http://www.moses.uklinux.net/patches/lki.sgml

Esta guía es ahora parte del Proyecto de Documentación de Linux y también puede ser descargada en varios formatos desde:

http://www.linuxdoc.org/guides.html

o puede ser leída en linea (la última versión en Inglés) en:

http://www.moses.uklinux.net/patches/lki.html

 El autor esta trabajando como ingeniero decano del núcleo Linux en VERITAS Software Ltd y escribió este libro con el propósito de dar soporte al pequeño entrenamiento de cursos/charlas que dió sobre este tema, internamente en VERITAS. Gracias a: Juan J. Quintela (quintela@fi.udc.es), Francis Galiegue (fg@mandrakesoft.com), Hakjun Mun (juniorm@orgio.net), Matt Kraai (kraai@alumni.carnegiemellon.edu), Nicholas Dronen (ndronen@frii.com), Samuel S Chessman (chessman@tux.org), Nadeem Hasan (nhasan@nadmm.com) por varias correcciones y sugerencias. El capítulo del Caché de Páginas de Linux fue escrito por Christoph Hellwig (hch@caldera.de). El capítulo sobre los mecanismos IPC fue escrito por Russell Weight (weightr@us.ibm.com) y Mingming Cao (mcao@us.ibm.com)

Dentro del núcleo Linux 2.4

Tigran Aivazian tigran@veritas.com 23 Agosto 2001 (4 Elul 5761)

Contents

1 Arrancando

1.1 Construyendo la Imagen del Núcleo Linux

Esta sección explica los pasos dados durante la compilación del núcleo Linux y la salida producida en cada etapa. El proceso de construcción depende de la arquitectura, por lo tanto me gustaría enfatizar que sólo consideraremos la construcción de un núcleo Linux/x86.

Cuando el usuario escribe 'make zImage' o 'make bzImage' la imagen inicializable del núcleo resultante es almacenado como `arch/i386/boot/zImage` o `arch/i386/boot/bzImage` respectivamente. Aquí está como es construida la imagen:

1. Los archivos fuente en C y ensamblador son compilados en formato de objetos reasignables ELF (.o) y algunos de ellos son agrupados lógicamente en archivos (.a) usando **ar(1)**.
2. Usando **ld(1)**, los anteriores .o y .a son enlazados en `vmlinux`, el cual es un fichero ejecutable ELF 32-bits LSB 80386 estáticamente enlazado al que no se le han eliminado los símbolos de depuración.
3. `System.map` es producido por **nm vmlinux**, donde los símbolos irrelevantes o que no interesan son desechados.
4. Se entra en el directorio `arch/i386/boot`.
5. El código de ensamblador del sector de arranque `bootsect.S` es preprocesado con o sin **-D__BIG_KERNEL__**, dependiendo de cuando el objetivo es bzImage o zImage, en `bbootsect.s` o `bootsect.s` respectivamente.
6. `bbootsect.s` es ensamblado y entonces convertido en la forma 'raw binary' llamada `bbootsect` (o `bootsect.s` ensamblado y convertido a raw en `bootsect` para zImage).
7. El código de configuración `setup.S` (`setup.S` incluye `video.S`) es preprocesado en `bsetup.s` para bzImage o `setup.s` para zImage. De la misma forma que el código del sector de arranque, la diferencia radica en que **-D__BIG_KERNEL__** está presente para bzImage. El resultado es entonces convertido en la forma 'raw binary' llamada `bsetup`.
8. Se entra en el directorio `arch/i386/boot/compressed` y se convierte `/usr/src/linux/vmlinux` a $tmppiggy (nombre temporal) en el formato binario raw, borrando las secciones ELF `.note` y `.comment`.
9. **gzip -9 < $tmppiggy > $tmppiggy.gz**
10. Se enlaza $tmppiggy.gz en ELF reasignable (**ld -r**) `piggy.o`.
11. Compila las rutinas de compresión `head.S` y `misc.c` (todavía en el directorio `arch/i386/boot/compressed`) en los objetos ELF `head.o` y `misc.o`.
12. Se enlazan todas ellas: `head.o`, `misc.o` y `piggy.o` en `bvmlinux` (o `vmlinux` para zImage, ¡no confundas esto con `/usr/src/linux/vmlinux`!). Destacar la diferencia entre **-Ttext 0x1000** usado para `vmlinux` y **-Ttext 0x100000** para `bvmlinux`, esto es, el cargador de compresión para bzImage es cargado más arriba.
13. Se convierte `bvmlinux` a 'raw binary' `bvmlinux.out` borrando las secciones ELF `.note` y `.comment`.
14. Se vuelve atrás al directorio `arch/i386/boot` y, usando el programa **tools/build**, se concatenan todos ellos: `bbootsect`, `bsetup` y `compressed/bvmlinux.out` en `bzImage` (borra la 'b' extra anterior para `zImage`). Esto escribe variables importantes como `setup_sects` y `root_dev` al final del sector de arranque.

El tamaño del sector de arranque es siempre de 512 bytes. El tamaño de la configuración debe ser mayor que 4 sectores, pero está limitado superiormente sobre los 12k - la regla es:

0x4000 bytes >= 512 + sectores_configuración * 512 + espacio para la pila mientras está funcionando el sector de arranque/configuración

Veremos más tarde de dónde viene esta limitación.

El límite superior en el tamaño de bzImage producido en este paso está sobre los 2.5M para arrancarcon LILO y 0xFFFF párrafos ((0xFFFF0 = 1048560 bytes) para arrancar imágenes directamentee, por ejemplo desde un diskette o CD-ROM (con el modo de emulación El-Torito).

Destacar que mientras que **tools/build** valida el tamaño del sector de arranque, la imagen del núcleo y el límite inferior del tamaño de la configuración, no chequea el límite *superior* de dicho tamaño de configuración. Entonces, es fácil construir un núcleo defectuoso justamente sumándole algún gran ".espacio" al final de `setup.S`.

1.2 Arrancando: Vista General

Los detalles del proceso de arranque son específicos de cada arquitectura, por lo tanto centraremos nuestra atención en la arquitectura IBM PC/IA32. Debido al diseño antiguo y a la compatibilidad hacia atrás, el firmware del PC arranca el sistema operativo a la vieja usanza. Este proceso puede ser separado en las siguientes seis etapas lógicas:

1. La BIOS selecciona el dispositivo de arranque.
2. La BIOS carga el sector de arranque del dispositivo de arranque.
3. El sector de arranque carga la configuración, las rutinas de descompresión y la imagen del núcleo comprimida.
4. El núcleo es descomprimido en modo protegido.
5. La inicialización de bajo nivel es realizada por el código ensamblador.
6. Inicialización de alto nivel en C.

1.3 Arrancando: BIOS POST

1. La fuente de alimentación inicia el generador de reloj y aserta la señal #POWERGOOD en el bus.
2. La linea CPU #RESET es asertada (CPU está ahora en modo real 8086).
3. %ds=%es=%fs=%gs=%ss=0, %cs=0xFFFF0000,%eip = 0x0000FFF0 (código ROM BIOS POST).
4. Todos los chequeos POST son realizados con las interrupciones deshabilitadas.
5. La TVI (Tabla de Vectores de Interrupción) es inicializada en la dirección 0.
6. La función de la BIOS de cargador de la rutina de arranque es llamada a través de la **int0x19**, con %dl conteniendo el dispositivo de arranque 'número de controladora'. Esto carga la pista 0, sector 1 en la dirección física 0x7C00 (0x07C0:0000).

1.4 Arrancando: sector de arranque y configuración

El sector de arranque usado para arrancar el núcleo Linux puede ser uno de los siguientes:

- Sector de arranque de Linux (`arch/i386/boot/bootsect.S`),
- Sector de arranque de LILO (u otros cargadores de arranque) o
- sin sector de arranque (loadlin, etc)

Consideraremos aquí el sector de arranque de Linux en detalle. Las primeras lineas inicializan las macros convenientes para ser usadas por los valores de segmento:

```
29 SETUPSECS = 4                /* tamaño por defecto de los sectores de configuración */
30 BOOTSEG   = 0x07C0           /* dirección original del sector de arranque */
31 INITSEG   = DEF_INITSEG      /* movemos el arranque aquí - lejos del camino */
32 SETUPSEG  = DEF_SETUPSEG     /* la configuración empieza aquí */
33 SYSSEG    = DEF_SYSSEG       /* el sistema es cargado en 0x10000 (65536) */
34 SYSSIZE   = DEF_SYSSIZE      /* tamaño del sistema: # de palabras de 16 bits */
```

(los números a la izquierda son los números de linea del archivo bootsect.S) Los valores de `DEF_INITSEG`, `DEF_SETUPSEG`, `DEF_SYSSEG` y `DEF_SYSSIZE` son tomados desde `include/asm/boot.h`:

```
/* No toques esto, a menos que realmente sepas lo que estás haciendo. */
#define DEF_INITSEG     0x9000
#define DEF_SYSSEG      0x1000
#define DEF_SETUPSEG    0x9020
#define DEF_SYSSIZE     0x7F00
```

Ahora, consideremos el código actual de `bootsect.S`:

```
54          movw    $BOOTSEG, %ax
55          movw    %ax, %ds
56          movw    $INITSEG, %ax
57          movw    %ax, %es
58          movw    $256, %cx
59          subw    %si, %si
60          subw    %di, %di
61          cld
62          rep
63          movsw
64          ljmp    $INITSEG, $go

65  # bde - cambiado 0xff00 a 0x4000 para usar el depurador después de 0x6400 (bde).
66  # No tendríamos que preocuparnos por esto si chequeamos el límite superior
67  # de la memoria. También mi BIOS puede ser configurada para poner las tablas
68  # wini de controladoras en la memoria alta en vez de en la tabla de vectores.
69  # La vieja pila quizás tenga que ser insertada en la tabla de controladores.

70  go:     movw    $0x4000-12, %di         # 0x4000 es un valor arbitrario >=
71                                          # longitud de sector de arranque + longitud de la
72                                          # configuración + espacio para la pila;
73                                          # 12 es el tamaño parm del disco.
74          movw    %ax, %ds                # ax y es ya contienen INITSEG
75          movw    %ax, %ss
76          movw    %di, %sp                # pone la pila en INITSEG:0x4000-12.
```

Las lineas 54-63, mueven el código del sector de arranque desde la dirección 0x7C00 a 0x90000. Esto es realizado de la siguiente manera:

1. establece %ds:%si a $BOOTSEG:0 (0x7C0:0 = 0x7C00)
2. establece %es:%di a $INITSEG:0 (0x9000:0 = 0x90000)
3. establece el número de palabras de 16 bits en %cx (256 palabras = 512 bytes = 1 sector)
4. limpia la bandera DF (dirección) en EFLAGS a direcciones auto-incrementales (cld)

5. va allí y copia 512 bytes (rep movsw)

El motivo por el que este código no usa `rep movsd` es intencionado (hint - .code16).

La línea 64 salta a la etiqueta `go:` en la nueva copia hecha del sector de arranque, esto es, en el segmento 0x9000. Esto y las tres instruciones siguientes (lineas 64-76) preparan la pila en $INITSEG:0x4000-0xC, esto es, %ss = $INITSEG (0x9000) y %sp = 0x3FF4 (0x4000-0xC). Aquí es de dónde viene el límite del tamaño de la configuración que mencionamos antes (ver Construyendo la Imagen del Núcleo Linux).

Las lineas 77-103 parchean la tabla de parámetros del disco para el primer disco para permitir lecturas multi-sector:

```
 77  # Las tablas por defecto de parámetros del disco de muchas BIOS
 78  # no reconocerán lecturas multi-sector más allá del número máximo especificado
 79  # en las tablas de parámetros del diskette por defecto - esto
 80  # quizás signifique 7 sectores en algunos casos.

 82  # Como que las lecturas simples de sectores son lentas y fuera de la cuestión
 83  # tenemos que tener cuidado con esto creando nuevas tablas de parámetros
 84  # (para el primer disco) en la RAM. Estableceremos la cuenta máxima de sectores
 85  # a 36 - el máximo que encontraremos en un ED 2.88.
 86  #
 87  # Lo grande no hace daño. Lo pequeño si.
 88  #
 89  # Los segmentos son como sigue: ds = es = ss = cs - INITSEG, fs = 0,
 90  # y gs queda sin usar.

 91          movw    %cx, %fs                # establece fs a 0
 92          movw    $0x78, %bx              # fs:bx es la dirección de la tabla de parámetros
 93          pushw   %ds
 94          ldsw    %fs:(%bx), %si          # ds:si es el código
 95          movb    $6, %cl                 # copia 12 bytes
 96          pushw   %di                     # di = 0x4000-12.
 97          rep                             # no necesita cld -> hecho en la linea 66
 98          movsw
 99          popw    %di
100          popw    %ds
101          movb    $36, 0x4(%di)           # parchea el contador de sectores
102          movw    %di, %fs:(%bx)
103          movw    %es, %fs:2(%bx)
```

El controlador de diskettes es reinicializado usando el servicio de la BIOS int 0x13 función 0 (reinicializa FDC) y los sectores de configuración son cargados inmediatamente después del sector de arranque, esto es, en la dirección física 0x90200 ($INITSEG:0x200), otra vez usando el servicio de la BIOS int 0x13, función 2 (leer sector(es)). Esto sucede durante las lineas 107-124:

```
107  load_setup:
108          xorb    %ah, %ah                # reinicializa FDC
109          xorb    %dl, %dl
110          int     $0x13
111          xorw    %dx, %dx                # controladora 0, cabeza 0
112          movb    $0x02, %cl              # sector 2, pista 0
113          movw    $0x0200, %bx            # dirección = 512, en INITSEG
114          movb    $0x02, %ah              # servicio 2, "leer sector(es)"
115          movb    setup_sects, %al        # (asume todos en la cabeza  0, pista 0)
116          int     $0x13                   # los lee
```

```
        jnc     ok_load_setup           # ok - continua

        pushw   %ax                     # vuelca el código de error
        call    print_nl
        movw    %sp, %bp
        call    print_hex
        popw    %ax
        jmp     load_setup

ok_load_setup:
```

Si la carga falla por alguna razón (floppy defectuoso o que alguien quitó el diskette durante la operación), volcamos el código de error y se intenta en un bucle infinito. La única forma de salir de él es reiniciando la máquina, a menos que los reintentos tengan éxito, pero usualmente no lo tienen (si algo está mal sólo se pondrá peor).

Si la carga de los sectores setup_sects del código de configuración es realizada con éxito, saltamos a la etiqueta `ok_load_setup:`.

Entonces procedemos a cargar la imagen comprimida del núcleo en la dirección física 0x10000. Esto es realizado para preservar las áreas de datos del firmware en la memoria baja (0-64K). Después de que es cargado el núcleo, saltamos a $SETUPSEG:0(`arch/i386/boot/setup.S`). Una vez que los datos no se necesitan mas (ej. no se realizan más llamadas a la BIOS) es sobreescrito moviendo la imagen entera (comprimida) del núcleo desde 0x10000 a 0x1000 (direcciones físicas, por supuesto). Esto es hecho por `setup.S`, el cual prepara las cosas para el modo protegido y salta a 0x1000, que es el comienzo del núcleo comprimido, esto es, `arch/386/boot/compressed/{head.S,misc.c}`. Esto inicializa la pila y llama a `decompress_kernel()`, que descomprime el núcleo en la dirección 0x100000 y salta a ella.

Destacar que los viejos cargadores de arranque (viejas versiones de LILO) sólo podían cargar los 4 primeros sectores de la configuración, el cual es el motivo por el que existe código en la configuración para cargar el resto de si mismo si se necesita. También, el código en la configuración tiene que tener cuidado de varias combinaciones de tipo/versión del cargador vs zImage/bzImage y esto es altamente complejo.

Examinemos este truco en el código del sector de arranque que nos permite cargar un núcleo grande, también conocido como "bzImage".

Los sectores de configuración son cargados usualmente en la dirección 0x90200, pero el núcleo es cargado en fragmentos de 64k cada vez usando una rutina de ayuda especial que llama a la BIOS para mover datos desde la memoria baja a la memoria alta. Esta rutina de ayuda es referida por `bootsect_kludge` en `bootsect.S` y es definida como `bootsect_helper` en `setup.S`. La etiqueta `bootsect_kludge` en `setup.S` contiene el valor del segmento de configuración y el desplazamiento del código `bootsect_helper` en él, por lo que el sector de arranque puede usar la instrucción `lcall` para saltar a él (salto entre segmentos). El motivo por lo cual esto es realizado en `setup.S` es simplemente porque no existe más espacio libre en bootsect.S (lo cual no es estrictamente verdad - hay aproximadamente 4 bytes dispersos y al menos 1 byte disperso en `bootsect.S`, pero que obviamente no es suficiente). Esta rutina usa el servicio de la BIOS int 0x15 (ax=0x8700) para moverlo a la memoria alta y restablecer %es al punto de siempre 0x10000. Esto asegura que el código en `bootsect.S` no se va fuera de memoria cuando está copiando datos desde disco.

1.5 Usando LILO como cargador de arranque

Existen varias ventajas en usar un cargador de arranque especializado (LILO) sobre un esqueleto desnudo de un sector de arranque:

1. Habilidad para escoger entre varios núcleos Linux o incluso múltiples Sistemas Operativos.

2. Habilidad para pasar parámetros a la linea de comandos del núcleo (existe un parche llamado BCP que añade esta habilidad al esqueleto desnudo de sector de arranque + configuración).
3. Habilidad para cargar núcleos bzImage más grandes - hasta los 2.5M vs 1M.

Viejas versiones de LILO (v17 y anteriores) no podían cargar núcleos bzImage. Las versiones más nuevas (como las de hace un par de años y posteriores) usan la misma técnica que el sector de arranque + configuración de mover datos desde la memoria baja a la memoria alta mediante los servicios de la BIOS. Alguna gente (notablemente Peter Anvin) argumentan que el soporte para zImage debería de ser quitado. El motivo principal (de acuerdo con Alan Cox) para que permanezca es que aparentemente existen algunas BIOS defectuosas que hacen imposible arrancar núcleos bzImage, mientras que la carga de núcleos zImage se realiza correctamente.

La última cosa que hace LILO es saltar a `setup.S` y entonces las cosas prosiguen de la forma normal.

1.6 Inicialización de Alto Nivel

Por "Inicialización de Alto Nivel" consideramos cualquier cosa que no está directamente relacionada con la fase de arranque, incluso aquellas partes del código que están escritas en ensamblador, esto es `arch/i386/kernel/head.S`, que es el comienzo del núcleo descomprimido. Los siguientes pasos son realizados:

1. Inicializa los valores de segmento (%ds = %es = %fs = %gs = __KERNEL_DS = 0x18).
2. Inicializa las tablas de páginas.
3. Habilita el paginamiento estableciendo el bit PG en %cr0.
4. Limpia a cero BSS (en SMP, sólo la primera CPU realiza esto).
5. Copia los primeros 2k de los parámetros de arranque (linea de comandos del núcleo).
6. Chequea el tipo de CPU usando EFLAGS y, si es posible, cpuid, capaz de detectar 386 y superiores.
7. La primera CPU llama a `start_kernel()`,y si ready=1 todas las otras llaman a `arch/i386/kernel/smpboot.c:initialize_secondary()` el cual recarga esp/eip y no retorna.

La función `init/main.c:start_kernel()` está escrita en C y realiza lo siguiente:

1. Realiza un cierre global del núcleo (es necesario para que sólo una CPU realice la inicialización).
2. Realiza configuraciones específicas de la arquitectura (análisis de la capa de memoria, copia de la linea de comandos de arranque otra vez, etc.).
3. Muestra el "anuncio" del núcleo Linux conteniendo la versión, el compilador usado para construirlo, etc ..., a la memoria intermedia con forma de anillo del núcleo para los mensajes. Esto es tomado desde la variable linux_banner definida en init/version.c y es la misma cadena mostrada por **cat /proc/version**.
4. Inicializa las traps.
5. Inicializa las irqs.
6. Inicializa los datos requeridos por el planificador (scheduler).
7. Inicializa el tiempo manteniendo los datos.

8. Inicializa el subsistema softirq.

9. Analiza las opciones del arranque de la linea de comandos.

10. Inicializa la consola.

11. Si el soporte para módulos ha sido compilado en el núcleo, inicializa la facilidad para la carga dinámica de módulos.

12. Si la linea de comandos "profile=" ha sido suministrada, inicializa los perfiles de memoria intermedia.

13. `kmem_cache_init()`, inicializa la mayoría del asignador slab.

14. Habilita las interrupciones.

15. Calcula el valor BogoMips para esta CPU.

16. Llama a `mem_init()`, que calcula `max_mapnr`, `totalram_pages` y `high_memory` y muestra la linea "Memory: ...".

17. `kmem_cache_sizes_init()`, finaliza la inicialización del asignador slab.

18. Inicializa las estructuras de datos usadas por procfs.

19. `fork_init()`, crea `uid_cache`, inicializa `max_threx_threads` basándose en la cantidad de memoria disponible y configura `RLIMIT_NPROC` para que `init_task` sea `max_threads/2`.

20. Crea varias antememorias slab necesitadas para VFS, VM, la antememoria intermedia, etc.

21. Si el soporte para System V IPC ha sido compilado en el núcleo, inicializa el subsistema. Nótese que para System V shm, esto incluye el montaje de una instancia (dentro del núcleo) del sistema de archivos shmfs.

22. Si el soporte de quota ha sido compilado en el núcleo, crea e inicializa una antememoria slab especial para él.

23. Realiza "chequeos de fallos" específicos de la arquitectura y, cuando es posible, activa las correcciones para los fallos de procesadores/bus/etc. Comparando varias arquitecturas vemos que "ia64 no tiene fallos" e "ia32 tiene unos pocos". Un buen ejemplo es el "fallo f00f", el cual es sólo chequeado si el núcleo ha sido compilado para menos de un 686 y corregido adecuadamente.

24. Establece una bandera para indicar que un planificador debería de ser llamado en la "siguiente oportunidad" y crea un hilo del núcleo `init()` que ejecuta execute_command si este ha sido suministrado a través del parámetro de inicio "init=", o intenta ejecutar **/sbin/init**, **/etc/init**, **/bin/init**, **/bin/sh** en este orden; si todos estos fallan, ocurre una situación de pánico con la "sugerencia" de usar el parámetro "init=".

25. Se va a un bucle vacío, este es un hilo vacío con pid=0.

Una cosa importante que hay que hacer notar aquí es que el hilo del núcleo `init()` llama a `do_basic_setup()`, el cual cuando vuelve llama a `do_initcalls()`, que va a través de la lista de funciones registradas por medio de las macros `__initcall` o `module_init()` y las invoca. Estas funciones no dependen de otras o sus dependencias han sido manualmente arregladas por el orden de enlazado en los Makefiles. Esto significa que, dependiendo de las posición de los directorios en los árboles y de las estructuras en los Makefiles, el orden en el cual estas funciones de inicialización son llamadas puede cambiar. A veces esto es importante, imagínate dos subsistemas A y B, con B dependiendo de alguna inicialización realizada por A. Si A es compilada estáticamente y B es un módulo entonces el punto de entrada de B está garantizado para ser llamado después de que A prepare todo el entorno necesario. Si A es un módulo, entonces

B es también necesariamente un módulo para que no existan problemas. Pero, ¿qué pasa si A y B están estáticamente enlazadas en el núcleo? El orden en el cual son llamadas depende del desplazamiento relativo del punto de entrada en la sección ELF `.initcall.init` de la imagen del núcleo. Rogier Wolff propuso introducir una infraestructura jerárquica de "prioridades" donde los módulos pueden dejar que el enlazador conozca en que orden (relativo) deberían de ser enlazados, pero todavía no existen parches disponibles que implementen esto de una forma suficientemente elegante para ser aceptada en el núcleo. Por consiguiente, asegúrate de que el orden de enlace es correcto, Si, en el ejemplo anterior, A y B trabajan bien cuando han sido compilados estáticamente una vez, trabajarán siempre, tal como han sido listados secuencialmente en el mismo Makefile. Si no trabajan, cambia el orden en el cual sus archivos objetos son listados.

Otra cosa de algún valor es la habilidad de Linux de ejecutar un "programa init alternativo" por medio del pase de la linea de comandos "init=". Esto es útil para la recuperación desde un /**sbin**/**init** accidentalmente sobreescrito o para depurar a mano los guiones de inicialización (rc) y `/etc/inittab`, ejecutándolos de uno en uno.

1.7 Arranque SMP en x86

En SMP, el BP (Procesador de arranque) va a través de la secuencia normal del sector de arranque, configuración, etc... hasta que llega a `start_kernel()`, y entonces sobre `smp_init()` y especialmente `src/i386/kernel/smpboot.c:smp_boot_cpus()`. La función `smp_boot_cpus()` entra en un buche para cada apicid (identificador de cada APIC), hasta `NR_CPUS`, y llama a `do_boot_cpu()` en él. Lo que hace `do_boot_cpu()` es crear (esto es: `fork_by_hand`) una tarea vacía para la cpu de destino y escribe en localizaciones bien conocidas definidas por la especificación Intel MP (0x467/0x469) el EIP del código del trampolín encontrado en `trampoline.S`. Entonces genera STARTUP IPI a la cpu de destino la cual hace que este AP (Procesador de Aplicación) ejecute el código en `trampoline.S`.

La CPU de arranque crea una copia del código trampolín para cada CPU en la memoria baja. El código del AP escribe un número mágico en su propio código, el cual es verificado por el BP para asegurarse que el AP está ejecutando el código trampolín. El requerimiento de que el código trampolín tenga que estar en la memoria baja es forzado por la especificación Intel MP.

El código trampolín simplemente establece el registro %bx a uno, entra en modo protegido y salta a startup_32, que es la entrada principal a `arch/i386/kernel/head.S`.

Ahora, el AP empieza ejecutando `head.S` y descubriendo que no es un BP, se salta el código que limpia BSS y entonces entra en `initialize_secondary()`, el cual justamente entra en la tarea vacía para esta CPU - recalcar que `init_tasks[cpu]` ya había sido inicializada por el BP ejecutando `do_boot_cpu(cpu)`.

Destacar que init_task puede ser compartido, pero cada hilo vacío debe de tener su propio TSS. Este es el motivo por el que `init_tss[NR_CPUS]` es una array.

1.8 Liberando datos y código de inicialización

Cuando el sistema operativo se inicializa a si mismo, la mayoría del código y estructuras de datos no se necesitarán otra vez. La mayoría de los sistemas operativos (BSD, FreeBSD, etc.) no pueden deshacerse de esta información innecesaria, gastando entonces valiosa memoria física del núcleo. El motivo que ellos no lo realizan (ver el libro de McKusick 4.4BSD) es que "el código relevante está propagado a través de varios subsistemas y por lo tanto no es factible liberarlo". Linux, por supuesto, no puede usar tal escusa porque bajo Linux "si en principio algo es posible, entonces ya está implementado o alguien está trabajando en ello".

