anterior para que sea como *echo(1)*, y podemos hacer que la opción "`-v`" imprima cada argumento entre corchetes, para distinguirlos mejor. Este es el programa:

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

static char *argv0;

static void
usage(void)
{
    fprintf(stderr, "usage: %s [-nv] [args...]\n", argv0);
    exit(1);
}
```

```
int
main(int argc, char* argv[])
{
    int i, nflag, vflag;
    char *arg, *sep;

    nflag = vflag = 0;
    argv0 = argv[0];
    argv++;
    argc--;
    while(argc > 0 && argv[0][0] == '-') {
        if (strcmp(argv[0], "--") == 0) {
            argv++;
            argc--;
            break;
        }
        for(arg = argv[0]+1; *arg != 0; arg++) {
            switch(*arg) {
            case 'n':
                nflag = 1;
                break;
            case 'v':
                vflag = 1;
                break;
            default:
                usage();
            }
        }
        argv++;
        argc--;
    }
    sep = "";
    for(i = 0; i < argc; i++){
        if (vflag) {
            printf("%s[%s]", sep, argv[i]);
        } else {
            printf("%s%s", sep, argv[i]);
        }
        sep = " ";
    }
    if (!nflag) {
        printf("\n");
    }
    exit(0);
}
```

Primero guardamos en la global `argv0` el nombre del programa. Esto lo usamos en la función `usage` para imprimir un recordatorio del uso del programa empleando el nombre tal y como lo hemos recibido en `argv[0]`. Una vez hecho esto, quitamos del vector del argumentos el primer string haciendo que `argv` apunte al siguiente elemento (y actualizando `argc` para que refleje el número de strings en `argv`).

El bucle `while` procesa las opciones (argumentos que empiezan por "`-`") para detectar si se ha indicado "`-n`" o "`-v`". Como verás, el programa también entiende dichas opciones si se suministra "`-nv`" o "`-vn`", como suelen hacer los programas en UNIX. Además, un argumento "`--`" indica el fin de las opciones, para

hacer que el resto de argumentos no se procesen como opciones. Esto es lo habitual en UNIX. Por ejemplo, podemos ejecutar el programa como sigue, si lo tenemos compilado y enlazado en el ejecutable "`eco2`".

```
unix$ eco2 -v -- -n hola
[-n] [hola]
unix$
```

Hemos optado por procesar los argumentos de tal forma que una vez procesadas todas las opciones, `argc` y `argv` contienen el resto de argumentos. Esto es cómodo en general.

Si llamamos al programa de forma incorrecta, obtenemos un recordatorio de cómo hay que usarlo:

```
unix$ eco2 -vx hola
usage: eco2 [-nv] [args...]
unix$
```

## 5. Muerte

Habrás notado que "`main`" termina llamando a `exit`. Esta llamada al sistema pide a UNIX que termine la ejecución del proceso (y por tanto de su programa). El entero que pasamos como argumento se espera que sea cero si el programa ha conseguido hacer su trabajo y distinto de cero en caso contrario. A este valor se le suele llamar **exit status** (en inglés). Todo esto es muy importante. Habitualmente es un programa el que ejecuta otros programas (por ejemplo el shell). Si dichos programas no informan correctamente respecto a si pudieron hacer su trabajo o no, el programa que los ejecuta podría hacer cosas que no esperamos.

Si un programa no comprueba si sus argumentos son correctos y/o no suministra el estatus de salida adecuado (cero o distinto de cero), no está correctamente programado y es posible que no pueda usarse en la práctica.

Habrás visto que `main` retorna un entero. Dado que *exit(3)* termina la ejecución, no retornamos nada en nuestro programa. No obstante, retornar de `main` hace que el valor retornado se utilice para llamar a *exit(3)* con dicho valor. Esto es, podríamos haber terminado nuestro programa escribiendo

```
int
main(int argc, char* argv[])
{
    ...
    return 0;
}
```

No hay magia en esto. En realidad el programa principal es una función que hace algo equivalente a

```
exit(main(argc, argv));
```

lo que explica que podamos retornar de `main`.

Podemos utilizar el shell para ver que tal le fue a un comando que ejecutamos anteriormente:

```
unix$ ls /blah
ls: /blah: No such file or directory
unix$ echo $?
1
unix$ echo $?
0
```

Aquí, "`$?`" es una *variable de entorno* (más adelante veremos lo que es esto) que contiene como valor un string correspondiente al estatus de salida del último comando ejecutado. ¿Puedes decir por qué la segunda vez el estatus es 0?

### 6. En la salida

En ocasiones resulta útil ejecutar código cuando un programa termina. Aunque no se debe abusar de este mecanismo, puede resultar útil para, por ejemplo, borrar ficheros temporales o realizar alguna otra tarea incluso si una función decide llamar a `exit`.

La función *atexit(3)* permite instalar punteros a función de tal forma que dichas funciones ejecuten durante la llamada a *exit(3)*. En realidad, `exit` llama las funciones que ha instalado `atexit` y luego llama a `_exit`, que realmente termina la ejecución del proceso.

Recuerda que el programa principal es el realidad

```
exit(main(argc, argv));
```

La función de librería `exit` se parece a...

```
void
exit(int sts)
{
    int i;
    for (i = 0; i < numexitfns; i++) {
        exitfns[i]();
    }
    _exit(sts);
}
```

Para ejecutar algo cuando el programa termine llamando a *exit*, podemos llamar a *atexit* como en el siguiente programa:

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

static void
exitfn1(void)
{
    puts("dentro de exitfn1");
}

static void
exitfn2(void)
{
    puts("dentro de exitfn2");
}
```

```
int
main(int argc, char* argv[])
{
    atexit(exitfn1);
    atexit(exitfn2);
    puts("a punto de salir...");
    return 0;
}
```

Cuando lo ejecutemos, suponiendo que el ejecutable es `atexit`, veremos algo como...

```
unix$ atexit
a punto de salir...
dentro de exitfn2
dentro de exitfn1
```

Como verás, el kernel de UNIX no sabe nada de nada de tus *atexit*. Simplemente la función de librería *exit(3)* se ocupa de hacer los honores. ¿Qué pasaría si llamas a *exit* desde una función instalada con *atexit*? ¡Pruébalo! ¿Entiendes por qué?

