Sistemas Informáticos

Curso 2005-2006

Establecimiento externo de límites de procesos en Linux

Ibón Cano Sanz

Bálder Carraté Martínez

Manuel Rodríguez Pascual

Dirigido por:
Prof. D. Manuel Prieto Matías
Dpto. Arquitectura de Computadores y Automática

Facultad de Informática

Universidad Complutense de Madrid

"If it compiles, it is good, if it boots up, it is perfect."

Linus Torvalds

Resumen

En todo sistema operativo multitarea han de existir límites para regular el uso equitativo de los recursos.

Un límite consiste en un valor que acota el uso de un determinado recurso por parte de un proceso dado.

Hasta este momento, las llamadas al sistema existentes en Linux sólamente permitían consultar o modificar los límites de un proceso cualquiera desde dentro del propio proceso.

El trabajo realizado en el código fuente del kernel aporta las siguientes mejoras:

- permite que los límites se lean o escriban de forma externa a cada proceso.
- define e implementa políticas de modificación (hasta ahora inexistentes).
- incluye los límites de procesos dentro del pseudosistema de ficheros */proc*, permitiendo su tratamiento como un fichero de texto legible.

Así mismo, se ha desarrollado la aplicación gráfica *Qlimits* utilizando las librerías *Qt* que permite el seguimiento y establecimiento de los límites en un interfaz sencillo.

Abstract

In any multitasking operating system, there should exist limits to regulate fair resources' use.

A limit consists in a value that bounds resources' usage by a given process.

Up to now, the solely support provided by existent system calls in Linux permits any process to consult or modify its own limits internally.

Work carried out over kernel's source code provides the following improvements:

- allows limits to be externally read or written outside each process.
- defines and implements writing policies (non-existents up to now).
- includes processes' limits in */proc* pseudo-filesystem and lets the limits to be processed as a readable text file.

We've also developed the graphic application *Qlimits* using *Qt* graphic libraries to allow process' tracking and setting in a friendly interface.

Índice general

Índice de figuras

Capítulo 1

Conceptos relacionados

Puesto que este trabajo, como ya se ha dicho, consiste en realizar modificaciones para que sea posible modificar externamente los límites de los procesos, es fundamental presentar la forma en que se representan, crean, planifican y destruyen los procesos en Linux.

1.1. Procesos en Linux

1.1.1. Conceptos generales

Un proceso se puede concebir como el resultado de tener un programa en ejecución y sus recursos asociados, tales como el estado del procesador, el espacio de direcciones, los ficheros abiertos por el propio proceso o las señales pendientes del mismo.

Un proceso estará compuesto por uno o varios hilos, cada uno de de los cuales poseerá un contador de programa, pila de proceso y conjunto de registros del procesador propios. Con esta nomenclatura, conviene aclarar que el kernel planifica hilos, no procesos.

Linux tiene una implementación única para hilos y procesos, no habiendo distinción alguna de cara al planificador. Un hilo se representa con la misma estructura de datos que un proceso y la única diferencia entre ambos consistirá en que un hilo comparte ciertos recursos con otros procesos (o hilos). En concreto, varios hilos que compongan un mismo proceso compartirán el espacio de direcciones. De esta forma, existe un único planificador que no debe hacer distinción alguna entre distintos tipos de procesos, pues todos son igual de importantes y

representan la misma carga de trabajo, en contraste con la idea que presentan otros sistemas operativos en los que los procesos son una estructura muy pesada y cuentan con los hilos como *procesos ligeros* que permiten una planificación y ejecución más rápida que los procesos convencionales.

Los procesos de los sistemas operativos modernos proporcionan un procesador virtual y memoria virtual, de forma que cada proceso puede trabajar como si tuviese a su disposición un sistema completo, sin preocuparse de estar en realidad compartiéndolo con otros procesos. Cabe destacar que los hilos comparten la memoria virtual, mientras que cada uno recibe un procesador virtual.

Un proceso se crea mediante la llamada al sistema `fork()`, que crea un duplicado del proceso desde el cual se ejecutó la llamada. Este último proceso recibe el nombre de proceso padre, mientras que el proceso que se crea con `fork()` será llamado proceso hijo. Normalmente, tras la creación del proceso se emplea la llamada al sistema `exec()` para crear un nuevo espacio de direcciones y cargar en él otro programa.

Los procesos terminan con la llamada al sistema `exit()`, que además de finalizar el proceso liberan todos sus recursos. Cualquier proceso padre puede conocer el estado de un proceso hijo terminado mediante la llamada al sistema `wait4()`, que permite a un proceso esperar la terminación de un proceso concreto. Cuando un proceso termina, pasa a estado zombie hasta que su padre ejecuta `wait()` o `waitpid()`.

Dentro del kernel de Linux, los procesos son conocidos como tareas (*tasks*) y toda la información asociada a cada proceso se almacena en el descriptor de proceso, que es una estructura de datos de tipo

`struct task_struct` (ver *Implementación de procesos en el kernel*)

y es la forma de acceso más común a los datos de un proceso. Esta estructura de datos se encuentra definida en el fichero `<linux/sched.h>`. El kernel almacena todos los procesos en una lista circular doblemente enlazada llamada lista de procesos cuyos elementos son descriptores de proceso.

Cada proceso se identifica de forma unívoca dentro del sistema con un identificador de proceso o *PID*, que es un valor numérico entero positivo (normalmente comprendido entre 1 y 32768, lo que implica que el número total de procesos en el sistema no pueda superar este valor máximo). Este identificador se almacena en un campo

```
pid_t pid;
```

del descriptor de proceso, siendo `pid_t` un tipo de datos redefinido equivalente a `short int`.

El proceso `init` (cuyo `pid` es 1) es un proceso que el kernel inicia en el último paso del proceso de arranque. `init` es el encargado de crear todos los demás procesos como se indique en los *scripts* de inicio. Si recordamos que la única forma de crear procesos es emplear la función `fork()`, que crea un proceso hijo, parece evidente que todos los procesos tienen un origen común y existe, por tanto, una jerarquía de procesos que se puede ver como un árbol cuya raíz es el proceso `init`. Siempre podemos acceder a `init` mediante la variable estática `init_task`.

1.1.2. Creación de procesos

Como ya se ha mencionado anteriormente, la creación de procesos en Linux (y en todos los sistemas basados en Unix) se divide en dos pasos:

`pid_t fork(void);`

Con esta llamada se crea un proceso hijo que se diferenciará del proceso padre únicamente en su *PID* y *PPID* (*Parent PID* o *PID* del proceso padre) y en que inicialmente todos los recursos del proceso se inicializan a 0, ya que se utiliza una técnica conocida como *COW* (*Copy On Write*). *COW* permite que ambos procesos compartan una copia de su espacio de direcciones mientras éste no sea modificado. En el momento en que se realiza una escritura sobre dicha zona de memoria (lo que normalmente ocurre inmediatamente con la llamada a `exec()`), se duplican los datos y cada proceso recibe su propia copia.

El proceso recién creado tampoco hereda los bloqueos de ficheros ni las señales pendientes. El empleo de *COW* hace que el consumo de tiempo y memoria se reduzca a la creación de las tablas de páginas y la creación de la nueva estructura `task_struct` para albergar el proceso hijo.

En caso de éxito la función devuelve el *PID* del nuevo proceso en el hilo de ejecución del padre y 0 en el hijo. En caso de fallo no se creará un nuevo proceso, el padre (el proceso que hizo la llamada) recibirá -1 como valor devuelto y `errno` contendrá el valor apropiado.

Resulta interesante destacar que una de las posibles causas de fallo es que el proceso padre halla excedido el límite de número de procesos creados (`RLIMIT_NPROC`). El proceso podrá superar este límite si se la dotado con las opciones `CAP_SYS_ADMIN`

o `CAP_SYS_RESOURCE`.
La implementación de `fork()` se basa en la llamada al sistema

```
int clone(int flags, void *child_stack);
```

que crea un proceso compartirá con el padre los recursos especificados en el parámetro `flags`. De esta forma, `clone()` permite la creación de hilos que, como ya se ha explicado anteriormente, compartan con su padre (y entre sí) los recursos indicados por el programador. Entre las opciones permitidas por los `flags`, podemos compartir, por ejemplo, espacio de direcciones, ficheros abiertos, manejadores de señales o bien hacer que el hijo herede el PPID de su padre, de manera que ambos sean hijos del mismo proceso.

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

Esta función ejecuta el programa indicado en `filename`. Este programa podrá ser un fichero binario ejecutable o bien un script cuya primera línea sea de la forma

```
#! interpreter [arg]
```

indicando el intérprete que se empleará para el script y los argumentos con los que se debe invocar dicho intérprete.
El parámetro `argv` es un array de cadenas de caracteres que contiene los argumentos del programa a ejecutar, mientras que `envp` es otro array de cadenas de la forma *clave=valor* que almacena las variables de entorno para el nuevo programa.
Si esta función tiene éxito, no regresa al programa que la invocó, sino que comienza la nueva ejecución, heredando el *PID* del proceso que la invocó y todos los descriptores de fichero que no se han declarado *close-on-exec*; se eliminan las señales pendientes y se sobreescriben los datos, bss y la pila con la del nuevo programa.
En caso de error, la función sí regresa con valor de salida -1 y `errno` tendrá un valor que explique la causa del fallo.

1.1.3. Terminación de procesos

Un proceso puede terminar por las siguientes razones:

- que el propio proceso realice una llamada a `exit()` para finalizar la ejecución de forma voluntaria y explícita.

- que el proceso alcance el final del programa a ejecutar. En este caso, los compiladores suelen introducir de forma automática una llamada a la función de salida.
- que durante la ejecución del proceso se reciba una excepción o señal que el proceso no sepa manejar y se vea obligado a abortar la ejecución.

En cualquiera de las situaciones anteriores, ante la terminación de un proceso el sistema operativo ejecutará la función `do_exit()` (incluida en el fichero `<kernel/exit.c>`) para liberar todos los recursos utilizados de forma exclusiva por el proceso de forma segura, como los temporizadores del kernel, semáforos, el espacio de direcciones del proceso, descriptores de fichero, archivos de sistema y manejadores de señal.

Tras liberar los recursos, el proceso debe notificar a su padre que ha finalizado. En caso de que el proceso que termina tenga algún hijo, hay que indicar a estos procesos hijo que su padre ya no existe y reubicarlos como hijos de otro hilo de su grupo o de `init`. Por último, el proceso debe cambiar su estado a *TASK_ZOMBIE* para indicar al planificador que ha finalizado e invocar a dicho planificador para que ponga en ejecución otra tarea.

De esta manera, el proceso sólo ocupa la memoria de su pila del kernel y el necesario para alojar las estructuras `thread_info` y `task_struct`. Estos datos quedan disponibles para que el padre pueda acceder a ellos si lo quisiera. Cuando esto ocurra o el padre indique que no está interesado en dicha información, la memoria ocupada se liberará para que el kernel la utilice cuando sea necesario.

1.1.4. Implementación de procesos en el kernel

El kernel contiene una lista con todos los procesos que se están ejecutando en un momento dado. Esta estructura está implementada como una lista circular doblemente enlazada llamada `task list`. Cada elemento de la tabla es un descriptor de proceso del tipo `task_struct` (ver figura 1.1) definido en `<linux/sched.h>`. El descriptor de proceso contiene, como ya se ha explicado, toda la información necesaria acerca de un proceso: programa ejecutándose, ficheros abiertos, espacio de direcciones y otra mucha información que escapa del alcance de este proyecto.

```
struct task_struct {
  volatile long state; /*Estado del proceso:
      -1 no ejecutable,
      0 ejecutable,
      >0 parado*/
  struct thread_info *thread_info; /*thread_info asociada*/
  pid_t pid; /*id proceso*/
  pid_t tgid; /*id grupo tareas*/
  struct task_struct *parent; /*proceso padre*/
  struct list_head children; /*lista hijos*/
  struct list_head sibling; /*enlace a la lista de hijos
      del proceso padre*/
  struct task_struct *group_leader; /*proceso líder
      del grupo de tareas*/
  struct fs_struct *fs; /*info sistema ficheros*/
  struct files_struct *files; /*info ficheros abiertos*/
  struct namespace *namespace; /*namespace*/
  struct signal_struct *signal; /**/
}
```

Figura 1.1: Estructura `task_struct`

Históricamente, el descriptor de proceso `task_struct` se alojaba al final de la pila del kernel de cada proceso. Esto permitía a las arquitecturas como pocos registros (como la X86) acceder a su contenido sin tener que usar un registro extra para calcular la dirección. Sin embargo, en la versión 2.6 del kernel se introduce una nueva estructura, `struct thread_info` (ver figura 1.2), que modifica esta disposición: es ésta la que se aloja al final de la pila e incluye un puntero al descriptor de procesos, además de añadir una serie de funcionalidades. La definición de esta estructura y sus funciones asociadas se encuentra en `<asm/thread_info.h>`)

1.2. Concurrencia y sincronización

1.2.1. Introducción

En todo sistema que cuenta con memoria compartida se debe asegurar la consistencia de datos ante los accesos concurrentes a los mismos datos por parte de distintos procesos realizando una gestión explícita

```
struct thread_info {
   struct task_struct *task; /*descriptor de proceso*/
   struct exec_domain *exec_domain; /*dominio ejecución*/
   unsigned long flags; /*flags bajo nivel*/
   unsigned long status; /*flags sincronismo procesos*/
   __u32 cpu; /*CPU actual*/
   int preempt_count; /*0 =>expropiable, <0 =>FALLO*/
   mm_segment_t addr_limit; /*espacio direcciones:
       0-0xBFFFFFFF proceso de usuario
       0-0xFFFFFFFF proceso del kernel*/
   struct restart_block restart_block;
   unsigned long previous_esp;
   __u8 supervisor_stack[0];
}
```

Figura 1.2: Estructura `thread_info`

de dichos accesos. En máquinas que cuentan con un solo procesador, solamente se encontrará concurrencia si se produce una interrupción, se ejecuta código del kernel o se cambiase el proceso en ejecución. En entornos multiprocesador, esta tarea se complica enormemente, pues distintos procesadores pueden intentar un acceso que sea concurrente en el tiempo a ciertos recursos compartidos.

Como regla general, hay que tener en cuenta que cualquier objeto creado por código del núcleo que va a ser compartido total o parcialmente debe existir y funcionar de acuerdo a su especificación hasta que no exista ninguna referencia externa a dichos datos. Se deduce que

- ningún objeto debe ser accesible para el kernel hasta que esté en condiciones de operar adecuadamente.
- se debe llevar algún tipo de registro del número de referencias que existen en cada momento a un recurso compartido. En muchas ocasiones, este control lo realiza el kernel de forma "automática".