Por lo tanto, como he dicho anteriormente, el núcleo Linux sólo puede ser compilado como un binario ELF, y ahora adivinamos el motivo (o uno de los motivos) para ello. El motivo referente a deshechar el código/datos de inicialización es que Linux suministra dos macros para ser usadas:

- `__init` - para el código de inicialización
- `__initdata` - para datos

Estas evalúan al atributo especificador gcc (también conocido como "gcc magic") tal como ha sido definido en `include/linux/init.h`:

```
#ifndef MODULE
#define __init        __attribute__ ((__section__ (".text.init")))
#define __initdata    __attribute__ ((__section__ (".data.init")))
#else
#define __init
#define __initdata
#endif
```

Lo que esto significa es que si el código es compilado estáticamente en el núcleo (MODULO no está definido), entonces es colocado en la sección especial ELF `.text.init`, el cual es declarado en el mapa del enlazado en `arch/i386/vmlinux.lds`. En caso contrario (si es un módulo) las macros no evalúan nada.

Lo que pasa durante el arranque es que el hilo del núcleo "init" (función `init/main.c:init()`) llama a la función específica de la arquitectura `free_initmem()` la cual libera todas las páginas entre las direcciones `__init_begin` e `__init_end`.

En un sistema típico (mi estación de trabajo), esto resulta en la liberación de unos 260K de memoria.

Las funciones registradas a través de `module_init()` son colocadas en `.initcall.init` el cual es también liberado en el caso estático. La actual tendencia en Linux, cuando se está diseñando un subsistema (no necesariamente un módulo), es suministrar puntos de entrada init/exit desde las etapas tempranas del diseño para que en el futuro, el subsistema en cuestión, pueda ser modularizado si se necesita. Un ejemplo de esto es pipefs, ver `fs/pipe.c`. Incluso si un subsistema nunca fuese convertido a módulo, ej. bdflush (ver `fs/buffer.c`), aún es bonito y arreglado usar la macro `module_init()` contra su función de inicialización, suministrada aunque no haga nada cuando la función es precisamente llamada.

Hay dos macros más, las cuales trabajan de una manera similar, llamadas `__exit` y `__exitdata`, pero ellas están más directamente conectadas al soporte de módulos por lo que serán explicadas en una sección posterior.

1.9 Procesando la linea de comandos del núcleo

Déjanos recalcar qué es lo que le pasa a la linea de comandos cuando se le pasa al núcleo durante el arranque:

1. LILO (o BCP) acepta la linea de comandos usando los servicios de teclado de la BIOS y los almacena en una localización bien conocida en la memoria física, también como una firma diciendo que allí existe una linea de comando válida.

2. `arch/i386/kernel/head.S` copia los primeros 2k de ella fuera de la página cero. Nótese que la actual versión de LILO (21) corta la linea de comandos a los 79 bytes. Esto es un fallo no trivial en LILO (cuando el soporte para EBDA grandes está activado) y Werner prometió arreglarlo próximamente. Si realmente necesitas pasarle lineas de comando más grandes de los 79 bytes, entonces puedes usar BCP o codificar tu linea de comandos en la función `arch/i386/kernel/setup.c:parse_mem_cmdline()`.

3. `arch/i386/kernel/setup.c:parse_mem_cmdline()` (llamada por `setup_arch()`, y esta llamada por `start_kernel()`) copia 256 bytes de la página cero a `saved_command_line` la cual es mostrada por `/proc/cmdline`. Esta misma rutina procesa la opción "mem=" si está presente y realiza los ajustes apropiados en los parámetros de la VM.

4. Volvemos a la línea de comandos en `parse_options()` (llamada por `start_kernel()`) el cual procesa algunos parámetros "dentro del núcleo" (actualmente "init=" y entorno/argumentos para init) y pasa cada palabra a `checksetup()`.

5. `checksetup()` va a través del código en la sección ELF `.setup.init` y llama a cada función, pasándole la palabra si corresponde. Nótese que usando el valor de retorno de 0 desde la función registrada a través de `__setup()`, es posible pasarle el mismo "variable=value" a más de una función con el "value" inválido a una y válido a otra. Jeff Garzik comentó: "los hackers que hacen esto son buenos :)" ¿Por qué? Porque esto es claramente específico del orden de enlazado, esto es, el enlazado del núcleo en un orden tendrá a la llamada de la funciónA antes de la funciónB y otro los tendrá en orden inverso, con el resultado dependiendo del orden.

Por lo tanto, ¿cómo debemos de escribir el código que procesa la linea de comandos del arranque? Nosotros usamos la macro `__setup()` definida en `include/linux/init.h`:

```
/*
 * Usado por la configuración de parámetros de la línea de comandos
 * del núcleo
 */
struct kernel_param {
        const char *str;
        int (*setup_func)(char *);
};

extern struct kernel_param __setup_start, __setup_end;

#ifndef MODULE
#define __setup(str, fn) \
   static char __setup_str_##fn[] __initdata = str; \
   static struct kernel_param __setup_##fn __initsetup = \
   { __setup_str_##fn, fn }

#else
#define __setup(str,func) /* nada */
endif
```

Por lo tanto, típicamente la usarás en tu código de esta forma (tomado del código del controlador real, BusLogic HBA `drivers/scsi/BusLogic.c`):

```
static int __init
BusLogic_Setup(char *str)
{
        int ints[3];

        (void)get_options(str, ARRAY_SIZE(ints), ints);

        if (ints[0] != 0) {
                BusLogic_Error("BusLogic: Obsolete Command Line Entry "
                                "Format Ignored\n", NULL);
                return 0;
        }
        if (str == NULL || *str == '\0')
                return 0;
        return BusLogic_ParseDriverOptions(str);
```

```
}

__setup("BusLogic=", BusLogic_Setup);
```

Destacar que `__setup()` no hace nada por los módulos, por lo tanto el código que quiere procesar la linea de comandos del arranque, que puede ser un módulo o estar estáticamente enlazado, debe de ser llamado pasándole la función manualmente en la rutina de inicialización del módulo. Esto también significa que es posible escribir código que procese los parámetros cuando es compilado como un módulo pero no cuando es estático o viceversa.

2 Procesos y Manejo de Interrupciones

2.1 Estructura de Tareas y Tabla de Procesos

Cada proceso bajo Linux es dinámicamente asignado a una estructura `struct task_struct`. El número máximo de procesos que pueden ser creados bajo Linux está solamente limitado por la cantidad de memoria física presente, y es igual a (ver `kernel/fork.c:fork_init()`):

```
        /*
         * El número máximo por defecto de hilos es establecido
         * a un valor seguro: las estructuras de hilos pueden ocupar al
         * menos la mitad de la memoria.
         */
        max_threads = mempages / (THREAD_SIZE/PAGE_SIZE) / 2;
```

lo cual, en la arquitectura IA32, básicamente significa `num_physpages/4`. Como ejemplo, en una máquina de 512M, puedes crear 32k de hilos. Esto es una mejora considerable sobre el límite de 4k-epsilon para los núcleos viejos (2.2 y anteriores). Es más, esto puede ser cambiado en tiempo de ejecución usando el KERN_MAX_THREADS **sysctl(2)**, o simplemente usando la interfaz procfs para el ajuste del núcleo:

```
# cat /proc/sys/kernel/threads-max
32764
# echo 100000 > /proc/sys/kernel/threads-max
# cat /proc/sys/kernel/threads-max
100000
# gdb -q vmlinux /proc/kcore
Core was generated by ‘BOOT_IMAGE=240ac18 ro root=306 video=matrox:vesa:0x118’.
#0  0x0 in ?? ()
(gdb) p max_threads
$1 = 100000
```

El conjunto de procesos en el sistema Linux está representado como una colección de estructuras `struct task_struct`, las cuales están enlazadas de dos formas:

1. como una tabla hash, ordenados por el pid, y
2. como una lista circular doblemente enlazada usando los punteros `p->next_task` y `p->prev_task`.

La tabla hash es llamada `pidhash[]` y está definida en `include/linux/sched.h`:

```
/* PID hashing. (¿no debería de ser dinámico?) */
#define PIDHASH_SZ (4096 >> 2)
extern struct task_struct *pidhash[PIDHASH_SZ];

#define pid_hashfn(x)   ((((x) >> 8) ^ (x)) & (PIDHASH_SZ - 1))
```

Las tareas son ordenadas por su valor pid y la posterior función de ordenación se supone que distribuye los elementos uniformemente en sus dominios de (`0` a `PID_MAX-1`). La tabla hash es usada para encontrar rápidamente una tarea por su pid usando `find_task_pid()` dentro de `include/linux/sched.h`:

```
static inline struct task_struct *find_task_by_pid(int pid)
{
        struct task_struct *p, **htable = &pidhash[pid_hashfn(pid)];

        for(p = *htable; p && p->pid != pid; p = p->pidhash_next)
                ;

        return p;
}
```

Las tareas en cada lista ordenada (esto es, ordenadas por el mismo valor) son enlazadas por `p->pidhash_next/pidhash_pprev` el cual es usado por `hash_pid()` y `unhash_pid()` para insertar y quitar un proceso dado en la tabla hash. Esto es realizado bajo la protección del spinlock read/write (lectura/escritura) llamado `tasklist_lock` tomado para ESCRITURA.

La lista circular doblemente enlazada que usa `p->next_task/prev_task` es mantenida para que uno pueda ir fácilmente a través de todas las tareas del sistema. Esto es realizado por la macro `for_each_task()` desde `include/linux/sched.h`:

```
#define for_each_task(p) \
        for (p = &init_task ; (p = p->next_task) != &init_task ; )
```

Los usuarios de `for_each_task()` deberían de coger la tasklist_lock para LECTURA. Destacar que `for_each_task()` está usando `init_task` para marcar el principio (y el final) de la lista - esto es seguro porque la tarea vacía (pid 0) nunca existe.

Los modificadores de los procesos de la tabla hash y/o los enlaces de la tabla de procesos, notablemente `fork()`, `exit()` y `ptrace()`, deben de coger la `tasklist_lock` para ESCRITURA. El motivo por el que esto es interesante es porque los escritores deben de deshabilitar las interrupciones en la CPU local. El motivo para esto no es trivial: la función `send_sigio()` anda por la lista de tareas y entonces coge `tasklist_lock` para ESCRITURA, y esta es llamada desde `kill_fasync()` en el contexto de interrupciones. Este es el motivo por el que los escritores deben de deshabilitar las interrupciones mientras los lectores no lo necesitan.

Ahora que entendemos cómo las estructuras `task_struct` son enlazadas entre ellas, déjanos examinar los miembros de `task_struct`. Ellos se corresponden débilmente con los miembros de las estructuras de UNIX 'struct proc' y 'struct user' combinadas entre ellas.

Las otras versiones de UNIX separan la información del estado de las tareas en una parte, la cual deberá de ser mantenida en memoria residente durante todo el tiempo (llamada 'proc structure' la cual incluye el estado del proceso, información de planificación, etc.), y otra parte, la cual es solamente necesitada cuando el proceso está funcionando (llamada 'u_area' la cual incluye la tabla de descriptores de archivos, información sobre la cuota de disco etc.). El único motivo para este feo diseño es que la memoria era un recurso muy escaso. Los sistemas operativos modernos (bueno, sólo Linux por el momento, pero otros, como FreeBSD (que parece que avanzan en esta dirección, hacia Linux) no necesitan tal separación y entonces mantienen el estado de procesos en una estructura de datos del núcleo residente en memoria durante todo el tiempo.

La estructura task_struct está declarada en `include/linux/sched.h` y es actualmente de un tamaño de 1680 bytes.

El campo de estado es declarado como:

```
volatile long state;    /* -1 no ejecutable, 0 ejecutable, >0 parado */
```

```
#define TASK_RUNNING            0
#define TASK_INTERRUPTIBLE      1
#define TASK_UNINTERRUPTIBLE    2
#define TASK_ZOMBIE             4
#define TASK_STOPPED            8
#define TASK_EXCLUSIVE          32
```

¿Por qué `TASK_EXCLUSIVE` está definido como 32 y no cómo 16? Porque 16 fue usado por `TASK_SWAPPING` y me olvidé de cambiar `TASK_EXCLUSIVE` cuando quité todas las referencias a `TASK_SWAPPING` (en algún sitio en 2.3.x).

La declaración `volatile` en `p->state`significa que puede ser modificada asincrónicamente (desde el manejador de interrupciones);

1. **TASK_RUNNING**: significa que la tarea está "supuestamente" en la cola de ejecución. El motivo por lo que quizás no esté aún en la cola de ejecución es porque marcar una tarea como `TASK_RUNNING` y colocarla en la cola de ejecución no es atómico. Necesitarás mantener el spinlock read/write `runqueue_lock` en lectura para mirar en la cola de ejecución. Si lo haces, verás que cada tarea en la cola de ejecución está en el estado `TASK_RUNNING`. Sin embargo, la conversión no es verdad por los motivos explicados anteriormente. De una forma parecida, los controladores pueden marcarse a ellos mismos (o en realidad, en el contexto del proceso en el que están) como `TASK_INTERRUPTIBLE` (o `TASK_UNINTERRUPTIBLE`) y entonces llaman a `schedule()`, el cual entonces los quita de la cola de ejecución (a memos que exista una señal pendiente, en tal caso permanecen en la cola de ejecución).
2. **TASK_INTERRUPTIBLE**: significa que la tarea está durmiendo pero que puede ser despertada por una señal o por la terminación de un cronómetro.
3. **TASK_UNINTERRUPTIBLE**: lo mismo que `TASK_INTERRUPTIBLE`, excepto que no puede ser despertado.
4. **TASK_ZOMBIE**: tareas que han terminado pero que no tienen su estado reflejado (para `wait()`-ed) por el padre (natural o por adopción).
5. **TASK_STOPPED**: tarea que fue parada, por señales de control de trabajos o por **ptrace(2)**.
6. **TASK_EXCLUSIVE**: este no es un estado separado pero puede ser uno de `TASK_INTERRUPTIBLE` o `TASK_UNINTERRUPTIBLE`. Esto significa que cuando esta tarea está durmiendo o un una cola de espera con otras tareas, puede ser despertada sóla en vez de causar el problema de "movimiento general" despertando a todos los que están esperando.

Las banderas de las tareas contienen información sobre los estados de los procesos, los cuales no son mutuamente exclusivos:

```
unsigned long flags;    /* banderas para cada proceso, definidas abajo */
/*
 * Banderas para cada proceso
 */
#define PF_ALIGNWARN    0x00000001      /* Imprime mensajes de peligro de alineación */
                                        /* No implementada todavía, solo para 486 */
#define PF_STARTING     0x00000002      /* Durante la creación */
#define PF_EXITING      0x00000004      /* Durante la destrucción */
#define PF_FORKNOEXEC   0x00000040      /* Dividido pero no ejecutado */
#define PF_SUPERPRIV    0x00000100      /* Usados privilegios de super-usuario */
#define PF_DUMPCORE     0x00000200      /* Núcleo volcado */
#define PF_SIGNALED     0x00000400      /* Asesinado por una señal */
#define PF_MEMALLOC     0x00000800      /* Asignando memoria */
```

```
#define PF_VFORK        0x00001000      /* Despertar al padre en mm_release */
#define PF_USEDFPU      0x00100000      /* La tarea usó de FPU este quantum (SMP) */
```

Los campos `p->has_cpu`, `p->processor`, `p->counter`, `p->priority`, `p->policy` y `p->rt_priority` son referentes al planificador y serán mirados más tarde.

Los campos `p->mm` y `p->active_mm` apuntan respectivamente a la dirección del espacio del proceso descrita por la estructura `mm_struct` y al espacio activo de direcciones, si el proceso no tiene una verdadera (ej, hilos de núcleo). Esto ayuda a minimizar las descargas TLB en los intercambios del espacio de direcciones cuando la tarea es descargada. Por lo tanto, si nosotros estamos planificando el hilo del núcleo (el cual no tiene `p->mm`) entonces su `next->active_mm` será establecido al `prev->active_mm` de la tarea que fue descargada, la cual será la misma que `prev->mm` si `prev->mm != NULL`. El espacio de direcciones puede ser compartido entre hilos si la bandera `CLONE_VM` es pasada a las llamadas al sistema **clone(2)** o **vfork(2)**.

Los campos `p->exec_domain` y `p->personality` se refieren a la personalidad de la tarea, esto es, la forma en que ciertas llamadas al sistema se comportan para emular la "personalidad" de tipos externos de UNIX.

El campo `p->fs` contiene información sobre el sistema de archivos, lo cual significa bajo Linux tres partes de información:

1. dentry del directorio raíz y punto de montaje,
2. dentry de un directorio raíz alternativo y punto de montaje,
3. dentry del directorio de trabajo actual y punto de montaje.

Esta estructura también incluye un contador de referencia porque puede ser compartido entre tareas clonadas cuando la bandera `CLONE_FS` es pasada a la llamada al sistema **clone(2)**.

El campo `p->files` contiene la tabla de descriptores de ficheros. Esto también puede ser compartido entre tareas, suministrando `CLONE_FILES` el cual es especificado con **clone(2)**.

El campo `p->sig` contiene los manejadores de señales y puede ser compartido entre tareas clonadas por medio de `CLONE_SIGHAND`.

2.2 Creación y terminación de tareas e hilos del núcleo

Diferentes libros sobre sistemas operativos definen un "proceso" de diferentes formas, empezando por "instancia de un programa en ejecución" y finalizando con "lo que es producido por las llamadas del sistema clone(2) o fork(2)". Bajo Linux, hay tres clases de procesos:

- el/los hilo(s) vacío(s),
- hilos del núcleo,
- tareas de usuario.

El hilo vacío es creado en tiempo de compilación para la primera CPU; es entonces creado "manualmente" para cada CPU por medio de la función específica de la arquitectura `fork_by_hand()` en `arch/i386/kernel/smpboot.c`, el cual desenrolla la llamada al sistema **fork(2)** a mano (en algunas arquitecturas). Las tareas vacías comparten una estructura init_task pero tienen una estructura privada TSS, en la matriz de cada CPU `init_tss`. Todas las tareas vacías tienen pid = 0 y ninguna otra tarea puede compartir el pid, esto es, usar la bandera `CLONE_PID` en **clone(2)**.

Los hilos del núcleo son creados usando la función `kernel_thread()` la cual invoca a la llamada al sistema **clone(2)** en modo núcleo. Los hilos del núcleo usualmente no tienen espacio de direcciones de usuario, esto es p->mm = NULL, porque ellos explicitamente hacen exit_mm(), ej. a través de la función daemonize(). Los hilos del núcleo siempre pueden acceder al espacio de direcciones del núcleo directamente. Ellos son asignados a números pid en el rango bajo. Funcionando en el anillo del procesador 0 (en x86) implica que los hilos del núcleo disfrutan de todos los privilegios de E/S y no pueden ser pre-desocupados por el planificador.

Las tareas de usuario son creadas por medio de las llamadas al sistema **clone(2)** o **fork(2)**, las cuales internamente invocan a **kernel/fork.c:do_fork()**.

Déjenos entender qué pasa cuando un proceso de usuario realiza una llamada al sistema **fork(2)**. Como **fork(2)** es dependiente de la arquitectura debido a las diferentes formas de pasar la pila y registros de usuario, la actual función subyacente do_fork() que hace el trabajo es portable y está localizada en kernel/fork.c.

Los siguientes pasos son realizados:

1. La variable local retval es establecida a -ENOMEM, ya que este es el valor al que errno debería de ser establecida si **fork(2)** falla al asignar una nueva estructura de tarea.
2. Si CLONE_PID es establecido en clone_flags entonces devuelve un error (-EPERM), a menos que el llamante sea el hilo vacío (sólo durante el arranque). Por lo tanto, un hilo de un usuario normal no puede pasar CLONE_PID a **clone(2)** y esperar que tenga éxito. Para **fork(2)**, es irrelevante que clone_flags sea establecido a SIFCHLD - esto es sólo relevante cuando do_fork() es invocado desde sys_clone() el cual pasa clone_flags desde el valor pedido desde el espacio de usuario.
3. current->vfork_sem es inicializado (es más tarde limpiado en el hijo). Esto es usado por sys_vfork() (la llamada al sistema **vfork(2)** corresponde a clone_flags = CLONE_VFORK|CLONE_VM|SIGCHLD) para hacer que el padre duerma mientras el hijo hace mm_release(), por ejemplo como resultado de exec() (ejecutar) otro programa o **exit(2)**.
4. Una nueva estructura de tarea es asignada usando la macro dependiente de la arquitectura alloc_task_struct(). En x86 es justo un gfp a la prioridad GFP_KERNEL. Este es el primer motivo por el que la llamada **fork(2)** quizás duerma. Si la primera asignación falla, devolvemos -ENOMEM.
5. Todos los valores de la actual estructura de tareas del proceso son copiadas en la nueva, usando un asignamiento de estructura *p = *current. ¿Quizás debería de ser reemplazada por un establecimiento de memoria? Más tarde, los campos que no deberían de ser heredados por el hijo son establecidos a los valores correctos.
6. Un gran cierre del núcleo es tomado durante el resto del código ya que en otro caso sería no reentrante.
7. Si el padre no tiene recursos de usuario (un concepto de UID, Linux es suficientemente flexible para hacer de ellos una cuestión mejor que un hecho), entonces verifica si el límite blando de usuario RLIMIT_NPROC ha sido excedido - si lo es, falla con -EAGAIN, si no, incrementa la cuenta de procesos con el uid dado p->user->count.
8. Si el número de tareas a lo largo del sistema excede el valor de max_threads (recordar que es ajustable), falla con -EAGAIN.
9. Si el binario que se ejecuta pertenece a un dominio modularizado de ejecución, incrementa el contador de referencia del correspondiente módulo.
10. Si el binario que se ejecuta pertenece a un formato binario modularizado, incrementa el contador de referencia del módulo correspondiente.
11. El hijo es marcado como 'no tiene que ser ejecutado' (p->did_exec = 0)
12. El hijo es marcado como 'no intercambiable' (p->swappable = 0)

13. EL hijo es puesto en el estado 'durmiendo no interrumpible', esto es `p->state = TASK_UNINTERRUPTIBLE` (POR HACER: ¿por qué es realizado esto? Creo que no se necesita - librarse de el, Linus confirma que no se necesita)
14. Las `p->flags` del hijo son establecidas de acuerdo a los valores de clone_flags; para **fork(2)** limpias, esto será `p->flags = PF_FORKNOEXEC`.
15. El pid del hijo `p->pid` es establecido usando el algoritmo rápido en `kernel/fork.c:get_pid()` (POR HACER: el spinlock `lastpid_lock` puede ser redundante ya que `get_pid()` siempre es llamado bajo un gran cierre del núcleo desde `do_fork()`, también quita los argumentos bandera de `get_pid()`, parche enviado a Alan el 20/06/2000 - mirar después).
16. El resto del código en `do_fork()` inicializa el resto de la estructura de la tarea del hijo. Muy al final, la estructura de tarea del hijo es ordenada en la tabla hash `pidhash` y el hijo es despertado. (POR HACER: `wake_up_process(p)` establece `p->state = TASK_RUNNING` y añade el proceso a la cola de ejecución, entonces probablemente no necesita establecer `p->state` a `TASK_RUNNING` tempranamente en `do_fork()`). La parte interesante es establecer `p->exit_signal` a `clone_flags & CSIGNAL`, la cual para **fork(2)** significa justamente `SIGCHLD` y establece `p->pdeath_signal` a 0. La `pdeath_signal` es usada cuando un proceso 'olvida' el padre original (durante la muerte) y puede ser establecido/tomado por medio del comando `PR_GET/SET_PDEATHSIG` de la llamada al sistema **prctl(2)** (Tu quizás argumentes que la forma en la que el valor de `pdeath_signal` es devuelto a través de un argumento de un puntero del espacio de usuario en **prctl(2)** es un poco tonto - mea culpa, después de que Andries Brouwer actualizara la página man era muy tarde para arreglarlo ;)

Entonces las tareas son creadas. Hay varias formas para la terminación de tareas:

1. haciendo la llamada al sistema **exit(2)**;
2. enviando un señal con la disposición por defecto de morir;
3. siendo forzado a morir bajo ciertas excepciones;
4. llamando **bdflush(2)** con `func == 1` (esto es específico de Linux, para compatibilización de viejas distribuciones que todavía tienen la linea 'update' en `/etc/inittab` - hoy en día el trabajo de update es hecho por el hilo del núcleo `kupdate`).

Las funciones implementando llamadas al sistema bajo Linux son prefijadas con `sys_`, pero ellas son usualmente concernientes sólo al chequeo de argumentos o a formas específicas de la arquitectura de pasar alguna información y el trabajo actual es realizado por las funciones `do_`. Por lo tanto, es con `sys_exit()` el cual llama a `do_exit()` para hacer el trabajo. Aunque otras partes del núcleo a veces invocan a `sys_exit()` mientras que deberían realmente de llamar a `do_exit()`.

La función `do_exit()` es encontrada en `kernel/exit.c`. Los puntos que destacar sobre `do_exit()` son;

- Usa un cierre global del núcleo (cierra pero no abre).
- Llama `schedule()` al final, el cual nunca regresa.
- Establece el estado de tareas a `TASK_ZOMBIE`.
- Notifica cualquier hijo con `current->pdeath_signal`, si no 0.
- Notifica al padre con una `current->exit_signal`, el cual es usualmente igual a `SIGCHLD`.
- Libera los recursos asignador por fork, cierra los archivos abiertos, etc,
- En arquitecturas que usan FPU lentas (ia64, mips, mips64) (POR HACER: quitar el argumento 'flags' de sparc, sparc64), realiza lo que el hardware requiera para pasar la FPU al dueño (si el dueño es el actual) a "none" (ninguno).

2.3 Planificador Linux

El trabajo de un planificador es decidir el acceso a la actual CPU entre múltiples procesos. El planificador está implementado en el 'archivo principal del núcleo' `kernel/sched.c`. El archivo de cabeceras correspondiente `include/linux/sched.h` está incluido virtualmente (explícita o implicitamente) en todos los archivos de código fuente del núcleo.