## 7. Errores

En el programa que hemos hecho y en los siguientes vamos a hacer muchas llamadas al sistema. Bueno, en realidad muchas serán llamadas a la librería de C y otras serán llamadas al sistema. Nos da un poco igual (salvo por la sección del manual en que están documentadas, que será la 2 para llamadas al sistema y la 3 para llamadas a la librería de C). En cualquier caso, son llamadas para utilizar UNIX.

En muchos casos no habrá problema en la llamada que hagamos podrá hacer su trabajo. Pero en otros casos no será así. O bien habremos utilizado argumentos incorrectos en la llamada, o bien tendremos un problema de permisos o de otro tipo. Por ejemplo, cada proceso tiene un directorio de trabajo (como sabes) y podemos utilizar la llamada *chdir(2)* para cambiarlo. ¡Pero es posible que intentemos cambiar a un directorio que no existe!

Todas las llamadas devuelven un valor que, entre otras cosas, nos indicará al menos si han podido o no hacer su trabajo. Es responsabilidad nuestra comprobar dicho valor y actuar en consecuencia. Habitualmente el valor devuelto por una función suele ser un valor absurdo en el caso de errores. Si la función devuelve un puntero, devolverá `NULL`, si devuelve un entero positivo, devolverá `-1`, etc. La tradición en UNIX es que si el valor devuelto es sólo para indicar un error, suele devolverse `-1` en caso de error y `0` en caso de éxito. Las páginas de manual de cada función indican qué se devuelve en caso de error.

Cuando programes en C deberías seguir el mismo convenio. Todas tus funciones deberían informar al que las llama de si hubo algún error o no (salvo en aquellos casos en que nunca pueda producirse un error y en aquellos casos en que si se produce un error la función termine la ejecución de todo el programa).

Debes **siempre comprobar si hubo errores** en tus programas, en todas las llamadas. Si no lo haces, el programa será un poltergueist. ¿Y qué debería hacerse cuando hubo un error? No hay una respuesta correcta a esta pregunta. Depende de lo que estés programando y de qué función tenga el error. Simplemente piensa qué te gustaría que pasara si en una llamada sufres un error. Imagina que el programa lo has tomado prestado y piensa en lo que crees que debería hacer en cada caso.

Una vez has comprobado que una llamada no ha podido hacer su trabajo (¡eso es lo que significa que sufrió un error!, los errores no son magia), deberías ver a qué se debió el error. En UNIX, puedes utilizar la global `errno` para ver el código del error (un entero) y puedes utilizar *strerror(3)* para obtener un string correspondiente a dicho código.

Por ejemplo, este programa intenta cambiar de directorio:

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>

int
main(int argc, char* argv[])
{
    if (chdir("foo") < 0){
        printf("errno is %d\n", errno);
        printf("err string is '%s'\n", strerror(errno));
        exit(1);
    }
    /* ... do other things ... */
    exit(0);
}
```

Suponiendo que el ejecutable es `err` y que no existe el directorio `foo`, cuando lo ejecutamos...

```
unix$ err
errno is 2
err string is 'No such file or directory'
unix$
```

En la práctica, deberíamos imprimir el nombre del programa, lo que intentábamos hacer cuando hemos sufrido el error, y cuál es la causa:

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>

int
main(int argc, char* argv[])
{
    if (chdir("foo") < 0){
        fprintf(stderr, "%s: chdir: foo: %s\n", argv[0], strerror(errno));
        exit(1);
    }
    /* ... do other things ... */
    exit(0);
}
```

Si ahora ejecutamos el programa y este no puede cambiar su directorio, obtenemos un mensaje que ayudará a resolver el problema:

```
unix$ err
err: chdir: foo: No such file or directory
unix$
```

Esto es tan habitual, que la función *warn(3)* hace justo eso. Por ejemplo:

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <err.h>

int
main(int argc, char* argv[])
{
    if (chdir("foo") < 0){
        warn("chdir: %s", "foo");
        exit(1);
    }
    /* ... do other things ... */
    exit(0);
}
```

Y ahora tenemos...

```
unix$ err
err: chdir: foo: No such file or directory
unix$
```

Como verás, `warn` se usa de un modo similar a *printf(3)* e imprime un mensaje de error indicando el nombre del programa y el string de error correspondiente a `errno`. Si has leído la página de manual *warn(3)* (¡Cosa que deberías haber hecho en este punto!) habrás visto que la función *err(3)* es similar a `warn` pero llama a `exit` tras imprimir el mensaje con el estatus que le pasas en el primer argumento.

Un último comentario. No tiene sentido utilizar `errno` si no es justo después de una llamada que ha fallado (y ha indicado su error con el valor de retorno). Sólo cuando una llamada a UNIX tiene un problema, dicha llamada actualiza `errno` para informar de la causa del error.

### 8. Variables de entorno

Otra forma de darle información a un proceso (además de usando sus argumentos) es utilizar las llamadas **variables de entorno**. Cada proceso tiene un array de strings de tal forma que cada string tiene el aspecto

"*nombre=valor*"

y define una variable de entorno con nombre "*nombre*" y valor "*valor*".

Cuando UNIX inicializa la memoria para un nuevo programa se ocupa de que dicho array esté inicializado con las variables de entorno que se han indicado en la llamada al sistema utilizada para ejecutar un nuevo programa. Además de `argv`, ahora tienes que considerar que tienes un array de variables de entorno.

El propósito de estas variables es definir elementos que se desea que estén disponibles para los programas que ejecutas sin tener que pasar dichos elementos como argumento todas las veces. Naturalmente, tanto el nombre de las variables como su valor son strings, dado que esta abstracción (variables de entorno) consiste en el array de strings que hemos descrito.

Podemos definir variables de entorno utilizando el shell. Por ejemplo:

```
unix$ v=33
```

Cuando el shell ve que una palabra en una línea de comandos comienza por un "$", entiende que se desea

utilizar el valor de la variable de entorno cuyo nombre sigue y cambia dicha palabra por el valor de la variable (¡que siempre es un string!).

Por ejemplo, ya conoces *echo(1)* y sabes que simplemente escribe sus argumentos tal cual los recibe en `argv`, sin ningún tipo de magia. Pues bien, mira lo que escribe *echo*:

```
unix$ v=33
unix$ echo $v
33
unix$
```

Te resultará útil definir variables para nombres que utilices a menudo. Por ejemplo, si sueles cambiar de directorio a un directorio dado, puedes utilizar una variable de entorno para escribir menos:

```
unix$ d=/un/path/muy/largo
unix$ cd $d
unix$
```

Es tan habitual utilizar el directorio *home* de un usuario, que cuando haces un *login* se define una variable de entorno `HOME` que contiene el path de tu directorio *home*. Así pues:

```
unix$ cd $HOME
unix$ pwd
/home/nemo
unix$ cd
unix$ pwd
/home/nemo
```

El shell expande variables de entorno (reemplaza `$x` por el valor de la variable `x`) en cualquier sitio de la línea de comandos. Fíjate en esto:

```
unix$ p=pwd
unix$ $p
/home/nemo
unix$
```

Pero, si intentamos utilizar este truco para ejecutar "`ls -l`" nos llevamos una sorpresa...

```
unix$ l=ls -l
sh: -l: command not found
```

Necesitamos escribir *una sóla palabra* tras el operador "`=`". Esta es la sintaxis del shell para definir variables. Como hay un espacio entre "`ls`" y "`-l`", el shell ha intentado ejecutar "`-l`" como uno comando.

¿Y si probamos...?

```
unix$ l=ls-l
unix$
```

¡Ha funcionado!, ¿O no?

```
unix$ $l
sh: ls-l: command not found
unix$
```

¿Comprendes lo que ha sucedido? Seguro que sí, y, si no es así, piensa... ¿Cuál es el nombre del comando

que has ejecutado?

La solución a nuestro problema es utilizar sintaxis de shell para indicarle que cierto texto que escribimos en la línea de comandos ha de entenderse como una sóla palabra. Veámoslo:

```
unix$ l='ls -l'
unix$
```

Y ahora podemos...

```
unix$ $l
total 240
drwxr-xr-x   2 nemo  staff   1156 Jul 25 12:36 zxbib
drwxr-xr-x   5 nemo  staff   1734 Jul 25 13:01 zxdoc
...
```

Vamos a utilizar nuestro programa `eco2` para ver qué argumentos pasa el shell cuando utilizamos las comillas.

```
unix$ eco2 -v 'ls -l' ls -l
[ls -l] [ls] [-l]
```

Lo escrito entre comillas simples está en un sólo string en el vector `argv` de `eco2`. Esto es, el shell ha tomado literalmente lo que hay entre comillas simples como una sóla palabra. Ya lo sabías si recuerdas el primer tema de este curso.

También podemos utilizar comillas dobles para hacer que el shell tome su contenido como una sóla palabra. La diferencia entre estas y las simples radica en que el shell, en el caso de las comillas dobles, expande variables de entorno dentro de ellas. Por ejemplo:

```
unix$ cmd=ls
unix$ arg=-l
unix$ line="$cmd $arg"
unix$ $line
total 240
drwxr-xr-x   2 nemo  staff   1156 Jul 25 12:36 zxbib
drwxr-xr-x   5 nemo  staff   1734 Jul 25 13:01 zxdoc
...
```

En cambio...

```
unix$ cmd=ls
unix$ arg=-l
unix$ line='$cmd $arg'
unix$ $line
sh: $cmd: command not found
unix$ echo $line
$cmd $arg
```

En un programa en C podemos consultar variables de entorno llamando a *getenv(3)*. Por ejemplo, este programa cambia su directorio actual al directorio casa e imprime el path de dicho directorio antes de continuar con su trabajo.

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

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

    home = getenv("HOME");
    if (home == NULL){
        fprintf(stderr, "we are homeless\n");
        exit(1);
    }
    if (chdir(home) < 0) {
        err(1, "can't cd to home");
    }
    printf("working at home: %s\n", home);

    // ... do some other things...

    exit(0);
}
```

Si lo tenemos compilado en `env` y lo ejecutamos, podemos ver lo que hace...

```
unix$ env
working at home: /home/nemo
unix$
```

Si la variable no está definida, `getenv` devuelve `NULL`. Esto no es un error, y `errno` no se actualizará en tal caso. Simplemente la variable puede no estar definida. O dicho de otro modo... puede que ningún string en tu vector de variables de entorno comience por "`HOME=`" (aunque no es lo que uno espera en UNIX para la variable `HOME`).

¿Qué haría el programa si lo cambiamos para que ejecute

```
home = getenv("$HOME");
```

en lugar de hacer que hacía antes? ¡Pruébalo! ¿Puedes ver por qué? ¿Cómo se llama la variable? Recuerda que "`$HOME`" es sintaxis del shell. ¿Ejecuta el shell en algún caso cuando tu programa en C ejecuta la línea anterior? ¡Desde luego que no!

¿Y qué haría si lo cambiamos para que utilice

```
if (chdir("$HOME") < 0) {
    err(1, "can't cd to home");
}
```

o para que utilice

```
if (chdir("HOME") < 0) {
    err(1, "can't cd to home");
}
```

en lugar de lo que teníamos antes?