Normalmente se conoce como *región crítica* al fragmento de código que accede a algún recurso compartido y, por tanto, no debe ser ejecutada por más de un proceso al mismo tiempo. Una de las formas más sencillas de evitar problemas provocados por la concurrencia consistirá en garantizar que las regiones críticas se ejecuten de forma

atómica, como si se tratara de una única instrucción. Las instrucciones sencillas como la comparación o el incremento de un dato se encuentran implementadas de forma atómica en el repertorio de instrucciones de distintas arquitecturas hardware y no suelen suponer un problema. Si nos enfrentamos a una región crítica que incluya, por ejemplo, la modificación de una estructura de datos relativamente grande, debemos proporcionar alguna solución más elaborada.

El kernel de Linux ofrece distintos mecanismos de sincronización que se encargan de garantizar que los accesos concurrentes se realicen de forma completamente segura, sin corrupción ni pérdida de información.

La técnica de *exclusión mutua* consiste en asegurar que en ningún momento haya más de un proceso accediendo al mismo recurso. A continuación veremos varios mecanismos de sincronización que se basan en el principio de exclusión mutua.

1.2.2. Operaciones atómicas

El kernel proporciona dos tipos de operaciones atómicas: una para operar sobre datos enteros y otra que opera sobre bits. Recordemos que estas operaciones suelen estar directamente implementadas en el repertorio de instrucciones de las distintas arquitecturas soportadas por Linux.

Operaciones atómicas sobre enteros

El kernel ofrece un conjunto de funciones para operar de forma atómica sobre enteros. Estas funciones no utilizan el tipo `int` original de *C*, sino que se emplea el tipo `atomic_t` por las siguientes razones:

- el hecho de que estas funciones utilicen este tipo especial para sus argumentos de entrada y salida garantiza que sólo puedan ser invocadas con parámetros de tipo `atomic_t`. De la misma manera, se asegura que los datos que deben ser compartidos (y por ello han sido declarados de este tipo) no sean utilizados en otras funciones.
- evitar que el compilador realice optimizaciones no deseadas que puedan conducir a situaciones de error.
- la utilización de `atomic_t` oculta los detalles que dependen de la arquitectura y ofrece un interfaz estable y común.

Aunque algunas funciones atómicas para enteros sólo están disponibles en ciertas arquitecturas, existe un repertorio de instrucciones de este tipo que funciona sobre cualquier máquina, pudiéndose encontrar la declaración de estas operaciones en el fichero `<asm/atomic.h>` para cada arquitectura.

Los valores de tipo `atomic_t` se declaran de la forma habitual. Las operaciones trabajan con punteros a las variables declaradas. La figura 1.3 muestra un ejemplo de uso de este tipo de datos.

```
atomic_t a = ATOMIC_INIT(0);
a = atomic_op(&a);
a = atomic_add(2, &a);
```

Figura 1.3: Ejemplo de uso de operaciones atómicas sobre enteros

Operaciones atómicas sobre bits

La implementación de estas operaciones, como en el caso de las operaciones atómicas para enteros, también es dependiente de la arquitectura y podemos hallar las declaraciones en los ficheros `<asm/bitops.h>`.

Estas funciones no tienen un tipo de datos específico, sino que trabajan con posiciones de memoria genéricas, recibiendo como argumentos un puntero y un número que indica el bit al que se quiere acceder (siendo 0 el bit menos significativo de la palabra). Podemos encontrar un ejemplo de uso de estas operaciones en la figura 1.4.

```
unsigned long x = 0;
set_bit(0, &x);
clear_bit(0, &x);
```

Figura 1.4: Ejemplo de uso de operaciones atómicas sobre bits

1.2.3. Cerrojos

Los *cerrojos* son el mecanismo de exclusión mutua más utilizado en el kernel de Linux. Un cerrojo es una variable que solamente puede tener dos valores: *bloqueado* y *desbloqueado*.

Si bien la implementación de los cerrojos es dependiente de la arquitectura y se escribe en ensamblador, en general los cerrojos se implementan utilizando un único bit de una variable entera. Si el bit vale 0, el cerrojo está desbloqueado, y si vale 1, estará bloqueado. De esta manera, cuando un proceso va a entrar en su región crítica, comprueba si el cerrojo está desbloqueado y, de ser así, lo bloquea y entra a ejecutar el código. Si encuentra el cerrojo bloqueado, entra en un bucle de espera en el que comprueba si el cerrojo se desbloquea. El cerrojo se debe desbloquear cuando un proceso abandone la región crítica. Lógicamente, las operaciones de tipo "test-and-set" que comprueban y modifican el estado del cerrojo se deben ejecutar de forma atómica para garantizar un funcionamiento correcto.

Hay que tener en cuenta que los procesos que están esperando a que se desbloquee un cerrojo realizan una *espera activa* y por lo tanto consumen tiempo de procesador. Por ello, los cerrojos se deben utilizar cuando la región crítica sea "pequeña" o siempre que el proceso que compite por acceder a la región crítica no pueda dormir, como en el caso de los manajadores de interrupciones.

Cuando el código del kernel bloquea un cerrojo, automáticamente se desactiva el kernel expropiativo para evitar que un proceso que haya bloqueado un cerrojo pueda ser expropiado. Esta situación podría llevar a bloqueos no deseados si otro proceso intenta obtener ese cerrojo, ya que entrará en un bucle de espera que puede llevarnos al bloqueo completo del sistema. Por ello es recomendable que el código que ejecute un proceso que tenga un cerrojo bloqueado se ejecute de forma atómica y no duerma ni abandone el procesador salvo que tenga que tratar una interrupción.

En caso de que algún manejador de interrupción pueda necesitar obtener un determinado cerrojo, debemos garantizar que cada vez que se obtenga dicho cerrojo se desactiven las interrupciones en el procesador en que se ejecuta el proceso que haya bloqueado el cerrojo. En caso contrario, nos encontraríamos con otra situación de bloqueo: el código de interrupción espera indefinidamente a que el cerrojo se libere y el proceso no podrá terminar su ejecución (y desbloquear el cerrojo) hasta que no finalice la rutina de tratamiento de interrupción. Más adelan-

te se muestra una familia de funciones para manejar los cerrojos que tienen en cuenta estas necesidades.

El archivo de cabeceras en el que encontramos la definición de los cerrojos es `<linux/spinlock.h>`. Un cerrojo se representa en el kernel con un tipo especial `spinlock_t` que, como ya se ha dicho, varía en la implementación real según la arquitectura del procesador. Las funciones más relevantes (y elementales) son:

`spinlock_t cerrojo = SPIN_LOCK_UNLOCKED;`
: Declara una variable `cerrojo` de tipo `spinlock_t` y la inicializa estáticamente (en general, un cerrojo empezará estando desbloqueado).

`void spin_lock_init(spinlock_t *cerrojo);`
: Inicializa dinámicamente la variable `cerrojo` (quedando éste desbloqueado).

`void spin_lock(spinlock_t *cerrojo);`
: Trata de obtener `cerrojo` y bloquearlo antes de entrar en una sección crítica según el algoritmo de espera ya explicado.

`int spin_trylock(spinlock_t *cerrojo);`
: Es análoga a `spin_lock(cerrojo)`, pero no realiza espera y por lo tanto no es bloqueante. La función devolverá un valor distinto de 0 cuando obtenga `cerrojo` y 0 en caso contrario.

`void spin_unlock(spinlock_t *cerrojo);`
: Desbloquea `cerrojo` tras abandonar la sección crítica. Se emplea si el cerrojo se ha obtenido mediante `spin_lock(cerrojo)` o con `spin_trylock(cerrojo)`.

```
void spin_lock_irqsave(spinlock_t *cerrojo,
          unsigned long flags);
```
: Al obtener `cerrojo`, desactiva las interrupciones en el procesador local y almacena el estado de las mismas en `flags`. Se debe emplear cuando sea posible que ya se hayan desactivado las interrupciones, para garantizar que se devuelven al mismo estado al liberar `cerrojo`.

```
void spin_lock_irqrestore(spinlock_t *cerrojo,
          unsigned long flags);
```
: Al liberar `cerrojo`, activa las interrupciones en el procesador local y fija el estado de las mismas a `flags`. Se debe emplear cuando se haya obtenido el cerrojo con `spin_lock_irqsave(cerrojo,flags)`.

Se debe bloquear y desbloquear el cerrojo en el mismo procedimiento o función, ya que en caso contrario, el código podría fallar en algunas arquitecturas.

`void spin_lock_irq(spinlock_t *cerrojo);`
: Al obtener `cerrojo`, desactiva las interrupciones en el procesador local sin guardar el estado. Se debe utilizar esta función cuando haya certeza de que nadie haya desctivado previamente las interrupciones y que al liberar `cerrojo` las interrupciones deben quedar activadas.

`void spin_unlock_irq(spinlock_t *cerrojo);`
: Al liberar `cerrojo`, activa las interrupciones en el procesador local.
Se debe utilizar sólo cuando se haya obtenido el cerrojo mediante `spin_lock_irq(cerrojo)`.

`void spin_lock_bh(spinlock_t *cerrojo);`
: Al obtener `cerrojo`, desactiva las interrupciones *software* en el procesador local sin guardar el estado. Las interrupciones *hardware* seguirán activadas.

`int spin_trylock_bh(spinlock_t *cerrojo);`
: Es análoga a `spin_lock_bh(cerrojo)`, pero no realiza espera y por lo tanto no es bloqueante. La función devolverá un valor distinto de 0 cuando obtenga `cerrojo` y 0 en caso contrario.

`void spin_unlock_bh(spinlock_t *cerrojo);`
: Al liberar `cerrojo`, activa las interrupciones *software* en el procesador local. Se debe emplear cuando se haya obtenido el cerrojo con `spin_lock_bh(cerrojo)` o `spin_trylock_bh(cerrojo)`.

El kernel también nos ofrece cerrojos de tipo *lectores/escritores* (*l/e*). Este tipo de cerrojos permite a un número de lectores determinado por el programador acceder de forma simultánea a la región crítica. Los escritores, en cambio, requieren acceso exclusivo.

Un cerrojo de *l/e* se representa en el kernel con un tipo especial `rwlock_t` que cuenta con una serie funciones análogas a las existentes para el tipo `spinlock_t`. En la figura 1.5 se muestran algunos métodos de cerrojos de *l/e*. Las funciones *read* serán las usadas por los lectores; los escritores emplearán las funciones *write*:

```
void read_lock(rwlock_t *cerrojo);
void read_unlock(rwlock_t *cerrojo);
void write_lock(rwlock_t *cerrojo);
void write_unlock(rwlock_t *cerrojo);
```

Figura 1.5: Ejemplo de funciones de cerrojos *lectores/escritores*

1.2.4. Semáforos

Los semáforos son otro mecanismo de sincronización muy conocido. Podemos ver un semáforo como una variable entera sobre la que se pueden realizar las siguientes tres operaciones:

1. Iniciarlizar el semáforo con un valor no negativo mayor que cero.
2. Cuando un proceso desea acceder a su región crítica, comprobará si el valor del semáforo es mayor que cero. Si es así, decrementará el valor del semáforo y continuará su ejecución. En caso de que el semáforo tenga valor 0, el proceso esperará hasta que alguien libere el semáforo. Normalmente esta función de acceso al semáforo recibe el nombre de `p()` (del alemán *proberen*, probar).
3. Cuando un proceso salga de la región crítica, incrementará el valor del semáforo y, en caso de que haya otros procesos esperando en ese semáforo, los despertará para informarles de que el semáforo está libre. La función de liberar el semáforo se suele llamar `v()` (del alemán *verhogen*, incrementar).

Un semáforo empleado para exlusión mutua recibe el nombre de *mutex*, y lógicamente se inicializa con valor 1. La mayoría de los semáforos empleados en el kernel son de este tipo.

Funcionalmente, podemos entender que en Linux los semáforos son cerrojos que permiten a los procesos "apuntarse a una lista de espera" y realizar la espera durmiendo, en vez de llevarla a cabo activamente; esto deja el procesador libre mientras aguardan. Cuando un proceso finaliza de ejecutar el código crítico, despertará al primer proceso que haya en la cola de espera para que éste tome el semáforo y continúe su ejecución.

Podemos destacar las siguientes características de los semáforos:

- Dado que los procesos en espera pueden dormir, el tiempo durante el que un semáforo está bloqueado puede ser mayor que el de un cerrojo sin pérdida de eficiencia.
- Por la misma razón, los semáforos no son recomendables en situaciones en las que el tiempo de bloqueo sea pequeño, ya que la carga adicional de mantener la cola de espera y dormir y despertar procesos puede llegar a sobrepasar el tiempo durante el que se bloquea el semáforo.
- El hecho de que la espera se realice durmiendo hace que los semáforos sólo se empleen en contexto de procesos, ya que las interrupciones no se planifican de la misma manera.
- Un proceso puede dormir tras haber obtenido un semáforo, ya que no bloqueará a los demás procesos que traten de bloquearlo: irán a dormir y esperarán su turno.
- Un proceso no debe tratar de obtener un semáforo si tiene un cerrojo bloqueado, ya que la espera del semáforo le puede obligar a dormir, y un proceso que posea un cerrojo no debe dormir.

Teniendo en cuenta lo anterior, parece claro que tendremos que usar un semáforo en lugar de un cerrojo siempre que el código tenga que dormir o cuando queramos permitir esta flexibilidad (que a menudo simplifica las precauciones que haya que tomar). Cuando la capacidad de dormir no sea un condicionante, basaremos nuestra elección en la duración del bloqueo, utilizando cerrojos para bloqueos cortos y semáforos para aquellos bloqueos de mayor duración. Por último, hay que resaltar que los semáforos no desactivan la *expropiatividad* del kernel, por lo que el código que haya adquirido un semáforo puede ser *expropiado* y el bloqueo no aumenta la latencia del planificador.

La implementación de los semáforos es dependiente de la arquitectura. Podemos encontrar los distintos ficheros de cabeceras en `<asm/semaphore.h>`. La estructura utilizada para implementar los semáforos se muestra en la figura 1.6.

A continuación se presenta una lista de los métodos utilizados para manejar los semáforos:

`static DECLARE_SEMAPHORE_GENERIC(name, count);`
: Declaración estática de un semáforo con nombre `name` y contador inicializado con el valor entero `count`.

```
struct semaphore {
   atomic_t count; /*contador del semáforo*/
   int sleepers; /*número de procesos esperando*/
   wait_queue_head_t wait; /*cola de espera*/
};
```

Figura 1.6: Estructura `semaphore`

`static DECLARE_MUTEX(name);`
: Declaración estática de un *mutex* con nombre `name` y valor inicial 1.