Los campos de una estructura de tareas relevante a planificar incluyen:

- `p->need_resched`: este campo es establecido si `schedule()` debería de ser llamado en la 'siguiente oportunidad'.
- `p->counter`: número de ticks de reloj que quedan en esta porción de tiempo del planificador, decrementada por un cronómetro. Cuando este campo se convierte a un valor menor o igual a cero, es reinicializado a 0 y `p->need_resched` es establecido. Esto también es llamado a veces 'prioridad dinámica' de un proceso porque puede cambiarse a si mismo.
- `p->priority`: la prioridad estática del proceso, sólo cambiada a través de bien conocidas llamadas al sistema como **nice(2)**, POSIX.1b **sched_setparam(2)** o 4.4BSD/SVR4 **setpriority(2)**.
- `p->rt_priority`: prioridad en tiempo real.
- `p->policy`: la política de planificación, específica a la clase de planificación que pertenece la tarea. Las tareas pueden cambiar su clase de planificación usando la llamada al sistema **sched_setscheduler(2)**. Los valores válidos son `SCHED_OTHER` (proceso UNIX tradicional), `SCHED_FIFO` (proceso FIFO en tiempo real POSIX.1b) y `SCHED_RR` (proceso en tiempo real round-robin POSIX). Uno puede también `SCHED_YIELD` a alguno de esos valores para significar que el proceso decidió dejar la CPU, por ejemplo llamando a la llamada al sistema **sched_yield(2)**. Un proceso FIFO en tiempo real funcionará hasta que: a) se bloquee en una E/S, b) explícitamente deje la CPU, o c) es predesocupado por otro proceso de tiempo real con un valor más alto de `p->rt_priority`. `SCHED_RR` es el mismo que `SCHED_FIFO`, excepto que cuando su porción de tiempo acaba vuelve al final de la cola de ejecutables.

EL algoritmo de planificación es simple, olvídate de la gran complejidad aparente de la función `schedule()`. La función es compleja porque implementa tres algoritmos de planificación en uno y también porque disimula los específicos de SMP.

Las aparentemente 'inservibles' etiquetas (gotos) en `schedule()` están allí con el propósito de generar el mejor código optimizado (para i386). También, destacar que el planificador (como la mayoría del núcleo) fue totalmente reescrito para el 2.4, entonces la discusión de más abajo no se aplica a los núcleos 2.2 o anteriores.

Déjanos mirar la función en detalle:

1. Si `current->active_mm == NULL` entonces algo está mal. El actual proceso, incluso un hilo del núcleo (`current->mm == NULL`) debe de tener un `p->active_mm` válido durante todo el tiempo.
2. Si hay algo que hacer en la cola de tareas `tq_scheduler`, entonces se procesa ahora. La cola de tareas suministra al núcleo un mecanismo para planificar la ejecución de las funciones más tarde. Lo miraremos en detalle en otra parte.
3. Se inicializan las variables locales `prev` y `this_cpu` a las tareas y CPUs actuales, respectivamente.
4. Se chequea si `schedule()` fue llamada desde el controlador de interrupciones (debido a un fallo) y provoca un pánico si ha sido así.
5. Se quita el cierre global del núcleo.

6. Si hay algún trabajo que hacer a través del mecanismo de softirq, se hace ahora.

7. Se inicializa el puntero local `struct schedule_data *sched_data` para que apunte a cada CPU (alineado de la linea de antememoria para prevenir que la linea de antememoria salte) planificando el área de datos, el cual contiene el valor TSC de `last_schedule` y el puntero a la última estructura planificada (POR HACER: `sched_data` es usada sólo en SMP, ¿pero porqué inicializa también `init_idle()` en UP (monoprocesadores)?

8. Es tomado el spinlock `runqueue_lock`. Destacar que usamos `spin_lock_irq()` porque en `schedule()` garantizamos que las interrupciones están habilitadas. Por esto, cuando abrimos `runqueue_lock`, podemos rehabilitarlas en vez de salvar/restaurar las eflags (variante `spin_lock_irqsave/restore`).

9. Estado de tareas de la máquina: si la tarea está en el estado `TASK_RUNNING` entonces se deja sólo, si está en el estado `TASK_INTERRUPTIBLE` y hay una señal pendiente, es movido al estado `TASK_RUNNING`. En todos los otros casos es borrado de la cola de ejecución.

10. `next` (mejor candidato para ser planificado) es establecido a la tarea vacía de esta CPU. En todo caso, la virtud de este candidato es establecida a un valor muy bajo (-1000), con la esperanza de que haya otro mejor que él.

11. Si la tarea `prev` (actual) está en el estado `TASK_RUNNING` entonces la actual virtud es establecida a su virtud y es marcado como mejor candidato para ser planificado que la tarea vacía.

12. Ahora la cola de ejecución es examinada y una virtud de cada proceso que puede ser planificado en esta CPU es comparado con el actual valor; el proceso con la virtud más alta gana. Ahora el concepto de "puede ser planificado en esta CPU" debe de ser clarificado: en UP, todos los procesos en la cola de ejecución son elegibles para ser planificados; en SMP, sólo los procesos que no estean corriendo en otra CPU son elegibles para ser planificados en esta CPU. La virtud es calculada por una función llamada `goodness()`, la cual trata los procesos en tiempo real haciendo sus virtudes muy altas (`1000 + p->rt_priority`), siendo mayor que 1000 se garantiza que no puede ganar otro proceso `SCHED_OTHER`; por lo tanto sólo compiten con los otros procesos en tiempo real que quizás tengan un mayor `p->rt_priority`. La función virtud devuelve 0 si la porción de tiempo del proceso (`p->counter`) se acabó. Para procesos que no son en tiempo real, el valor inicial de la virtud es establecido a `p->counter` - por este camino, el proceso tiene menos posibilidades para alcanzar la CPU si ya la tuvo por algún tiempo, esto es, los procesos interactivos son favorecidos más que el límite de impulsos de la CPU. La constante específica de la arquitectura `PROC_CHANGE_PENALTY` intenta implementar la "afinidad de cpu" (esto es, dar ventaja a un proceso en la misma CPU). También da una ligera ventaja a los procesos con mm apuntando al actual `active_mm` o a procesos sin espacio de direcciones (de usuario), esto es, hilos del núcleo.

13. si el actual valor de la virtud es 0 entonces la lista entera de los procesos (¡no sólo los de la lista de ejecutables!) es examinada y sus prioridades dinámicas son recalculadas usando el simple algoritmo:

```
recalculate:
        {
                struct task_struct *p;
                spin_unlock_irq(&runqueue_lock);
                read_lock(&tasklist_lock);
                for_each_task(p)
                        p->counter = (p->counter >> 1) + p->priority;
                read_unlock(&tasklist_lock);
                spin_lock_irq(&runqueue_lock);
        }
```

Destacar que tiramos el `runqueue_lock` antes de recalcular. El motivo para esto es que vamos a través del conjunto entero de procesos; esto puede llevar un gran tiempo, durante el cual el `schedule()` puede ser llamado por otra CPU y seleccionar un proceso con la suficiente virtud para esta CPU, mientras que nosotros en esta CPU seremos obligados a recalcular. Muy bien, admitamos que esto es algo inconsistente porque mientras que nosotros (en esta CPU) estamos seleccionando un proceso con la mejor virtud, `schedule()` corriendo en otra CPU podría estar recalculando las prioridades dinámicas.

14. Desde este punto, es cierto que `next` apunta a la tarea a ser planificada, por lo tanto debemos de inicializar `next->has_cpu` a 1 y `next->processor` a `this_cpu`. La `runqueue_lock` puede ahora ser abierta.

15. Si estamos volviendo a la misma tarea (`next == prev`) entonces podemos simplemente readquirir un cierre global del núcleo y volver, esto es, saltar todos los niveles hardware (registros, pila, etc.) y el grupo relacionado con la VM (Memoria Virtual) (cambiar la página del directorio, recalcular `active_mm` etc.)

16. La macro `switch_to()` es específica de la arquitectura. En i386, es concerniente con: a) manejo de la FPU (Unidad de Punto Flotante), b) manejo de la LDT, c) recargar los registros de segmento, d) manejo de TSS y e) recarga de los registros de depuración.

2.4 Implementación de la lista enlazada (de) Linux

Antes de ir a examinar las implementación de las colas de espera, debemos de informarnos con la implementación estándar de la lista doblemente enlazada Linux. Las colas de espera (igual que todo lo demás en Linux) hacen un uso fuerte de ellas y entonces son llamadas en la jerga "implementación list.h" porque el archivo más relevante es `include/linux/list.h`.

La estructura de datos fundamental aquí es `struct list_head`:

```
struct list_head {
        struct list_head *next, *prev;
};

#define LIST_HEAD_INIT(name) { &(name), &(name) }

#define LIST_HEAD(name) \
        struct list_head name = LIST_HEAD_INIT(name)

#define INIT_LIST_HEAD(ptr) do { \
        (ptr)->next = (ptr); (ptr)->prev = (ptr); \
} while (0)

#define list_entry(ptr, type, member) \
        ((type *)((char *)(ptr)-(unsigned long)(&((type *)0)->member)))

#define list_for_each(pos, head) \
        for (pos = (head)->next; pos != (head); pos = pos->next)
```

Las tres primeras macros son para inicializar un lista vacía apuntando los punteros `next` y `prev` a ellos mismos. Esto es obvio debido a las restricciones sintácticas de C, las cuales deberían de ser usadas aquí - por ejemplo, `LIST_HEAD_INIT()` puede ser usada para la inicialización de elementos de la estructura en la declaración, la segunda puede ser usada para la inicialización de las declaraciones de variables estáticas y la tercera puede ser usada dentro de la función.

La macro `list_entry()` da acceso individual a los elementos de la lista, por ejemplo (desde `fs/file_table.c:fs_may_remount_ro()`):

```
struct super_block {
   ...
   struct list_head s_files;
   ...
} *sb = &some_super_block;

struct file {
   ...
   struct list_head f_list;
   ...
} *file;

struct list_head *p;

for (p = sb->s_files.next; p != &sb->s_files; p = p->next) {
     struct file *file = list_entry(p, struct file, f_list);
     haz algo a 'file'
}
```

Un buen ejemplo del uso de la macro `list_for_each()` está en el planificador, donde andamos a través de la cola de ejecución buscando al proceso con la virtud más alta:

```
static LIST_HEAD(runqueue_head);
struct list_head *tmp;
struct task_struct *p;

list_for_each(tmp, &runqueue_head) {
    p = list_entry(tmp, struct task_struct, run_list);
    if (can_schedule(p)) {
        int weight = goodness(p, this_cpu, prev->active_mm);
        if (weight > c)
            c = weight, next = p;
    }
}
```

Aquí, `p->run_list` es declarada como `struct list_head run_list` dentro de la estructura `task_struct` y sirve como ancla de la lista. Quitando y añadiendo (al principio o al final de la lista) un elemento de la lista es hecho por las macros `list_del()/list_add()/list_add_tail()`. Los ejemplos siguientes están añadiendo y quitando una tarea de la cola de ejecución:

```
static inline void del_from_runqueue(struct task_struct * p)
{
        nr_running--;
        list_del(&p->run_list);
        p->run_list.next = NULL;
}

static inline void add_to_runqueue(struct task_struct * p)
{
        list_add(&p->run_list, &runqueue_head);
        nr_running++;
}
```

```
static inline void move_last_runqueue(struct task_struct * p)
{
        list_del(&p->run_list);
        list_add_tail(&p->run_list, &runqueue_head);
}

static inline void move_first_runqueue(struct task_struct * p)
{
        list_del(&p->run_list);
        list_add(&p->run_list, &runqueue_head);
}
```

2.5 Colas de espera

Cuando un proceso solicita el núcleo para hacer algo que es actualmente imposible pero que quizás sea posible más tarde, el proceso es puesto a dormir y es despertado cuando la solicitud tiene más probabilidades de ser satisfecha. Uno de los mecanismos del núcleo usados para esto es llamado 'cola de espera'.

La implementación de Linux nos permite despertar usando la bandera `TASK_EXCLUSIVE`. Con las colas de espera, también puedes usar una cola bien conocida y entonces simplificar `sleep_on/sleep_on_timeout/interruptible_sleep_on/interruptible_sleep_on_timeout`, o puedes definir tu propia cola de espera y usar `add/remove_wait_queue` para añadir y quitarte desde ella y `wake_up/wake_up_interruptible` para despertar cuando se necesite.

Un ejemplo del primer uso de las colas de espera es la interacción entre el asignador de páginas (en `mm/page_alloc.c:__alloc_pages()`) y el demonio del núcleo `kswapd` (en `mm/vmscan.c:kswap()`), por medio de la cola de espera `kswapd_wait`, declarada en `mm/vmscan.c`; el demonio `kswapd` duerme en esta cola, y es despertado cuando el asignador de páginas necesita liberar algunas páginas.

Un ejemplo del uso de una cola de espera autónoma es la interacción entre la solicitud de datos de un proceso de usuario a través de la llamada al sistema **read(2)** y el núcleo funcionando en el contexto de interrupción para suministrar los datos. Un manejador de interrupciones quizás se parezca a (`drivers/char/rtc_interrupt()` simplificado):

```
static DECLARE_WAIT_QUEUE_HEAD(rtc_wait);

void rtc_interrupt(int irq, void *dev_id, struct pt_regs *regs)
{
        spin_lock(&rtc_lock);
        rtc_irq_data = CMOS_READ(RTC_INTR_FLAGS);
        spin_unlock(&rtc_lock);
        wake_up_interruptible(&rtc_wait);
}
```

Por lo tanto, el manejador de interrupciones obtiene los datos leyendo desde algún puerto de E/S específico del dispositivo (la macro `CMOS_READ()` devuelve un par de `outb/inb`) y entonces despierta a quien esté durmiendo en la cola de espera `rtc_wait`.

Ahora, la llamada al sistema **read(2)** puede ser implementada como:

```
ssize_t rtc_read(struct file file, char *buf, size_t count, loff_t *ppos)
{
        DECLARE_WAITQUEUE(wait, current);
        unsigned long data;
        ssize_t retval;
```

```
        add_wait_queue(&rtc_wait, &wait);
        current->state = TASK_INTERRUPTIBLE;
        do {
                spin_lock_irq(&rtc_lock);
                data = rtc_irq_data;
                rtc_irq_data = 0;
                spin_unlock_irq(&rtc_lock);

                if (data != 0)
                        break;

                if (file->f_flags & O_NONBLOCK) {
                        retval = -EAGAIN;
                        goto out;
                }
                if (signal_pending(current)) {
                        retval = -ERESTARTSYS;
                        goto out;
                }
                schedule();
        } while(1);
        retval = put_user(data, (unsigned long *)buf);
        if (!retval)
                retval = sizeof(unsigned long);

out:
        current->state = TASK_RUNNING;
        remove_wait_queue(&rtc_wait, &wait);
        return retval;
}
```

Lo que pasa en `rtc_read()` es esto:

1. Declaramos un elemento de la cola de espera apuntando al contexto del proceso actual.
2. Añadimos este elemento a la cola de espera `rtc_wait`
3. Marcamos el actual contexto como `TASK_INTERRUPTIBLE` lo que significa que no será replanificado después de la próxima vez que duerma.
4. Chequeamos si no hay datos disponibles; si los hay empezamos, copiamos los datos a la memoria intermedia del usuario, nos marcamos como `TASK_RUNNING`, nos quitamos de la cola de espera y regresamos.
5. Si todavía no hay datos, chequeamos cuando el usuario especificó una E/S no bloqueante, y si es así entonces fallamos con `EAGAIN` (el cual es el mismo que `EWOULDBLOCK`)
6. También chequeamos si hay alguna señal pendiente y si por lo tanto informamos a las "capas superiores" para reinicializar la llamada al sistema si es necesario. Por "si es necesario" yo entiendo los detalles de la disposición de la señal tal como están especificadas en la llamada al sistema **sigaction(2)**.
7. Entonces "salimos", esto es, nos dormimos, hasta que sea despertado por el manejador de interrupciones. Si no nos marcamos como `TASK_INTERRUPTIBLE` entonces el planificador nos podrá planificar tan pronto como los datos estean disponibles, causando así procesamiento no necesario.

Es también valioso apuntar que, usando una cola de espera, es bastante más fácil implementar la llamada al sistema **poll(2)**:

```
static unsigned int rtc_poll(struct file *file, poll_table *wait)
{
        unsigned long l;

        poll_wait(file, &rtc_wait, wait);

        spin_lock_irq(&rtc_lock);
        l = rtc_irq_data;
        spin_unlock_irq(&rtc_lock);

        if (l != 0)
                return POLLIN | POLLRDNORM;
        return 0;
}
```

Todo el trabajo es realizado por la función independiente del dispositivo `poll_wait()` la cual hace las manipulaciones necesarias en la lista de espera; todo lo que necesitamos hacer es apuntarla a la cola de espera la cual es despertada por nuestro manejador de interrupciones específico del dispositivo.

2.6 Cronómetros del núcleo

Ahora déjanos poner nuestra atención en los cronómetros del núcleo. Los cronómetros del núcleo son usados para expedir la ejecución de una función particular (llamada 'manejador de cronómetros') en un tiempo especificado en el futuro. La estructura de datos principal es `struct timer_list` declarada en `include/linux/timer.h`:

```
struct timer_list {
        struct list_head list;
        unsigned long expires;
        unsigned long data;
        void (*function)(unsigned long);
        volatile int running;
};
```

El campo `list` es para enlazar con la lista interna, protegida por el spinlock `timerlist_lock`. El campo `expires` es el valor de `jiffies` cuando el manejador `function` debería de ser invocado con `data` pasado como parámetro. El campo `running` es usado en SMP para probar si el manejador de cronómetros está actualmente funcionando en otra CPU

Las funciones `add_timer()` y `del_timer()` añaden y quitan un cronómetro determinado de la lista. Cuando un cronómetro se termina, este es borrado automáticamente. Antes de que el cronómetro sea usado, DEBE de ser inicializado por medio de la función `init_timer()`. Y entonces es añadido, los campos `function` y `expires` deben de ser establecidos.

2.7 Bottom Halves

A veces es razonable partir la cantidad de trabajo para ser realizada dentro de un manejador de interrupciones en un trabajo inmediato (ej. agradecer la interrupción, actualizar las estadísticas, etc.) y el trabajo que puede ser postpuesto para más tarde, cuando las interrupciones están habilitadas (ej, para realizar algún post-procesamiento sobre los datos, despertar a los procesos esperando por estos datos, etc).

Los bottom halves son el mecanismo más viejo para posponer la ejecución de una tarea del núcleo y están disponibles desde Linux 1.x. En Linux 2.0, un nuevo mecanismo fue añadido, llamado 'colas de tareas', las cuales serán el título de la siguiente sección.

Los bottom halves son serializados por el spinlock `global_bh_lock`, esto es, sólo puede haber un bottom half funcionando en cualquier CPU a la vez. De cualquier modo, cuando se intenta ejecutar el manejador, si no está disponible `global_bh_lock`, el bottom half es marcado (esto es planificado) para ejecución - por lo tanto el procesamiento puede continuar, en opuesto a un bucle ocupado en `global_bh_lock`.

Sólo puede haber 32 bottom halves registrados en total. Las funciones requeridas para manipular los bottom halves son las siguientes (todas exportadas a módulos);

- `void init_bh(int nr, void (*routine)(void))`: instala un manejador de bottom half apuntado por el argumento `routine` en el slot `nr`. El slot debe de estar numerado en `include/linux/interrupt.h` en la forma `XXXX_BH`, ej. `TIMER_BH` o `TQUEUE_BH`. Típicamente, una rutina de inicialización del subsistema (`init_module()` para los módulos) instala el bottom half requerido usando esta función.
- `void remove_bh(int nr)`: hace lo opuesto de `init_bh()`, esto es, desinstala el bottom half instalado en el slot `nr`. No hay chequeos de errores realizados aquí, por lo tanto, como ejemplo `remove_bh(32)` rompe el sistema. Típicamente, una rutina de limpieza del subsistema (`cleanup_module()` para los módulos) usa esta función para liberar el slot, que puede ser reusado por algún otro subsistema. (POR HACER: ¿no sería bonito tener una lista `/proc/bottom_halves` con todos los bottom halves en el sistema? Esto significa que `global_bh_lock` deberían hacer lecturas/escrituras, obviamente).
- `void mark_bh(int nr)`: marca el bottom half en el slot `nr` para ejecución. Típicamente, un manejador de interrupciones marcará este bottom half (¡de aquí el nombre!) para ejecución en un "tiempo seguro".

Los bottom halves son tasklets globalmente cerrados, por lo tanto la pregunta "¿cúando es el manejador bottom half ejecutado?" es realmente "¿cuándo son los tasklets ejecutados?". Y la respuesta es, en dos sitios: a) en cada `schedule()` y b) en cada camino de retorno de interrupciones/llamadas al sistema en `entry.S` (POR HACER: entonces, el caso `schedule()` es realmente aburrido - parece añadir todavía otra interrupción muy muy lenta, ¿por qué no desembarazarse de la etiqueta `handle_softirq` de `schedule()` en su conjunto?).

2.8 Colas de Tareas

Las colas de tareas pueden ser entendidas como una extensión dinámica de los viejos bottom halves. En realidad, en el código fuente son a veces referidas como los "nuevos" bottom halves. Más específicamente, los viejos bottom halves discutidos en la sección anterior tienen estas limitaciones:

1. Sólo hay un número fijo de ellos (32).
2. Cada bottom half sólo puede estar asociado con una función de manejador.
3. Los Bottom halves son consumidos con un spinlock mantenido, por lo tanto no pueden bloquear.

Por lo tanto, con las colas de tareas, un número arbitrario de funciones pueden ser encadenadas y procesadas una después de otra en un tiempo posterior. Uno crea una nueva cola de tareas usando la macro `DECLARE_TASK_QUEUE()` y encola la tarea en él usando la función `queue_task()`. La cola de tareas entonces puede ser procesada usando `run_task_queue()`. En vez de crear nuestra propia cola de tareas (y tener que consumirla manualmente) puedes usar una de las colas de tareas predefinidas en Linux las cuales son consumidas en puntos bien conocidos:

1. **tq_timer**: la cola de tareas de cronómetros, funciona en cada interrupción del cronómetro y cuando se libera un dispositivo tty (cerrando o liberando un dispositivo de terminal medio abierto). Desde que el manejador de cronómetro funciona en el contexto de interrupción, la tarea `tq_timer` también funciona en el contexto de interrupción y de este modo tampoco puede bloquearse.

2. **tq_scheduler**: la cola de tareas del planificador, consumida por el planificador (y también cuando se cierran dispositivos tty, como `tq_timer`). Como el planificador es ejecutado en el contexto de los procesos siendo re-planificados, las tareas `tq_scheduler` pueden hacer todo lo que quieran, esto es bloquear, usar los datos del contexto de los procesos (pero porque ellos quieren), etc .

3. **tq_immediate**: esto es realmente un bottom half `IMMEDIATE_BH`, por lo tanto los controladores pueden `queue_task(task, &tq_immediate)` y entonces `mark_bh(IMMEDIATE_BH)` ser consumido en el contexto de interrupción.

4. **tq_disk**: usado por un acceso de dispositivo de bloqueo de bajo nivel (y RAID) para empezar la actual petición. Esta cola de tareas es exportada a los módulos pero no debería de ser usada excepto para los propósitos especiales para los que fue diseñada.

A menos que un controlador use su propia cola de tareas, no necesita llamar a `run_tasks_queues()` para procesar la cola, excepto bajo ciertas circunstancias explicadas a continuación.

El motivo por el que la cola de tareas `tq_timer/tq_scheduler` no es consumida sólo en los sitios usuales sino en otras partes (cerrando un dispositivo tty, pero no el único ejemplo) se aclara si uno recuerda que el controlador puede planificar tareas en la cola, y estas tareas solo tienen sentido mientras una instancia particular del dispositivo sea todavía válida - lo cual usualmente significa hasta que la aplicación la cierre. Por lo tanto, el controlador quizás necesite llamar a `run_task_queue()` para encender las tareas que el (y alguno más) ha puesto en la cola, porque permitiéndoles funcionar en un tiempo posterior quizás no tenga sentido - esto es, las estructuras de datos relevantes quizás no hayan sido liberadas/reusadas por una instancia diferente. Este es el motivo por el que ves `run_task_queue()` en `tq_timer` y `tq_scheduler` en otros lugares más que el cronómetro de interrupciones y `schedule()` respectivamente.

2.9 Tasklets

Todavía no, estarán en una revisión futura

2.10 Softirqs

Todavía no, estarán en una revisión futura

2.11 ¿Cómo son las llamadas al sistema implementadas en la arquitectura i386?

Existen dos mecanismos bajo Linux para implementar las llamadas al sistema:

- las llamadas puerta lcall7/lcall27 ;
- interrupción software int 0x80.

Los programas nativos de Linux utilizan int 0x80 mientras que los binarios de los distintos tipos de UNIX (Solaris, UnixWare 7 etc.) usan el mecanismo lcall7. El nombre 'lcall7' es históricamente engañoso porque también cubre lcall27 (ej. Solaris/x86), pero la función manejadora es llamada lcall7_func.

Cuando el sistema arranca, la función `arch/i386/kernel/traps.c:trap_init()` es llamada, la cual inicializa el IDT, por lo tanto el vector 0x80 (del tipo 15, dpl 3) apunta a la dirección de la entrada system_call desde `arch/i386/kernel/entry.S`.

Cuando una aplicación del espacio de usuario realiza una llamada del sistema, los argumentos son pasados a través de los registros y la aplicación ejecuta la instrucción 'int 0x80'. Esto causa un reajuste en el modo núcleo y el procesador salta al punto de entrada system_call en `entry.S`. Lo que esto hace es:

1. Guarda los registros.
2. Establece %ds y %es a KERNEL_DS, por lo tanto todas las referencias de datos (y segmentos extras) son hechas en el espacio de direcciones del núcleo.
3. Si el valor de %eax es mayor que `NR_syscalls` (actualmente 256) fallará con el error `ENOSYS`.
4. Si la tarea está siendo ptraced (`tsk->ptrace & PF_TRACESYS`), realiza un procesamiento especial. Esto es para soportar programas como strace (análogo a SVR4 **truss(1)**) o depuradores.
5. Llamada `sys_call_table+4*(syscall_number desde %eax)`. Esta tabla es inicializada en el mismo archivo (`arch/i386/kernel/entry.S`) para apuntar a los manejadores individuales de las llamadas al sistema, los cuales bajo Linux son (usualmente) prefijados con `sys_`, ej. `sys_open`, `sys_exit`, etc. Estos manejadores de las llamadas al sistema de C encontrarán sus argumentos en la pila donde `SAVE_ALL` las almacenó.
6. Entra en el 'camino de retorno de la llamada al sistema'. Esta es una etiqueta separada porque es usada no sólo por int 0x80 sino también por lcall7, lcall27. Esto es, relacionado con el manejo de tasklets (incluyendo bottom halves), chequeando si un `schedule()` es necesitado (`tsk->need_resched !=0`), chequeando si hay señales pendientes y por lo tanto manejándolas.