Para definir una variable de entorno en el proceso en que ejecutamos un programa en C podemos utilizar la

función *putenv(3)* o *setenv(3)*. Ambas están descritas en *getenv(3)*, así que si has seguido con la buena costumbre de leer la página de manual de cada llamada la primera vez que la usas, ya las conoces. Esto es un ejemplo en cualquier caso:

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

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

    putenv("tempdir=/tmp");
    temp = getenv("tempdir");
    if (temp == NULL){
        fprintf(stderr, "no temp dir\n");
        exit(1);
    }
    printf("temp dir is: %s\n", temp);

    exit(0);
}
```

Y esta es su ejecución:

```
unix$ penv
temp dir is: /tmp
```

Naturalmente, sería absurdo utilizar la variable de entorno `tempdir` como lo hemos hecho si no pensamos ejecutar nuevos procesos desde nuestro programa en C. Si tan sólo queremos definir una variable para guardar el path de un directorio temporal, C ya tiene variables.

Hasta ahora todo bien. Pero... ¿Puedes explicar esto?

```
unix$ penv
temp dir is: /tmp
unix$ echo $tempdir
unix$
```

Resulta que tras ejecutar nuestro programa en C que define una variable de entorno, ¡el shell no la tiene definida! Pero esto es normal. Piensa que la variable la define el proceso que ejecuta el programa en C, y que el shell es otro proceso. Cada proceso tiene sus propias variables de entorno. ¿Lo ves normal ahora?

Cuando no entiendas algo, piensa que no hay magia y piensa en qué programas intervienen en lo que estás haciendo y qué hace cada uno paso a paso. Si sigues sin poder explicar el resultado es que te estás perdiendo algo respecto a lo que hace cada programa. Recurre al manual y haz programas de prueba que intenten explicar tus hipótesis respecto a qué está pasando.

En relación con esto, resulta interesante mirar lo que sucede en este otro caso:

```
unix$ x=foo
unix$ echo $x
foo
unix$ sh
unix$ echo $x

unix$ exit
unix$ echo $x
foo
unix$
```

Hemos ejecutado un shell (ejecutando el comando `sh`) y dicho shell no tiene definida la variable `x` (`$x` no tiene nada). Cuando terminamos dicho shell con `exit` volvemos al shell original y ahí si que tenemos la variable de entorno definida.

Es como si las variables que defines fuesen locales al shell y los comandos que ejecutamos desde ese shell ya no las tienen. Vamos a comprobarlo con un programa en C:

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

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

    x = getenv("x");
    if (x == NULL){
        fprintf(stderr, "no x\n");
        exit(1);
    }
    printf("%s\n", x);

    exit(0);
}
```

Si lo compilamos y dejamos el ejecutable en `penvx` podemos ver lo que sucede:

```
unix$ x=foo
unix$ penvx
no x
unix$
```

Efectivamente, el shell no ha pedido a UNIX que ejecute `penvx` con la variable `x` definida en el entorno. Pero hay formas de hacer que lo haga:

```
unix$ export x
unix$ penvx
foo
unix$
```

El comando `export` es un comando construido dentro del shell. Es una *primitiva* o *builtin* del shell. Su propósito es indicarle al shell que *exporte* una o más variables de entorno a los procesos que ejecute dicho shell para ejecutar nuevos comandos.

Hoy día suele ser habitual hacer ambas cosas a la vez:

```
unix$ export x=foo
unix$ penvx
foo
unix$
```

## 9. Procesos y nombres

Ya sabes que un proceso es tan sólo una abstracción que implementa el kernel del sistema operativo. Dentro del kernel tenemos un array de records de tal forma que cada record será de tipo `Proc` (o cualquier otro nombre) y definirá el tipo de datos *proceso*.

Como te imaginarás en este punto, cada uno de esos records contiene el path para el directorio actual en que ejecuta el proceso y cualquier otra cosa que UNIX necesite recordar sobre ese proceso. Por ejemplo, qué segmentos de memoria utiliza. Y, naturalmente, los segmentos son también una abstracción y serán tan sólo records que contienen la información que UNIX necesite saber sobre cada uno de ellos. Así de simple.

El nombre de un proceso es parte de la abstracción *proceso* en UNIX. Pero dicho nombre *no* es el nombre del programa que está ejecutando (lo que sería `argv[0]` en la función `main` de dicho programa). El nombre de un proceso es un número entero que identifica el proceso y se conoce como **identificador de proceso** o *process id* o *pid*.

Cuando UNIX crea un proceso le asigna un *pid* que ningún otro proceso ha utilizado antes. Es como un "DNI" para el proceso. Todas las llamadas al sistema que necesitan que indiques sobre qué proceso deben operar reciben un *pid* para que nombres el proceso sobre el que han de trabajar. Pero recuerda que hay muchas llamadas que operan sobre el proceso que hace la llamada y, como es natural, no necesitan que indiques sobre qué proceso han de actuar.

Por ejemplo, cuando un programa llama a `chdir`, UNIX sabe qué proceso está haciendo la llamada (puesto que ha sido UNIX el que lo puso a ejecutar), y dicha llamada opera sobre el proceso en cuestión.

El siguiente programa utiliza la llamada al sistema *getpid(2)* para averiguar el *pid* del proceso que ejecuta el programa.