`static DECLARE_MUTEX_LOCKED(name);`
: Declaración estática de un *mutex* con nombre `name` y valor inicial 0.

`void sema_init(struct semaphore *sem, int val);`
: Inicializa dinámicamente un semáforo `sem` con el valor entero `val`.

`void init_MUTEX(struct semaphore *sem);`
: Inicializa dinámicamente un mutex `sem` con el valor 1.

`void init_MUTEX(struct semaphore *sem);`
: Inicializa dinámicamente un mutex `sem` con el valor entero 0.

`void down(struct semaphore *sem);`
: La función para obtener el semáforo, `p()`, recibe el nombre de `down()` en Linux debido al hecho de que decrementa el contador.

`int down_interruptible(struct semaphore *sem);`
: Esta versión permite que la operación sea interrumpida. En tal caso, la función devolverá un valor distinto de 0 y el semáforo no será adquirido.

`int down_trylock(struct semaphore *sem);`
: Esta versión no duerme esperando a que se libere el semáforo en caso de que esté ocupado; en su lugar, vuelve inmediatamente y devuelve un valor distinto de 0.

`void up(struct semaphore *sem);`
: La función para liberar el semáforo, `v()`, recibe el nombre de `up()` en Linux debido a que incrementa el contador.

Hay que resaltar que es fundamental asegurar que un semáforo se obtiene con una única llamada a `down()` y se libera *siempre* con una única llamada a `up()`.

Por último, cabe destacar que el kernel también proporciona semáforos de tipo *lectores/escritores* (*l/e*) que se implementan con la estructura de datos `struct rw_semaphore` cuya definición se puede encontrar en `<linux/rwsem.h>`. En este tipo de semáforos, como ocurría en el caso de los cerrojos, se permite a un número de lectores determinado por el programador acceder de forma simultánea a la región crítica. Los escritores requieren nuevamente acceso exclusivo y siempre tienen prioridad sobre los lectores, y esto puede llevar a casos de inanición de los lectores. Por ello este tipo de semáforos sólo se emplea cuando el número de accesos de escritura es muy inferior al de accesos de lectura. Desde el punto de vista del programador, este tipo de semáforos no utiliza contador, ya que la propia política de *l/e* impone las restricciones de acceso.

Requisitos	**Bloqueo a emplear**
Bajo coste	Cerrojo
Bloqueo de corta duración	Cerrojo
Bloqueo de larga duración	Semáforo
Necesidad de bloquear desde contexto de interrupción	Cerrojo
Necesidad de dormir durante el bloqueo	Semáforo

Figura 1.7: Comparación de uso de cerrojos y semáforos

1.2.5. Variables condición

Las variables condición son un mecanismo que ofrece una forma sencilla de sincronizar dos tareas en el kernel cuando una de ellas está a la espera de que cierto evento tenga lugar en la otra tarea. Más concretamente, una tarea quedará esperando en la variable condición mientras la otra realiza sus cómputos; en el momento en que esta tarea haya finalizado el trabajo, usará la variable para despertar al proceso que estaba esperando. Este mecanismo no es más que una simplificación de los semáforos, también cuenta con una cola de espera para ordenar los procesos en espera. Este tipo de bloqueo se utiliza, por ejemplo, en la llamada `vfork()` para despertar al proceso padre cuando el hijo realiza una llamada a `exec()` o `exit()`.

Las variables condición se declaran en `<linux/completion.h>` y se representan con la estructura mostrada en la figura 1.8.

```
struct completion {
  unsigned int done; /*=0 si tarea completa
      <>0 en otro caso*/
  wait_queue_head_t wait; /*cola de espera*/
};
```

Figura 1.8: Estructura `completion`

Las funciones empleadas para manejar las variables condición son:

`DECLARE_COMPLETION(c);`
: Declaración estática de una variable condición `c`.

`void init_completion(struct completion *c);`
: Inicialización dinámica de una variable condición `c`.

`int wait_for_completion(struct completion *c);`
: Esta función realiza una espera no interrumpible sobre la variable `c` en el proceso que la invoca. Si no se llega a completar la tarea, el proceso queda bloqueado y no se puede matar con un *KILL*.

```
unsigned long wait_for_completion_timeout(
          struct completion *c, unsigned long timeout);
```
: Esta función realiza una espera no interrumpible en la variable `c` y especifica un tiempo máximo de espera `timeout`.

`int wait_for_completion_interruptible(struct completion *c);`
: Esta versión proporciona espera interrumpible en la variable `c`.

```
unsigned long wait_for_completion_interruptible_timeout(
         struct completion *c, unsigned long timeout);
```
: Esta versión nos ofrece espera interrumpible en la variable `c` con un tiempo máximo de espera `timeout`.

`void complete(struct completion *c);`
: Esta función despierta al primer proceso en la cola de espera de la variable condición `c` para indicarle que la tarea se ha completado.

`void complete_all(struct completion *c);`
: Esta función despierta a todos los procesos en la cola de espera de la variable `c`.

1.3. Límites de procesos

En esta sección se presentará una definición e introducción a los límites y mostrará la localización e implementación de los mismos de los procesos dentro del código del kernel.

1.3.1. Definición de límite

Un *límite* se puede entender como *un valor numérico que acota el uso de un determinado recurso por parte de un proceso dado.*

Cada límite está formado por dos valores, el *límite blando* y el *límite duro*. El *límite blando* es el que tiene un determinado proceso en un momento para cada recurso. El *duro* es el valor más alto que puede alcanzar el límite actual. La idea es que un proceso tenga como valor del *límite blando* unos valores razonables según la función que esté realizando. Los *límites duros* se incluyen para acotar los *blandos* e impedir que se establezcan valores que pueden poner en peligro la estabilidad del sistema.

1.3.2. Localización de los límites

Entre la información que encontramos en el descriptor de proceso o `task_struct` (que se muestra en la figura 1.1) de cada tarea, hay un puntero a una estructura del tipo `signal_struct`, de la cual se muestra un fragmento mínimo en la figura 1.9; el resto de los campos se omiten por no presentar relación con este proyecto. Dicha estructura está definida en el archivo de cabeceras `<linux/sched.h>`.

```
struct signal_struct {
  [...]
  struct rlimit rlim[RLIM_NLIMITS]; /*vector de los límites
      del proceso*/
  [...]
};
```

Figura 1.9: Estructura `signal_struct`

```
struct rlimit {
  unsigned long rlim_cur; /*límite blando*/
  unsigned long rlim_max; /*límite duro*/
};
```

Figura 1.10: Estructura `rlimit`

Como se puede ver en la figura 1.9, `struct signal_struct` tiene un campo `rlim[RLIM_NLIMITS]`, que es un array de `struct rlimit`. La estructura `rlimit` que implementa cada uno de estos límites está definida en `<linux/resource.h>` y se muestra en la figura 1.10.

1.3.3. Descripción de los límites

Para acceder a las distintas posiciones del array de límites `rlim` descrito en la figura 1.9 se emplean como índices ciertas constantes numéricas definidas en `<asm-generic/resource.h>`. Estas constantes se corresponden con cada uno de los límites existentes y su nombre y descripción se encuentran en la figura 1.11.

1.3.4. Inicialización

Los límites toman su valor inicial durante la creación de `init`, la primera tarea del sistema. Después, son copiados por la función `fork()` en la creación de cada nuevo proceso. Por tanto, cada proceso hijo hereda los límites del proceso padre.

El valor inicial para todos los límites que presenta la figura 1.12 se encuentra en `<asm-generic/resource.h>`.

El formato que sigue la inicialización de cada límite es

`[LIMITE] = {VALOR_MIN, VALOR_MAX}`

Algunos de estos valores de `VALOR_MIN` y `VALOR_MAX` son, a su vez, constantes. La figura 1.13 muestra estas constantes, su valor y el archivo de cabeceras en las que han sido definidas. Cada una de estas constantes se declara en el ámbito con el que están directamente relacionadas. Por ejemplo, el tamaño máximo de una cola de mensajes que se emplea en el límite RLIMIT_MSGQUEUE está definido en `<linux/mqueue.h>`.

1.4. *procfs*

En los sistemas UNIX, *procfs* (abreviatura de *process file system*) es un sistema de archivos virtual (no está asociado a ningún dispositivo de

```
#define RLIMIT_CPU 0 /*tiempo CPU en ms*/
#define RLIMIT_FSIZE 1 /*máx tamaño fichero*/
#define RLIMIT_DATA 2 /*máx tamaño datos*/
#define RLIMIT_STACK 3 /*máx tamaño pila*/
#define RLIMIT_CORE 4 /*máx tamaño ficheros core*/
#define RLIMIT_RSS 5 /*máx tamaño páginas en memoria*/
#define RLIMIT_NPROC 6 /*máx núm de procesos creados*/
#define RLIMIT_NOFILE 7 /*máx núm ficheros abiertos*/
#define RLIMIT_MEMLOCK 8 /*máx tamaño memoria bloqueable*/
#define RLIMIT_AS 9 /*máx tamaño espacio direcciones*/
#define RLIMIT_LOCKS 10 /*máx núm bloqueos fichero*/
#define RLIMIT_SIGPENDING 11 /*máx núm señales pendientes*/
#define RLIMIT_MSGQUEUE 12 /*máx bytes en mqueues POSIX*/
#define RLIMIT_NICE 13 /*máx valor prioridad nice*/
#define RLIMIT_RTPRIO 14 /*máx prioridad tiempo real*/
#define RLIM_NLIMITS 15 /*núm total de límites*/
```

Figura 1.11: Definición y descripción de los límites

```
[RLIMIT_CPU] = {RLIM_INFINITY, RLIM_INFINITY}
[RLIMIT_FSIZE] = {RLIM_INFINITY, RLIM_INFINITY}
[RLIMIT_DATA] = {RLIM_INFINITY, RLIM_INFINITY}
[RLIMIT_STACK] = {_STK_LIM, _STK_LIM_MAX}
[RLIMIT_CORE] = {0, RLIM_INFINITY}
[RLIMIT_RSS] = {RLIM_INFINITY, RLIM_INFINITY}
[RLIMIT_NPROC] = {0, 0}
[RLIMIT_NOFILE] = {INR_OPEN, INR_OPEN}
[RLIMIT_MEMLOCK] = {MLOCK_LIMIT, MLOCK_LIMIT}
[RLIMIT_AS] = {RLIM_INFINITY, RLIM_INFINITY}
[RLIMIT_LOCKS] = {RLIM_INFINITY, RLIM_INFINITY}
[RLIMIT_SIGPENDING] = {0, 0}
[RLIMIT_MSGQUEUE] = {MQ_BYTES_MAX, MQ_BYTES_MAX}
[RLIMIT_NICE] = {0, 0}
[RLIMIT_RTPRIO] = {0, 0}
```

Figura 1.12: Límites de *init*

RLIM_INFINITY
representación de valor infinito dependiente de la arquitectura. Definido en <*arquitectura*/resource.h>.

MQ_BYTES_MAX
tiene un valor de 819200 bytes y está definido en <linux/mqueue.h>.

_STK_LIM
equivale a 8MB y está definido en <linux/resource.h>.

MLOCK_LIM
tiene un valor de 32MB definido en <*arquitectura*/resource.h>.

INR_OPEN
vale 1024 y está definido en <linux/fs.h>.

Figura 1.13: Constantes de valores de límites

bloques y sólo reside en memoria) que se monta en el directorio */proc*. Fue inicialmente diseñado para facilitar el acceso a la información sobre los procesos en un directorio propio identificado con el *PID* de cada uno. Existe un enlace simbólico */proc/self* que en todo momento apunta al proceso en ejecución.

Poco a poco se han incluido en */proc* otros elementos del hardware, de la configuración de módulos (ficheros */proc/modules*), elementos con información relevante del kernel, u otros que se refieren a todo el sistema (como */proc/meminfo*, que almacena las estadísticas de utilización de memoria). Hay que destacar que los archivos en */proc/sys* son archivos *sysctl*: contienen opciones del kernel que se pueden modificar dinámicamente, no pertenecen a *procfs* y son controlados de forma independiente.

procfs desempeña un papel de gran relevancia en la comunicación entre *espacio de usuario* y *espacio del kernel*. procfs permitirá que un comando como ps pueda obtener toda la información relativa a procesos en espacio de usuario, sin acceder a las estructuras del kernel.

Uno de los objetivos de este proyecto es incluir un fichero *rlimits* dentro de */proc/pid* para permitir el tratamiento de los límites como un fichero de texto legible desde espacio de usuario. De esta manera

permitimos que, por ejemplo, un usuario con los privilegios adecuados pueda acceder a los límites de un proceso desde el terminal en lugar de tener que implementar un programa que utilice las llamadas al sistema correspondientes. Los detalles de implementación correspondientes se encuentran en la sección 2.3.2.

Capítulo 2

Implementación

En esta sección se abordará la implementación de la nueva funcionalidad dentro del código del kernel. También se explicará la interfaz a través de la cual se proporciona al espacio de usuario. Aunque la implementación que realiza la operación de modificación propiamente dicha es única, existen dos maneras distintas de proporcionar la nueva funcionalidad al usuario: mediante llamadas al sistema o través del pseudosistema de ficheros *procfs*.

2.1. Introducción a la implementación

El objetivo que se persigue conseguir dentro del kernel de Linux es la incorporación de la funcionalidad que describe el título de este proyecto: la posibilidad de modificar los valores de los límites de un proceso de forma externa a éste, ya sea por parte del propio sistema operativo o de otro proceso que cumpla los requisitos de seguridad necesarios, es decir, que disponga de los permisos para poder realizar tal operación.

2.2. Código de modificación de los límites

El código que implementa la funcionalidad de consulta y modificación de los límites de un proceso cualquiera está inspirado en el de las llamadas `getrlimit()` y `setrlimit()` del kernel. Éstas se encuentran en el fichero `<kernel/sys.c>` raíz de la de las fuentes del kernel. Las cabeceras se muestran en la figura 2.1.

```
asmlinkage long sys_getrlimit(unsigned int resource,
                              struct rlimit __user *rlim)

asmlinkage long sys_setrlimit(unsigned int resource,
                              struct rlimit __user *rlim)
```

Figura 2.1: Cabeceras de las llamadas `getrlimit()` y `setrlimit()`

Estas llamadas se invocan desde el proceso que va cambiar sus límites. De sus dos parámetros, `resource` indica el código del límite a leer o escribir, y `*rlim` es un puntero a la estructura que almacenará el valor leído o el nuevo que se establecerá.