Linux soporta hasta 6 argumentos para las llamadas al sistema. Ellas son pasadas en %ebx, %ecx, %edx, %esi, %edi (y %ebp usado temporalmente, ver `_syscall6()` en `asm-i386/unistd.h`). El número de llamadas al sistema es pasado a través de %eax.

2.12 Operaciones Atómicas

Hay dos tipos de operaciones atómicas: bitmaps (mapas de bits) y `atomic_t`. Los bitmaps son muy convenientes para mantener un concepto de unidades "asignadas" o "libres" para alguna colección grande donde cada unidad es identificada por algún número, por ejemplo tres inodos o tres bloques. Son ampliamente usados para un simple cierre, por ejemplo para suministrar acceso exclusivo para abrir un dispositivo. Un ejemplo de esto puede ser encontrado en `arch/i386/kernel/microcode.c`:

```
/*
 *  Bits en microcode_status. (31 bits de espacio para una futura expansión)
 */
#define MICROCODE_IS_OPEN       0       /* establece si el dispositivo está en uso */

static unsigned long microcode_status;
```

No hay necesidad de inicializar `microcode_status` a 0 ya que BSS es limpiado a cero explícitamente bajo Linux.

```
/*
 * Forzamos a un sólo usuario a la vez aquí con open/close.
 */
static int microcode_open(struct inode *inode, struct file *file)
{
        if (!capable(CAP_SYS_RAWIO))
                return -EPERM;

        /* uno de cada vez, por favor */
        if (test_and_set_bit(MICROCODE_IS_OPEN, &microcode_status))
                return -EBUSY;
```

```
        MOD_INC_USE_COUNT;
        return 0;
}
```

Las operaciones en los bitmaps son:

- **void set_bit(int nr, volatile void *addr)**: establece el bit `nr` en el bitmap apuntado por `addr`.
- **void clear_bit(int nr, volatile void *addr)**: limpia el bit `nr` en el bitmap apuntado por `addr`.
- **void change_bit(int nr, volatile void *addr)**: cambia el bit `nr` (si está establecido limpia, si está limpio establece) en en el bitmap apuntado por `addr`.
- **int test_and_set_bit(int nr, volatile void *addr)**: atómicamente establece el bit `nr` y devuelve el viejo valor del bit.
- **int test_and_clear_bit(int nr, volatile void *addr)**: atómicamente limpia el bit `nr` y devuelve el viejo valor del bit.
- **int test_and_change_bit(int nr, volatile void *addr)**: atómicamente cambia el bit `nr` y devuelve el viejo valor del bit.

Estas operaciones usan la macro `LOCK_PREFIX`, la cual en núcleos SMP evalúa la instrucción prefijo de cierre del bus y no hace nada en UP. Esto garantiza la atomicidad del acceso en el entorno SMP.

A veces las manipulaciones de bits no son convenientes, pero en cambio necesitamos realizar operaciones aritméticas - suma, resta, incremento decremento. Los casos típicos son cuentas de referencia (ej. para los inodos). Esta facilidad es suministrada por el tipo de datos `atomic_t` y las siguientes operaciones:

- **atomic_read(&v)**: lee el valor de la variable `atomic_t` `v`.
- **atomic_set(&v, i)**: establece el valor de la variable `atomic_t` `v` al entero `i`.
- **void atomic_add(int i, volatile atomic_t *v)**: suma un entero `i` al valor de la variable atómica apuntado por `v`.
- **void atomic_sub(int i, volatile atomic_t *v)**: resta el entero `i` del valor de la variable atómica apuntada por `v`.
- **int atomic_sub_and_test(int i, volatile atomic_t *v)**: resta el entero `i` del valor de la variable atómica apuntada por `v`; devuelve 1 si el nuevo valor es 0, devuelve 0 en otro caso.
- **void atomic_inc(volatile atomic_t *v)**: incrementa el valor en 1.
- **void atomic_dec(volatile atomic_t *v)**: decrementa el valor en 1.
- **int atomic_dec_and_test(volatile atomic_t *v)**: decrementa el valor; devuelve 1 si el nuevo valor es 0, devuelve 0 en otro caso.
- **int atomic_inc_and_test(volatile atomic_t *v)**: incrementa el valor; devuelve 1 si el nuevo valor es 0, devuelve 0 en otro caso.
- **int atomic_add_negative(int i, volatile atomic_t *v)**: suma el valor de `i` a `v` y devuelve 1 si el resultado es negativo. Devuelve 0 si el resultado es mayor o igual a 0. Esta operación es usada para implementar semáforos.

2.13 Spinlocks, Spinlocks Read-Write y Spinlocks Big-Reader

Desde los primeros días del soporte Linux (al principio de los 90, en el siglo XX), los desarrolladores se encararon con el clásico problema de acceder a datos compartidos entre los diferentes tipos de contexto (procesos de usuario vs interrupciones) y diferentes instancias del mismo contexto para múltiples cpus.

El soporte SMP fue añadido a Linux 1.3.42 el 15 de Noviembre de 1995 (el parche original fue hecho para el 1.3.37 en Octubre del mismo año).

Si la región crítica del código puede ser ejecutada por el contexto de los procesos y el contexto de las interrupciones, entonces la forma de protegerlo usando las instrucciones `cli/sti` en UP es:

```
unsigned long flags;

save_flags(flags);
cli();
/* código crítico */
restore_flags(flags);
```

Mientras que esto está bien en UP, obviamente no lo está usándolo en SMP porque la misma secuencia de código quizás sea ejecutada simultáneamente en otra cpu, y mientras `cli()` suministra protección contra las carreras con el contexto de interrupciones en cada CPU individualmente, no suministra protección contra todas las carreras entre los contextos funcionando en diferentes CPUs. Es aquí donde los spinlocks son útiles.

Hay tres tipos de spinlocks: vanilla (básico), read-write y spinlocks big-reader. Los spinlocks read-write deberían de ser usados cuando existe una tendencia natural de 'muchos lectores y pocos escritores'. Un ejemplo de esto es el acceso a las lista de sistemas de archivos registrados (ver `fs/super.c`). La lista es guardada por el spinlock read-write `file_systems_lock` porque uno necesita acceso exclusivo sólo cuando se está registrando/desregistrando un sistema de archivos, pero cualquier proceso puede leer el archivo `/proc/filesystems` o usar la llamada al sistema **sysfs(2)** para forzar un escaneo de sólo lectura de la lista file_systems. Esto lo hace sensible a usar spinlocks read-write. Con los spinlocks read-write, uno puede tener múltiples lectores a la vez pero sólo un escritor y no puede haber lectores mientras hay un escritor. Por el camino, sería bonito si nuevos lectores no isaran un cierre mientras hay un escritor intentando usar un cierre, esto es, si Linux pudiera distribuir correctamente la solución del hambre potencial del escritor por los múltiples lectores. Esto quiere significar que los lectores deben de ser bloqueados mientras exista un escritor intentando usar el cierre. Este no es actualmente el caso y no es obvio cuando debería de ser arreglado - el argumento para lo contrario es - los lectores usualmente ocupan el cierre por un instante de tiempo muy pequeño, por lo tanto, ¿ellos realmente deberían de tener hambre mientras el escritor usa el cierre para periodos potencialmente más largos?

Los spinlocks Big-reader son una forma de spinlocks read-write altamente optimizados para accesos de lectura muy ligeros, con una penalización para los escritores. Hay un número limitado de spinlocks big-reader - actualmente sólo existen dos, de los cuales uno es usado sólo en sparc64 (irq global) y el otro es usado para redes. En todos los demás casos donde el patrón de acceso no concuerda con ninguno de estos dos escenarios, se debería utilizar los spinlocks básicos. No puedes bloquear mientras mantienes algún tipo de spinlock.

Los Spinlocks vienen en tres tipos: plano, `_irq()` y `_bh()`.

1. `spin_lock()/spin_unlock()` plano: si conoces que las interrupciones están siempre deshabilitadas o si no compites con el contexto de interrupciones (ej. desde un manejador de interrupciones), entonces puedes utilizar este. No toca el estado de interrupción en la actual CPU.

2. `spin_lock_irq()/spin_unlock_irq()`: si sabes que las interrupciones están siempre habilitadas entonces puedes usar esta versión, la cual simplemente deshabilita (en el cierre) y re-habilita (en el desbloqueo) las interrupciones en la actual CPU. Por ejemplo, `rtc_read()` usa `spin_lock_irq(&rtc_lock)`

(las interrupciones están siempre habilitadas dentro de `read()`) mientras que `rtc_interrupt()` usa `spin_lock(&rtc_lock)` (las interrupciones están siempre deshabilitadas dentro del manejador de interrupciones). Nótese que `rtc_read()` usa `spin_lock_irq()` y no el más genérico `spin_lock_irqsave()` porque en la entrada a cualquier llamada al sistema las interrupciones están siempre habilitadas.

3. `spin_lock_irqsave()/spin_unlock_irqrestore()`: la forma más fuerte, es usada cuando el estado de las interrupciones no es conocido, pero sólo si las interrupciones no importan nada, esto es, no hay punteros usándolo si nuestro manejador de interrupciones no ejecuta ningún código crítico.

El motivo por el que no puedes usar el `spin_lock()` plano si compites contra el manejador de interrupciones es porque si lo usas y después un interrupción viene en la misma CPU, el esperará ocupado por el bloqueo para siempre: el que tenga el bloqueo, habiendo sido interrumpido, no continuará hasta el que manejador de interrupciones vuelva.

El uso más común de un spinlock es para acceder a estructuras de datos compartidas entre el contexto de proceso de usuario y el manejador de interrupciones:

```
spinlock_t my_lock = SPIN_LOCK_UNLOCKED;

my_ioctl()
{
        spin_lock_irq(&my_lock);
        /* sección crítica */
        spin_unlock_irq(&my_lock);
}

my_irq_handler()
{
        spin_lock(&lock);
        /* sección crítica */
        spin_unlock(&lock);
}
```

Hay un par de cosas que destacar sobre este ejemplo:

1. El contexto del proceso, representado aquí como un método típico de un controlador - `ioctl()` (argumentos y valores de retorno omitidos para una mayor claridad), deben de usar `spin_lock_irq()` porque conocen que las interrupciones están siempre habilitadas mientras se ejecuta un método de dispositivo `ioctl()`.

2. El contexto de interrupciones, representado aquí por `my_irq_handler()` (otra vez los argumentos son omitidos para una mayor claridad) pueden usar la forma `spin_lock()` plana porque las interrupciones están deshabilitadas dentro del manejador de interrupciones.

2.14 Semáforos y semáforos read/write

A veces, mientras se está accediendo a estructuras de datos compartidas, uno debe realizar operaciones que puedan bloquear, por ejemplo copiar datos al espacio de usuario. La directiva del cierre disponible para tales escenarios bajo Linux es llamada semáforo. Hay dos tipos de semáforos: básicos y semáforos read/write. Dependiendo del valor inicial del semáforo, pueden ser usados para exclusión mutua (valor inicial a 1) o para suministrar un tipo más sofisticado de acceso.

Los semáforos read-write difieren de los semáforos básicos de la misma forma que los spinlocks read-write difieren de los spinlocks básicos: uno puede tener múltiples lectores a la vez pero sólo un escritor y no puede haber lectores mientras hay escritores - esto es, el escritor bloquea a todos los lectores, y los nuevos lectores se bloquean mientras un escritor está esperando.

También, los semáforos básicos pueden ser interrumpidos - justamente usan las operaciones `down/up_interruptible()` en vez del `down()/up()` plano y chequean el valor devuelto desde `down_interruptible()`: no será cero si la operación fue interrumpida.

El uso de semáforos para exclusión mutua es ideal para situaciones donde una sección crítica de código quizás sea llamada por funciones de referencia desconocidas registradas por otros subsistemas/módulos, esto es, el llamante no conoce a priori cuando la función bloquea o no.

Un ejemplo simple del uso de semáforos está en la implementación de las llamadas al sistema **gethostname(2)**/**sethostname(2)** en `kernel/sys.c`.

```
asmlinkage long sys_sethostname(char *name, int len)
{
        int errno;

        if (!capable(CAP_SYS_ADMIN))
                return -EPERM;
        if (len < 0 || len > __NEW_UTS_LEN)
                return -EINVAL;
        down_write(&uts_sem);
        errno = -EFAULT;
        if (!copy_from_user(system_utsname.nodename, name, len)) {
                system_utsname.nodename[len] = 0;
                errno = 0;
        }
        up_write(&uts_sem);
        return errno;
}

asmlinkage long sys_gethostname(char *name, int len)
{
        int i, errno;

        if (len < 0)
                return -EINVAL;
        down_read(&uts_sem);
        i = 1 + strlen(system_utsname.nodename);
        if (i > len)
                i = len;
        errno = 0;
        if (copy_to_user(name, system_utsname.nodename, i))
                errno = -EFAULT;
        up_read(&uts_sem);
        return errno;
}
```

Los puntos a destacar en este ejemplo son:

1. Las funciones quizás bloqueen mientras están copiando datos desde/al espacio de usuario en `copy_from_user()/copy_to_user()`. Entonces no pueden usar ninguna forma de spinlock aquí.

2. El tipo de semáforo escogido es read-write en oposición al básico porque quizás existan un montón de peticiones concurrentes **gethostname(2)** las cuales no tienen que ser mutuamente exclusivas.

Aunque la implementación de Linux de los semáforos y los semáforos de read-write es muy sofisticada, existen posibles escenarios que uno puede pensar en los cuales no están todavía implementados, por ejemplo, no existe el concepto de semáforos read-write interrumpible. Eso es obvio porque no hay situaciones en el mundo real que requieran estos tipos exóticos de directivas.

2.15 Soporte del Núcleo para la Carga de Módulos

Linux es un sistema operativo monolítico y olvídate de todos los dichos modernos sobre algunas "ventajas" ofrecidas por los sistemas operativos basados en el diseño micro-núcleo, la verdad permanece (cita de Linus Torvalds):

> `... el paso de mensajes como la operación fundamental del SO es sólo un ejercicio de masturbación de la ciencia de la computación. Quizás suene bien, pero actualmente no tienes nada HECHO.`

Entonces, Linux esta y siempre estará basado en un diseño monolítico, lo cual significa que todos los subsistemas funcionan en el mismo modo privilegiado y comparten el mismo espacio de direcciones, la comunicación entre ellos es realizada por medio de las llamadas usuales de funciones de C.

De cualquier modo, aunque la separación de la funcionalidad del núcleo en "procesos" separados realizada en los micro-núcleos es definitivamente una mala idea, separándolo en módulos del núcleo dinámicamente cargados bajo demanda es deseable en algunas circunstancias (ej, en máquinas con poca memoria o para núcleos de instalación, los cuales de otra forma pueden contener controladores de dispositivos ISA auto-probables que son mutuamente exclusivos). La decisión de cuando incluir soporte para la carga de módulos es hecha en tiempo de compilación y es determinada por la opción `CONFIG_MODULES`. El soporte para la auto-carga de módulos a través del mecanismo `request_module()` es una opción separada de compilación (`CONFIG_KMOD`).

Las siguientes funcionalidades pueden ser implementadas como módulos cargables bajo Linux:

1. Controladores de dispositivos de bloque y de carácter, incluyendo los controladores de dispositivos misc.
2. Disciplinas de linea de Terminal.
3. Ficheros virtuales (regulares) en `/proc` y en devfs (ej. `/dev/cpu/microcode` vs `/dev/misc/microcode`).
4. Formatos de Ficheros Binarios (ej. ELF, aout, etc).
5. Dominios de Ejecución (ej. Linux, UnixWare7, Solaris, etc).
6. Sistemas de ficheros.
7. System V IPC.

Hay unas pocas cosas que no pueden ser implementadas como módulos bajo Linux (probablemente porque no tienen sentido el ser modularizadas):

1. Algoritmos de planificación.
2. Políticas de VM (Memoria Virtual).
3. Antememoria intermedia, antememoria de páginas y otras antememoria.

Linux suministra varias llamadas al sistema para asistir en la carga de módulos:

1. `caddr_t create_module(const char *name, size_t size)`: asigna `size` bytes usando `vmalloc()` y mapea una estructura de un módulo al principio de este. Este nuevo módulo es enlazado en la cabecera de la lista por module_list. Sólo un proceso con `CAP_SYS_MODULE` puede llamar a esta llamada al sistema, otros verán como se les retorna `EPERM`.
2. `long init_module(const char *name, struct module *image)`: carga la imagen del módulo reasignado y motiva que la rutina de inicialización del módulo sea invocada. Sólo un proceso con `CAP_SYS_MODULE` puede llamar a esta llamada al sistema, otros verán como se les retorna `EPERM`.
3. `long delete_module(const char *name)`: intenta descargar el módulo. Si `name == NULL`, el intento es hecho para descargar todos los módulos no utilizados.
4. `long query_module(const char *name, int which, void *buf, size_t bufsize, size_t *ret)`: devuelve información sobre un módulo (o sobre todos los módulos).

La interfaz de comandos disponible a los usuarios consiste en:

- **insmod**: inserta un módulo simple.
- **modprobe**: inserta un módulo incluyendo todos los otros módulos de los cuales dependa.
- **rmmod**: quita un módulo.
- **modinfo**: imprime alguna información sobre un módulo, ej. autor, descripción, parámetros que acepta el módulo, etc.

Aparte de ser capaz de cargar un módulo manualmente usando **insmod** o **modprobe**, también es posible tener el módulo insertado automáticamente por el núcleo cuando una funcionalidad particular es requerida. La interface del núcleo para esto es la función llamada `request_module(name)` la cual es exportada a los módulos, por lo tanto los módulos también pueden cargar otros módulos. La `request_module(name)` internamente crea un hilo del núcleo el cual ejecuta el comando del espacio de usuario **modprobe -s -k module_name**, usando la interfaz estándar del núcleo `exec_usermodehelper()` (que es también exportado a los módulos). La función devuelve 0 si es exitosa, de cualquier forma no es usualmente válido chequear el código de retorno desde `request_module()`. En vez de esto, el idioma de programación es:

```
if (check_some_feature() == NULL)
    request_module(module);
if (check_some_feature() == NULL)
    return -ENODEV;
```

Por ejemplo, esto es realizado por `fs/block_dev.c:get_blkfops()` para cargar un módulo `block-major-N` cuando el intento es hecho para abrir un dispositivo de bloque con número mayor `N`. Obviamente, no existe tal módulo llamado `block-major-N` (los desarrolladores Linux solo escogen nombres sensibles para sus módulos) pero es mapeado al propio nombre del módulo usando el archivo `/etc/modules.conf`. De cualquier forma, para la mayoría de los números mayores bien conocidos (y otros tipos de módulos) los comandos **modprobe**/**insmod** conocen qué módulo real cargar sin necesitar una declaración explícita de un alias en `/etc/modules.conf`.

Un buen ejemplo de la carga de un módulo está dentro de la llamada del sistema **mount(2)**. La llamada al sistema **mount(2)** acepta el tipo de sistema de archivos como una cadena `fs/super.c:do_mount()` la cual entonces pasa a `fs/super.c:get_fs_type()`:

```
static struct file_system_type *get_fs_type(const char *name)
{
        struct file_system_type *fs;

        read_lock(&file_systems_lock);
        fs = *(find_filesystem(name));
        if (fs && !try_inc_mod_count(fs->owner))
                fs = NULL;
        read_unlock(&file_systems_lock);
        if (!fs && (request_module(name) == 0)) {
                read_lock(&file_systems_lock);
                fs = *(find_filesystem(name));
                if (fs && !try_inc_mod_count(fs->owner))
                        fs = NULL;
                read_unlock(&file_systems_lock);
        }
        return fs;
}
```

Hay que destacar unas pocas cosas en esta función:

1. Primero intentamos encontrar el sistema de archivos con el nombre dado entre aquellos ya registrados. Esto es hecho bajo la protección de `file_systems_lock` tomado para lectura (ya que no estamos modificando la lista registrada de sistemas de archivos).

2. Si tal sistema de archivos es encontrado intentamos coger una nueva referencia a él intentando incrementar la cuenta mantenida del módulo. Esto siempre devuelve 1 para sistemas de archivos enlazados dinámicamente o para módulos que actualmente no se han borrados. Si `try_inc_mod_count()` devuelve 0 entonces lo consideraremos un fallo - esto es, si el módulo está allí pero está siendo borrado, es tan bueno como si no estuviera allí en absoluto.

3. Tiramos el `file_systems_lock` porque lo siguiente que vamos a hacer (`request_module()`) es una operación bloqueante, y entonces no podemos mantener un spinlock sobre el. Actualmente, en este caso específico, podríamos tirar `file_systems_lock` de cualquier forma, incluso si `request_module()` fuera garantizada para ser no bloqueante y la carga de módulos fuera ejecutada en el mismo contexto atómicamente. El motivo para esto es que la función de inicialización de módulos intentará llamar a `register_filesystem()`, la cual tomará el mismo spinlock read-write `file_systems_lock` para escritura.

4. Si el intento de carga tiene éxito, entonces cogemos el spinlock `file_systems_lock` e intentamos situar el nuevamente registrado sistema de archivos en la lista. Nótese que esto es ligeramente erróneo porque es posible en un principio que un fallo en el comando modprobe pueda causar un volcado del núcleo después de cargar con éxito el módulo pedido, en tal caso `request_module()` fallará incluso aunque el nuevo sistema de archivos halla sido registrado, y todavía no lo encontrará `get_fs_type()`.

5. Si el sistema de archivos es encontrado y es capaz de obtener una referencia a el, la devolvemos. En otro caso devolvemos NULL.

Cuando un módulo es cargado en el núcleo, puede ser referido por cualquier símbolo que sea exportado como público por el núcleo usando la macro `EXPORT_SYMBOL()` o por otros módulos actualmente cargados. Si el módulo usa símbolos de otro módulo, es marcado como dependiente de ese módulo durante el recálculo de dependencias, realizado funcionando el comando **depmod -a** en el arranque (ej. después de instalar un nuevo núcleo).

Usualmente, uno debe comprobar el conjunto de los módulos con la versión de las interfaces del núcleo que usan, lo cual bajo Linux simplemente significa la "versión del núcleo" ya que no hay versionados especiales del mecanismo de interfaces del núcleo en general. De cualquier forma, hay una funcionalidad limitada llamada "versionamiento de módulos" o `CONFIG_MODVERSIONS` la cual nos permite eliminar el recompilamiento de módulos cuando cambiamos a un nuevo núcleo. Lo que pasa aquí es que la tabla de símbolos de núcleo es tratada de forma diferente para el acceso interno y para el acceso de los módulos. Los elementos de la parte pública (exportada) de la tabla de símbolos son construidos en la declaración de C de suma de control 32bit. Por lo tanto, en orden de resolver un símbolo usado por un módulo durante la carga, el cargador debe comprobar la representación total del símbolo que incluye la suma de control; será rechazada para cargar el módulo si estos símbolos difieren. Esto sólo pasa cuando el núcleo y el módulo son compilados con el versionamiento de módulos habilitado. Si ninguno de los dos usa los nombres originales de los símbolos el cargador simplemente intenta comprobar la versión del núcleo declarada por el módulo y el exportado por el núcleo y rechaza cargarlo si difieren.

3 Sistemas de Archivos Virtuales (VFS)

3.1 Caché de Inodos e Interacción con Dcache

En orden para soportar múltiples sistemas de archivos, Linux contiene un nivel especial de interfaces del núcleo llamado VFS (Interruptor de Sistemas de Ficheros Virtuales). Esto es muy similar a la interfaz vnode/vfs encontrada en los derivados de SVR4 (originalmente venían de BSD y de las implementaciones originales de Sun).

La antememoria de inodos de Linux es implementada en un simple fichero, `fs/inode.c`, el cual consiste de 977 lineas de código. Es interesante notar que no se han realizado muchos cambios en él durante los últimos 5-7 años: uno todavía puede reconocer algún código comparando la última version con, digamos, 1.3.42.

La estructura de la antememoria de inodos Linux es como sigue:

1. Una tabla global hash, `inode_hashtable`, donde cada inodo es ordenado por el valor del puntero del superbloque y el número de inodo de 32bit. Los inodos sin un superbloque (`inode->i_sb == NULL`) son añadidos a la lista doblemente enlazada encabezada por `anon_hash_chain` en su lugar. Ejemplos de inodos anónimos son los conectores creados por `net/socket.c:sock_alloc()`, llamado por `fs/inode.c:get_empty_inode()`.
2. Una lista global del tipo "en_uso" (`inode_in_use`), la cual contiene los inodos válidos con `i_count>0` y `i_nlink>0`. Los inodos nuevamente asignados por `get_empty_inode()` y `get_new_inode()` son añadidos a la lista `inode_in_use`.
3. Una lista global del tipo "sin_usar" (`inode_unused`), la cual contiene los inodos válidos con `i_count = 0`.
4. Una lista por cada superbloque del tipo "sucia" (`sb->s_dirty`) que contiene los inodos válidos con `i_count>0`, `i_nlink>0` y `i_state & I_DIRTY`. Cuando el inodo es marcado como sucio, es añadido a la lista `sb->s_dirty` si el está también ordenado. Manteniendo una lista sucia por superbloque de inodos nos permite rápidamente sincronizar los inodos.
5. Una antememoria propia de inodos - una antememoria SLAB llamada `inode_cachep`. Tal como los objetos inodos son asignados como libres, ellos son tomados y devueltos a esta antememoria SLAB.

Los tipos de listas son sujetadas desde `inode->i_list`, la tabla hash desde `inode->i_hash`. Cada inodo puede estar en una tabla hash y en uno, y en sólo uno, tipo de lista (en_uso, sin_usar o sucia).

Todas estas listas están protegidas por un spinlock simple: `inode_lock`.

El subsistema de caché de inodos es inicializado cuando la función `inode_init()` es llamada desde `init/main.c:start_kernel()`. La función es marcada como `__init`, lo que significa que el código será lanzado posteriormente. Se le pasa un argumento simple - el número de páginas físicas en el sistema. Esto es por lo que la antememoria de inodos puede configurarse ella misma dependiendo de cuanta memoria está disponible, esto es, crea una tabla hash más grande si hay suficiente memoria.

Las únicas estadísticas de información sobre la antememoria de inodos es el número de inodos sin usar, almacenados en `inodes_stat.nr_unused` y accesibles por los programas de usuario a través de los archivos `/proc/sys/fs/inode-nr` y `/proc/sys/fs/inode-state`.