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

int
main(int argc, char* argv[])
{
    pid_t p;

    p = getpid();
    printf("pid is %d\n", p);
    exit(0);
}
```

El tipo `pid_t` es tan solo un tipo de `int`, no hay magia en eso tampoco. Vamos a ejecutar el programa varias veces, tras compilarlo en `pid`:

```
unix$ pid
pid is 91795
unix$ pid
pid is 91800
unix$ pid
pid is 91805
unix$
```

Y cada vez tendrá un nuevo *pid*. En un mismo programa, pero los procesos son distintos.

El comando *ps(1)* lista los procesos que están ejecutando

```
unix$ ps
  PID TT  STAT       TIME COMMAND
15891 p0  Ss      0:00.11 -ksh (ksh)
14610 p0  R+      0:00.00 ps
12349 C0- I       0:00.00 acme
unix$
```

La salida varía mucho de un tipo de UNIX a otro. En este caso el shell que ejecutamos no es `sh`, sino `ksh` y tenemos un editor en el programa `acme` que está ejecutando. Además, podemos ver cómo `ps` está ejecutando también.

Habitualmente `ps` lista sólo los procesos que ejecutan a nombre del mismo usuario que ejecuta `ps`. Pero `ps` tiene muchas opciones y te permite listar todos los procesos que ejecutan en la máquina así como ver muchas otras cosas sobre cada proceso (cuánta memoria virtual está utilizando cada proceso, cuánta memoria real, qué tiempo ha consumido de CPU, cuánto hace que está ejecutando, etc.). Lo mejor es que utilices el manual y recuerdes que en el caso de `ps` las opciones y las columnas que imprime para cada proceso suelen variar de un tipo de UNIX a otro.

Eso sí, lo normal suele ser que se imprima el *pid* de cada proceso (en la primera columna en la mayoría de los UNIX) y el vector de argumentos para cada proceso (normalmente al final de cada línea).

Por ejemplo, en el UNIX que estamos utilizando (un BSD) podemos utilizar estos flags en `ps` para listar todos los procesos y más información sobre cada uno de ellos:

```
unix$ ps -auxw
USER       PID %CPU %MEM   VSZ   RSS TT  STAT  STARTED       TIME COMMAND
elf      15891  0.0  0.0   680   856 p0  Ss    12:48PM    0:00.11 -ksh (ksh)
elf      27386  0.0  0.0   420   452 p0  R+    12:56PM    0:00.00 ps -auw
elf      12349  0.0  0.0   624   692 C0- I     14Jul16    0:00.00 ksh -c /zx/bin/xcmd -s sh -v
elf      26019  0.0  0.0 17720  5040 C0- I     14Jul16    2:36.19 /zx/bin/xcmd
root      5830  0.0  0.0   492  1148 C0  Is+   14Jul16    0:00.01 /usr/libexec/getty ttyC0
root     15869  0.0  0.0   500  1148 C1  Is+   14Jul16    0:00.00 /usr/libexec/getty ttyC1
...
```

Aquí, `VSZ` es la cantidad de memoria virtual y `RSS` es la cantidad de memoria física en uso. La columna `STAT` describe el estado de planificación del proceso (ya sabes... *ejecutando*, *listo para ejecutar*, *bloqueado*). Mira el manual de *ps* para ver qué significan las cosas en la salida que obtengas en tu UNIX.

### 10. Usuarios

¿A nombre de qué usuario ejecutamos? Veámoslo...

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

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

    uid = getuid();
    printf("uid is %d\n", uid);
    exit(0);
}
```

El nombre de usuario para UNIX es otro entero (como sabes) o *identificador de usuario*, o *uid*. La llamada *getuid(2)* te permite obtener el *uid* del usuario a nombre de quien ejecuta el proceso. Si compilamos el programa en `guid`, podemos ejecutarlo...

```
unix$ guid
uid is 501
unix$
```

Si ejecutamos el comando *id(1)* podemos ver que es correcto:

```
unix$ id
uid=501(nemo) gid=20(staff) groups=20(staff)
unix$
```

En este caso y en este UNIX, mi usuario es el `501`. Igualmente, podemos averiguar a nombre de qué grupo de usuarios está ejecutanto nuestro proceso.

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

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

    gid = getgid();
    printf("gid is %d\n", gid);
    exit(0);
}
```

Si compilamos el programa en `ggid`, podemos ejecutarlo...

```
unix$ ggid
gid is 20
unix$
```

¡Ya sabes! Tanto el *uid* como el *gid* son atributos de cada proceso.

¿Y qué nombre de usuario corresponde a cada *uid*? En realidad, estamos adentrándonos en un campo que varía de un UNIX a otro. En general, las cuentas de usuario se abren editando un fichero llamado `/etc/passwd`, y en dicho fichero se suele incluir una línea para cada usuario que detalla su nombre, *uid*, *gid* para cada grupo al que pertenece el usuario, directorio casa, shell que ejecuta el usuario cuando hace un

login, etc. Del mismo modo, el fichero `/etc/group` suele utilizarse para incluir una línea por cada grupo de usuarios con el nombre del grupo y el *gid* del grupo, al menos.

En la mayoría de los UNIX podemos utilizar *getpwuid* para recuperar desde C un record que describe una entrada en *passwd* para un *uid* dado.

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <pwd.h>
#include <uuid/uuid.h>
#include <err.h>

int
main(int argc, char* argv[])
{
    int uid;
    struct passwd *p;

    uid = getuid();
    p = getpwuid(uid);
    if (p == NULL) {
        err(1, "no passwd for uid");
    }
    printf("user name is %s\n", p->pw_name);
    exit(0);
}
```

Lee *getpwuid(3)* y echa un vistazo a las otras funciones y a los campos de `passwd`. Pero antes vamos a ejecutar este programa:

```
unix$ uidname
user name is nemo
unix$
```

Cualquier usuario puede cambiar su password utilizando *passwd(1)*. Naturalmente el comando *passwd* ejecuta a nombre del usuario que lo ejecuta. La pregunta entonces es... ¿Cómo puede tener *passwd* permisos para cambiar el password? Si tu usuario tiene permisos para editar `/etc/passwd` (o el fichero donde quiera que se guarden los passwords, encriptados) entonces tu usuario tendría acceso a todas las cuentas de usuario del sistema, lo que no es razonable salvo para el superusuario (o *root*, o *uid* `0`).