Nuestro objetivo es la implementación de dos nuevas funciones que hagan lo mismo que estas dos llamadas al sistema, pero que además permitan hacerlo para un proceso cualquiera (cuyo *pid* se pasará como parámetro) desde él mismo o desde otro proceso (siempre que tenga los permisos necesarios). Estas funciones las situaremos en `<kernel/sched.c>`. A partir de éstas, reimplementaremos las antiguas llamadas. Por ello, además de los dos parámetros de las llamadas al sistema anteriores, se introduce un nuevo parámetro de entrada `pid` que indica el número de proceso sobre el que tiene efecto la operación. Hay que prestar especial atención a los permisos a la hora de consultar y modificar el valor de los límites, ya que su incumplimiento implicaría un grave "agujero" de seguridad.

Las antiguas llamadas `getrlimit()` y `setrlimit()` de la figura 2.1, implementan el código de modificación dentro del propio cuerpo de la llamada. Como se puede observar el parámetro `rlim` apunta a datos pertenecientes al espacio de usuario, por tanto hay que copiarlos al espacio del kernel antes de realizar ninguna operación con ellos (mediante las funciones `copy_to_user()` y `copy_from_user()`) lo que impide que se puedan utilizar directamente desde el código del kernel. Es por ello que el código de las dos nuevas llamadas que vamos a incorporar no se encuentra en el cuerpo de la función de las mismas, sino en las funciones `do_getprlimit()` y `do_setprlimit()` (figura 2.2), para poder ser utilizadas por el propio kernel. Se puede observar que en éstas el parámetro `*rlim` no proviene del espacio de usuario. El código de estas funciones puede ser consultado en el apéndice D.

2.2.1. Políticas de modificación

La modificación del valor de los límites de un proceso plantea una serie de problemas a los que se les pueden aplicar diverso número de soluciones. Así por ejemplo, ¿qué hacer cuando se limita el número de ficheros abiertos por un proceso a un determinado valor cuando la cantidad actual de ficheros abiertos excede dicho valor?. En este caso podríamos adoptar diversas soluciones. La más sencilla sería, sin duda, prohibir la aplicación de ese nuevo valor del límite. Pero puesto que la modificación de límites es una operación excepcional y puede que de ella dependa una situación crítica, también podríamos considerar como prioritario obligar al proceso a cerrar ficheros hasta que cumpla con el nuevo valor del límite, o incluso finalizar el propio proceso. La solución adoptada en este tipo de situaciones es lo que denominamos políticas de modificación. Las hemos elegido basándonos, principalmente, en su interés experimental y asumiendo que siempre debe forzarse el cumplimiento del nuevo valor del límite. A continuación se presenta una breve descripción de estas políticas. En la sección 2.2.3 se da una explicación más detallada de la implementación de cada una de ellas.

Límite máximo de tiempo de CPU en segundos (RLIMIT_CPU)
Cuando se establece el valor del límite a un valor inferior al del tiempo de CPU (tiempo de CPU en modo kernel más tiempo de CPU en modo usuario) permitido para el proceso, se establece el valor de éste al del nuevo límite.

Límite máximo de ficheros abiertos (RLIMIT_NOFILE)
Cuando se establece el valor del límite a un valor inferior al del número de ficheros actualmente abiertos por el proceso, se cierran los n últimos, donde n es la diferencia entre el número total de ficheros abiertos y el nuevo valor del límite máximo.

```
asmlinkage long do_getprlimit(pid_t pid,
      unsigned int resource, struct rlimit *rlim)

asmlinkage long do_setprlimit(pid_t pid,
      unsigned int resource, struct rlimit *rlim)
```

Figura 2.2: Cabeceras de las funciones `do_getprlimit()` y `do_setprlimit()`

Tamaño máximo de un fichero abierto (RLIMIT_FSIZE)
Cuando se establece el valor del límite a un valor inferior al del tamaño de algún fichero abierto por el proceso, se cierran todos aquellos de tamaño superior al especificado por el nuevo límite.

Valor de *nice* más prioritario posible (RLIMIT_NICE)
Cuando se establece el valor del límite a un valor menos privilegiado que el actual, se degrada la prioridad del proceso hasta igualarla a la del valor del límite.

Tamaño máximo del segmento de datos del proceso (RLIMIT_DATA)
Cuando se establece el valor del límite a un valor inferior al del actual tamaño del segmento de datos, se reduce el tamaño de éste al especificado por el límite.

2.2.2. Implementación de `do_getprlimit()`

Como se ha visto con anterioridad, la función `do_getprlimit()` devuelve el valor del límite especificado en sus parámetros en una estructura `rlimit` (ver figura 1.10). Para llevar este proceso a cabo realiza los siguientes pasos:

1. Comprueba que el *pid* y el código de identificación del límite son válidos. En caso contrario devuelve un error de argumentos no válidos.

2. Adquiere un puntero a la estructura `task_struct` del proceso identificado por el parámetro `pid`. Si no tiene éxito devuelve un error que indica la no existencia del proceso.

3. Comprueba si el usuario del proceso que realiza la llamada tiene permisos de lectura de los límites. Esta comprobación se realiza mediante la función `pr_lim_check_perm()` de la figura 2.3, implementada también en `<kernel/sched.c>`.

4. Una vez que se han comprobado los permisos se procede a leer el valor del límite del array `rlimit`, que se encuentra en la estructura `signal_struct` (ver figura 1.9) del `task_struct` (ver figura 1.1) del proceso.

2.2.3. Implementación de `do_setprlimit()`

La función `do_setprlimit()` establece el valor del límite especificado al del parámetro `rlim` pasado a la función. Para ello realiza los siguientes pasos.

1. Comprueba que el *pid* y el código de identificación del límite son válidos. También comprueba que el valor del nuevo *límite blando* no excede el valor del nuevo *límite duro*. En caso contrario devuelve un error de argumentos no válidos.
2. Adquiere un puntero a la estructura `task_struct` del proceso identificado por el parámetro `pid`. Si no tiene éxito devuelve un error que indica la no existencia del proceso.
3. Comprueba si el usuario del proceso que realiza la llamada tiene permisos de escritura sobre los límites mediante la función `pr_lim_check_perm()` de la figura 2.3.
4. Lee el valor actual del límite y si se está intentando aumentar el *límite duro* comprueba si ésto es posible.
5. Se asigna el nuevo valor del límite al proceso.
6. Finalmente se aplican las políticas de modificación descritas en el apartado 2.2.1 y cuya implementación se detalla a continuación:

   RLIMIT_CPU
   Comprueba que el nuevo valor no es infinito. Si no es así mira

```
static inline int prlim_check_perm(task_t *task) {
  return ((current->uid == task->euid) &&
      (current->uid == task->suid) &&
      (current->uid == task->uid) &&
      (current->gid == task->egid) &&
      (current->gid == task->sgid) &&
      (current->gid == task->gid)) ||
      capable(CAP_SYS_PTRACE);
}
```

Figura 2.3: Función `prlim_check_perm()`

si el valor establecido en el campo `it_prof_expires` del campo `signal` (de tipo `struct signal_struct`) es cero (infinito) o mayor que el nuevo límite máximo. Este campo indica el total de tiempo de CPU (modo usuario más modo kernel) permitido para el proceso. En el supuesto que sea mayor que el nuevo valor del límite se cambia a este último mediante la función `set_process_cpu_timer()`. A esta función se le pasa el identificador del temporizador al que vamos a cambiar el valor, en este caso `CPUCLOCK_PROF`.

RLIMIT_NOFILE

Obtiene el número de ficheros abiertos por el proceso en la variable `files_open`. A continuación mira si el nuevo valor del límite es más pequeño que `files_open`. En ese caso procede a cerrar los `n` últimos archivos abiertos apuntados por el array de punteros a `struct file` denominado `fd` del campo `fdt`, que a su vez es un campo del campo `files` del `task_struct` del proceso. La cantidad `n` es la diferencia entre el número de ficheros abiertos y el nuevo valor del límite, es decir, `n = files_open - rlim->rlim_cur`. Cabe hacer notar que los ficheros correspondientes a los `fd` 0, 1 y 2 no se deben cerrar nunca, ya que se corresponden con la E/S estándar.

RLIMIT_FSIZE

Recorre el array de punteros `fd` mediante un bucle, leyendo del `inode` de cada fichero el tamaño que ocupa en bytes. En el caso de que éste exceda el tamaño establecido por el valor del nuevo límite, lo cierra. Como en el caso anterior, ésto no afecta a los ficheros de la E/S estándar.

RLIMIT_NICE

Lee el valor del nice del proceso y lo guarda en la variable `nice`. A continuación calcula el valor mínimo permitido del nice, con el nuevo límite, en la variable `min_nice`. Normaliza `min_nice` para que no se salga del rango (-20, +19) y lo compara con `nice`. En el caso de que sea mayor se establece el valor del nice del proceso al valor de `min_nice` haciendo uso de la función `set_one_prio()`.

RLIMIT_DATA

El primer paso es conseguir una referencia al campo `mm` de la estructura `task_struct` del proceso. `mm` es una estructura `mm_struct` del mapa de memoria del proceso, una de las más importantes del kernel. Después calcula la dirección máxima donde finalizaría el segmento de datos con el nuevo valor del límite. Ésta se calcula en la variable `new_brk`, sumando la

longitud máxima a la dirección inicial del segmento de datos (especificada por el campo `start_data` de `mm`). Finalmente compara la dirección actual donde finaliza el segmento con la máxima calculada en `new_brk`, y en el caso de que la primera sea mayor la establece a la segunda valíendose de la función `sys_brk()`.

2.3. Aplicación desde el espacio de usuario

En el apartado anterior hemos visto como se implementaban las nuevas funciones (figura 2.2) que van a permitir consultar y modificar el valor de los límites de un proceso dado. Estas funciones pertenecen al espacio del kernel, por lo que se pueden utilizar desde el código de éste sin restricciones.Pero para poder utilizar las funciones de modificación de los límites desde el espacio de usuario tenemos que proporcionarlas a través de una interfaz adecuada.

2.3.1. Llamadas al sistema

Aquí se describe la forma en que se implementa la interfaz de las llamadas al sistema, que proporciona a las aplicaciones a nivel de usuario la capacidad de consulta y modificación de los límites de los procesos, funcionalidad implementada a nivel de kernel. En las figuras 2.4 y 2.5 vemos el código de las funciones de estas llamadas, que hemos incorporado en `<kernel/sys.c>`. El código es sencillo: llaman a las funciones que realizan las operaciones sobre los límites (figuras 2.4 y 2.5) y pasan la información al/desde el espacio del kernel del parámetro `rlim` mediante las funciones `copy_from_user()` y `copy_to_user()` respectivamente.

De forma similar reimplementamos las llamadas `getrlimit()` y `setrlimit()` como se puede ver en las figuras 2.6 y 2.7.

2.3.2. Interfaz en *procfs*

Una forma alternativa a las llamadas al sistema de acceder a la información del kernel es el pseudosistema de ficheros `/proc` . Este pseudosistema sigue la filosofía UNIX de que “todo es un fichero” para

```
asmlinkage long sys_getprlimit(pid_t pid,
      unsigned int resource, struct rlimit __user *rlim) {
  struct rlimit value;
  int retval;

  retval = do_getprlimit(pid, resource, &value);

  if (copy_to_user(rlim, &value, sizeof(value))) {
    retval = -EFAULT;
  }
  return retval;
}
```

Figura 2.4: Código de la llamada `getprlimit()`

```
asmlinkage long sys_setprlimit(pid_t pid,
      unsigned int resource, struct rlimit __user rlim) {
  struct rlimit new_rlim;

  if(copy_from_user(&new_rlim, rlim, sizeof(rlim))) {
    return -EFAULT;
  }
  else {
    return do_setprlimit(pid, resource, &new_rlim);
  }
}
```

Figura 2.5: Código de la llamada `setprlimit()`

permitir la comunicación entre el espacio de usuario y el del kernel. Dentro de `/proc` existe un subdirectorio para cada proceso denominado con el PID de dicho proceso, en el que hemos incluido un fichero de nombre `rlimits` para consultar y modificar los límites. Existen dos operaciones asociadas con este fichero: leer y escribir. Cuando leemos del fichero el contenido que obtenemos es una lista de todos los límites y el valor asociado a éstos para el proceso. La escritura es un poco más complicada, ya que hay que usar un formato determinado para conseguir el resultado deseado, de lo contrario se producirá un error de escritura. Lo que tratamos de conseguir con la escritura es modificar

```
asmlinkage long sys_getrlimit(unsigned int resource,
      struct rlimit __user *rlim) {
  struct rlimit value;
  int retval;

  retval = do_getprlimit(0, resource, &value);

  if (copy_to_user(rlim, &value, sizeof(value))) {
    retval = -EFAULT;
  }

  return retval;
}
```

Figura 2.6: Código de la llamada `getrlimit()`

el valor de un determinado límite para el proceso que se corresponde con el directorio donde se encuentra el fichero `rlimits` que tratamos de escribir. Definimos ambas operaciones de la siguiente forma:

Lectura. La operación de lectura consiste simplemente en leer el contenido del fichero `/proc/`*`pid`*`/rlimits`.

Escritura. La operación de escritura modifica el valor del límite indicado. Para ello hay que escribir una cadena de texto en el fichero `/proc/`*`pid`*`/rlimits` con el siguiente formato:

"NUM SOFT HARD" ó "NUM SOFT"

Donde NUM es el número del límite, SOFT el nuevo valor para el *límite blando* y HARD el nuevo valor para el *límite duro*.

En las figuras 2.9 y 2.8 se muestra un ejemplo de su uso, utilizando los comandos `cat` y `echo` para leer y escribir el valor del límite que específica el número máximo de ficheros abiertos desde la consola.