Podemos examinar una de las listas desde **gdb** en un núcleo en funcionamiento de esta forma:

```
(gdb) printf "%d\n", (unsigned long)(&((struct inode *)0)->i_list)
8
(gdb) p inode_unused
$34 = 0xdfa992a8
(gdb) p (struct list_head)inode_unused
$35 = {next = 0xdfa992a8, prev = 0xdfcdd5a8}
(gdb) p ((struct list_head)inode_unused).prev
$36 = (struct list_head *) 0xdfcdd5a8
(gdb) p (((struct list_head)inode_unused).prev)->prev
$37 = (struct list_head *) 0xdfb5a2e8
(gdb) set $i = (struct inode *)0xdfb5a2e0
(gdb) p $i->i_ino
$38 = 0x3bec7
(gdb) p $i->i_count
$39 = {counter = 0x0}
```

Destacar que restamos 8 de la dirección 0xdfb5a2e8 para obtener la dirección de `struct inode` (0xdfb5a2e0) de acuerdo a la definición de la macro `list_entry()` de `include/linux/list.h`.

Para entender cómo trabaja la antememoria de inodos, déjanos seguir un tiempo de vida de un inodo de un fichero regular en el sistema de ficheros ext2, el cómo es abierto y cómo es cerrado:

```
fd = open("file", O_RDONLY);
close(fd);
```

La llamada al sistema **open(2)** es implementada en la función `fs/open.c:sys_open` y el trabajo real es realizado por la función `fs/open.c:filp_open()`, la cual está dividida en dos partes:

1. `open_namei()`: rellena la estructura nameidata conteniendo las estructuras dentry y vfsmount.
2. `dentry_open()`: dado dentry y vfsmount, esta función asigna una nueva `struct file` y las enlaza a todas ellas; también llama al método específico del sistema de ficheros `f_op->open()` el cual fue inicializado en `inode->i_fop` cuando el inodo fue leído en `open_namei()` (el cual suministra el inodo a través de `dentry->d_inode`).

La función `open_namei()` interactúa con la antememoria dentry a través de `path_walk()`, el cual en el regreso llama a `real_lookup()`, el cual llama al método específico del sistema de ficheros `inode_operations->lookup()`. La misión de este método es encontrar la entrada en el directorio padre con el nombre correcto y entonces hace `iget(sb, ino)` para coger el correspondiente inodo - el cual nos trae la antememoria de inodos. Cuando el inodo es leido, el dentry es instanciado por medio de `d_add(dentry, inode)`. Mientras estamos en él, nótese que en los sistemas de ficheros del estilo UNIX que tienen el concepto de número de inodos en disco, el trabajo del método lookup es mapear su bit menos significativo al actual formato de la CPU, ej. si el número de inodos en la entrada del directorio sin formato (específico del sistema de ficheros) está en el formato de 32 bits little-endian uno haría:

```
unsigned long ino = le32_to_cpu(de->inode);
inode = iget(sb, ino);
d_add(dentry, inode);
```

Por lo tanto, cuando abrimos un fichero nosotros llamamos a `iget(sb, ino)` el cual es realmente `iget4(sb, ino, NULL, NULL)`, el cual hace:

1. Intenta encontrar un inodo con el superbloque emparejado y el número de inodo en la tabla hash bajo la protección de `inode_lock`. Si el inodo es encontrado, su cuenta de referencia (`i_count`) es incrementada; si era 0 anteriormente al incremento y el inodo no estaba sucio, es quitado de cualquier tipo de lista (`inode->i_list`) en la que esté (tiene que estar en la lista `inode_unused`, por supuesto) e insertado en la lista del tipo `inode_in_use`; finalmente `inodes_stat.nr_unused` es decrementado.
2. Si el inodo está actualmente bloqueado, esperaremos hasta que se desbloquee, por lo tanto está garantizado que `iget4()` devolverá un inodo desbloqueado.
3. Si el inodo no fue encontrado en la tabla hash entonces es la primera vez que se pide este inodo, por lo tanto llamamos a `get_new_inode()`, pasándole el puntero al sitio de la tabla hash donde debería de ser insertado.
4. `get_new_inode()` asigna un nuevo inodo desde la antememoria SLAB `inode_cachep`, pero esta operación puede bloquear (asignación `GFP_KERNEL`), por lo tanto el spinlock que guarda la tabla hash tiene que ser quitado. Desde que hemos quitado el spinlock, entonces debemos de volver a buscar el inodo en la tabla; si esta vez es encontrado, se devuelve (después de incrementar la referencia por `__iget`) el que se encontró en la tabla hash y se destruye el nuevamente asignado. Si aún no se ha encontrado en la tabla hash, entonces el nuevo inodo que tenemos acaba de ser asignado y es el que va a ser usado; entonces es inicializado a los valores requeridos y el método específico del sistema de ficheros `sb->s_op->read_inode()` es llamado para propagar el resto del inodo. Esto nos proporciona desde la antememoria de inodos la vuelta al código del sistema de archivos - recuerda que venimos de la antememoria de inodos cuando el método específico del sistema de ficheros `lookup()` llama a `iget()`. Mientras el método `s_op->read_inode()` está leyendo el inodo del disco, el inodo está bloqueado (`i_state = I_LOCK`); él es desbloqueado después de que el método `read_inode()` regrese y todos los que están esperando por el hayan sido despertados.

Ahora, veamos que pasa cuando cerramos este descriptor de ficheros. La llamada al sistema **close(2)** está implementada en la función `fs/open.c:sys_close()`, la cual llama a `do_close(fd, 1)` el cual rompe (reemplaza con NULL) el descriptor del descriptor de ficheros de la tabla del proceso y llama a la función `filp_close()`, la cual realiza la mayor parte del trabajo. La parte interesante sucede en `fput()`, la cual chequea si era la última referencia al fichero, y si es así llama a `fs/file_table.c:_fput()` la cual llama a `__fput()` en la cual es donde sucede la interacción con dcache (y entonces con la memoria intermedia de inodos - ¡recuerda que dcache es la memoria intermedia de inodos Maestra!). El `fs/dcache.c:dput()` hace `dentry_iput()` la cual nos brinda la vuelta a la memoria intermedia de inodos a través de `iput(inode)`, por lo tanto déjanos entender `fs/inode.c:iput(inode)`:

1. Si el parámetro pasado a nosotros es NULL, no hacemos nada y regresamos.
2. Si hay un método específico del sistema de archivos `sb->s_op->put_inode()`, es llamada inmediatamente sin mantener ningún spinlock (por lo tanto puede bloquear).

3. El spinlock `inode_lock` es tomado y `i_count` es decrementado. Si NO era la última referencia a este inodo entonces simplemente chequeamos si hay muchas referencias a el y entonces `i_count` puede urdir sobre los 32 bits asignados a el si por lo tanto podemos imprimir un mensaje de peligro y regresar. Nótese que llamamos a `printk()` mientras mantenemos el spinlock `inode_lock` - esto está bien porque `printk()` nunca bloquea, entonces puede ser llamado absolutamente en cualquier contexto (¡incluso desde el manejador de interrupciones!).
4. Si era la última referencia activa entonces algún trabajo necesita ser realizado.

EL trabajo realizado por `iput()` en la última referencia del inodo es bastante complejo, por lo tanto lo separaremos en una lista de si misma:

1. Si `i_nlink == 0` (ej. el fichero fué desenlazado mientras lo manteníamos abierto) entonces el inodo es quitado de la tabla hash y de su lista de tipos; si hay alguna página de datos mantenida en la antememoria de páginas para este inodo, son borradas por medio de `truncate_all_inode_pages(&inode->i_data)`. Entonces el método específico del sistema de archivos `s_op->delete_inode()` es llamado, el cual típicamente borra la copia en disco del inodo. Si no hay un método `s_op->delete_inode()` registrado por el sistema de ficheros (ej. ramfs) entonces llamamos a `clear_inode(inode)`, el cual llama `s_op->clear_inode()` si está registrado y si un inodo corresponde a un dispositivo de bloques, esta cuenta de referencia del dispositivo es borrada por `bdput(inode->i_bdev)`.
2. Si `i_nlink != 0` entonces chequeamos si hay otros inodos en el mismo cubo hash y si no hay ninguno, entonces si el inodo no está sucio lo borramos desde su tipo de lista y lo añadimos a la lista `inode_unused` incrementando `inodes_stat.nr_unused`. Si hay inodos en el mismo cubo hash entonces los borramos de la lista de tipo y lo añadimos a la lista `inode_unused`. Si no había ningún inodo (NetApp .snapshot) entonces lo borramos de la lista de tipos y lo limpiamos/destruimos completamente.

3.2 Registro/Desregistro de sistemas de Ficheros

El núcleo Linux suministra un mecanismo para los nuevos sistemas de ficheros para ser escritos con el mínimo esfuerzo. Los motivos históricos para esto son:

1. En el mundo donde la gente aún usa sistemas operativos no Linux para proteger sus inversiones en el software legado, Linux tiene que suministrar interoperabilidad para soportar una gran multitud de sistemas de ficheros diferentes - la mayoría no merecen existir pero sólo por compatibilidad con los existentes sistemas operativos no Linux.
2. La interfaz para los escritores de sistemas de ficheros tiene que ser muy simple para que la gente pueda intentar hacer ingeniería inversa con los sistemas de ficheros existentes para escribir versiones de sólo lectura de ellos. Entonces el VFS de Linux hace muy fácil implementar sistemas de ficheros de sólo lectura: el 95% del trabajo está por finalizar añadiéndole un soporte total para escritura. Como un ejemplo concreto leí sistemas de ficheros BFS para Linux en modo sólo lectura en unas 10 horas, pero llevó varias semanas completarlo para tener un soporte total de escritura (e incluso hoy algunos puristas dicen que no está completo porque no tiene soporte de compactación).
3. La interfaz VFS es exportada, y entonces todos los sistemas de ficheros Linux pueden ser implementados como módulos.

Déjanos considerar los pasos requeridos para implementar un sistema de ficheros bajo Linux. El código para implementar un sistema de ficheros puede ser un módulo dinámicamente cargado o estár estáticamente enlazado en el núcleo, el camino es realizado por Linux trasparentemente. Todo lo que se necesita es rellenar una estructura `struct file_system_type` y registrarla con el VFS usando la función `register_filesystem()` como en el siguiente ejemplo de `fs/bfs/inode.c`:

```
#include <linux/module.h>
#include <linux/init.h>

static struct super_block *bfs_read_super(struct super_block *, void *, int);

static DECLARE_FSTYPE_DEV(bfs_fs_type, "bfs", bfs_read_super);

static int __init init_bfs_fs(void)
{
        return register_filesystem(&bfs_fs_type);
}

static void __exit exit_bfs_fs(void)
{
        unregister_filesystem(&bfs_fs_type);
}

module_init(init_bfs_fs)
module_exit(exit_bfs_fs)
```

Las macros `module_init()/module_exit()` aseguran que, cuando BFS es compilado como un módulo, las funciones `init_bfs_fs()` y `exit_bfs_fs()` se convierten en `init_module()` y `cleanup_module()` respectivamente; si BFS está estáticamente enlazado en el núcleo el código `exit_bfs_fs()` lo hace innecesario.

La `struct file_system_type` es declarada en `include/linux/fs.h`:

```
struct file_system_type {
        const char *name;
        int fs_flags;
        struct super_block *(*read_super) (struct super_block *, void *, int);
        struct module *owner;
        struct vfsmount *kern_mnt; /* For kernel mount, if it's FS_SINGLE fs */
        struct file_system_type * next;
};
```

Los campos anteriores son explicados de esta forma:

- **name**: nombre humano leíble, aparece en el fichero `/proc/filesystems` y es usado como clave para encontrar un sistema de ficheros por su nombre; este mismo nombre es usado por el tipo de sistema de ficheros en **mount(2)**, y debería de ser único; (obviamente) sólo puede haber un sistema de ficheros con un nombre dado. Para los módulos, los nombres de los punteros al espacio de direcciones del módulo no son copiados: esto significa que **cat /proc/filesystems** puede fallar si el módulo fue descargado pero el sistema de ficheros aún está registrado.
- **fs_flags**: una o mas (ORed) de las banderas: `FS_REQUIRES_DEV` para sistemas de ficheros que sólo pueden ser montados como dispositivos de bloque, `FS_SINGLE` para sistemas de ficheros que pueden tener sólo un superbloque, `FS_NOMOUNT` para los sistemas de ficheros que no pueden ser montados desde el espacio de usuario por medio de la llamada al sistema **mount(2)**: ellos pueden de todas formas ser montados internamente usando la interfaz `kern_mount()`, ej, pipefs.
- **read_super**: un puntero a la función que lee el superbloque durante la operación de montaje. Esta función es requerida; si no es suministrada, la operación de montaje (desde el espacio de usuario o

desde el núcleo) fallará siempre excepto en el caso `FS_SINGLE` donde fallará en `get_sb_single()`, intentando desreferenciar un puntero a NULL en `fs_type->kern_mnt->mnt_sb` con (`fs_type->kern_mnt = NULL`).

- **owner**: puntero al módulo que implementa este sistema de ficheros. Si el sistema de ficheros está enlazado estáticamente en el núcleo entonces esto es NULL. No necesitas establecer esto manualmente puesto que la macro `THIS_MODULE` lo hace automáticamente.
- **kern_mnt**: sólo para sistemas de ficheros `FS_SINGLE`. Esto es establecido por `kern_mount()` (POR HACER: `kern_mount()` debería de rechazar montar sistemas de ficheros si `FS_SINGLE` no está establecido).
- **next**: enlaza a la cabecera de la lista simplemente enlazada `file_systems` (ver `fs/super.c`). La lista está protegida por el spinlock read-write `file_systems_lock` y las funciones `register/unregister_filesystem()` modificada por el enlace y desenlace de la entrada de la lista.

El trabajo de la función `read_super()` es la de rellenar los campos del superbloque, asignando el inodo raiz e inicializando cualquier información privada del sistema de ficheros asociadas por esta instancia montada del sistema de ficheros. Por lo tanto, tipicamente el `read_super()` hará:

1. Lee el superbloque desde el dispositivo especificado a través del argumento `sb->s_dev`, usando la función de la antememoria intermedia `bread()`. Si se anticipa a leer unos pocos más bloques de metadatos inmediatamente subsecuentes, entonces tiene sentido usar `breada()` para planificar el leer bloque extra de forma asíncrona.
2. Verifica que el superbloque contiene el número mágico válido y todo "parece" correcto.
3. Inicializa `sb->s_op` para apuntar a la estructura `struct super_block_operations`. Esta estructura contiene las funciones específicas del sistema de ficheros implementando las operaciones como "leer inodo", "borrar inodo", etc.
4. Asigna el inodo y dentry raiz usando `d_alloc_root()`.
5. Si el sistema de ficheros no está montado como sólo lectura entonces establece `sb->s_dirt` a 1 y marca la antememoria conteniendo el superbloque como sucio (POR HACER: ¿porqué hacemos esto? Yo lo hice en BFS porque MINIX lo hizo ...)

3.3 Administración de Descriptores de Ficheros

Bajo Linux hay varios niveles de rodeos entre el descriptor de ficheros del usuario y la estructura de inodos del núcleo. Cuando un proceso realiza la llamada al sistema **open(2)**, el núcleo devuelve un entero pequeño no negativo el cual puede ser usado para operaciones de E/S subsecuentes en este fichero. Cada estructura de fichero apunta a dentry a través de `file->f_dentry`. Y cada dentry apunta a un inodo a través de `dentry->d_inode`.

Cada tarea contiene un campo `tsk->files` el cual es un puntero a `struct files_struct` definida en `include/linux/sched.h`:

```
/*
 * Abre la estructura tabla del fichero
 */
struct files_struct {
        atomic_t count;
        rwlock_t file_lock;
```

```
        int max_fds;
        int max_fdset;
        int next_fd;
        struct file ** fd;      /* actualmente una matriz de descriptores de ficheros */
        fd_set *close_on_exec;
        fd_set *open_fds;
        fd_set close_on_exec_init;
        fd_set open_fds_init;
        struct file * fd_array[NR_OPEN_DEFAULT];
};
```

El `file->count` es una cuenta de referencia, incrementada por `get_file()` (usualmente llamada por `fget()`) y decrementada por `fput()` y por `put_filp()`. La diferencia entre `fput()` y `put_filp()` es que `fput()` hace más trabajo usualmente necesitado para ficheros regulares, como la liberación de conjuntos de bloqueos, liberación de dentry, etc, mientras que `put_filp()` es sólo para manipular las estructuras de tablas de ficheros, esto es, decrementa la cuenta, quita el fichero desde `anon_list` y lo añade a la `free_list`, bajo la protección del spinlock `files_lock`.

El `tsk->files` puede ser compartido entre padre e hijo si el hilo hijo fue creado usando la llamada al sistema `clone()` con la bandera `CLONE_FILES` establecida en los argumentos de las banderas de clone. Esto puede ser visto en `kernel/fork.c:copy_files()` (llamada por `do_fork()`) el cual sólo incrementa el `file->count` si `CLONE_FILES` está establecido, en vez de la copia usual de la tabla de descriptores de ficheros en la tradición respetable en el tiempo de los clásicos **fork(2)** de UNIX.

Cuando un fichero es abierto, la estructura del fichero asignada para él es instalada en el slot `current->files->fd[fd]` y un bit `fd` es establecido en el bitmap `current->files->open_fds`. Todo esto es realizado bajo la protección de escritura del spinlock read-write `current->files->file_lock`. Cuando el descriptor es cerrado un bit `fd` es limpiado en `current->files->open_fds` y `current->files->next_fd` es establecido igual a `fd` como una indicación para encontrar el primer descriptor sin usar la próxima vez que este proceso quiera abrir un fichero.

3.4 Administración de estructuras de ficheros

La estructura de ficheros es declarada en `include/linux/fs.h`:

```
struct fown_struct {
        int pid;                /* pid o -pgrp donde SIGIO debería de ser enviado  */
        uid_t uid, euid;        /* uid/euid del proceso estableciendo el dueño */
        int signum;             /* posix.1b rt señal para ser enviada en ES */
};

struct file {
        struct list_head        f_list;
        struct dentry           *f_dentry;
        struct vfsmount         *f_vfsmnt;
        struct file_operations  *f_op;
        atomic_t                f_count;
        unsigned int            f_flags;
        mode_t                  f_mode;
        loff_t                  f_pos;
        unsigned long           f_reada, f_ramax, f_raend, f_ralen, f_rawin;
        struct fown_struct      f_owner;
        unsigned int            f_uid, f_gid;
        int                     f_error;
```

```
        unsigned long           f_version;

        /* necesitado para este controlador tty, y quizás por otros */
        void                    *private_data;
};
```

Déjanos mirar varios de los campos de `struct file`:

1. **f_list**: este campo enlaza la estructura del fichero con una (y sólo una) de las listas: a) `sb->s_files` lista de todos los ficheros abiertos en este sistema de ficheros, si el correspondiente inodo no es anónimo, entonces `dentry_open()` (llamado por `filp_open()`) enlaza el fichero en esta lista; b) `fs/file_table.c:free_list`, conteniendo las estructuras de ficheros sin utilizar; c) `fs/file_table.c:anon_list`, cuando una nueva estructura de ficheros es creada por `get_empty_filp()` es colocada en esta lista. Todas estas listas son protegidas por el spinlock `files_lock`.
2. **f_dentry**: la dentry (entrada de directorio) correspondiente a este fichero. La dentry es creada en tiempo de búsqueda de nombre y datos (nameidata) por `open_namei()` (o más bien `path_walk()` la cual lo llama a él) pero el campo actual `file->f_dentry` es establecido por `dentry_open()` para contener la dentry de esta forma encontrada.
3. **f_vfsmnt**: el puntero a la estructura `vfsmount` del sistema de ficheros conteniendo el fichero. Esto es establecido por `dentry_open()`, pero es encontrado como una parte de la búsqueda de nameidata por `open_namei()` (o más bien `path_init()` la cual lo llama a él).
4. **f_op**: el puntero a `file_operations`, el cual contiene varios métodos que pueden ser llamados desde el fichero. Esto es copiado desde `inode->i_fop` que es colocado aquí durante la búsqueda nameidata. Miraremos los métodos `file_operations` en detalle más tarde en esta sección.
5. **f_count**: cuenta de referencia manipulada por `get_file/put_filp/fput`.
6. **f_flags**: banderas `O_XXX` desde la llamada al sistema **open(2)** copiadas allí (con ligeras modificaciones de `filp_open()`) por `dentry_open()` y después de limpiar `O_CREAT`, `O_EXCL`, `O_NOCTTY`, `O_TRUNC` - no hay sitio para almacenar estas banderas permanentemente ya que no pueden ser modificadas por las llamadas **fcntl(2)** `F_SETFL` (o consultadas por `F_GETFL`).
7. **f_mode**: una combinación de banderas del espacio de usuario y modos, establecido por `dentry_open()`. El punto de conversión es almacenar los accesos de lectura y escritura en bits separados, por lo tanto uno los chequearía fácilmente como `(f_mode & FMODE_WRITE)` y `(f_mode & FMODE_READ)`.
8. **f_pos**: la actual posición en el fichero para la siguiente lectura o escritura. Bajo i386 es del tipo `long long`, esto es un valor de 64 bits.
9. **f_reada, f_ramax, f_raend, f_ralen, f_rawin**: para soportar readahead - muy complejo para ser discutido por mortales ;)
10. **f_owner**: dueño del archivo de E/S a recibir las modificaciones de E/S asíncronas a través del mecanismo `SIGIO` (ver `fs/fcntl.c:kill_fasync()`).
11. **f_uid, f_gid** - establece el identificador del usuario y el identificador del grupo del proceso que abrió el fichero, cuando la estructura del fichero es creada por `get_empty_filp()`. Si el fichero es un conector, usado por netfilter ipv4.
12. **f_error**: usado por el cliente NFS para devolver errores de escritura. Esto es establecido en `fs/nfs/file.c` y chequeado en `mm/filemap.c:generic_file_write()`.

13. **f_version** - mecanismo de versionado para la invalidación de antememorias, incrementado (usando un `event` global) cuando cambia `f_pos`.

14. **private_data**: datos privados para cada fichero, los cuales pueden ser usados por los sistemas de ficheros (ej. coda almacena las credenciales aquí) o por otros controladores de dispositivos. Los controladores de dispositivos (en la presencia de devfs) pueden usar este campo para diferenciar entre múltiples instancias, en vez del clásico número menor codificado en `file->f_dentry->d_inode->i_rdev`.

Ahora déjanos mirar en la estructura `file_operations` la cual contiene los métodos que serán llamados en los archivos. Déjanos recalcar que es copiado desde `inode->i_fop` donde es establecido por el método `s_op->read_inode()`. Se declara en `include/linux/fs.h`:

```
struct file_operations {
        struct module *owner;
        loff_t (*llseek) (struct file *, loff_t, int);
        ssize_t (*read) (struct file *, char *, size_t, loff_t *);
        ssize_t (*write) (struct file *, const char *, size_t, loff_t *);
        int (*readdir) (struct file *, void *, filldir_t);
        unsigned int (*poll) (struct file *, struct poll_table_struct *);
        int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned long);
        int (*mmap) (struct file *, struct vm_area_struct *);
        int (*open) (struct inode *, struct file *);
        int (*flush) (struct file *);
        int (*release) (struct inode *, struct file *);
        int (*fsync) (struct file *, struct dentry *, int datasync);
        int (*fasync) (int, struct file *, int);
        int (*lock) (struct file *, int, struct file_lock *);
        ssize_t (*readv) (struct file *, const struct iovec *, unsigned long, loff_t *);
        ssize_t (*writev) (struct file *, const struct iovec *, unsigned long, loff_t *);
};
```

1. **owner**: un puntero al módulo que es dueño del subsistema en cuestión. Sólo los controladores necesitan establecerlo a `THIS_MODULE`, los sistemas de ficheros puede felizmente ignorarlos porque sus cuentas de módulos son controladas en el tiempo de montaje/desmontaje, en cambio los controladores necesitan controlarlo en tiempo de apertura/liberación.

2. **llseek**: implementa la llamada al sistema **lseek(2)**. Usualmente es omitida y es usada `fs/read_write.c:default_llseek()`, la cual hace lo correcto (POR HACER: fuerza a todos aquellos que establecen a NULL actualmente a usar default_llseek - que es el camino por el que salvamos una `if()` en `llseek()`).

3. **read**: implementa la llamada al sistema `read(2)`. Los sistemas de ficheros pueden usar `mm/filemap.c:generic_file_read()` para ficheros regulares y `fs/read_write.c:generic_read_dir()` (la cual simplemente devuelve `-EISDIR`) para directorios aquí.

4. **write**: implementa la llamada al sistema **write(2)**. Los sistemas de ficheros pueden usar `mm/filemap.c:generic_file_write()` para ficheros regulares e ignorarlo para directorios aquí.

5. **readdir**: usado por los sistema de ficheros. Ignorado por los ficheros regulares e implementa las llamadas al sistema **readdir(2)** y **getdents(2)** para directorios.

6. **poll**: implementa las llamadas al sistema **poll(2)** y **select(2)**.

7. **ioctl**: implementa el controlador o los ioctls específicos del sistema de ficheros. Nótese que los ioctls genéricos de los ficheros como `FIBMAP`, `FIGETBSZ`, `FIONREAD` son implementados por niveles más altos y por lo tanto nunca leerán el método `f_op->ioctl()`.

8. **mmap**: implementa la llamada al sistema **mmap(2)**. Los sistemas de ficheros pueden usar aquí **generic_file_mmap** para ficheros regulares e ignorarlo en los directorios.

9. **open**: llamado en tiempo de **open(2)** por `dentry_open()`. Los sistemas de ficheros raramente usan esto, ej. coda intenta almacenar el fichero localmente en tiempo de apertura.

10. **flush**: llamada en cada **close(2)** de este fichero, no necesariamente el último (ver el método `release()` a continuación). El único sistema de ficheros que lo utiliza es en un cliente NFS para pasar a disco todas las páginas sucias. Nótese que esto puede devolver un error el cual será retornado al espacio de usuario que realizó la llamada al sistema **close(2)**.

11. **release**:llamado por la última **close(2)** de este fichero, esto es cuando `file->f_count` llega a 0. Aunque definido como un entero de retorno, el valor de retorno es ignorado por VFS (ver `fs/file_table.c:__fput()`).

12. **fsync**: mapea directamente a las llamadas al sistema **fsync(2)**/**fdatasync(2)**, con el último argumento especificando cuando es fsync o fdatasync. Por lo menos no se realiza trabajo por VFS sobre esto, excepto el mapear el descriptor del fichero a una estructura de fichero (`file = fget(fd)`) y bajar/subir el semáforo `inode->i_sem`. El sistema de ficheros Ext2 ignora el último argumento y realiza lo mismo para **fsync(2)** y **fdatasync(2)**.