La respuesta suele ser que el fichero con el ejecutable de *passwd* tiene un permiso puesto, el *set uid bit*, que indica que dicho comando debe ejecutar de forma efectiva a nombre del dueño del fichero, y no a nombre del usuario que ejecuta dicho fichero. Dado que `/bin/passwd` (o el fichero de que se trate) pertenece normalmente a *root*, cualquier usuario puede cambiar su password.

Fíjate en esto:

```
unix$ which passwd
/usr/bin/passwd
unix$ ls -l /usr/bin/passwd
-r-sr-xr-x  1 root  bin  23800 Mar  8  2015 /usr/bin/passwd*
```

El comando *which(1)* imprime el nombre del fichero que corresponde a un comando dado. Como puedes

ver, hay una `s` como permiso de ejecución para el dueño de `passwd`. Eso quiere decir que cualquier usuario que ejecute dicho fichero obtiene un proceso a nombre de *root*.

Podemos poner o quitar ese permiso como cualquier otro, siempre que tengamos permiso para ello:

```
unix$ cc -o eco eco.c
unix$ chmod u+s eco
```

Existe otro bit similar para el grupo, se llama el *set gid bit*, y se puede activar como en...

```
unix$ chmod g+s eco
```

Ahora podemos decir que en realidad los procesos tienen dos *uid* y dos *gid*. Tienen los reales y tienen los *efectivos*. Las llamadas *setuid(2)* y *seteuid(2)* permiten cambiar los *uid* real y efectivo de un proceso, y las llamadas *setgid(2)* y *setegid(2)* permiten cambiar los *gid* real y efectivo.

Como podrás suponer, en realidad son los identificadores efectivos los que se utilizan para comprobar permisos. Los reales suelen utilizarse para saber quién está en realidad ejecutando qué.

El comando *su(1)* (o *switch user*) permite ejecutar un shell a nombre de otro usuario (de *root* si no se indica nombre de usuario) y naturalmente utiliza el mecanismo de *set uid* para conseguirlo. Por ejemplo:

```
unix$ su
Password:
unix# id
uid=0(root) gid=0(wheel) groups=0(wheel),1(daemon)
unix# exit
unix$
```

**11. ¿Qué más tiene un proceso?**

En este punto sabes que los procesos tienen un *pid*, segmentos de memoria, valores para los registros (mientras no ejecutan, ¡cuando ejecutan dichos valores se guardan en los registros de verdad!), un directorio actual, variables de entorno, el identificador de usuario o *uid* a nombre del cual ejecutan, etc.

De aquí en adelante veremos otros recursos que tienen los procesos tales como ficheros abiertos y otros muchos. Piensa siempre que se trata de otros campos en el record que implementa cada proceso e intenta imaginar las estructuras de datos que los implementan. Es una buena forma de que vea que no hay magia por ningún lado.

Pero... ¿Cómo vas a ser capaz de recordar todo esto? ¡No lo hagas! ¡Usa el manual!

**12. Cuando las cosas se tuercen...**

De vez en cuando un proceso ejecutará un programa que tendrá un error y no hará lo que esperamos o hará algo demasiado grave que casuse que UNIX lo mate (por ejemplo, accediendo a una dirección de memoria a la que no tiene permiso para acceder por estar fuera de un segmento o por ser una escritura de un segmento de sólo lectura).

En la mayoría de los errores, incluir algún `printf` para depurar tras pensar en qué puede estar fallando es la mejor herramienta. En otros casos, necesitaremos ayuda para ver qué sucede. Para eso podemos utilizar un depurador. Cuando un programa hace algo que cause que UNIX lo mate, UNIX permite que se haga un volcado del estado de la memoria del proceso a un fichero (y del valor de los registros en el momento de la muerte). Dicho fichero suele llamarse `core` dado que es un volcado de la memoria y, cuando se hizo

UNIX, la memoria podía ser de núcleos (o "cores") de ferrita!). Según una de las personas que hizo UNIX, ¡las cosas empezaron a ir mal cuando los bits dejaron de verse a simple vista!

Vamos a hacer un programa que se comporte mal.

```
#include <stdlib.h>

static void
clearstr(char *p)
{
    p[0] = 0;
}

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

    p = NULL;

    clearstr(p);
    // ... do other things here...
    exit(0);
}
```

Y vamos a ejecutarlo:

```
unix$ bad
Segmentation fault: 11
unix$
```

El programa ha intentado utilizar una dirección de memoria que no tiene y la causado una violación de segmento. Cuando pasa esto, el hardware eleva una excepción al intentar traducir la dirección de memoria y el manejador de la excepción (el kernel) ve que el proceso no debería hacer eso. Como resultado, el kernel termina la ejecución del proceso. En nuestro caso, el shell se ha dado cuenta de lo que ha sucedido y ha impreso un mensaje para informar de ello.

Para conseguir un `core`, hemos de cambiar uno de los límites que tiene el proceso. Concretamente, el límite que indica el tamaño máximo de `core` que queremos obtener. Una vez más, los límites son también atributos del proceso. Desde C puedes usar *setrlimit(2)* para cambiar un límite y *getrlimit(2)* para consultar el valor actual. Desde el shell podemos ver los límites que tenemos con el comando *ulimit(1)*:

```
unix$ ulimit -a
core file size          (blocks, -c) 0
data seg size           (kbytes, -d) unlimited
file size               (blocks, -f) unlimited
max locked memory       (kbytes, -l) unlimited
max memory size         (kbytes, -m) unlimited
open files                      (-n) 256
pipe size            (512 bytes, -p) 1
stack size              (kbytes, -s) 8192
cpu time               (seconds, -t) unlimited
max user processes              (-u) 709
virtual memory          (kbytes, -v) unlimited
```

Y cambiar el tamaño máximo de core como sigue:

```
unix$ ulimit -c unlimited
unix$
```