Ésta interfaz permite consultar modificar el valor de los límites directamente por el usuario sin necesidad de un programa específico para ello que haga uso de las llamadas al sistema. También puede ser usado por programas que actúen como un *frontend* realizando las lecturas y escrituras sobre el fichero `rlimits` de manera transparente.

```
asmlinkage long sys_setrlimit(pid_t pid,
      unsigned int resource, struct rlimit __user *rlim) {
  struct rlimit new_rlim;

  if(copy_from_user(&new_rlim, rlim, sizeof(*rlim))) {
    return -EFAULT;
  }
  else {
    return do_setprlimit(0, resource, &new_rlim);
  }
}
```

Figura 2.7: Código de la llamada `setrlimit()`

La implementación del subsistema *procfs* se encuentra en el directorio `<fs/proc>` de la raíz de las fuentes. La parte principal, que es donde se ha implementado la la nueva funcionalidad, se encuentra en el fichero `base.c` de dicho subdirectorio. Lo primero que hay que hacer en `base.c` es registrar los *inumber* para la nueva entrada `rlimits`. Esto se consigue añadiendo a la definición del tipo enumerado `pid_directory_inos` los valores PROC_TGID_RLIMITS y PROC_TID_RLIMITS. El pri-

Figura 2.8: Escritura de `rlimits` en *procfs*

Figura 2.9: Lectura de `rlimits` en *procfs*

mero identifica la entrada `rlimits` con un *tgid* (grupo de tareas) y el segundo con un *tid* (tarea individual, se corresponde con el *pid*). Una vez registrados los *inumber* el siguiente paso es utilizarlos para crear el fichero `rlimits`. Esto se consigue añadiendo a la inicialización de `tgid_base_stuff[]` los siguientes valores:

```
E(PROC_TGID_RLIMITS, ‘‘rlimits’’, S_IFREG|S_IRUGO)
E(PROC_TID_RLIMITS, ‘‘rlimits’’, S_IFREG|S_IRUGO)
```

El paso siguiente es registrar las operaciones sobre el fichero `rlimits`. Éstas se reducen a la lectura y la escritura. Las operaciones de las entradas del `/proc/`*`pid`*`/` se definen en la función de la figura 2.10.

Cada operación se implementa en su respectiva función, aunque el mecanismo por el que se registra la función de escritura y la de lectura es distinto.

Las operaciones se registran en el campo `i_fop` de la variable `inode` dentro de la función de la figura 2.10. El campo `i_fop` es un puntero a un tipo `struct file_operations`. La estructura instanciada y apuntada por este campo es la de la figura 2.11.

El registro de las operaciones se realiza mediante el código mostrado en la figura 2.12. Aquí se puede ver como se registra la estructura de la figura 2.11 en el *inode* de la entrada `rlimits`. También se observa como se registra la función `proc_pid_read_rlimits` en la estructura

```
static struct dentry *proc_pident_lookup(struct inode *dir,
      struct dentry *dentry, struct pid_entry *ents)
```

Figura 2.10: Función donde se registran las operaciones de `rlimits`

```
static struct file_operations proc_rlimits_operations = {
  .read = proc_info_read,
  .write = proc_pid_write_rlimits,
};
```

Figura 2.11: Instancia de las estructura `file_operations`

`proc_inode`, apuntada por `ei`.

```
[...]
case PROC_TID_RLIMITS:
  inode->i_fop = &proc_rlimits_operations;
  ei->op.proc_read = proc_pid_read_rlimits;
  break;
case PROC_TGID_RLIMITS:
  inode->i_fop = &proc_rlimits_operations;
  ei->op.proc_read = proc_pid_read_rlimits;
  break;
[...]
```

Figura 2.12: Registro de las operaciones de rlimits

La función de lectura `proc_info_read` tiene la cabecera mostrada en la figura 2.13. Lo único que hace es copiar el parámetro `*buf` en el espacio del kernel y obtener el `struct task_struct` correspondiente de la tarea (guardándolo en la variable `task`). Con estos valores llama a la función de lectura registrada en el `struct proc_inode` de la entrada `rlimits`.

La función de lectura registrada en el `struct proc_inode` es `proc_pid_read_rlimits`, cuya cabecera podemos ver en la figura 2.14. Esta función es la que realiza realmente la operación de lectura de los límites. Su parámetro `task` apunta a la `struct task_struct` del proceso y se

```
static ssize_t proc_info_read(struct file * file,
      pchar __user * buf, size_t count, loff_t *ppos)
```

Figura 2.13: Función `proc_info_read()`

usa para leer los límites; el parámetro `buffer` es la cadena de texto que devolverá el fichero al ser leído. Por tanto lee los valores de los límites del primero y los escribe, ya formateados, en el segundo.

```
int proc_pid_read_rlimits(struct task_struct *task,
      char *buffer)
```

Figura 2.14: Función `proc_pid_read_rlimits()`

`proc_pid_read_rlimits` también se registra en la función `proc_pident_lookup` (figura 2.10), en la variable `ei` que es un puntero al `proc_inode` de la entrada `rlimits`.

2.4. Pruebas y depuración

Una de las principales dificultades que se nos plantea a la hora de escribir código para el kernel es la imposibilidad de depurarlo mediante ningún tipo de herramienta. Ésto, unido al hecho de la inexistencia de ningún tipo de protección de memoria, dificulta enormemente la fase de pruebas y de solución de bugs. A pesar de ello existen una serie de procedimientos que pueden ayudar a la hora de descubrir y solucionar errores:

- La primera y más importante práctica que se debe adoptar es ser muy cuidadoso a la hora de escribir el código. Esto puede parecer algo de sentido común, pero la mayor parte de los errores cometidos son consecuencia de la falta de atención durante el desarrollo. Hay que tener en cuenta que los errores en el kernel suponen un esfuerzo mucho mayor a la hora de solucionarlos que las aplicaciones a nivel de usuario, y por tanto es imprescindible un doble esfuerzo a la hora de implementar para minimizarlos. Es necesario

tener las ideas claras, saber lo que se está haciendo exactamente en cada momento y repasar el código antes de compilar y ejecutar.

- A falta de herramientas podemos valernos de la función `printk()` para imprimir mensajes y poder realizar cierta "depuración".

- Si se implementa la funcionalidad en un módulo externo en lugar de directamente en el código del kernel (al menos durante la fase de desarrollo y depuración), ésto permitirá cargar y descargar los cambios en tiempo de ejecución, evitando la necesidad de reiniciar el equipo en cada nueva compilación (excepto, claro está, cuando se produzca un error crítico que produzca un cuelgue).Desgraciadamente ésto sólo suele ser posible cuando se están escribiendo drivers o añadiendo funciones nuevas, y no cuando se están modificando partes críticas y estructuras del kernel (como es el caso).

- Es recomendable disponer de un equipo de pruebas conectado en red al de desarrollo. De esta manera se evita tener que parar y reiniciar el equipo sobre el que se está trabajando para realizar las pruebas. Si no se dispone de uno para poder dedicarlo a las pruebas una gran alternativa es usar un software de *virtualización*, como QEmu o VMWare.

  Nosotros hemos optado por esta última alternativa, realizando todas las pruebas sobre una máquina virtual VMWare corriendo una instalación mínima de Debian 3.1. La comunicación entre la máquina real y la virtual se lleva a cabo mediante un servidor Samba que comparte un directorio en la máquina real al que accede la otra mediante una conexión virtual de red. La figura 2.15 muestra una captura de la máquina virtual en funcionamiento sobre el equipo de desarrollo.

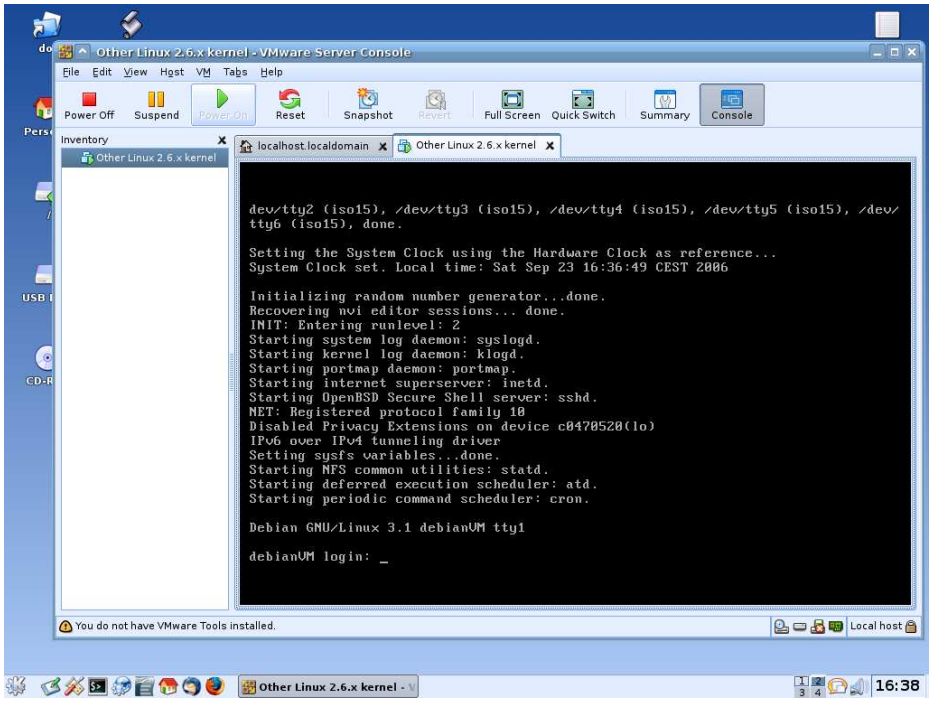

Figura 2.15: Captura de VMWare ejecutando Debian 3.1

Capítulo 3

Qlimits

3.1. Introducción

Qlimits es una pequeña aplicación que hemos realizado como parte del proyecto. Mediante una intuitiva interfaz, nos permite gestionar de manera eficiente los límites de todos los procesos en ejecución del sistema. Está programada en *C++* utilizando la biblioteca *Qt4*, y consiste básicamente en una aplicación gráfica que se vale de las llamadas `getprlimit()` y `setprlimit()` para la administración de límites.

3.2. Descripición

Qlimits es un programa sencillo, que tan sólo consta de 2 ventanas. La ventana principal muestra todos los procesos en ejecución, junto con una información básica de cada uno (PID, propietario, ...). Tras elegir el proceso del cual queremos consultar o modificar los límites, pasamos a una segunda ventana. Aquí se nos muestra el valor de todos los límites del proceso seleccionado, y nos permite modificarlos libremente.

En cualquier caso Qlimits impone ciertas restricciones a la hora de modificar los límites de un proceso, de modo que el sistema siga siendo estable. Dichas restricciones son las siguientes:

- El valor del límite ha de ser un número entero mayor o igual que -1. El valor -1 corresponde a infinito.

- No permitimos que el *límite blando* sea mayor que el *límite duro* correspondiente. Esta es una restricción que impone la función *setrlimit*.

Las figuras 3.1 y 3.2 muestran la aplicación en ejecución.

3.3. Implementación

La implementación de Qlimits está dividida en dos partes. Por un lado, las clases `MainWindow` y `ProcessDialog` implementan el interfaz con que interactuará el usario. Por otro, `ProcessProperties` proporciona métodos auxiliares para obtener la información necesaria sobre cada proceso.

3.3.1. MainWindow

La clase `MainWindow` es la encargada de mostrar la ventana principal de la aplicación. Se encarga de mostrar una lista con todos los procesos en ejecución, junto con información básica de cada uno. Permite que el usuario seleccione uno de dichos procesos, y llama a `ProcessDialog`. Su diseño está inspirada en la del aministrador de tareas KSysguard. Utiliza los *widgets* predefinidos en *Qt* para botones, etiquetas y tablas, y nuestra clase `ProcesProperties` para obtener las propiedades de cada proceso.

3.3.2. ProcessDialog

La clase `ProcessDialog` muestra los límites del proceso seleccionado y permite modificarlos, actuando como un interfaz gráfico de las llamadas `getprlimits` y `setprlimits`. Al iniciarse o pulsar el botón de actualizar utiliza repetidamente nuestra `getprlimits` para averiguar el valor de todos los límites. Cuando queremos cambiar el valor de un límite, comprueba que el nuevo valor sea válido y llama a `setprlimits` para efectuar la operación. A continuación informa del resultado, así como de cualquier error que haya podido producirse,

3.3.3. ProcessProperties

La clase `ProcessProperties` obtiene información relevante sobre un proceso determinado a partir de su *PID*. Esto incluye:

- Nombre del proceso.
- Propietario del proceso.
- Orden completa con la que se lanzó el proceso (si lo hay)
- Valor `nice`,la prioridad de planificación.
- La cantidad total de memoria virtual (en kBytes) que usa el proceso.
- La cantidad total de memoria física (en kBytes) que usa el proceso.

Para obtener estos datos hace uso tanto de llamadas al sistema como de los ficheros que cada proceso proporciona en su entrada del `/proc` para mostrar su estado.