13. **fasync**: este método es llamado cuando cambia `file->f_flags & FASYNC`.

14. **lock**: parte del mecanismo de bloqueo de la región del **fcntl(2)** POSIX de la porción específica del sistema de ficheros. El único fallo aquí es porque es llamado antes por una porción independiente del sistema de ficheros (`posix_lock_file()`), si tiene éxito pero el código de bloqueo estandart POSIX falla, entonces nunca será desbloqueado en un nivel dependiente del sistema de ficheros...

15. **readv**: implementa la llamada al sistema **readv(2)**.

16. **writev**: implementa la llamada al sistema **writev(2)**.

3.5 Administración de Puntos de Montaje y Superbloque

Bajo Linux, la información sobre los sistemas de ficheros montados es mantenida en dos estructuras separadas - `super_block` y `vfsmount`. El motivo para esto es que Linux permite montar el mismo sistema de ficheros (dispositivo de bloque) bajo múltiples puntos de montaje, lo cual significa que el mismo `super_block` puede corresponder a múltiples estructuras `vfsmount`.

Déjanos mirar primero en `struct super_block`, declarado en `include/linux/fs.h`:

```
struct super_block {
        struct list_head        s_list;         /* Mantiene esto primero */
        kdev_t                  s_dev;
        unsigned long           s_blocksize;
        unsigned char           s_blocksize_bits;
        unsigned char           s_lock;
        unsigned char           s_dirt;
        struct file_system_type *s_type;
        struct super_operations *s_op;
        struct dquot_operations *dq_op;
        unsigned long           s_flags;
        unsigned long           s_magic;
        struct dentry           *s_root;
        wait_queue_head_t       s_wait;
```

```
        struct list_head        s_dirty;        /* inodos sucios */
        struct list_head        s_files;

        struct block_device     *s_bdev;
        struct list_head        s_mounts;       /* vfsmount(s) de este */
        struct quota_mount_options s_dquot;     /* Opciones Específicas de Diskquota */

       union {
                struct minix_sb_info    minix_sb;
                struct ext2_sb_info     ext2_sb;
                ..... todos los sistemas de archivos necesitan sb-private ...
                void                    *generic_sbp;
        } u;
       /*
         * El siguiente campo es *sólo* para VFS. Los sistemas de ficheros
         * no tienen trabajo alguno mirando en él. Has sido avisado.
         */
        struct semaphore s_vfs_rename_sem;      /* Truco */

        /* El siguiente campo es usado por knfsd cuando convierte un
         * manejador de ficheros (basado en el número de inodo) en
         * una dentry. Tal como construye un camino en el árbol dcache
         * desde el fondo hasta arriba, quizás exista durante algún
         * tiempo un subcamino de dentrys que no están conectados al
         * árbol principal. Este semáforo asegura que hay sólo
         * siempre un camino libre por sistema de ficheros. Nótese que
         * los ficheros no conectados (o otros no directorios) son
         * permitidos, pero no los directorios no conectados.
         */
        struct semaphore s_nfsd_free_path_sem;
};
```

Las diversos campos en la estructura `super_block` son:

1. **s_list**: una lista doblemente enlazada de todos los superbloques activos; nótese que no he dicho "de todos los sistemas de ficheros montados" porque bajo Linux uno puede tener múltiples instancias de un sistema de ficheros montados correspondientes a un superbloque simple.

2. **s_dev**: para sistemas de ficheros que requieren un bloque para ser montado en él. Esto es para los sistemas de ficheros `FS_REQUIRES_DEV`, esto es la `i_dev` del dispositivo de bloques. Para otros (llamados sistemas de ficheros anónimos) esto es un entero `MKDEV(UNNAMED_MAJOR, i)` donde `i` es el primer bit no establecido en la matriz `unnamed_dev_in_use`, entre 1 y 255 incluidos. Ver `fs/super.c:get_unnamed_dev()/put_unnamed_dev()`. Ha sido sugerido muchas veces que los sistemas de ficheros anónimos no deberían de usar el campo `s_dev`.

3. **s_blocksize, s_blocksize_bits**: tamaño del bloque y log2(tamaño del bloque).

4. **s_lock**: indica cuando un superbloque está actualmente bloqueado por `lock_super()/unlock_super()`.

5. **s_dirt**: establece cuando el superbloque está modificado, y limpiado cuando es vuelto a ser escrito a disco.

6. **s_type**: puntero a `struct file_system_type` del sistema de ficheros correspondiente. El método `read_super()` del sistema de ficheros no necesita ser establecido como VFS

fs/super.c:read_super(), lo establece para ti si el read_super() que es específico del sistema de ficheros tiene éxito, y se reinicializa a NULL si es que falla.

7. **s_op**: puntero a la estructura super_operations, la cual contiene métodos específicos del sistema de ficheros para leer/escribir inodos, etc. Es el trabajo del método read_super() del sistema de ficheros inicializar s_op correctamente.

8. **dq_op**: operaciones de cuota de disco.

9. **s_flags**: banderas de superbloque.

10. **s_magic**: número mágico del sistema de ficheros. Usado por el sistema de ficheros de minix para diferenciar entre múltiples tipos del mismo.

11. **s_root**: dentry de la raiz del sistema de ficheros. Es trabajo de read_super() leer el inodo raiz desde el disco y pasárselo a d_alloc_root() para asignar la dentry e instanciarlo. Algunos sistemas de ficheros dicen "raiz" mejor que "/" y por lo tanto usamos la función más genérica d_alloc() para unir la dentry a un nombre, ej. pipefs se monta a si mismo en "pipe:" como su raiz en vez de "/".

12. **s_wait**: cola de espera de los procesos esperando para que el superbloque sea desbloqueado.

13. **s_dirty**: una lista de todos los inodos sucios. Recalcar que si un inodo está sucio (inode->i_state & I_DIRTY) entonces su lista sucia específica del superbloque es enlazada a través de inode->i_list.

14. **s_files**: una lista de todos los ficheros abiertos en este superbloque. Util para decidir cuándo los sistemas de archivos pueden ser remontados como de sólo lectura, ver fs/file_table.c:fs_may_remount_ro() el cual va a través de la lista sb->s_files y deniega el remontar si hay ficheros abiertos para escritura (file->f_mode & FMODE_WRITE) o ficheros con desenlaces pendientes (inode->i_nlink == 0).

15. **s_bdev**: para FS_REQUIRES_DEV, esto apunta a la estructura block_device describiendo el dispositivo en el que el sistema de ficheros está montado.

16. **s_mounts**: una lista de todas las estructuras vfsmount, una por cada instancia montada de este superbloque.

17. **s_dquot**: más miembros de diskquota.

Las operaciones de superbloque son descritas en la estructura super_operations declarada en include/linux/fs.h:

```
struct super_operations {
        void (*read_inode) (struct inode *);
        void (*write_inode) (struct inode *, int);
        void (*put_inode) (struct inode *);
        void (*delete_inode) (struct inode *);
        void (*put_super) (struct super_block *);
        void (*write_super) (struct super_block *);
        int (*statfs) (struct super_block *, struct statfs *);
        int (*remount_fs) (struct super_block *, int *, char *);
        void (*clear_inode) (struct inode *);
        void (*umount_begin) (struct super_block *);
};
```

1. **read_inode**: lee el inodo desde el sistema de archivos. Es sólo llamado desde fs/inode.c:get_new_inode(), y desde iget4() (y por consiguiente iget()). Si un sistema de ficheros quiere usar iget() entonces read_inode() debe de ser implementado - en otro caso get_new_inode()

fallará. Mientras el inodo está siendo leido está bloqueado (`inode->i_state = I_LOCK`). Cuando la función regresa, todos los que están esperando en `inode->i_wait` son despertados. El trabajo del método `read_inode()` del sistema de ficheros es localizar el bloque del disco que contiene el inodo a ser leído y usar la funcion de la antememoria intermedia `bread()` para leerlo e inicializar varios campos de la estructura de inodos, por ejemplo el `inode->i_op` y `inode->i_fop` para que los niveles VFS conozcan qué operaciones pueden ser efectuadas en el inodo o fichero correspondiente. Los sistemas de ficheros que no implementan `read_inode()` son ramfs y pipefs. Por ejemplo, ramfs tiene su propia función de generación de inodos `ramfs_get_inode()` con todas las operaciones de inodos llamándola cuando se necesita.

2. **write_inode**: escribe el inodo de vuelta al disco. Similar a `read_inode()` en que necesita localizar el bloque relevante en el disco e interactuar con la antememoria intermedia llamando a `mark_buffer_dirty(bh)`. Este método es llamado en los inodos sucios (aquellos marcados como sucios por `mark_inode_dirty()`) cuando el inodo necesita ser sincronizado individualmente o como parte de la actualizacion entera del sistema de ficheros.
3. **put_inode**: llamado cuando la cuenta de referencia es decrementada.
4. **delete_inode**: llamado cuando `inode->i_count` y `inode->i_nlink` llegan a 0. El sistema de ficheros borra la copia en disco del inodo y llama a `clear_inode()` en el inodo VFS para "terminar con él con el perjuicio extremo".
5. **put_super**: llamado en las últimas etapas de la llamada al sistema **umount(2)** para notificar al sistema de ficheros que cualquier información mantenida por el sistema de ficheros sobre esa instancia tiene que ser liberada. Típicamente esto `brelse()` el bloque conteniendo el superbloque y `kfree()` cualesquiera bitmaps asignados para bloques libres, inodos, etc.
6. **write_super**: llamado cuando el superbloque necesita ser vuelto a escribir en el disco. Debería de encontrar el bloque conteniendo el superbloque (usualmente mantenido en el área `sb-private`) y `mark_buffer_dirty(bh)`. También debería de limpiar la bandera `sb->s_dirt`.
7. **statfs**: implementa las llamadas al sistema **fstatfs(2)/statfs(2)**. Nótese que el puntero a `struct statfs` pasado como argumento, es el puntero del núcleo, no un puntero del usuario, por lo tanto no necesitamos hacer ninguna E/S al espacio de usuario. Si no está implementada entonces `statfs(2)` fallará con `ENOSYS`.
8. **remount_fs**: llamado cuando el sistema de ficheros está siendo remontado.
9. **clear_inode**: llamado desde el nivel VFS `clear_inode()`. Los sistemas que atacan datos privados a la estructura del inodo (a través del campo `generic_ip`) deben liberarse aquí.
10. **umount_begin**: llamado durante el desmontaje forzado para notificarlo al sistema de ficheros de antemano, por lo tanto puede ser lo mejor para asegurarse que nada mantiene al sistema de ficheros ocupado. Actualmente usado sólo por NFS. Esto no tiene nada que hacer con la idea del soporte de desmontaje forzado del nivel genérico de VFS

Por lo tanto, déjanos mirar qué pasa cuando montamos un sistema de ficheros en disco (`FS_REQUIRES_DEV`). La implementación de la llamada al sistema **mount(2)** está en `fs/super.c:sys_mount()` que es justo un envoltorio que copia las opciones, el tipo del sistema de ficheros y el nombre del dispositivo para la función `do_mount()`, la cual realiza el trabajo real:

1. El controlador del sistema de ficheros es cargado si se necesita y la cuenta de referencia del módulo es incrementada. Nótese que durante la operación de montaje, la cuenta del sistema de ficheros es incrementada dos veces - una vez por `do_mount()` llamando a `get_fs_type()` y otra vez por `get_sb_dev()`

llamando a `get_filesystem()` si `read_super()` tuvo éxito. El primer incremento es para prevenir la descarga del módulo mientras estamos dentro del método `read_super()`, y el segundo incremento es para indicar que el módulo está en uso por esta instancia montada. Obviamente, `do_mount()` decrementa la cuenta antes de regresar, por lo tanto, después de todo, la cuenta sólo crece en 1 después de cada montaje.

2. Desde, que en nuestro caso, `fs_type->fs_flags & FS_REQUIRES_DEV` es verdad, el superbloque es inicializado por una llamada a `get_sb_bdev()`, la cual obtiene la referencia al dispositivo de bloques e interactúa con el método `read_super()` del sistema de ficheros para rellenar el superbloque. Si todo va vien, la estructura `super_block` es inicializada y tenemos una referencia extra al módulo del sistema de ficheros y una referencia al dispositivo de bloques subyacente.

3. Una nueva estructura `vfsmount` es asignada y enlazada a la lista `sb->s_mounts` y a la lista global `vfsmntlist`. El campo `vfsmount` de `mnt_instances` nos permite encontrar todas las instancias montadas en el mismo superbloque que este. El campo `mnt_list` nos permite encontrar todas las instancias para todos los superbloques a lo largo del sistema. El campo `mnt_sb` apunta a este superbloque y `mnt_root` tiene una nueva referencia a la dentry `sb->s_root`.

3.6 Ejemplo de un Sistema de Ficheros Virtual: pipefs

Como un ejemplo simple del sistema de ficheros de Linux que no requiere un dispositivo de bloque para montar, déjanos considerar pipefs desde `fs/pipe.c`. El preámbulo del sistema de ficheros es bastante directo y requiere una pequeña explicación:

```
static DECLARE_FSTYPE(pipe_fs_type, "pipefs", pipefs_read_super,
        FS_NOMOUNT|FS_SINGLE);

static int __init init_pipe_fs(void)
{
        int err = register_filesystem(&pipe_fs_type);
        if (!err) {
                pipe_mnt = kern_mount(&pipe_fs_type);
                err = PTR_ERR(pipe_mnt);
                if (!IS_ERR(pipe_mnt))
                        err = 0;
        }
        return err;
}

static void __exit exit_pipe_fs(void)
{
        unregister_filesystem(&pipe_fs_type);
        kern_umount(pipe_mnt);
}

module_init(init_pipe_fs)
module_exit(exit_pipe_fs)
```

El sistema de ficheros es del tipo `FS_NOMOUNT|FS_SINGLE`, lo que significa que no puede ser montado desde el espacio de usuario y sólo puede haber uno en el sistema. El fichero `FS_SINGLE` también significa que debe de ser montado a través de `kern_mount()` después de que haya sido registrado con éxito a través de `register_filesystem()`, lo cual es exactamente lo que pasa en `init_pipe_fs()`. El único fallo en esta función es que si `kern_mount()` falla (ej. porque `kmalloc()` falló en `add_vfsmnt()`) entonces el sistema de ficheros es dejado como registrado pero la inicialización del módulo falla. Esto causará que **cat**

/proc/filesystems falle (justamente acabo de enviar un parche a Linus mencionándole que esto no es un fallo real hoy en día porque pipefs no puede ser compilado como módulo, debería de ser escrito desde el punto de vista de que en el futuro pudiera ser modularizado).

El resultado de `register_filesystem()` es que `pipe_fs_type` es enlazado en la lista `file_systems`, por lo tanto uno puede leer `/proc/filesystems` y encontrar la entrada "pipefs" allí con la bandera "nodev" indicando que `FS_REQUIRES_DEV` no fue establecida. El archivo `/proc/filesystems` debería realmente de ser capaz de soportar todas las nuevas banderas `FS_` (y yo he hecho un parche que lo hace) pero no puede ser realizado porque hará fallar a todas las aplicaciones de usuario que lo utilicen. A pesar de que los interfaces del núcleo Linux cambian cada minuto (sólo para mejor) cuando se refiere a la compatibilidad del espacio de usuario, Linux es un sistema operativo muy conservador que permite que muchas aplicaciones sean usadas durante un largo periodo de tiempo sin ser recompiladas.

El resultado de `kern_mount()` es que:

1. Un nuevo número de dispositvo sin nombre (anónimo) es asignado estableciendo un bit en el bitmap `unnamed_dev_in_use`; si no hay más bits entonces `kern_mount()` fallará con `EMFILE`.
2. Una nueva estructura superbloque es asignada por medio de `get_empty_super()`. La función `get_empty_super()` camina a través de las cabeceras de las lista de superbloques por `super_block` y busca una entrada vacía, esto es `s->s_dev == 0`. Si no se encuentra dicho superbloque vacío entonces uno nuevo es asignado usando `kmalloc()` con la prioridad `GFP_USER`. El número máximo de superbloques en el sistema es chequeado en `get_empty_super()`, por lo tanto empieza fallando, uno puede modificar el parámetro ajustable `/proc/sys/fs/super-max`.
3. Un método específico del sistema de ficheros `pipe_fs_type->read_super()`, esto es `pipefs_read_super()`, es invocada, la cual asigna el inodo y la dentry raiz `sb->s_root`, y establece `sb->s_op` para ser `&pipefs_ops`.
4. Entonces `kern_mount()` llama a `add_vfsmnt(NULL, sb->s_root, "none")` la cual asigna una nueva estructura `vfsmount` y la enlaza en `vfsmntlist` y `sb->s_mounts`.
5. El `pipe_fs_type->kern_mnt` es establecido a esta nueva estructura `vfsmount` y es devuelta. El motivo por el que el valor de retorno de `kern_mount()` es una estructura `vfsmount` es porque incluso los sistemas de ficheros `FS_SINGLE` pueden ser montados múltiples veces y por lo tanto sus `mnt->mnt_sb` deberían apuntar a la misma cosa, que sería tonto devolverla desde múltiples llamadas a `kern_mount()`.

Ahora que el sistema de ficheros está registrado y montado dentro del núcleo podemos usarlo. El punto de entrada en el sistema de ficheros pipefs es la llamada al sistema **pipe(2)**, implementada por una función dependiente de la arquitectura `sys_pipe()`, pero el trabajo real es realizado por un función portable `fs/pipe.c:do_pipe()`. Déjanos mirar entonces en `do_pipe()`. La interacción con pipefs sucede cuando `do_pipe()` llama a `get_pipe_inode()` para asignar un nuevo inodo pipefs. Para este inodo, `inode->i_sb` es establecido al superbloque de pipefs `pipe_mnt->mnt_sb`, las operaciones del archivo `i_fop` son establecidas a `rdwr_pipe_fops` y el número de lectores y escritores (mantenidos en `inode->i_pipe`) es establecido a 1. El motivo por el que hay un campo de inodos separado `i_pipe` en vez de mantenerlo en la unión `fs-private` es que pipes y FIFOs comparten el mismo código y los FIFOs puede existir en otros sistemas de ficheros los cuales usan otros caminos de acceso con la misma unión, lo cual en C es muy malo y puede trabajar sólo con pura suerte. Por lo tanto, sí, los núcleos 2.2.x trabajan por pura suerte y pararán de trabajar tan pronto como tu retoques ligeramente los campos en el inodo.

Cada llamada al sistema **pipe(2)** incrementa una cuenta de referencia en la instancia de montaje `pipe_mnt`.

Bajo Linux, los pipes no son simétricos (pipes STREAM o bidireccionales) esto es, dos caras del mismo fichero tienes diferentes operaciones `file->f_op` - la `read_pipe_fops` y `write_pipe_fops` respectivamente. La escritura en la cara de lectura devuelve un `EBADF` y lo mismo si se lee en la cara de escritura.

3.7 Ejemplo de Sistema de Ficheros de Disco: BFS

Como un simple ejemplo de un sistema de ficheros Linux en disco, déjanos considerar BFS. El preámbulo del módulo de BFS está en `fs/bfs/inode.c`:

```
static DECLARE_FSTYPE_DEV(bfs_fs_type, "bfs", bfs_read_super);

static int __init init_bfs_fs(void)
{
        return register_filesystem(&bfs_fs_type);
}

static void __exit exit_bfs_fs(void)
{
        unregister_filesystem(&bfs_fs_type);
}

module_init(init_bfs_fs)
module_exit(exit_bfs_fs)
```

Una declaracion especial de la macro del sistema de ficheros `DECLARE_FSTYPE_DEV()` es usada, la cual establece el `fs_type->flags` a `FS_REQUIRES_DEV` para significar que BFS requiere un dispositivo de bloque real para ser montado.

La función de inicialización del módulo registra el sistema de ficheros con VFS y la función de limpieza (sólo presente cuando BFS está configurada para ser un módulo) lo desregistra.

Con el sistema de ficheros registrado, podemos proceder a montarlo, lo cual invocará al método `fs_type->read_super()` que es implementado en `fs/bfs/inode.c:bfs_read_super()`.. El realiza lo siguiente:

1. `set_blocksize(s->s_dev, BFS_BSIZE)`: desde que nosotros vamos a interactuar con la capa de dispositivos de bloque a través de la antememoria intermedia, debemos inicializar unas pocas cosas, esto es, establecer el tamaño del bloque y también informar a VFS a través de los campos `s->s_blocksize` y `s->s_blocksize_bits`.
2. `bh = bread(dev, 0, BFS_BSIZE)`: leemos el bloque 0 del dispositivo a través de `s->s_dev`. Este bloque es el superbloque del sistema.
3. El superbloque es validado contra el número `BFS_MAGIC` y, si es válido, es almacenado en el campo sb-private `s->su_sbh` (el cual es realmente `s->u.bfs_sb.si_sbh`).
4. Entonces asignamos el bitmap del inodo `kmalloc(GFP_KERNEL)` y limpiamos todos sus bits a 0, excepto los dos primeros, los cuales estableceremos a 1 para indicar que nunca deberemos asignar los inodos 0 y 1. El inodo 2 es la raiz y el correspondiente bit será establecido a 1 unas pocas lineas después de cualquier forma - ¡el sistema de ficheros debe de tener un inodo raiz válido en tiempo de montaje!
5. Entonces inicializamos `s->s_op`, lo cual significa que podemos desde este punto llamar a la memoria intermedia de inodos a través de `iget()`, lo cual resulta en la invocación de `s_op->read_inode()`. Esto encuentra el bloque que contiene el inodo especificado (por `inode->i_ino` y `inode->i_dev`) y lo lee. Si fallamos al obtener el inodo raiz entonces liberamos el bitmap de inodos y descargaremos la antememoria de superbloque a la antememoria intermedia y devolveremos NULL. Si el inodo raiz fue leido correctamente, entonces asignamos una dentry con el nombre `/` (como convirtiéndolo en raiz) y lo instanciamos con este inodo.

6. Ahora vamos a través de todos los inodos del sistema de ficheros y los leemos en orden a establecer los bits correspondientes en nuestro bitmap interno de inodos y también calculamos otros parámetros internos como el desplazamiento del último inodo y el comienzo/final del último fichero. Cada inodo que leemos es devuelto atrás a la memoria intermedia de inodos a través de `iput()` - no mantenemos una referencia a él más tiempo del necesario.

7. Si el sistema de ficheros no fue montado como de sólo lectura, marcamos la memoria intermedia del superbloque como sucio y establecemos la bandera `s->s_dirt` (POR HACER: ¿Porqué hago esto? Originalmente, lo hice porque lo hacía `minix_read_super()` pero ni minix ni BFS parecen modificar el superbloque en el `read_super()`).

8. Todo está bién, por lo tanto regresamos atrás a este superbloque inicializado para el llamante en el nivel VFS, esto es, `fs/super.c:read_super()`.

Después de que la función `read_super()` regrese con éxito, VFS obtiene la referencia al módulo del sistema de ficheros a través de la llamada a `get_filesystem(fs_type)` en `fs/super.c:get_sb_bdev()` y una referencia al dispositivo de bloques.

Ahora, déjanos examinar qué pasa cuando hacemos una E/S en el sistema de ficheros. Ya hemos examinado cómo los inodos son leidos cuando `iget()` es llamado y cómo son quitados en `iput()`. Leyendo inodos, configura entre otras cosas, `inode->i_op` y `inode->i_fop`; abriendo un fichero propagará `inode->i_fop` en `file->f_op`.

Déjanos examinar el camino de código de la llamada al sistema **link(2)**. La implementación de la llamada al sistema está en `fs/namei.c:sys_link()`:

1. Los nombres del espacio de usuario son copiados en el espacio del núcleo por medio de la función `getname()` la cual realiza el chequeo de errores.

2. Estos nombres son convertidos a datos usando `path_init()/path_walk()` interactuando con dcache. El resultado es almacenado en las estructuras `old_nd` y `nd`.

3. Si `old_nd.mnt != nd.mnt` entonces "enlace a través de dispositvos" `EXDEV` es devuelto - uno no puede enlazar entre sistemas de ficheros, en Linux esto se traduce en - uno no puede enlazar entre múltiples instancias de un sistema de ficheros (o, en particular entre sistemas de ficheros).

4. Una nueva dentry es creada correspondiente a `nd` por `lookup_create()` .

5. Una función genérica `vfs_link()` es llamada, la cual chequea si podemos crear una nueva entrada en el directorio e invoca el método `dir->i_op->link()`, que nos trae atrás a la función específica del sistema de ficheros `fs/bfs/dir.c:bfs_link()`.

6. Dentro de `bfs_link()`, chequeamos si estamos intentando enlazar un directorio, y si es así, lo rechazamos con un error `EPERM`. Este es el mismo comportamiento que el estándar (ext2).

7. Intentamos añadir una nueva entrada de directorio al directorio especificado por la función de ayuda `bfs_add_entry()` la cual va a través de todas las entradas buscando un slot sin usar (`de->ino == 0`) y, cuando lo encuentra, escribe en el par nombre/inodo en el bloque correspondiente y lo marca como sucio (a una prioridad no-superbloque).

8. Si hemos añadido con éxito la entrada de directorio entonces no hay forma de fallar la operación y por lo tanto incrementamos `inode->i_nlink`, actualizamos `inode->i_ctime` y marcamos este inodo como sucio a la vez que instanciamos la nueva dentry con el inodo.

Otras operaciones de inodos relacionadas como `unlink()/rename()` etc. trabajan en una forma similar, por lo tanto no se gana mucho examinándolas a todas ellas en detalle.

3.8 Dominios de Ejecución y Formatos Binarios

Linux soporta la carga de aplicaciones binarias de usuario desde disco. Más interesantemente, los binarios pueden ser almacenados en formatos diferentes y la respuesta del sistema operativo a los programas a través de las llamadas al sistema pueden desviarla de la norma (la norma es el comportamiento de Linux) tal como es requerido, en orden a emular los formatos encontrados en otros tipos de UNIX (COFF, etc) y también emular el comportamiento de las llamadas al sistema de otros tipos (Solaris, UnixWare, etc). Esto es para lo que son los dominios de ejecución y los formatos binarios.

Cada tarea Linux tiene una personalidad almacenada en su `task_struct` (`p->personality`). Las personalidades actualmente existentes (en el núcleo oficial o en el parche añadido) incluyen soporte para FreeBSD, Solaris, UnixWare, OpenServer y algunos otros sistemas operativos populares. El valor de `current->personality` es dividido en dos partes:

1. tres bytes altos - emulación de fallos: `STICKY_TIMEOUTS`, `WHOLE_SECONDS`, etc.
2. byte bajo - personalidad propia, un número único.