Ahora podemos ejecutar nuestro programa roto una vez más:

```
unix$ bad
Segmentation fault: 11 (core dumped)
unix$
```

El lugar en que UNIX guarda los ficheros `core` varía de un UNIX a otro. Hace tiempo era el directorio actual del proceso que muere. Hoy día depende mucho del tipo de UNIX. Por ejemplo, en MacOS (OS X), el manual dice...

```
CORE(5)                     BSD File Formats Manual                     CORE(5)

NAME
     core -- memory image file format

SYNOPSIS
     #include <sys/param.h>

DESCRIPTION
     A small number of signals which cause abnormal termination of a process
     also cause a record of the process's in-core state to be written to disk
     for later examination by one of the available debuggers.  (See
     sigaction(2).)  This memory image is written to a file named by default
     core.pid, where pid is the process ID of the process, in the /cores
     directory, provided the terminated process had write permission in the
     directory, and the directory existed.
```

Así pues, en este sistema:

```
unix$ ls -l /cores
total 1234232
-r--------  1 nemo  admin  631926784 Aug 19 15:25 core.92367
```

Donde `92367` era el *pid* del proceso que murió.

Podemos utilizar un depurador para ver un volcado de la pila en el momento de la muerte. Esto suele ser mas que suficiente para avergüar la causa del problema:

```
unix$ lldb -c /cores/core.92367
(lldb) target create --core "/cores/core.92433"
Core file '/cores/core.92433' (x86_64) was loaded.
(lldb) bt
* thread #1: tid = 0x0000, 0x0000000105a53f6c
    bad`clearstr(p=0x0000000000000000) + 12
    at bad.c:10, stop reason = signal SIGSTOP
  * frame #0: 0x0000000105a53f6c
    bad`clearstr(p=0x0000000000000000) + 12
    at bad.c:10
    frame #1: 0x0000000105a53f57
    bad`main(argc=1, argv=0x00007fff5a1acb20) + 39
    at bad.c:20
    frame #2: 0x00007fff94ad65ad libdyld.dylib`start + 1
(lldb) q
unix$
```

Hemos utilizado el depurador *lldb(1)*, que tiene la opción `-c` para indicarle que queremos inspeccionar un *core dump* (un fichero `core`). *LLdb* es un shell en el que podemos escribir comandos para depurar. El comando *bt* imprime un volcado o *backtrace* de la pila. El comando *q* termina la ejecución del depurador.

Como podrás ver, el program ha muerto en la función `clearstr` del ejecutable `bad`. Concretamente en la línea 10. A dicha función la ha llamado `main` en el ejecutable `bad`, desde la línea 20 de `bad.c`. Y a `main` lo ha llamado una función llamada `start` dentro del cargador de librerías dinámicas (mira el capítulo de introducción si no recuerdas lo que es una librería dinámica).

Pues bien, la línea 10 de `bad.c` es:

```
p[0] = 0;
```

Y podemos ver que el argumento de `clearstr`, `p`, tiene como valor `0`. Luego sucede que `p` es `NULL` y no podemos atravesar dicho puntero. Ese es el problema. Ahora habría que pensar en por qué ha sucedido y en qué deberíamos hacer para arreglarlo.

Un detalle importante es que para que el depurador pueda saber qué ficheros fuente y líneas corresponden a cada contador de programa, hay que pedir al compilador que incluya información de depuración en el ejecutable (que incluya una tabla con la correspondencia de nombre de fichero fuente y línea a contador de programa, y tal vez otra información como nombres de función y variables). Habitualmente, el flag `-g` del compilador de C consigue dicho efecto. Esto es, hemos compilado como sigue:

```
unix$ cc -g -o bad bad.c
```

Si no incluimos la información de depuración, el depurador no puede hacer magia. Podemos utilizar el comando *strip(1)* que elimina la información de depuración de un ejecutable y ver el resultado:

```
unix$ strip bad
unix$ bad
Segmentation fault: 11 (core dumped)
unix$
```

Y ahora...

```
unix$ ls /cores
core.92473
unix$ lldb -c /cores/core.92473
(lldb) target create --core "/cores/core.92473"
Core file ’/cores/core.92473’ (x86_64) was loaded.
(lldb) bt
* thread #1: tid = 0x0000, 0x00000001014c4f6c
    bad‘clearstr + 12, stop reason = signal SIGSTOP
  * frame #0: 0x00000001014c4f6c
    bad‘clearstr + 12
    frame #1: 0x00000001014c4f57
    bad‘main + 39
    frame #2: 0x00007fff94ad65ad libdyld.dylib‘start + 1
```

Como verás, sólo podemos ver el contador de programa en cada llamada registrada en la pila (en cada *registro de activación* de la pila). Concretamente, el programa murió en `bad‘clearstr + 12`. Esto es, 12 posiciones más allá del comienzo de `clearstr` en el fichero ejecutable `bad`. Pero claro, no sabemos ni el fichero fuente ni la línea.

Lo más aconsejable es compilar siempre con `-g` y dejar la información de depuración en los ejecutables.

Otro depurador muy popular es `gdb`. Lo mejor es que utilizes el manual o *google* para averigüar cómo obtener un volcado de pila de un fichero `core` con tu depurador, y que localices el directorio en que tu UNIX deja los *core dumps*.