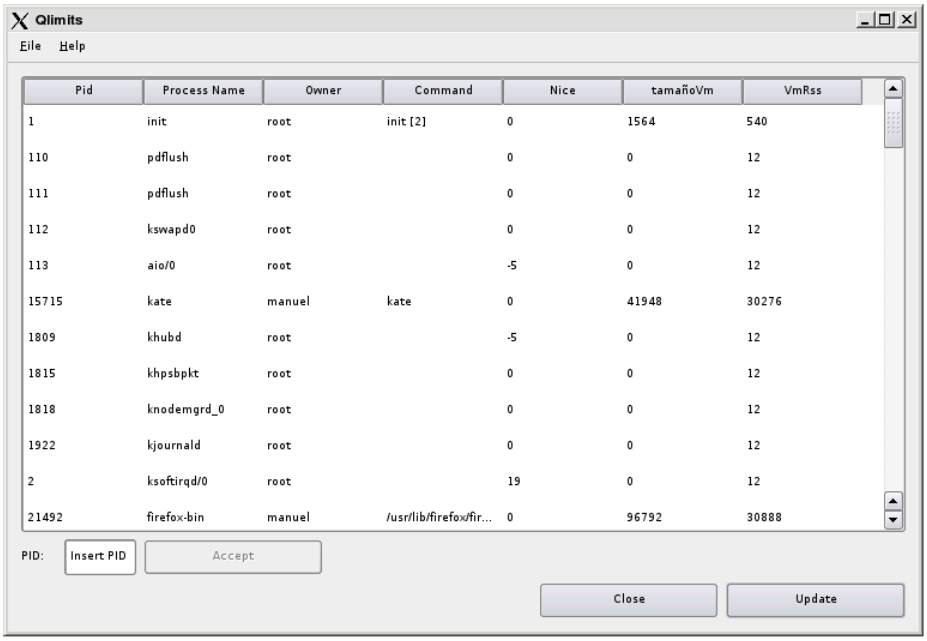

Figura 3.1: Ventana principal de *Qlimits*

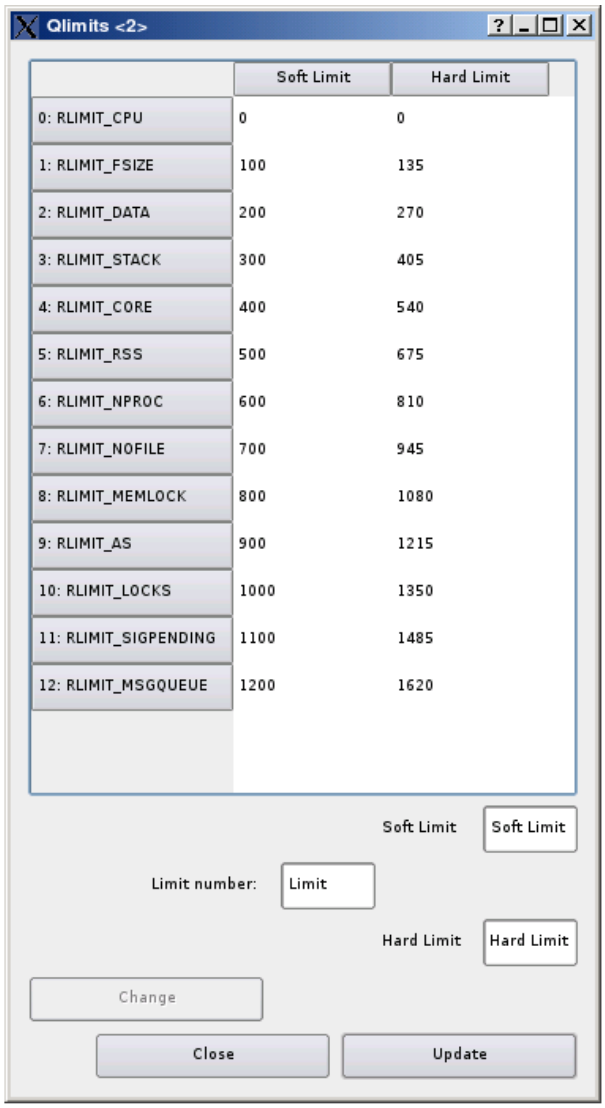

Figura 3.2: Ventana de modificación de límites de *Qlimits*

Apéndice A

Generación y aplicación de parches

Este apéndice describe el procedimiento para generar y aplicar parches sobre ficheros de texto. Éste es el método habitual por el que se distribuyen las modificaciones sobre el código fuente del kernel de Linux.

A.1. Introducción

Cuando se realiza algún cambio sobre el código de cualquier programa la mejor forma de distribuir estos cambios es a través de un parche. Los parches son ficheros de diferencias que registran los cambios existentes entra las antiguas versiones de los ficheros y las nuevas modificadas. Normalmente se guardan con extensión `patch` o `diff`. Los parches, obviamente, no sólo son aplicables al código fuente, sino que a cualquier fichero en formato de texto plano.

A.2. Generación de parches

La generación de parches se realiza utilizando el programa `diff`. Para realizar un parche para todo el contenido de un directorio tan sólo hay que escribir lo siguiente:

```
diff -Naur old_dir new_dir > nombre_del_parche
```

Esto generará un parche con los cambios realizados en `new_dir` respecto de `old_dir` y los guardará en el parche `nombre_del_parche` al que se ha redirigido la salida estandar mediante el operador `>`.

La función de las opciones utilizadas se describe a continuación.

-N Incluye los ficheros nuevos.

-a Procesa los ficheros como de texto.

-u Indica que se debe usar el formato unificado.

-r Recorre los directorios recursivamente.

Para más información acerca del resto de las opciones del comando `diff` consultar su correspondiente página del manual (`man diff` desde la consola).

A.3. Aplicación de parches

Para aplicar un parche hay que utilizar el siguiente comando desde el directorio que se quiere parchear:

```
patch -pnum < nombre_del_parche
```

La opción `-p`*num* indica el nivel de diretorio a partir del cual se quiere aplicar el parche. Por ejemplo, si se ha generado el parche de la manera expuesta en el apartado anterior, si parcheamos el directorio desde dentro del mismo directorio tendremos que usar la opción `p1`.

Apéndice B

Compilación del kernel 2.6

En este apéndice se describe el procedimiento para compilar las fuentes de un kernel 2.6.

B.1. Descarga de las fuentes

El primer paso a realizar es la descarga del código fuente. Éste puede descargarse de la página oficial del kernel de Linux (www.kernel.org). El código se puede bajar comprimido tanto en formato *tar.gz* como en *tar.bz2*. Elegimos el último formato por su mayor tasa de compresión. Una vez hecho esto lo descomprimimos en el directorio donde lo hemos descargado mediante el siguiente comando:

`tar xvjf linux-2.6.`*x*`.tar.bz2`

Al finalizar la descompresión del fichero descargado tendremos un directorio denominado `linux-2.6.`*x* en el que se encuentra la raíz del árbol de directorio de las fuentes.

B.2. Configuración

Ésta es la parte más importante a la hora de compilar un kernel nuevo. En este paso es donde se eligen las características y controladores que queremos incluir en nuestro kernel. No es un proceso muy complicado, pero exige un conocimiento mínimo de las características de la máquina y una ídea clara de las opciones que queremos tener incluidas en el kernel.

Existen varios métodos para configurar el kernel. Existen varias interfaces para configurar el kernel. A las tres principales se accede mediante los comandos `make config`, `make menuconfig` y `make xconfig`. La primera es la forma menos sofisticada y presenta un interfaz en texto bastante rudimentario. El segundo método es el favorito para usar en la consola en modo texto y consiste en un interfaz de menús que usa las librerías *ncurses*. El tercer método es el más fácil de usar y es un interfaz gráfico que funciona bajo *X-Window* implmentado mediante las librerías *Qt*. En los tres casos se os presenta una lista de los componentes que queremos tener incluidos en el kernel clasificados por categorías. Para la mayor parte de ellos se nos dan tres opciones: no incluirlo, incluirlo estáticamente en la imagen principal del kernel o incluirlo como módulo externo que se puede cargar y descargar en tiempo de ejecución. Normalmente hay ciertas características, como el controlador de disco por ejemplo, que deben estar incluidas directamente en la imagen para poder arrancar el sistema operativo. El resto es recomendable tenerlas como módulos independientes, ya que ésto reducirá el tamaño de la imagen principal.

Una vez hayamos quedado satisfechos con la configuración realizada, salvaremos los cambios y saldremos de la configuración. Ésta quedará guardada en el fichero `.config` del directorio raíz de las fuentes. Es recomendable realizar una copia de seguridad de éste fichero para ecitar perder y poder recuperar la configuración en el futuro.

B.3. Compilación

Tras el proceso de configuración llegamos al de compilación. El primer paso a realizar (aunque no obligatorio) es eliminar los ficheros objeto generados en los posibles procesos de compilación anteriores. Esto se realiza ejecutando desde la ráiz de las fuentes el siguiente comando:

```
make clean
```

A continuación para compilar la imagen principal escribimos los siguiente:

```
make bzImage
```

Una vez finalizado el proceso de compilación la imagen se guarda en el fichero `</arch/`*`arquitectura`*`/boot/bzImage>`. Ya sólo queda compilar los módulos con

```
make modules
```

e instalarlos con

```
make modules_install
```

con permisos de superusuario.

Después de realizar todos lo anterior ya sólo queda configurar el gestor de arranque instaldao para que inicie el sistema con la nueva imagen recién compilada. Para las instrucciones de como hacer ésto consultar la documentación correspondiente.

Apéndice C

Librerías gráficas *Qt*

C.1. Introducción

Qt es una biblioteca multiplataforma para desarrollar interfaces gráficas de usuario. Fue creada por la compañía noruega Trolltech. *Qt* es utilizada en KDE, un entorno de escritorio para sistemas como Linux o FreeBSD, entre otros. Utiliza el lenguaje de programación C++ de forma nativa y además existen bindings para C, Python, Perl y Ruby entre muchos.

Qt utiliza un enfoque "escribe una vez, compila en todos los sitios". Permite, utilizando el mismo código fuente, crear programas que correrán en Windows 95 hasta XP, Mac OS X, Linux, Solaris, HP-UX y otras versiones de Unix con X11. También hay una versión de *Qt* para sistemas linux empotrados, que utiliza la misma interfaz.

C.2. Historia

En 1990, Haarvard y Eirik estaban trabajando juntos en una base de datos en C++ para imágenes de ultrasonidos. El sistema necesitaba correr con una interfaz gráfica en Unix, Macintosh y Windows. Así pues, tras acabar el proyecto se pusieron a desarrollar el kit de desarrollo que más tarde se convertiría en *Qt*. En 1991 se añadió el paradigma de *"signals and slots"* ("señales y ranuras"), y en 1993 ya eran capaces de implementar sus primeros *widgets* . Al final del año decidieron fundar la compañía Trolltech para construir "el mejor kit de desarrollo para interfaces gráficas en C++".

El prefijo elegido para las clases, 'Q', tiene un origen curioso: lo eligieron porque esa letra era bonita en el tipo de letra que usaba Haavard. La letra 't' es de *toolkit* palabra inglesa que significa *kit de desarrollo*. Y así, en mayo de 1995, subieron *Qt* 0.9 a *sunsite.unc.edu* y *comps.os.linux.anounce*.

Qt podía ser utilizada en Unix y Windows con la misma *API*. Estaba disponible con dos licencias: una comercial, necesaria para desarrollos comerciales, y una gratuita para desarrollos de software libre. Durante los 10 primeros meses nadie compró una licencia comercial, hasta que en 1996 la Agencia Espacial Europea se convirtió en su primer cliente. Al final de año tenían 8 clientes. Desde ese momento, las ventas de Trolltech se han duplicado anualmente, y a día de hoy, la compañía tiene más de 4.000 clientes de 60 países distintos.

Paralelamente al desarrollo comercial, desde 1997 las *Qt* han sido utilizadas para construir en entorno de escritorio KDE. Esto ha convertido a *Qt* en el estándar de facto para interfaces gráficas con C++ en Linux.

En 1998 Matthias Ettrich, jefe del proyecto KDE, comenzó a trabajar en Trolltech. El año siguiente fue lanzado *Qt* 2.0, con muchos cambios estructurales, más potente y con una naturaleza más madura que su predecesor. La licencia que lo acompañaba, Q Public Licence (QPL) encajaba en la Open Source Definition. Ese mismo año, *Qt* ganó el premio de LinuxWorld a la mejor librería/programa.

En el año 2000 Trolltech lanzó *Qt*/Embedded, estaba diseñado para correr en dispositivos Linux empotrados. También proporcionaron su propio sistema de ventanas como un remplazo ligero de las X11. Tanto *Qt*/Embedded como *Qt*/X11 fueron ofrecidos bajo licencia GPL (además de las licencias comerciales).

En el año 2001 se lanzó *Qt*3. Esta vez estaba disponible para Windows, Unix, Linux, Linux empotrado y Mac OS X, todas bajo una doble licencia GPL/comercial, con lo que definitivamente se granjeó los afectos de la comunidad Linux. *Qt*3 proporcionaba 42 clases nuevas, y el código superó las 500.000 líneas.

Finalmente, en el año 2005 Trolltech lanzó *Qt*4, precedida de una gran expectación. Esta versión incluye nuevas tecnologías y funcionalidades, además de una renovación de lo existente. Ahora, se han simplificado algunas de las tareas habituales, haciendo más cómoda la programación. Además, se ha incluido soporte para Mac OS X con los nuevos procesadores Intel. Esta es la versión que utilizamos en nuestro proyecto.

C.3. Características

C.3.1. Características generales

Qt incluye un amplio conjunto de *widgets* ("controles" en la terminología de Windows) que proporcionan una funcionalidad estándar de la GUI. Introduce una innovadora alternativa para la comunicación entre objetos, llamada *"signals and slots"*, que sustituye la anticuada e insegura técnica de llamadas usada en la mayoría de los frameworks. Además, *Qt* proporciona un modelo de eventos convencional para gestionar clicks de ratón, teclas pulsadas y el resto de entradas de usuario. Las aplicaciones gráficas de *Qt* interplataforma soportan todas las funcionalidades requeridas por las aplicaciones modernas, tales como menús, menús contextuales, arrastrar y soltar y barras de herramientas.

Qt tiene un soporte excelente para gráficos 2D y 3D. De hecho, es el estándar de facto para la la programación con OpenGL. Su sistema de dibujo ofrece un renderizado de alta calidad en todas las plataformas soportadas.

Qt permite crear bases de datos independientes de la plataforma usando bases de datos estándar. Así, incluye drivers nativos para (entre otros) Oracle®, Microsoft® SQL Server, Sybase® Adaptive Server, IBM DB2®, PostgreSQLTM y MySQL®.

Los programas de *Qt* adoptan el aspecto nativo en todas las plataformas soportadas utilizando los estilos y temas de *Qt*. Con el mismo código fuente, basta con recompilarlo para producir aplicaciones para Windows, Linux, Mac OS X y demás. El compilador de *Qt* se encarga de generar los *Makefiles* o ficheros .dsp adecuados para la plataforma objetivo. Puede verse un ejemplo en la figuras C.1, C.2 y C.3, que muestra una misma ventana con las apariencias típicas de Linux, OS X y Windows XP.

Qt utiliza Unicode TM y tiene un soporte considerable para internacionalización. *Qt* incluye el *Qt Linguist* y otras herramientas para permitir la utilización de traductores. Las aplicaciones pueden fácilmente usar y mezclar texto en árabe, chino, inglés hebreo, japonés y el resto de los idiomas soportado por Unicode.

Qt incluye una gran variedad de clases apropiadas para dominios específicos. Por ejemplo, tiene un módulo XML que incluye clases SAX y DOM para leer y manipular datos almacenados en formatos basados en XML. Los objetos se pueden guardar en la memoria, y ser manejados

utilizando iteradores como los presentes en Java y las librerías estándar de C++ (STL). Las librerías de entrada/salida y trabajo en red de *Qt* te proporcionan la posibilidad de trabajar tanto con ficheros locales como remotos.

Las aplicaciones de *Qt* pueden extender su funcionalidad utilizando plugins y librerías dinámicas. Estos te proporcionan *códecs* adicionales, *drivers* para bases de datos, formatos de imagen, estilo y *widgets*. Cada plugin puede tener su propia licencia y ser vendido como un producto independiente.