Cambiando la personalidad, podemos cambiar la forma en la que el sistema operativo trata ciertas llamadas al sistema, por ejemplo añadiendo una `STICKY_TIMEOUT` a `current->personality` hacemos que la llamada al sistema **select(2)** preserve el valor del último argumento (timeout) en vez de almacenar el tiempo no dormido. Algunos programas defectuosos confían en sistemas operativos defectuosos (no Linux) y por lo tanto suministra una forma para emular fallos en casos donde el código fuente no está disponible y por lo tanto los fallos no pueden ser arreglados.

El dominio de ejecución es un rango contiguo de personalidades implementadas por un módulo simple. Usualmente un dominio de ejecución simple implementa una personalidad simple, pero a veces es posible implementar personalidades "cerradas" en un módulo simple sin muchos condicionantes.

Los dominios de ejecución son implementados en `kernel/exec_domain.c` y fueron completamente reescritos para el núcleo 2.4, comparado con el 2.2.x. La lista de dominios de ejecución actualmente soportada por el núcleo, a lo largo del rango de personalidades que soportan, está disponible leyendo el archivo `/proc/execdomains`. Los dominios de ejecución, excepto el `PER_LINUX`, pueden ser implementados como módulos dinámicamente cargados.

La interfaz de usuario es a través de la llamada al sistema **personality(2)**, la cual establece la actual personalidad del proceso o devuelve el valor de `current->personality` si el argumento es establecido a una personalidad imposible. Obviamente, el comportamiento de esta llamada al sistema no depende de la personalidad.

La interfaz del núcleo para el registro de dominios de ejecución consiste en dos funciones:

- `int register_exec_domain(struct exec_domain *)`: registra el dominio de ejecución enlazándolo en una lista simplemente enlazada `exec_domains` bajo la protección de escritura del spinlock read-write `exec_domains_lock`. Devuelve 0 si tiene éxito, distinto de cero en caso de fallo.
- `int unregister_exec_domain(struct exec_domain *)`: desregistra el dominio de ejecución desenlazándolo desde la lista `exec_domains`, otra vez usando el spinlock `exec_domains_lock` en modo de escritura. Retorna 0 si tiene éxito.

El motivo por el que `exec_domains_lock` es read-write es que sólo las peticiones de registro y desregistro modifican la lista, mientras haciendo **cat /proc/filesystems** llama `fs/exec_domain.c:get_exec_domain_list()`, el cual necesita sólo acceso de lectura a la lista. Registrando un nuevo dominio de ejecución define un "manejador lcall7" y un mapa de conversión de número de señales. Actualmente, el parche ABI extiende este concepto a los dominios de ejecución para incluir información extra (como opciones de conector, tipos de conector, familia de direcciones y mapas de números de errores).

Los formatos binarios son implementados de una forma similar, esto es, una lista simplemente enlazada de formatos es definida en `fs/exec.c` y es protegida por un cierre read-write `binfmt_lock`. Tal como con `exec_domains_lock`, el `binfmt_lock` es tomado para leer en la mayoría de las ocasiones excepto para el registro/desregistro de los formatos binarios. Registrando un nuevo formato binario intensifica la llamada al sistema **execve(2)** con nuevas funciones `load_binary()/load_shlib()`. Al igual que la habilidad para `core_dump()`. El método `load_shlib()` es usado sólo por la vieja llamada al sistema **uselib(2)** mientras que el método `load_binary()` es llamada por el `search_binary_handler()` desde `do_execve()` el cual implementa la llamada al sistema **execve(2)**.

La personalidad del proceso está determinada por el formato binario cargado por el método del correspondiente formato `load_binary()` usando algunas heurísticas. Por ejemplo, para determinar los binarios UnixWare7, uno primero marca el binario usando la utilidad **elfmark(1)**, la cual establece la cabecera de ELF `e_flags` al valor mágico 0x314B4455, el cual es detectado en tiempo de carga del ELF y `current->personality` es establecido a PER_SVR4. Si esta heurística falla entonces una más genérica como el tratamiento de los caminos del intérprete ELF como `/usr/lib/ld.so.1` o `/usr/lib/libc.so.1` para indicar un binario SVR4, es usado y la personalidad es establecida a PER_SVR4. Uno podría escribir un pequeño programa de utilidad que usara las capacidades del **ptrace(2)** de Linux para, en un simple paso, códificar y forzar a un programa funcionando a cualquier personalidad.

Una vez que la personalidad (y entonces `current->exec_domain`) es conocida, las llamadas al sistema son manejadas como sigue. Déjanos asumir que un proceso realiza una llamada al sistema por medio de la instrucción puerta lcall7. Esto transfiere el control a `ENTRY(lcall7)` de `arch/i386/kernel/entry.S` porque fue preparado en `arch/i386/kernel/traps.c:trap_init()`. Después de la apropiada conversión de la pila, `entry.S:lcall7` obtiene el puntero a `exec_domain` desde `current` y entonces un desplazamiento del manejador lcall7 con el `exec_domain` (el cual es codificado fuertemente como 4 en código ensamblador, por lo tanto no puedes desplazar el campo `handler` a través de la declaración en C de `struct exec_domain`) y salta a él. Por lo tanto, en C, se parecería a esto:

```
static void UW7_lcall7(int segment, struct pt_regs * regs)
{
       abi_dispatch(regs, &uw7_funcs[regs->eax & 0xff], 1);
}
```

donde `abi_dispatch()` es un envoltorio sobre la tabla de punteros de función que implementa las llamadas al sistema de esta personalidad `uw7_funcs`.

4 Memoria Intermedia de Páginas Linux

En este capítulo describimos la memoria intermedia de páginas de Linux 2.4. La memoria intermedia de páginas es - como sugiere el nombre - una memoria intermedia de páginas físicas. En el mundo UNIX el concepto de memoria intermedia de páginas se convirtió popular con la introdución de SVR4 UNIX, donde reemplazó a la antememoria intermedia para las operaciones de E/S.

Mientras la memoria intermedia de páginas de SVR4 es sólamente usada como memoria intermedia de datos del sistema de ficheros y estos usan la estructura vnode y un desplazamiento dentro del fichero como parámetros hash, la memoria intermedia de páginas de Linux está diseñada para ser más genérica, y entonces usa una estructura address_space (explicada posteriormente) como primer parámetro. Porque la memoria intermedia de páginas Linux está cerradamente emparejada a la notación del espacio de direcciones, necesitarás como mínimo un conocimiento previo del adress_spaces para entender la forma en la que trabaja la memoria intermedia de páginas. Un address_space es algún tipo de software MMU que mapea todas las páginas de un objeto (ej. inodo) a otro concurrentemente (tipicamente bloques físicos de disco). La estructura address_space está definida en `include/linux/fs.h` como:

```
struct address_space {
        struct list_head        clean_pages;
        struct list_head        dirty_pages;
        struct list_head        locked_pages;
        unsigned long           nrpages;
        struct address_space_operations *a_ops;
        struct inode            *host;
        struct vm_area_struct   *i_mmap;
        struct vm_area_struct   *i_mmap_shared;
        spinlock_t              i_shared_lock;

};
```

Para entender la forma en la que address_spaces trabaja, sólo necesitamos mirar unos pocos de estos campos: `clean_pages`, `dirty_pages` y `locked_pages` son listas doblemente enlazadas de páginas limpias, sucias y bloqueadas pertenecientes a este address_space, `nrpages` es el número total de páginas en este address_space. `a_ops` define los métodos de este objeto y `host` es un puntero perteneciente a este inodo address_space - quizás sea NULL, ej. en el caso del swapper (intercambiador) address_space (`mm/swap_state.c,`).

El uso de `clean_pages`, `dirty_pages`, `locked_pages` y `nrpages` es obvio, por lo tanto hecharemos un severo vistazo a la estructura `address_space_operations`, definida en la misma cabecera:

```
struct address_space_operations {
        int (*writepage)(struct page *);
        int (*readpage)(struct file *, struct page *);
        int (*sync_page)(struct page *);
        int (*prepare_write)(struct file *, struct page *, unsigned, unsigned);
        int (*commit_write)(struct file *, struct page *, unsigned, unsigned);
        int (*bmap)(struct address_space *, long);
};
```

Para una vista básica del principio de address_spaces (y de la memoria intermedia de páginas) necesitamos hechar una mirada a ->`writepage` y ->`readpage`, pero en la práctica necesitamos también mirar en ->`prepare_write` y ->`commit_write`.

Probablemente supongas que los métodos address_space_operations lo hacen en virtud de sus nombres solamente; no obstante, requieren hacer alguna explicación. Su uso en el camino de las E/S de los datos de los sistemas de ficheros, por lo lejos del más común camino a través de la memoria intermedia de páginas, suministra una buena forma para entenderlas. Como la mayoría de otros sistemas operativos del estilo UNIX, Linux tiene unas operaciones genéricas de ficheros (un subconjunto de las operaciones vnode SYSVish) para los datos E/S a través de la memoria intermedia de páginas. Esto significa que los datos no interactúan directamente con el sistema de ficheros en read/write/mmap, pero serán leidos/escritos desde/a la memoria intermedia de páginas cuando sea posible. La memoria intermedia de páginas tiene que obtener datos desde el sistema de ficheros actual de bajo nivel en el caso de que el usuario quiera leer desde una página que todavía no está en memoria, o escribir datos al disco en el caso de que la memoria sea insuficiente.

En el camino de lectura, los métodos genéricos primero intentarán encontrar una página que corresponda con la pareja buscada de inodo/índice.

```
hash = page_hash(inode->i_mapping, index);
```

Entonces testeamos cuando la página actualmente existe.

```
hash = page_hash(inode->i_mapping, index); page = __find_page_nolock(inode->i_mapping,
index, *hash);
```

Cuando no existe, asignamos una nueva página, y la añadimos al hash de la memoria intermedia de páginas.

```
page = page_cache_alloc(); __add_to_page_cache(page, mapping, index, hash);
```

Después de que la página haya sido hashed (ordenada) utilizamos la operación `->readpage` address_space para en este instante rellenar la página con datos (el fichero es una instancia abierta del inodo).

```
error = mapping->a_ops->readpage(file, page);
```

Finalmente podemos copiar los datos al espacio de usuario.

Para escribir en el sistema de archivos existen dos formas: una para mapeos escribibles (mmap) y otra para la familia de llamadas al sistema write(2). El caso mmap es muy simple, por lo tanto será el que primero discutamos. Cuando un usuario modifica los mapas, el subsistema VM marca la página como sucia.

```
SetPageDirty(page);
```

El hilo del núcleo bdflush que está intentando liberar páginas, como actividad en segundo plano o porque no hay suficiente memoria, intentará llamar a `->writepage` en las páginas que están explicitamente marcadas como sucias. El método `->writepage` tiene ahora que escribir el contenido de las páginas de vuelta al disco y liberar la página.

El segundo camino de escritura es _mucho_ más complicado. Para cada página que el usuario escribe, tenemos básicamente que hacer lo siguiente: (para el código completo ver `mm/filemap.c:generic_file_write()`).

```
page = __grab_cache_page(mapping, index, &cached_page); mapping->a_ops->prepare_write(file,
page, offset, offset+bytes); copy_from_user(kaddr+offset, buf, bytes);
mapping->a_ops->commit_write(file, page, offset, offset+bytes);
```

Por lo tanto intentamos encontrar la página ordenada o asignar una nueva, entonces llamamos al método `->prepare_write` address_space, copiamos la antememoria del usuario a la memoria del núcleo y finalmente llamamos al método `->commit_write`. Tal como probablemente has visto ->prepare_write y ->`commit_write` son fundamentalmente diferentes de `->readpage` y `->writepage`, porque ellas no sólo son llamadas cuando la E/S física se quiere actualizar sino que son llamadas cada vez que el usuario modifica el fichero. Hay dos (¿o más?) formas para manejar esto, la primero es usar la antememoria intermedia de Linux para retrasar la E/S física, rellenando un puntero `page->buffers` con buffer_heads, que será usado en try_to_free_buffers (`fs/buffers.c`) para pedir E/S una vez que no haya suficientemente memoria, y es usada de forma muy difundida en el actual núcleo. La otra forma justamente establece la página como sucia y confía en que `->writepage` realice todo el trabajo. Debido a la carencia de un bitmap de validez en la página de estructuras, esto no realiza todo el trabajo que tiene una granularidad más pequeña que `PAGE_SIZE`.

5 Mecanismos IPC

Este capítulo describe los mecanismos IPC semáforo, memoria compartida y cola de mensajes tal como han sido implementados en el núcleo Linux 2.4. Está organizado en 4 secciones. Las tres primeras secciones cubren las interfaces y las funciones de soporte para 5.1 (semaphores), 5.2 (message queues), y 5.3 (shared memory) respectivamente. La sección 5.4 (last) describe un conjunto de funciones comunes y estructuras de datos que son compartidas por los tres mecanismos.

5.1 Semáforos

Las funciones descritas en esta sección implementan el nivel de usuario de los mecanismos de los semáforos. Nótese que esta implementación ayuda en el uso de los spinlocks y semáforos del núcleo. Para eliminar esta confusión el término "semáforo del núcleo" será usado en referencia a los semáforos del núcleo. Todos los otros usos de la palabra "semáforo" será una referencia a los semáforos del nivel de usuario.

5.1.1 Interfaces de la Llamada al sistema de los Semáforos

sys_semget() La llamada entera a sys_semget() es protegida por el semáforo global del núcleo 5.4.2 (sem_ids.sem)

En el caso donde un nuevo conjunto de semáforos deben de ser creados, la función 5.1.3 (newary()) es llamada para crear e inicializar un nuevo conjunto de semáforos. La ID del nuevo conjunto es retornada al llamante.

En el caso donde un valor de llave es suministrado por un conjunto de semáforos existentes, 5.4.1 (ipc_findkey()) es llamado para buscar el correspondiente descriptor del semáforo en el índice de la matriz. Los parámetros y los permisos del llamante son verificados antes de devolver la ID del conjunto de semáforos.

sys_semctl() Para los comandos 5.1.3 (IPC_INFO), 5.1.3 (SEM_INFO), y 5.1.3 (SEM_STAT), 5.1.3 (semctl_nolock()) es llamado para realizar las funciones necesarias.

Para los comandos 5.1.3 (GETALL), 5.1.3 (GETVAL), 5.1.3 (GETPID), 5.1.3 (GETNCNT), 5.1.3 (GETZCNT), 5.1.3 (IPC_STAT), 5.1.3 (SETVAL), y 5.1.3 (SETALL), 5.1.3 (semctl_main()) es llamado para realizar las funciones necesarias.

Para los comandos 5.1.3 (IPC_RMID) y 5.1.3 (IPC_SET), 5.1.3 (semctl_down()) es llamada para realizar las funciones necesarias. Durante todas estas operaciones, es mantenido el semáforo global del núcleo 5.4.2 (sem_ids.sem).

sys_semop() Después de validar los parámetros de la llamada, los datos de las operaciones de los semáforos son copiados desde el espacio de usuario a una antememoria temporal. Si una pequeña antememoria temporal es suficiente, entonces es usada una antememoria de pila. En otro caso, es asignad una antememoria más grande. Después de copiar los datos de las operaciones de los semáforos, el spinlock global de los semáforos es cerrado, y la ID del conjunto de semáforos especificado por el usuario es validado. Los permisos de acceso para el conjunto de semáforos también son validados.

Todas las operaciones de los semáforos especificadas por el usuario son analizadas. Durante este proceso, es mantenida una cuenta para todas las operaciones que tienen la bandera SEM_UNDO establecida. Una bandera `decrease` es establecida si alguna de las operaciones quitan de un valor del semáforo, y una bandera `alter` es establecida si alguno de los valores de los semáforos es modificado (esto es, incrementados o decrementados). El número de cada semáforos a ser modificado es validado.

Si SEM_UNDO estaba asertado para alguna de las operaciones del semáforo, entonces la lista para deshacer la actual tarea es buscada por una estructura deshacer asociada con este conjunto de semáforos. Durante esta búsqueda, si la ID del conjunto de semáforos de alguna de las estructuras deshacer es encontrada será -1, entonces 5.1.3 (freeundos()) es llamada para liberar la estructura deshacer y quitarla de la lista. Si no se encuentra ninguna estructura deshacer para este conjunto de semáforos entonces 5.1.3 (alloc_undo()) es llamada para asignar e inicializar una.

La función 5.1.3 (try_atomic_semop()) es llamada con el parámetro `do_undo` igual a 0 en orden de ejecutar la secuencia de operaciones. El valor de retorno indica si ambas operaciones han tenido éxito, han sido fallidas, o que no han sido ejecutadas porque necesitaban bloquear. Cada uno de estos casos son más ampliamente descritos a continuación:

Operaciones de semáforos no bloqueantes La función 5.1.3 (try_atomic_semop()) devuelve cero para indicar que todas las operaciones en la secuencia han sido realizadas con éxito. Es este caso, 5.1.3 (update_queue()) es llamada para recorrer la cola de las operaciones pendientes del semáforo para el conjunto del semáforo y despertar cualquier tarea dormida que no necesite bloquear más. Esto completa la ejecución de la llamada al sistema sys_semop() para este caso.

Operaciones de Semáforo con fallos Si 5.1.3 (try_atomic_semop()) devuelve un valor negativo, entonces ha sido encontrada una condición de fallo. En este caso, ninguna de las operaciones han sido ejecutadas. Esto ocurre cuando una operación de un semáforo causaría un valor inválido del semáforo, o un operación marcada como IPC_NOWAIT es incapaz de completarse. La condición de error es retornada al llamante de sys_semop().

Antes de que sys_semop() retorne, es hecha una llamada a 5.1.3 (update_queue()) para recorrer la cola de operaciones pendientes del semáforo para el conjunto del semáforo y despierta cualquier tarea dormida que no necesite más bloqueos.

Operaciones de Semáforo bloqueantes La función 5.1.3 (try_atomic_semop()) devuelve un 1 para indicar que la secuencia de operaciones del semáforo no fue ejecutada porque uno de los semáforos bloquearía. Para este caso, un nuevo elemento 5.1.2 (sem_queue) es inicializado conteniendo estas operaciones del semáforo. Si alguna de estas operaciones afectaran al estado del semáforo, entonces un nuevo elemento de cola es añadido al final de la cola. En otro caso, el nuevo elemento es añadido al principio de la cola.

El elemento `semsleeping` de la tarea actual está establecido para indicar que la tarea está durmiendo en este elemento 5.1.2 (sem_queue). La tarea actual es marcada como TASK_INTERRUPTIBLE, y el elemento `sleeper` del 5.1.2 (sem_queue) es establecido para identificar esta tarea por el durmiente. El spinlock global del semáforo es entonces desbloqueado, y schedule() es llamado para poner la tarea actual a dormir.

Cuando es despertada, la tarea vuelve a cerrar el spinlock global del semáforo, determina por qué fue despertada, y cómo debería de responder. Los siguientes casos son manejados:

- Si el conjunto de semáforos ha sido borrado, entonces la llamada al sistema falla con EIDRM.
- Si el elemento `status` de la estructura 5.1.2 (sem_queue) está establecido a 1, entonces la tarea es despertada en orden a reintentar las operaciones del semáforo, Otra llamada a 5.1.3 (try_atomic_semop()) es realizada para ejecutar la secuencia de las operaciones del semáforo. Si try_atomic_sweep() devuelve 1, entonces la tarea debe de bloquearse otra vez como se describió anteriormente. En otro caso, se devuelve 0 en caso de éxito, o un código de error apropiado en caso de fallo.

  Antes de que sys_semop() regrese, current->semsleeping es limpiado, y 5.1.2 (sem_queue) es borrado de la cola. Si alguna de las operaciones del semáforo especificada eran operaciones alteradoras (incremento o decremento), entonces 5.1.3 (update_queue()) es llamado para recorrer la cola de operaciones pendientes del semáforo para el conjunto del semáforo y despertar cualquier tarea dormida que no necesite bloquear más.
- Si el elemento `status` de la estructura 5.1.2 (sem_queue) NO está establecida a 1, y el elemento 5.1.2 (sem_queue) no ha sido quitado de la cola, entonces la tarea ha sido despertada por una interrupción. Es este caso, la llamada al sistema falla con EINTR. Antes de regresar, current->semsleeping es limpiado, y 5.1.2 (sem_queue) es borrado de la cola. También 5.1.3 (update_queue()) es llamado si alguna de las operaciones eran operaciones alterantes.

- Si el elemento `status` de la estructura 5.1.2 (sem_queue) NO está establecido a 1, y el elemento 5.1.2 (sem_queue) ha sido quitado de la cola, entonces las operaciones del semáforo ya han sido ejecutadas por 5.1.3 (update_queue()). La cola `status`, la cual será 0 si se tiene éxito o un código negativo de error en caso de fallo, se convertirá en el valor de retorno de la llamada al sistema.

5.1.2 Estructuras Específicas de Soporte de Semáforos

Las siguientes estructuras son usadas específicamente para el soporte de semáforos:

struct sem_array

```
/* Una estructura de datos sem_array para cada conjunto de semáforos en el sistema. */
struct sem_array {
    struct kern_ipc_perm sem_perm; /* permisos .. ver ipc.h */
    time_t sem_otime; /* último tiempo de la operación con el semáforo */
    time_t sem_ctime; /* último tiempo de cambio */
    struct sem *sem_base; /* puntero al primer semáforo en el array */
    struct sem_queue *sem_pending; /* operaciones pendientes para ser procesadas  */
    struct sem_queue **sem_pending_last; /* última operación pendiente */
    struct sem_undo *undo; /* peticiones deshechas en este array * /
    unsigned long sem_nsems; /* número de semáforos en el array */
};
```

struct sem

```
/* Una estructura de semáforo para cada semáforo en el sistema. */
struct sem {
        int     semval;         /* valor actual */
        int     sempid;         /* pid de la última operación */
};
```

struct seminfo

```
struct  seminfo {
        int semmap;
        int semmni;
        int semmns;
        int semmnu;
        int semmsl;
        int semopm;
        int semume;
        int semusz;
        int semvmx;
        int semaem;
};
```

struct semid64_ds

```
struct semid64_ds {
        struct ipc64_perm sem_perm;             /* permisos.. ver ipc.h */
        __kernel_time_t sem_otime;              /* último tiempo de operación con el semáforo */
        unsigned long   __unused1;
        __kernel_time_t sem_ctime;              /* último tiempo de cambio */
        unsigned long   __unused2;
```

```
        unsigned long   sem_nsems;              /* número de semáforos en la matriz */
        unsigned long   __unused3;
        unsigned long   __unused4;
};
```

struct sem_queue

```
/* Una cola para cada proceso durmiendo en el sistema. */
struct sem_queue {
        struct sem_queue *      next;    /* siguiente entrada en la cola */
        struct sem_queue **     prev;    /* entrada anterior en la cola, *(q->prev) == q */
        struct task_struct*     sleeper; /* este proceso */
        struct sem_undo *       undo;    /* estructura deshacer */
        int                     pid;     /* id del proceso del proceso pedido */
        int                     status;  /* status de terminación de la operación  */
        struct sem_array *      sma;     /* matriz de semáforos para las operaciones */
        int                     id;      /* id interna del semáforo */
        struct sembuf *         sops;    /* matriz de operaciones pendientes*/
        int                     nsops;   /* número de operaciones */
        int                     alter;   /* operaciones que alterarán el semáforo  */
};
```

struct sembuf

```
/* las llamadas al sistema cogen una matriz de estas. */
struct sembuf {
        unsigned short  sem_num;        /* indice del semáforo en la matriz */
        short           sem_op;         /* operación del semáforo */
        short           sem_flg;        /* banderas de la operación */
};
```

struct sem_undo

```
/* Cada tarea tiene una lista de peticiones de deshacer. Ellas son
 * ejecutadas cuando el proceso sale.
 */
struct sem_undo {
        struct sem_undo *       proc_next;      /* siguiente entrada en este proceso */
        struct sem_undo *       id_next;        /* siguiente entrada en
                                                este conjunto de semáforos */
        int                     semid;          /* identificador del
                                                conjunto de semáforos */
        short *                 semadj;         /* matriz de ajustes, una
                                                por semáforo */

};
```

5.1.3 Funciones de Soporte de Semáforos

Las siguientes funciones son usadas específicamente para soportar los semáforos:

newary() newary() confía en la función 5.4.1 (ipc_alloc()) para asignar la memoria requerida para el nuevo conjunto de semáforos. El asigna suficiente memoria para el conjunto de descriptores del semáforo y para cada uno de los semáforos en el conjunto. La memoria asignada es limpiada, y la dirección del primer elemento del conjunto de descriptores del semáforo es pasada a 5.4.1 (ipc_addid()). 5.4.1 (ipc_addid()) reserva una entrada de la matriz para el conjunto de descriptores del semáforo e inicializa los datos (5.4.2 (struct kern_ipc_perm)) para el conjunto. La variavle global `used_sems` es actualizada por el número de semáforos en el nuevo conjunto y la inicialización de los datos (5.4.2 (struct kern_ipc_perm)) para el nuevo conjunto es completada. Otras inicializaciones realizadas para este conjunto son listadas a continuación:

- El elemento `sem_base` para el conjunto es inicializado a la dirección inmediatamente siguiente siguiendo la porción (5.1.2 (struct sem_array)) de los nuevos segmentos asignados. Esto corresponde a la localización del primer semáforon en el conjunto.
- La cola `sem_pending` es inicializada como vacía.

Todas las operaciones siguiendo la llamada a 5.4.1 (ipc_addid()) son realizadas mientras se mantiene el spinlock global de los semáforos. Después de desbloquear el spinlock global de los semáforos, newary() llama a 5.4.1 (ipc_buildid()) (a través de sem_buildid()). Esta función usa el índice del conjunto de descriptores del semáforo para crear una única ID, que es entonces devuelta al llamador de newary().

freeary() freeary() es llamada por 5.1.3 (semctl_down()) para realizar las funciones listadas a continuación. Es llamada con el spinlock global de los semáforos bloqueado y regresa con el spinlock desbloqueado.

- La función 5.4.1 (ipc_rmid()) es llamada (a través del envoltorio sem_rmid()) para borrar la ID del conjunto de semáforos y para recuperar un puntero al conjunto de semáforos.
- La lista de deshacer para el conjunto de semáforos es invalidada.
- Todos los procesos pendientes son despertados y son obligados a fallar con EIDRM.
- EL número de semáforos usados es reducido con el número de semáforos en el conjunto borrado.
- La memoria asociada con el conjunto de semáforos es liberada.

semctl_down() semctcl_down() suministra las operaciones 5.1.3 (IPC_RMID) y 5.1.3 (IPC_SET) de la llamada al sistema semctl(). La ID del conjunto de semáforos y los permisos de acceso son verificadas en ambas operaciones, y en ambos casos, el spinlock global del semáforo es mantenido a lo largo de la operación.