### 13. /proc

Existe otro interfaz para manipular procesos más allá de las llamadas al sistema habituales para ello. Concretamente, hay un directorio en (la mayoría de los sistemas) UNIX que aparenta tener ficheros relacionados con procesos:

```
unix$ ls /proc
1      153    25   43    701        dma          mtrr
10     154    26   44    778        driver       net
1001   16     27   45    8          execdomains  pagetypeinfo
1002   17     28   46    89         fb           partitions
103    174    29   47    898        filesystems  sched_debug
1046   175    3    48    9          fs           schedstat
1071   18     30   5     90         interrupts   scsi
11     19     31   50    900        iomem        self
1130   2      32   516   903        ioports      slabinfo
12     20     325  52    908        irq          softirqs
13     21     33   522   909        kallsyms     stat
14     21102  330  53    911        kcore        swaps
143    21104  34   54    930        key-users    sys
144    21227  35   541   acpi       keys         sysrq-trigger
145    21246  36   55    asound     kmsg         sysvipc
146    21247  368  5627  buddyinfo  kpagecount   timer_list
147    21248  369  613   bus        kpageflags   timer_stats
148    21301  37   629   cgroups    loadavg      tty
149    21302  38   637   cmdline    locks        uptime
15     21317  39   67    consoles   mdstat       version
150    215    40   68    cpuinfo    meminfo      version_signature
151    22     41   69    crypto     misc         vmallocinfo
15160  23     42   7     devices    modules      vmstat
152    24     421  70    diskstats  mounts       zoneinfo
```

En este caso, hemos utilizado un sistema Linux. Los ficheros en `/proc` no son ficheros reales en el disco. UNIX se los inventa para reflejar el estado de los procesos que están ejecutando y para dejarte averigüar cosas sobre ellos e incluso operar sobre ellos. Aún más, no sólo para operar sobre procesos, sino para operar sobre el sistema entero.

Cuando un proceso lee o escribe uno de estos ficheros, UNIX hace que la operación sobre el fichero se comporte normalmente, pero UNIX inventa el resultado de la operación para hacer creer que dichos ficheros corresponden en cada momento al estado del sistema. Son ficheros *sintéticos*. Dicho de otro modo, son falsos y UNIX se los inventa en cada momento.

Como puedes ver por el listado de ficheros en `/proc` hay un directorio por proceso, siendo el nombre del directorio el pid del proceso. Dicho directorio contiene ficheros que permiten ver y cambiar datos del proceso en cuestión. Por ejemplo, en un sistema Linux:

```
unix$ ps
  PID TTY          TIME CMD
21422 pts/0    00:00:00 bash
21438 pts/0    00:00:00 ps
unix$ ls -l /proc/21422
unix$ ls  /proc/21422
attr        coredump_filter  gid_map    mountinfo   oom_score      schedstat  status
autogroup   cpuset           io         mounts      oom_score_adj  sessionid  syscall
auxv        cwd              limits     mountstats  pagemap        setgroups  task
cgroup      environ          loginuid   net         personality    smaps      timers
clear_refs  exe              map_files  ns          projid_map     stack      uid_map
cmdline     fd               maps       numa_maps   root           stat       wchan
comm        fdinfo           mem        oom_adj     sched          statm
unix$
```

Hemos listado el directorio que corresponde al proceso del shell que estábamos ejecutando. Podemos ver cual es la línea de comandos para dicho proceso:

```
unix$ cat /proc/21422/cmdline
-bashunix$
```

Dado que era un shell de login (ejecutado al hacer el login en el sistema), el programa *login* que lo creó utilizó `-bash` como valor para el primer argumento (`argv[0]` en `main` en dicho proceso). El convenio en UNIX es que si en un shell, `argv[0]` comienza por "`-`", entonces se trata de un shell de login (y habitualmente dicho shell leerá `$HOME/.profile` u otro fichero para ejecutar los comandos que contenga como parte del proceso de inicialización).

En `/proc/21422/maps` tenemos una descripción de los segmentos de memoria que utiliza dicho proceso:

```
unix$ cat /proc/21422/maps
00400000-004ef000 r-xp 00000000 08:01 17039362                           /bin/bash
006ef000-006f0000 r--p 000ef000 08:01 17039362                           /bin/bash
006f0000-006f9000 rw-p 000f0000 08:01 17039362                           /bin/bash
006f9000-006ff000 rw-p 00000000 00:00 0
01f99000-021a0000 rw-p 00000000 00:00 0                                  [heap]
...
7f3164b43000-7f3164b44000 rw-p 00000000 00:00 0
7fff170f1000-7fff17112000 rw-p 00000000 00:00 0                          [stack]
unix$
```

Puedes ver las direcciones de comienzo y fin para los segmentos de texto, datos inicializados de sólo lectura, datos inicializados, BSS, y pila (entre otros). Hemos borrado del listado los segmentos correspondientes a las librerías dinámicas que utiliza dicho proceso. Resulta interesante ver que los segmentos de texto y datos inicializados proceden de `/bin/bash` para este proceso. Como verás, el texto tiene permiso de ejecución y los datos tienen permiso de lectura al menos. Los datos que no son constantes están en un segmento con permisos de lectura escritura.

A la vista de esto, aunque en C, `"hola"` es un array de `char`, no deberías intentar cambiar su contenido. Es muy posible que dicho array esté guardado en memoria en un segmento de datos de sólo lectura durante la ejecución del programa.

El fichero `/proc/21422/status` tiene información interesante sobre el estado del proceso:

```
unix$ cat /proc/21422/status
Name:    bash
State:    S (sleeping)
Pid:    21422
PPid:    21421
...
voluntary_ctxt_switches:    398
nonvoluntary_ctxt_switches:    247
unix$
```

Entre otras cosas, puedes ver que actualmente el proceso está bloqueado (durmiendo) y que unas 398 veces ha hecho llamadas al sistema que han provocado que se le expulse del procesador. Además, unas 247 veces ha sido UNIX el que lo ha expulsado sin que el proceso hiciera ninguna llamada que causara la expulsión. Simplemente, llegaría una interrupción de reloj y UNIX decidiría que ya era hora de empezar a ejecutar un rato otro proceso.

Recuerda que `/proc` (en la mayoría de los UNIX) es una abstracción y no corresponde a datos reales guardados en ningún disco. UNIX se inventa esos ficheros respondiendo a las llamadas al sistema que operan sobre dichos ficheros para que aparentemente `/proc` contenga una representación del estado de los procesos y el sistema.