Figura C.1: Vista de ventana con apariencia Windows XP

Figura C.2: Vista de ventana con apariencia Aqua

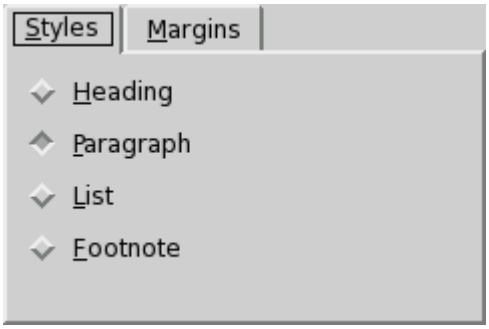

Figura C.3: Vista de ventana con apariencia Motif

C.3.2. Widgets

Los *widgets* son elementos visuales que pueden combinares para crear interfaces de usuario. Los botones, menús, barras de desplazamiento y ventanas de aplicación son ejemplos de *widgets*. En *Qt*, al contrario que otras plataformas de desarrollo, no hay división entre *controladores* y *contenedores*. En *Qt*, todos los *widgets* pueden ser utilizados como ambas cosas. Además, se pueden crear nuevos *widgets* como subclases de los existentes, o creándolos de la nada si fuera necesario.

Un *widget* puede tener un número arbitrario de *widgets* hijos. Los hijos se muestran en el área del padre. Si un *widgets* no tiene padre está en el primer nivel (una *ventana*) y normalmente tiene su propia entrada en la barra de programas del escritorio. *Qt* no impone ninguna limitación en este sentido a los *widgets*. Cualquiera puede ser una ventana, y cualquiera puede ser hijo de otro *widget*. Su posición se fija utilizando los *layout managers*, aunque también puede hacerse manualmente si se desea. Cuando un padre es deshabilitado, oculto o borrado, la misma acción se aplica recursivamente a todos sus hijos.

C.3.3. Signals and Slots

Las aplicaciones con interfaces gráficas de usuario responden a las acciones que realizan dichos usuarios. Por ejemplo, si un usuario pulsa un elemento del menú o un botón de la barra de herramientas, la aplicación ejecuta algo de código. Más en general, lo que se pretende con las *Qt* es que los objetos de cualquier tipo puedan comunicarse entre sí. El programador debe relacionar eventos con el código a ejecutar.

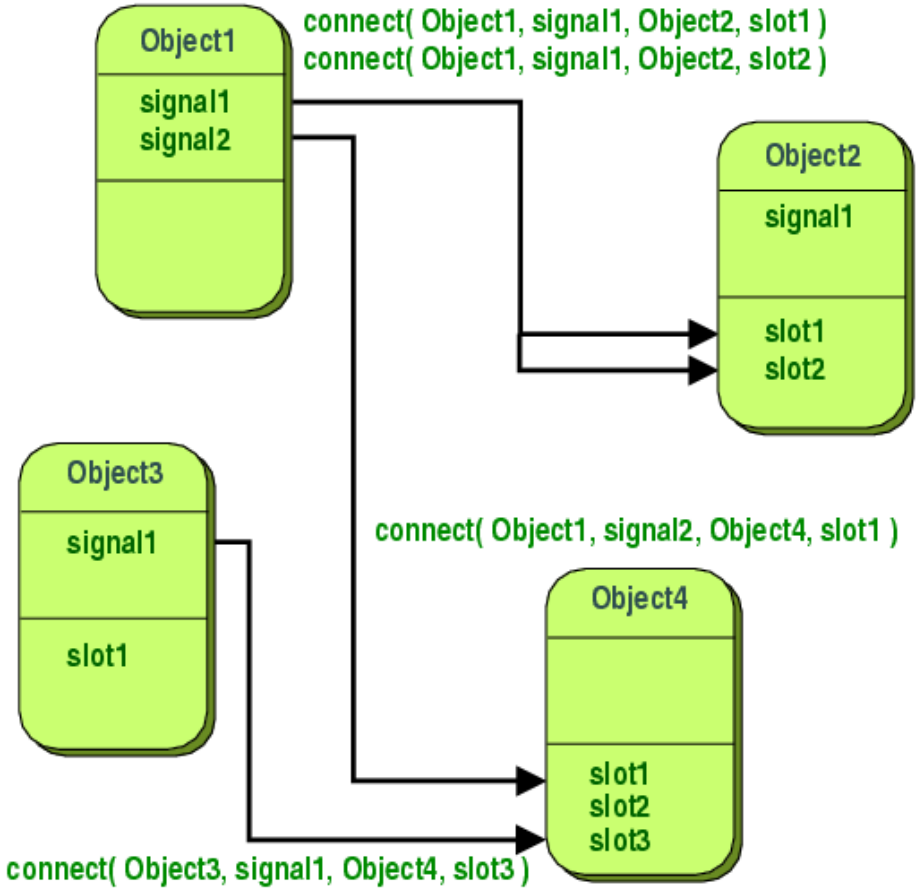

Figura C.4: Comunicación de objetos mediante *signals and slots*

Trolltech ha inventado una solución para esto llamada *signals and slots* (que se puede traducir aproximadamente por *señales y ranuras*). Es un poderoso mecanismo de comunicación completamente orientado a objetos, flexible e implementado en C++.

Los *widgets* de *Qt* emiten señales (*signals)* cuando ocurre un determinado evento. Por ejemplo, un botón emite una señal de "pulsado" cuando ha sido pulsado, o un campo de texto emite otra cuando su contenido ha sido modificado. El programador puede conectar esa señal con una ranura (*slot*) de otro *widget*, con lo que ambos quedan relacionados (ver figura C.4).

Utilizando este mecanismo, las clases que emiten y reciben las señales no tienen por qué conocerse, lo que hace mucho más fácil la tarea

del desarrollador a la hora de desarrollar clases altamente reusables. Además, el mecanismo de *signals and slots* realiza comprobaciones de tipo, con lo que un error no causa que la aplicación se caiga.

Por ejemplo, podemos querer conectar un botón de "salir" con la función de salida de una aplicación para que se finalice la ejecución cuando sea pulsado. Entonces, conectamos la señal de `clicked()` del botón con la ranura de `quit()` de la aplicación. Esto se realiza con la función `conect()` del modo siguiente:

```
connect(boton, SIGNAL(clicked()), aplicacion, SLOT(quit()));
```

Las conexiones pueden ser añadidas o eliminadas en cualquier momento de la ejecución de una aplicación de *Qt*, y tienen distintos modos de ejecución. Por ejemplo, puedes hacer que sean ejecutadas cada vez que se emite una señal o encolada para una ejecución posterior. Además, pueden hacerse conexiones entre objetos de diferentes hebras.

El mecanismo de *signals and slots* extiende la sintaxis de C++ sacando todo el partido posible a las características de este lenguaje orientadas a objetos. Este mecanismo realiza comprobaciones de tipo, las funciones pueden ser sobrecargadas o reescritas, y pueden aparecer en las partes pública ,protegida o privada de una clase.

Para beneficiarse de este mecanismo, una clase debe heredar de `QObject` o alguna de sus subclases, además de incluir la macro `Q_OBJECT`. Las señales y ranuras se declaran en sus secciones específicas de la clase.

C.3.4. Manejo del layout

El systema de layout de *Qt* proporciona un método simple y poderoso de especificar el layout de los *widgets* hijos. Especificando el layout lógico una sola vez, obtienes los siguientes beneficios:

- posicionar los widgets hijos
- especificar el tamaño mínimo, máximo y por defecto de las ventanas
- Control del layout y de las dimensiones de los *widget* hijo en las redimensiones
- Actualizaciones automáticas en los cambios de contexto: tamaño de texto, tipo de letra, muestra u ocultación de los *widgets* hijo.

Actualmente, *Qt* proporciona tres tipos de layout: vertical (`QVBoxLayout`), horizontal (`QHBoxLayout`) y en rejilla(`QGridLayout`). Sus diferencias pueden apreciarse en las figuras C.5, C.6 y C.7

La desventaja de este sistema es que es necesario compilar y ejecutar la aplicación cada vez que ejecutas un cambio en el layout, tarea que puede resultar tediosa en el caso de tener que realizar numerosas pruebas. Para agilizar este proceso Trolltech ha introducido el *Qt Designer*, una herramienta visual de diseño de interfaces gráficas de usuario. Su tarea es convertir el diseño que haces con el asistente en código C++ ejecutable.

En cualquier caso, para aplicaciones de amaño pequeño-mediano es extremadamente sencillo este asistente no resulta necesario. Por ejemplo, en la realización de la aplicación de este proyecto no utilizamos el asistente.

Figura C.5: Disposición de los elementos con `QVBoxLayout`

Figura C.6: Disposición de los elementos con `QHBoxLayout`

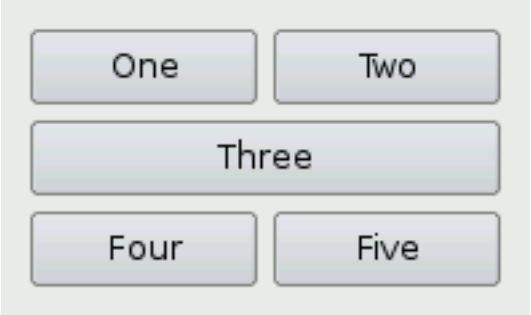

Figura C.7: Disposición de los elementos con `QGridLayout`

C.3.5. Internacionalización

Qt incluye herramientas para facilitar el proceso de traducción de los programas. Los programadores pueden marcar de un modo sencillo las partes del texto que necesitan traducción, y una herramienta extrae ese texto del código fuente. El *Qt Linguist* es una aplicación gráfica que lee el texto extraído del código y proporciona información para su traducción. Cuando esta se ha completado, el programa produce como salida un fichero con toda la información necesaria, que utilizarán las aplicaciones según el idioma en que deban ejecutarse. Además, es capaz de recordar las traducciones realizadas en aplicaciones anteriores y hacer sugerencias para la aplicación actual, lo que proporciona una mayor consistencia.

Qt también proporciona herramientas para adaptarse a las convenciones locales sobre símbolos monetarios y preferencias del lenguaje (utilizar coma o punto como separador de los números decimales, por ejemplo).

Qt incluye un potente motor renderizador para todo el texto que se presenta en pantalla. Este motor soporta características avanzadas tales como ajustes en los saltos de línea, escritura bidireccional y marcas diacríticas. Puede renderizar la mayoría de los métodos de escritura del mundo, incluyendo chino, griego, japonés, y latín. qt combina automáticamente las fuentes instaladas para renderizar el texto multilenguaje.

Por otro lado, *Qt* incluye la traducción de más de 400 cadenas de texto de inglés a francés y alemán (además de otros diccionarios que pueden descargarse). De este modo, basta con señalar con la llamada `tr(texto)` el texto a traducir, y será sustituido en cada país por su

equivalente. Por ejemplo, un botón con el texto

```
saveButton->setText(tr(‘‘Save’’));
```

será sustituido en francés por “Saver”. Para otros idiomas como el castellano, sin traducción automática, se conservará el texto original del botón.

Apéndice D

Código de las funciones de modificación rlim_do_getprlimit() y rlim_do_setprlimit()

```
/*****************************************************************
 *   SSII 2005-2006                                              *
 *   ESTABLECIMIENTO EXTERNO DE LIMITES DE PROCESOS EN LINUX     *
 *                                                               *
 *      Ibon Cano Sanz                                           *
 *      Bálder Carraté Martínez                                  *
 *      Manuel Rodríguez Pascual                                 *
 *****************************************************************/


/*
 * Comprueba que el usuario del proceso actual tiene los permisos
necesarios
 * para consultar y modificar los límites del proceso task
 */
static inline int prlim_check_perm(task_t *task){
  return ((current->uid == task->euid) &&
      (current->uid == task->suid) &&
      (current->uid == task->uid) &&
      (current->gid == task->egid) &&
```

```
      (current->gid == task->sgid) &&
      (current->gid == task->gid)) ||
      capable(CAP_SYS_PTRACE);

}


/*
 * Devuelve el valor del límite "resource" del proceso "pid" en
"rlim"
 */
inline long do_getprlimit(pid_t pid, unsigned int resource,
                          struct rlimit *rlim)
{
  /* Valor de retorno de la función (código de error) */
  int retval;
  /* Estructura del proceso */
  task_t *task;
  /* Almacenará temporalmente el valor del límite */
  struct rlimit value;

  retval = 0; /* Inicializa el valor de retorno a cero (éxito)
              */

  /* comprueba que "pid" y "resource" tienen valores válidos */
  if (pid >= 0 && resource < RLIM_NLIMITS) {

    read_lock(&tasklist_lock);    /* bloquea la lista de
    procesoss */

    /* obtiene el el task_struct del proceso correspondiente
    a pid */
    if (pid == 0) task = current;
    else {
      task = find_task_by_pid(pid);
    }

    if (task) {
      /* Comprueba permisos para leer el valor del límite
      */
      if (pid > 0 && prlim_check_perm(task)) {
        retval = security_task_rlimit(task, resource, 0);
```

```
      }
      else if (pid > 0 && !prlim_check_perm(task)) {
        retval = -EPERM;    /* Error. Permisos
        insuficientes */
      }

      /* Si se tienen permisos lee el valor del límite */
      if (retval == 0) {
        task_lock(task->group_leader);
        value = task->signal->rlim[resource];
        task_unlock(task->group_leader);
      }
    }
    else retval = -ESRCH;   /* Error. No existe el
    proceso */

    /* Si ha habido éxito se escriben el valor del límite al
    parámetro rlim */
    if (retval == 0) {
      rlim->rlim_cur = value.rlim_cur;
      rlim->rlim_max = value.rlim_max;
    }

    read_unlock(&tasklist_lock);    /* Desbloquea la
    lista de tareas */
  }
  else retval = -EINVAL;    /* Error. Argumentos no válidos
                            */

  return retval;
}


/*
 * Devuelve el número de ficheros abiertos en fdt (incluidos los
de la E/S estándar
 */
static int count_open_files(struct fdtable *fdt)
{
  /* lleva la cuenta de ficheros abiertos */
  int count;
  /* descriptor del fichero actual */
```

```
  int fd;

  count = 0;
  /* inicializa al mayor fd posible */
  fd = fdt->max_fds;

  /* mientras el fd sea mayor o igual a cero recorre el array
  de ficheros
  comprobando si el actual está o no abierto, y en caso
  afirmativo
  incrementando el contador en una unidad */
  while (fd >= 0) {
    if (fdt->fd[fd]) count++;
    fd--;
  }

  return count;
}


/* libera el descriptor fd de la estructura files para poder ser
vuelto a utilizar */
static inline void free_unused_fd(struct files_struct *files,
                                  unsigned int fd)
{
  struct fdtable *fdt = files_fdtable(files);
  __FD_CLR(fd, fdt->open_fds);
  if (fd < fdt->next_fd) fdt->next_fd = fd;
}


/*
 * Cierra el fichero con descriptor fd abierto por el proceso
task
 */
static long fd_close(struct task_struct *task, unsigned int fd)
{
  /* puntero a la estructura del fichero que se va a cerrar */
  struct file * filp;
  /* puntero a la estructura de ficheros del proceso */
  struct files_struct *files;
  /* puntero a la tabla de ficheros del proceso */
```

```
  struct fdtable *fdt;

  /* se inicializa files*/
  files = task->files;

  /* aquí es donde se realizan todos los pasos para cerrar el
  fichero.
  La acción de cerrar el fichero la realiza la función
  filp_close() */
  spin_lock(&files->file_lock);
  fdt = files_fdtable(files);
  if (fd >= fdt->max_fds)
    goto out_unlock;
  filp = fdt->fd[fd];
  if (!filp)
    goto out_unlock;
  rcu_assign_pointer(fdt->fd[fd], NULL);
  FD_CLR(fd, fdt->close_on_exec);
  free_unused_fd(files, fd);
  spin_unlock(&files->file_lock);
  return filp_close(filp, files);

out_unlock:
    spin_unlock(&files->file_lock);
  return -EBADF;
}


/* Establece la dirección final del segmento de datos de del
proceso task
a la de brk */
static unsigned long brk_task(struct task_struct *task, unsigned
    long brk)
{
  unsigned long rlim, retval;
  unsigned long newbrk, oldbrk;
  struct mm_struct *mm = task->mm;

  down_write(&mm->mmap_sem);

  if (brk < mm->end_code)
    goto out;
```

```
  newbrk = PAGE_ALIGN(brk);
  oldbrk = PAGE_ALIGN(mm->brk);
  if (oldbrk == newbrk)
    goto set_brk;

  /* en el caso de que haya que reducir el segmento, siempre se
  puede */
  if (brk <= mm->brk) {
    if (!do_munmap(mm, newbrk, oldbrk-newbrk))
      goto set_brk;
    goto out;
  }

  /* comprueba que no se vile el RLIMIT_DATA */
  rlim = current->signal->rlim[RLIMIT_DATA].rlim_cur;
  if (rlim < RLIM_INFINITY && brk - mm->start_data > rlim)
    goto out;

  /* comprueba que no haya conflictos */
  if (find_vma_intersection(mm, oldbrk, newbrk+PAGE_SIZE))
    goto out;

  /* modifica el valor */
  if (do_brk(oldbrk, newbrk-oldbrk) != oldbrk)
    goto out;
set_brk:
    mm->brk = brk;
out:
    retval = mm->brk;
  up_write(&mm->mmap_sem);
  return retval;
}


/*
 * Establece el valor del límite "resource" al de "rlim" en el
proceso "pid"
 */
inline long do_setprlimit(pid_t pid, unsigned int resource,
                          struct rlimit *rlim)
{
  /* guarda el valor de retorno */
```

```
int retval;

/* guarda el anterior valor del límite */
struct rlimit *old_value;

/* estructura del proceso */
task_t *task;

retval = 0;   /* inicializa retval a éxito */

/* comprueba que el "pid", el "resource" y el nuevo valor de
éste son
válidos y efectúa la modificación */
if ((pid >= 0 && resource < RLIM_NLIMITS) &&
     ((rlim->rlim_max == RLIM_INFINITY) ||
     ((rlim->rlim_max      != RLIM_INFINITY) && (rlim->rlim_cur
     <= rlim->rlim_max)))) {

  read_lock(&tasklist_lock);    /* bloquea la lista de
  procesos */

  /* si el proceso es el actual se asigna a "task" */
  if (pid == 0) {
    task = current;
  }
  /* si el proceso es otro distinto se busca en la lista de
  procesos */
  else {
    task = find_task_by_pid(pid);
  }

  if (task != NULL) {
    /* comprueba si el proceso actual tiene permisos
    sobre el que va
    a efectuar la modificación del límite */
    if (pid > 0 && !prlim_check_perm(task))
      retval = -EPERM;
    else {
      /* asigna a old_value la dirección del límite del
      proceso */
      old_value = task->signal->rlim + resource;
```

```
if ((rlim->rlim_max > old_value->rlim_max) &&
     !capable(CAP_SYS_RESOURCE))
  retval = -EPERM;

if (resource == RLIMIT_NOFILE && rlim->rlim_max >
    NR_OPEN)
  retval = -EPERM;

if (retval == 0) {
  retval = security_task_rlimit(task, resource,
                                rlim);
}

/* si no ha habido errores proceder a la
asignación del nuevo
valor del límite */
if (retval == 0) {

  task_lock(task->group_leader);
  *old_value = *rlim;
  task_unlock(task->group_leader);


  /****** POLÍTICAS ******/

  struct files_struct *files;

  task_lock(task);
  files = task->files;
  task_unlock(task);

  /* Si el valor de RLIMIT_CPU es menor que el
  valor de tiempo
  límite del proceso, se reduce éste al del
  nuevo valor del
  límite. Si no se deja como está.
  it_prof_expires es el valor
  máximo de tiempo en ejecución para el proceso
  en modo usuario
  más en modo kernel. Es el valor que limita
  RLIMIT_CPU, y por
  tanto no puede ser superior a éste*/
```

```
if (resource == RLIMIT_CPU && rlim->rlim_cur
    != RLIM_INFINITY &&


    (cputime_eq(task->signal->it_prof_expires,
     cputime_zero) ||
    rlim->rlim_cur <=

    cputime_to_secs(task->signal->it_prof_expires))) {

  cputime_t cputime =
      secs_to_cputime(rlim->rlim_cur);
  spin_lock_irq(&task->sighand->siglock);
  set_process_cpu_timer(task,
                        CPUCLOCK_PROF,
                        &cputime, NULL);
  spin_unlock_irq(&task->sighand->siglock);
    }

/* Caso en el que el límite de ficheros
    abiertos se
    establezca por debajo del
    número de ficheros
    abiertos
    actual */

    if (resource ==
        RLIMIT_NOFILE) {
      int files_open;

      spin_lock(&files->file_lock);
      files_open =

          count_open_files(files->fdt);

             printk("files open:
                 %d\n", files_open);
             printk("new max files
                 open: %u\n",
             rlim->rlim_cur);

             int n,fd;
```

```
                n = files_open -
                    rlim->rlim_cur;
                fd =
                    files->fdt->max_fds;

                spin_unlock(&files->file_lock);

                if (rlim->rlim_cur !=
                    RLIM_INFINITY &&

                    rlim->rlim_cur < files_open ) {

                  struct file *file
                      = NULL;

          /* No hay que cerrar los fd 0, 1, 2
                     por que son
                     la E/S estándar */
                     while (n > 0 && fd
                            > 2) {


                       spin_lock(&files->file_lock);

                       file =
                           files->fdt->fd[fd];

                       spin_unlock(&files->file_lock);

                       if (file) {

                         printk("about to close fd

                             %d... - ",fd);
                         if
                             (fd_close(task, fd) >= 0)

                         {

                           printk("success!\n");
                           n = n
                               - 1;
```

```
                    }
                    else
                      printk("failed!\n");

                  }
                  else
                    printk("not exist!\n");

                  fd--;
                       }

                  }

        }
/* Caso en el que el límite del tamaño de
        fichero se
        establezca por debajo de
        alguno abierto                      actualmente
*/
        else if (resource ==
                 RLIMIT_FSIZE){
          struct fdtable *fdt;
          struct file *file;
          struct inode *inode;


          spin_lock(&files->file_lock);
          fdt = files->fdt;
          int fd;
  /* Empieza en el fd=3 para no cerrar la
          E/S estándar*/
          for (fd = 3; fd <
               fdt->max_fds; fd++) {
                file =
                    fdt->fd[fd];

                if (file) {
                  inode =
                      file->f_dentry->d_inode;

                  if (inode) {
                    if
```

```
                        (inode->i_size >

                         rlim->rlim_cur) {
                printk("fd %d size:
                    %d\n", fd,
                inode->i_size);
                printk("closing fd %d: ",
                       fd);

                retval = fd_close(task,
                                  fd);

                if ( retval == 0) {
                  printk("success!\n");
                }
                else {
                  printk("failed! error
                      %d\n",
                  retval);
                }

                           }
                         }
                       }
                     }

                     spin_unlock(&files->file_lock);
                       }
        /* Caso en el que el límite del nice se
                       establezca a un
                       valor menos privilegiado
                       que el valor de nice

                       actual*/
                       else if (resource ==
                                RLIMIT_NICE) {
                         int nice, min_nice;
                         task_lock(task);

                         nice =
                             task_nice(task);
                         min_nice = 20 -
```

```
                        rlim->rlim_cur;

                    if (min_nice >= 20)
                      min_nice = 19;
                    else if (min_nice <=
                             -21) min_nice = -20;

                    printk("nice %d\n",
                           nice);
                    printk("min_nice
                        %d\n", min_nice);

                    if (nice < min_nice) {
                      retval =
                          set_one_prio( task,

                          min_nice, retval);
                      nice =
                          task_nice(task);
                      printk("new nice
                          %d\n", nice);
                    }
                    task_unlock(task);
                           }
/* Caso en el que el límite de tamaño de
                           segmento máximo de
                           memoria de datos se
                           establezca por debajo del

                           valor actual*/
                           else if (resource ==
                                    RLIMIT_DATA) {
                             struct mm_struct *mm;
                             mm = task->mm;

                printk("old data size:
                    %ld\n", mm->brk -

                    mm->start_data);
  printk("new max data size:
      %ld\n",rlim->rlim_cur);
```

```
            unsigned long new_brk;
            new_brk = mm->start_data +
                rlim->rlim_cur;
            if (rlim->rlim_cur != RLIM_INFINITY &&
                mm->brk > new_brk)
            {
              if (brk_task(task, new_brk) < 0) {
                printk("error changing max data
                    segment!\n");
              }
              printk("new data size: %ld\n",
                     mm->brk - mm->start_data);
            }
                                              }
        }
      }
    }
    else {
      retval = -ESRCH;
    }

    read_unlock(&tasklist_lock);  /* desbloquea la lista de
    procesos */
       }
       else retval = -EINVAL;

       return retval;
}


/* Llamada al sistema para leer los límites del proceso pid */
asmlinkage long sys_getprlimit(pid_t pid, unsigned int resource,
                               struct rlimit __user *rlim)
{
  struct rlimit value;
  int retval;

  retval = do_getprlimit(pid, resource, &value);

  if (copy_to_user(rlim, &value, sizeof(value))) {
    retval = -EFAULT;
  }
```

```
  return retval;
}


/* Llamada al sistema para leer los límites del proceso actual */
asmlinkage long sys_getrlimit(unsigned int resource, struct
    rlimit __user *rlim)
{
  struct rlimit value;
  int retval;

  retval = do_getprlimit(0, resource, &value);

  if (copy_to_user(rlim, &value, sizeof(value))) {
    retval = -EFAULT;
  }

  return retval;
}


/* Llamada al sistema para establecer los límites del proceso pid
*/
asmlinkage long sys_setprlimit(pid_t pid, unsigned int resource,
                               struct rlimit __user *rlim)
{
  struct rlimit new_rlim;

  if(copy_from_user(&new_rlim, rlim, sizeof(*rlim))) {
    return -EFAULT;
  }
  else {
    return do_setprlimit(pid, resource, &new_rlim);
  }
}


/* Llamada al sistema para establecer los límites del proceso
actual */
asmlinkage long sys_setrlimit(unsigned int resource, struct
    rlimit __user *rlim)
```

```
{
  struct rlimit new_rlim;

  if(copy_from_user(&new_rlim, rlim, sizeof(*rlim))) {
    return -EFAULT;
  }
  else {
    return do_setprlimit(0, resource, &new_rlim);
  }
}


/*****************************************************************/
```

Bibliografía

[**ARCOMANO_2003**] R. Arcomano.
Kernel Analysis HOWTO.
Publicado bajo GPL, 2003 (Kernel 2.4).
`http://www.tldp.org/HOWTO/KernelAnalysis-HOWTO.html`

[**AAS_2005**] J. Aas.
Understanding the Linux 2.6.8.1 CPU Scheduler.
Silicon Graphics, 2005.
`http://josh.trancesoftware.com/linux/`

[**BLANCHETTE_2004**] J. Blanchette, M. Summerfield.
C++ GUI Programming with Qt3.
Prentice Hall, 2004.

[**BOVET_2005**] D. P. Bovet, M. Cesati.
Understanding the Linux Kernel, 3rd Edition.
O'Reilly, 2005. ISBN 0-596-00565-2.

[**CORBET_2005**] J. Corbet, G. Kroah-Hartman, A. Rubini.
Linux Device Drivers, 3rd Edition.
O'Reilly, 2005. ISBN 0-596-00590-3.

[**LOVE_2005**] R. Love.
Linux Kernel Development, 2nd Edition.
Sams Publishing, 2005. ISBN 0-672-32720-1.

[**MARQUEZ_2004**] F. M. Márquez García.
UNIX. Programación Avanzada, 3ª Edición.
Ra-Ma. 2004. ISBN 8-478-97603-5.

[**SALZBERG_2005**] C. Salzberg Rodriguez, G. Fischer, S. Smolski.
Linux Kernel Primer: A Top-Down Approach for x86 and PowerPC Architectures.
Prentice Hall. 2005. ISBN 0-131-18163-7.

[**SARMA_2001**] D.Sarma, K.Thomas.
Read Copy Update HOWTO.
2001.
`http://lse.sourceforge.net/locking/rcu/HOWTO/index.html`

[**STALLINGS_2004**] W. Stallings.
Operating Systems: Internals and Desing principles, 5th Edition.
Prentice Hall, 2004. ISBN 0-131-47954-7.

[**TANEMBAUM_2006**] A. Tanenbaum.
Operating Systems: Design and Implementation, 3rd Edition.
Prentice Hall, 2006. ISBN 0-131-42938-8.

[**TROLLTECH_2006**] Trolltech
Qt 4.1 Whitepaper.
Trolltech, 2006.

[**VARIOS**] Varios Autores.
The Linux Kernel Mailing List FAQ.
Publicado bajo GPL.
`http://www.kernel.org/pub/linux/docs/lkml/`
`?bcsi_scan_037EBDF7009435C1=0`

[**WALL_2000**] K. Wall, M. Watson.
Linux Programming Unleashed, 2nd Edition.
Prentice Hall, 2000. ISBN 0-672-32021-5.

Índice alfabético