IPC_RMID La operación IPC_RMID llama a 5.1.3 (freeary()) para borrar el conjunto del semáforo.

IPC_SET La operación IPC_SET actualiza los elementos `uid`, `gid`, `mode`, y `ctime` del conjunto de semáforos.

semctl_nolock() semctl_nolock() es llamada por 5.1.1 (sys_semctl()) para realizar las operaciones IPC_INFO, SEM_INFO y SEM_STAT.

IPC_INFO y SEM_INFO IPC_INFO y SEM_INFO causan una antememoria temporal 5.1.2 (seminfo) para que sea inicializada y cargada con los datos estadísticos sin cambiar del semáforo, los elementos `semusz` y `semaem` de la estructura 5.1.2 (seminfo) son actualizados de acuerdo con el comando dado (IPC_INFO o SEM_INFO). El valor de retorno de las llamadas al sistema es establecido al conjunto máximo de IDs del conjunto de semáforos.

SEM_STAT SEM_STAT causa la inicialización de la antememoria temporal 5.1.2 (semid64_ds). El spinlock global del semáforo es entonces mantenido mientras se copian los valores `sem_otime`, `sem_ctime`, y `sem_nsems` en la antememoria. Estos datos son entonces copiados al espacio de usuario.

semctl_main() semctl_main() es llamado por 5.1.1 (sys_semctl()) para realizar muchas de las funciones soportadas, tal como se describe en la sección posterior. Anteriormente a realizar alguna de las siguientes operaciones, semctl_main() cierra el spinlock global del semáforo y valida la ID del conjunto de semáforos y los permisos. El spinlock es liberado antes de retornar.

GETALL La operación GETALL carga los actuales valores del semáforo en una antememoria temporal del núcleo y entonces los copia fuera del espacio de usuario. La pequeña pila de antememoria es usada si el conjunto del semáforo es pequeño. En otro caso, el spinlock es temporalmente deshechado en orden de asignar una antememoria más grande. El spinlock es mantenido mientras se copian los valores del semáforo en la antememoria temporal.

SETALL La operación SETALL copia los valores del semáforo desde el espacio de usuario en una antememoria temporal, y entonces en el conjunto del semáforo. El spinlock es quitado mientras se copian los valores desde el espacio de usuario a la antememoria temporal, y mientras se verifican valores razonables. Si el conjunto del semáforo es pequeño, entonces una pila de antememoria es usada, en otro caso una antememoria más grande es asignado. El spinlock es recuperado y mantenido mientras las siguientes operaciones son realizadas en el conjunto del semáforo:

- Los valores del semáforo son copiados en el conjunto del semáforo.
- Los ajustes del semáforon de la cola de deshacer para el conjunto del semáforo son limpiados.
- El valor `sem_ctime` para el conjunto de semáforos es establecido.
- La función 5.1.3 (update_queue()) es llamada para recorrer la cola de semops (operaciones del semáforo) pendientes y mirar por alguna tarea que pueda ser completada como un resultado de la operación SETALL. Cualquier tarea pendiente que no sea más bloqueada es despertada.

IPC_STAT En la operación IPC_STAT, los valores `sem_otime`, `sem_ctime`, y `sem_nsems` son copiados en una pila de antememoria. Los datos son entonces copiados al espacio de usuario después de tirar con el spinlock.

GETVAL Para GETVALL en el caso de no error, el valor de retorno para la llamada al sistema es establecido al valor del semáforo especificado.

GETPID Para GETPID en el caso de no error, el valor de retorno para la llamada al sistema es establecido al `pid` asociado con las última operación del semáforo.

GETNCNT Para GETNCNT en el caso de no error, el valor de retorno para la llamada al sistema de establecido al número de procesos esperando en el semáforo siendo menor que cero. Este número es calculado por la función 5.1.3 (count_semncnt()).

GETZCNT Para GETZCNT en la caso de no error, el valor de retorno para la llamada al sistema es establecido al número de procesos esperando en el semáforo estando establecido a cero. Este número es calculado por la función 5.1.3 (count_semzcnt()).

SETVAL Después de validar el nuevo valor del semáforo, las siguientes funciones son realizadas:

- La cola de deshacer es buscada para cualquier ajuste en este semáforo. Cualquier ajuste que sea encontrado es reinicializado a cero.
- El valor del semáforo es establecido al valor suministrado.
- El valor del semáforo `sem_ctime` para el conjunto del semáforo es actualizado.
- La función 5.1.3 (update_queue()) es llamada para recorrer la cola de semops (operaciones del semáforo) pendientes y buscar a cualquier tarea que pueda ser completada como resultado de la operación 5.1.3 (SETALL). Cualquier tarea que no vaya a ser más bloqueada es despertada.

count_semncnt() count_semncnt() cuenta el número de tareas esperando por el valor del semáforo para que sea menor que cero.

count_semzcnt() count_semzcnt() cuenta el número de tareas esperando por el valor del semáforo para que sea cero.

update_queue() update_queue() recorre la cola de semops pendientes para un conjunto de un semáforo y llama a 5.1.3 (try_atomic_semop()) para determinar qué secuencias de las operaciones de los semáforos serán realizadas. Si el estado de la cola de elementos indica que las tareas bloqueadas ya han sido despertadas, entonces la cola de elementos es pasada por alto. Para los otros elementos de la cola, la bandera `q-alter` es pasada como el parámetro deshacer a 5.1.3 (try_atomic_semop()), indicando que cualquier operación alterante debería de ser deshecha antes de retornar.

Si la secuencia de operaciones bloquearan, entonces update_queue() retornará sin hacer ningún cambio.

Una secuencia de operaciones puede fallar si una de las operaciones de los semáforos puede causar un valor inválido del semáforo, o una operación marcada como IPC_NOWAIT es incapaz de completarse. En este caso, la tarea que es bloqueada en la secuencia de las operaciones del semáforo es despertada, y la cola de status es establecida con un código de error apropiado. El elemento de la cola es también quitado de la cola.

Si la secuencia de las operaciones no es alterante, entonces ellas deberían de pasar un valor cero como parámetro deshacer a 5.1.3 (try_atomic_semop()). Si estas operaciones tienen éxito, entonces son consideradas completas y son borradas de la cola. La tarea bloqueada es despertada, y el elemento de la cola `status` es establecido para indicar el éxito.

Si la secuencia de las operaciones pueden alterar los valores del semáforo, pero puede tener éxito, entonces las tareas durmiendo que no necesiten ser más bloqueadas tienen que ser despertadas. La cola status es establecida a 1 para indicar que la tarea bloqueada ha sido despertada. Las operaciones no han sido realizadas, por lo tanto el elemento de la cola no es quitado de la cola. Las operaciones del semáforo serán ejecutadas por la tarea despertada.

try_atomic_semop() try_atomic_semop() es llamada por 5.1.1 (sys_semop()) y 5.1.3 (update_queue()) para determinar si una secuencia de operaciones del semáforo tendrán éxito. El determina esto intentando realizar cada una de las operaciones.

Si una operación bloqueante es encontrada, entonces el proceso es abortado y todas los operaciones son deshechas. -EAGAIN es devuelto si IPC_NOWAIT es establecido. En otro caso, es devuelto 1 para indicar que la secuencia de las operaciones del semáforo está bloqueada.

Si un valor del semáforo es ajustado más alla de los límites del sistema, entonces todas las operaciones son deshechas, y -ERANGE es retornado.

Si todas las operaciones de la secuencia tienen éxito, y el parámetro `do_undo` no es cero, entonces todas las operaciones son deshechas, y 0 es devuelto. Si el parámetro `do_undo` es cero, entonces todas las operaciones tienen éxito y continúan obligadas, y el `sem_otime`, campo del conjunto de semáforos es actualizado.

sem_revalidate() sem_revalidate() es llamado cuando el spinlock global del semáforo ha sido temporalmente tirado y necesita ser bloqueado otra vez. Es llamado por 5.1.3 (semctl_main()) y 5.1.3 (alloc_undo()). Valida la ID del semáforo y los permisos, y si tiene éxito retorna con el spinlock global de los semáforos bloqueado.

freeundos() freeundos() recorre las lista de procesos por deshacer en busca de la estructura deshacer deseada. Si es encontrada, la estructura deshacer es quitada de la lista y liberada. Un puntero a la siguiente estructura deshacer en la lista de procesos es devuelta.

alloc_undo() alloc_undo() espera ser llamada con el spinlock global de los semáforos cerrado. En el caso de un error, regresa con él desbloqueado.

El spinlock global de los semáforos es desbloqueado, y kmallock() es llamado para asignar suficiente memoria para la estructura 5.1.2 (sem_undo), y también para un array de uno de los valores de ajuste para cada semáforo en el conjunto. Si tiene éxito, el spinlock es recuperado con una llamada a 5.1.3 (sem_revalidate()).

La nueva estructura semundo es entonces inicializada, y la dirección de esta estructura es colocada en la dirección suministrada por el llamante. La nueva estructura deshacer es entonces colocada en la cabeza de la lista deshacer para la actual tarea.

sem_exit() sem_exit() es llamada por do_exit(), y es la responsable de ejecutar todos los ajustes deshacer para la tarea saliente.

Si el actual proceso fue bloqueado en un semáforo, entonces es borrado desde la lista 5.1.2 (sem_queue) mientras se mantiene el spinlock global de los semáforos.

La lista deshacer para la actual tarea es entonces recorrida, y las siguientes operaciones son realizadas mientras se mantienen y liberan los spinlocks globales de los semáforos a lo largo del procesamiento de cada elemento de la lista. Las siguientes operaciones son realizadas para cada uno de los elementos deshacer:

- La estructura deshacer y la ID del conjunto del semáforo son validadas.
- La lista deshacer del correspondiente conjunto de semáforos es buscada para encontrar una referencia a la misma estructura deshacer y para quitarla de esa lista.
- Los ajustes indicadores en la estructura deshacer son aplicados al conjunto de semáforos.
- El parámetro `sem_otime` del conjunto de semáforos es actualizado.
- 5.1.3 (update_queue()) es llamado para recorrer la cola de las semops pendientes y despertar cualquier tarea durmiento que no necesite ser bloqueada como resultado de la operación deshacer.
- La estructura deshacer es liberada.

Cuando el procesamiento de la lista está completo, el valor current->semundo es limpiado.

5.2 Colas de Mensajes

5.2.1 Interfaces de las llamadas al sistema de Colas

sys_msgget() La llamada entera a sys_msgget() es protegida por el semáforo global de la cola de mensajes (5.4.2 (msg_ids.sem)).

En el caso donde una nueva cola de mensajes tiene que ser creada, la función 5.2.3 (newque()) es llamada para crear e inicializar una nueva cola de mensajes, y la nueva ID de la cola es devuelta al llamante.

Si un valor llave es suministrado para una cola de mensajes existente, entonces 5.4.1 (ipc_findkey()) es llamada para mirar el índice correspondiente en la matriz de colas globales de descriptores de mensajes (msg_ids.entries). Los parámetros y los permisos del llamante son verificados antes de devolver la ID de la cola de mensajes. Las operaciónes de búsqueda y verificación son realizadas mientras el spinlock global de la cola de mensajes (msg_ids.ary) es mantenido.

sys_msgctl() Los parámetros pasados a sys_msgctl() son: una ID de una cola de mensajes (`msqid`), la operación (`cmd`), y un puntero al espacio de la antememoria 5.2.2 (msgid_ds) del tipo (`buf`). Seis operaciones son suministradas en esta función: IPC_INFO, MSG_INFO,IPC_STAT, MSG_STAT, IPC_SET y IPC_RMID. La ID de la cola de mensajes y los parámetros de la operación son validados; entonces, la operación (cmd) es realizada como sigue:

IPC_INFO (o MSG_INFO) La información de la cola global de mensajes es copiada al espacio de usuario.

IPC_STAT (o MSG_STAT) Una antememoria temporal del tipo 5.2.2 (struct msqid64_ds) es inicializado y el spinlock de la cola de mensajes global es cerrado. Después de verificar los permisos de acceso del proceso llamante, la información de la cola de mensajes asociada con la ID de la cola de mensajes es cargada en una antememoria temporal, el spinlock de la cola de mensajes global es abierto, y los contenidos de la antememoria temporal son copiados fuera del espacio de usuario por 5.2.3 (copy_msqid_to_user()).

IPC_SET Los datos del usuario son copiados a través de 5.2.3 (copy_msqid_to_user()). El semáforo de la cola de mensajes global y el spinlock son obtenidos y liberados al final. Después de que la ID de la cola de mensajes y los permisos de acceso del actual proceso hayan sido validados, la información de la cola de mensajes es actualizada con los datos suministrados por el usuario, Después, 5.2.3 (expunge_all()) y 5.2.3 (ss_wakeup()) son llamadas para despertar todos los procesos durmiendo en las colas de espera del emisor y del receptor de las colas de mensajes. Esto es el motivo por el que algunos receptores quizás sean ahora excluidos por permisos de acceso estrictos y alguno de los emisores sean capaces ahora de enviar el mensaje debido a un incremento del tamaño de la cola.

IPC_RMID El semáforo de la cola de mensajes global es obtenido y el spinlock global de la cola de mensajes es cerrado. Después de validar la ID de la cola de mensajes y los permisos de acceso de la actual tarea, 5.2.3 (freeque()) es llamado para liberar los recursos relacionados con la ID de la cola de mensajes. El semáforo de la cola de mensajes global y el spinlock son liberados.

sys_msgsnd() sys_msgsnd() recibe como parámetros una ID de una cola de mensajes (`msqid`), un puntero a la antememoria del tipo 5.2.2 (struct msg_msg) (`msgp`), el tamaño del mensaje a ser enviado (`msgsz`), y una bandera indicando esperar o no esperar (`msgflg`). Hay dos tareas esperando las colas y un mensaje esperando la cola asociada con la ID de la cola de mensajes. Si hay una nueva tarea en la cola de espera del receptor que está esperando por este mensaje, entonces el mensaje es enviado directamente al receptor. En otro caso, si hay suficiente espacio disponible en la cola de espera de mensajes, el mensaje es guardado en esta cola. Como último recurso, la tarea emisora se encola a si misma en la cola de espera del emisor. Una discusión con más profundidad de las operaciones realizadas por sys_msgsnd() es la siguiente:

1. Valida la dirección de la antememoria del usuario y el tipo de mensaje, entonces invoca a 5.2.3 (load_msg()) para cargar el contenido del mensaje del usuario en un objeto temporal `msg` del tipo 5.2.2 (struct msg_msg). El tipo de mensaje y los campos del tamaño del mensaje de `msg` también son inicializados.
2. Cierra el spinlock global de la cola de mensajes y coge el descriptor asociado con la cola de mensajes con la ID de la cola de mensajes. Si dicha cola de mensajes no existe, retorna EINVAL.
3. Invoca a 5.4.1 (ipc_checkid()) (a través de msg_checkid()) para verificar que la ID de la cola de mensajes es válida y llama a 5.4.1 (ipcperms()) para chequear los permisos de acceso de proceso llamante.
4. Chequea el tamaño del mensaje y el espacio que sobra en la cola de espera de mensajes para ver si hay suficiente espacio para almacenar el mensaje. Si no, los siguiens subpasos son realizados:
   (a) Si IPC_NOWAIT es especificado en `msgflg` el spinlock global de la cola de mensajes es abierto, los recursos de memoria para el mensaje son liberados, y EAGAIN es retornado.
   (b) Invoca a 5.2.3 (ss_add()) para encolar la actual tarea en la cola de espera del emisor. También abre al spinlock global de la cola de mensajes e invoca a schedule() para poner la actual tarea a dormir.
   (c) Cuando es despertado, obtiene el spinlock global otra vez y verifica que la ID de la cola de mensajes es todavía válida. Si la ID de la cola de mensajes no es valida, ERMID es retornado.
   (d) Invoca 5.2.3 (ss_del()) para quitar la tarea emisora de la cola de espera del emisor. Si hay alguna señal pendiente para la tarea, sys_msgsnd() abre el spinlock global, invoca a 5.2.3 (free_msg()) para liberar la antememoria del mensaje, y retorna EINTR. En otro caso, la función va a 3 (back) para chequear otra vez cuando hay otra vez suficiente sitio en la cola de espera de mensajes.
5. Invoca 5.2.3 (pipelined_send()) para intentar enviar el mensaje a la cola receptora directamente.
6. Si no hay receptores esperando por este mensaje, desencola `msg` en la cola de mensajes esperando (msq->q_messages). Actualiza los campos `q_cbytes` y `q_qnum` del descriptor de la cola de mensajes, al igual que las variables globales `msg_bytes` y `msg_hdrs`, las cuales indican el número de bytes total usados por los mensajes y el número total de mensajes a lo largo del sistema.
7. Si el mensaje ha sido enviado con éxito o encolado, actualiza los campos `q_lspid` y `q_stime` del descriptor de la cola de mensajes y abre el spinlock de colas de mensajes global.

sys_msgrcv() La función sys_msgrcv() recibe como parámetro una ID de una cola de mensajes (`msqid`), un puntero a una antememoria del tipo 5.2.2 (msg_msg) (`msgp`), el tamaño del mensaje deseado (`msgsz`), el tipo de mensaje (`msgtyp`), y las banderas (`msgflg`). Busca las colas de mensajes esperando asociadas con la ID de la cola de mensajes, encuentra el primer mensaje en la cola, la cual comprueba el tipo pedido y lo copia en la antememoria del usuario dado. Si tal mensaje no es encontrado en la cola de mensajes esperando, la tarea pedida es encolada en la cola de receptores esperando hasta que el mensaje deseado está disponible. Una discución más profunda de las operaciones realizadas por sys_msgrcv() es la que sigue:

1. Primero, invoca a 5.2.3 (convert_mode()) para derivar el modo de búsqueda desde `msgtyp`. sys_msgrcv() entonces cierra el spinlock global de la cola de mensajes y obtiene el descriptor de la cola de mensajes asociado con la ID de la cola de mensajes. Si tal cola de mensajes no existe, retorna EINVAL.

2. Chequea cuando la tarea actual tiene los permisos correctos para acceder a la cola de mensajes.
3. Empezando desde el primer mensaje en la cola de mensajes de espera, invoca a 5.2.3 (testmsg()) para chequear cuando el tipo del mensaje se empareja con el tipo requerido. sys_msgrcv() continúa buscando hasta que un mensaje emparejado es encontrado o la cola de espera entera está exausta. Si el modo de búsqueda es SEARCH_LESSEQUAL, entonces es buscado el primer mensaje en la cola con un tipo más bajo o igual a `msgtyp`.
4. Si un mensaje en encontrado, sys_msgrcv() realiza los siguientes subpasos:
   (a) Si el tamaño del mensaje es más grande que el tamaño deseado y `msgflg` indica que no se permiten errores, abre el spinlock de cola de mensajes global y retorna E2BIG.
   (b) Quita el mensaje de la cola de mensajes esperando y actualiza las estadísticas de la cola de mensajes.
   (c) Despierta todas las tareas durmiendo en las colas de espera de los emisores. El borrado de un mensaje de la cola en los pasos previos hace posible que uno de los emisores progresen. Va a través de 10 (last step).
5. Si no hay mensajes emparejados el criterio del receptor es encontrado en la cola de mensajes esperando, entonces `msgflg` es chequeado. Si IPC_NOWAIT está establecido, entonces el spinlock global de la cola de mensajes es abierto y ENOMSG es retornado. En otro caso, el receptor es encolado en la cola de espera del receptor como sigue:
   (a) Una estructura de datos 5.2.2 (msg_receiver) `msr` es asignada y es añadida en la cabeza de la cola de espera.
   (b) El campo `r_tsk` de `msr` es establecido a la tarea actual.
   (c) Los campos `r_msgtype` y `r_mode` son inicializados con el tipo y modo del mensaje deseado respectivamente.
   (d) Si `msgflg` indica MSG_NOERROR, entonces el campo r_maxsize de `msr` es establecido para ser el valor de `msgsz`. En otro caso es establecido para ser INT_MAX.
   (e) El campo `r_msg` es inicializado para indicar que todavía no ha sido recibido el mensaje.
   (f) Después de que la inicialización está completa, el estado de la tarea receptora es establecido a TASK_INTERRUPTIBLE, el spinlock global de colas de mensajes es abierto, y schedule() es invocado.
6. Después de que el receptor es despertado, el campo `r_msg` de `msr` es chequeado. Este campo es usado para almacenar el mensaje entubado o, en el caso de un error, almacenar el estado del error. Si el campo `r_msg` es rellenado con el mensaje deseado, entonces va a 10 (last step). En otro caso, el spinlock global de colas de mensajes es cerrado otra vez.
7. Después de obtener el spinlock, el campo `r_msg` es re-chequeado para ver si el mensaje fue recibido mientras se estaba esperando por el spinlock. Si el mensaje ha sido recibido, ocurre el 10 (last step).
8. Si el campo `r_msg` todavía está sin cambiar, entonces la tarea tiene que ser despertada en orden de reintentarlo. En este caso, `msr` es quitado de la cola. Si hay una señal pendiente para la tarea, entonces el spinlock de la cola de mensajes global es abierto y EINTR es retornado. En otro caso, la función necesita ir a 3 (back) y reintentarlo.
9. Si el campo `r_msg` muestra que ha ocurrido un error mientras estaba durmiendo, el spinlock de la cola de mensajes global es abierto y el error es devuelto.
10. Después de validar que la dirección de la antememoria del usuario `msp` es válida, el tipo de mensaje es cargado en el campo `mtype` de `msp`, y 5.2.3 (store_msg()) es invocado para copiar el contenido del mensaje al campo de `msp`. Finalmente la memoria para el mensaje es liberada por la función 5.2.3 (free_msg()).

5.2.2 Estructuras Específicas de Mensajes

Las estructuras de datos para las colas de mensajes están definidas en msg.c.

struct msg_queue

```
/* una estructura msq_queue para cada cola presente en el sistema */
struct msg_queue {
        struct kern_ipc_perm q_perm;
        time_t q_stime;                 /* último tiempo del mensaje enviado */
        time_t q_rtime;                 /* último tiempo del mensaje recibido */
        time_t q_ctime;                 /* útimo tiempo de cambio */
        unsigned long q_cbytes;         /* número actual de bytes en la cola */
        unsigned long q_qnum;           /* número de mensajes en la cola */
        unsigned long q_qbytes;         /* máximo número de bytes en la cola */
        pid_t q_lspid;                  /* último pid del mensaje recibido */
        pid_t q_lrpid;                  /* último pid del mensaje recibido */

        struct list_head q_messages;
        struct list_head q_receivers;
        struct list_head q_senders;
};
```

struct msg_msg

```
/* una estructura msg_msg para cada mensaje */
struct msg_msg {
        struct list_head m_list;
        long  m_type;
        int m_ts;           /* tamaño del mensaje de texto */
        struct msg_msgseg* next;
        /* el mensaje actual sigue inmediatamente */
};
```

struct msg_msgseg

```
/* segmento de mensaje para cada mensaje */
struct msg_msgseg {
        struct msg_msgseg* next;
        /* la siguiente parte del mensaje sigue inmediatamente */
};
```

struct msg_sender

```
/* un msg_sender para cada emisor durmiendo */
struct msg_sender {
        struct list_head list;
        struct task_struct* tsk;
};
```

struct msg_receiver

```
/* una estructura msg_receiver para cada receptor durmiendo */
struct msg_receiver {
        struct list_head r_list;
```

```
        struct task_struct* r_tsk;

        int r_mode;
        long r_msgtype;
        long r_maxsize;

        struct msg_msg* volatile r_msg;
};
```

struct msqid64_ds

```
struct msqid64_ds {
        struct ipc64_perm msg_perm;
        __kernel_time_t msg_stime;      /* último tiempo del mensaje enviado */
        unsigned long   __unused1;
        __kernel_time_t msg_rtime;      /* último tiempo del mensaje recibido */
        unsigned long   __unused2;
        __kernel_time_t msg_ctime;      /* último tiempo de cambio */
        unsigned long   __unused3;
        unsigned long  msg_cbytes;      /* número actual de bytes en la cola */
        unsigned long  msg_qnum;        /* número de mensajes en la cola */
        unsigned long  msg_qbytes;      /* número máximo de bytes en la cola */
        __kernel_pid_t msg_lspid;       /* pid del último mensaje enviado */
        __kernel_pid_t msg_lrpid;       /* pid del último mensaje recibido */
        unsigned long  __unused4;
        unsigned long  __unused5;
};
```

struct msqid_ds

```
 struct msqid_ds {
        struct ipc_perm msg_perm;
        struct msg *msg_first;          /* primer mensaje en la cola, no usado */
        struct msg *msg_last;           /* último mensaje en la cola, no usado */
        __kernel_time_t msg_stime;      /* último tiempo del mensaje enviado */
        __kernel_time_t msg_rtime;      /* último tiempo del mensaje recibido */
        __kernel_time_t msg_ctime;      /* último tiempo de cambio */
        unsigned long  msg_lcbytes;     /* reusar los campos borrados para 32 bit */
        unsigned long  msg_lqbytes;     /* idem */
        unsigned short msg_cbytes;      /* número actual de bytes en la cola */
        unsigned short msg_qnum;        /* número de mensajes en la cola */
        unsigned short msg_qbytes;      /* número máximo de bytes en la cola */
        __kernel_ipc_pid_t msg_lspid;   /* último pid del mensaje enviado */
        __kernel_ipc_pid_t msg_lrpid;   /* último pid del mensaje recibido */
};
```

msg_setbuf

```
struct msq_setbuf {
        unsigned long   qbytes;
        uid_t           uid;
        gid_t           gid;
        mode_t          mode;
};
```

5.2.3 Funciones de Soporte de Mensajes

newque() newqueue() asigna la memoria para un nuevo descriptor de una cola de mensajes (5.2.2 (struct msg_queue)) y entonces llama a 5.4.1 (ipc_addid()), la cual reserva una entrada de la matriz de colas de mensaje para el nuevo descriptor de cola de mensajes. El descriptor de cola de mensajes es inicializado como sigue:

- La estructura 5.4.2 (kern_ipc_perm) es inicializada.
- Los campos `q_stime` y `q_rtime` del descriptor de cola de mensajes son inicializados a 0. El campo `q_ctime` es establecido a CURRENT_TIME.
- El número máximo de bytes permitidos en esta cola de mensajes (`q_qbytes`) es establecida a MSGMNB, y el número de bytes actualmente usados por la cola (`q_cbytes`) es inicializada a 0.
- La cola de mensajes esperando (`q_messages`), la cola de receptores esperando (`q_receivers`), y la cola de emisores esperando (`q_senders`) son inicializadas como vacías.

Todas las operaciones siguiendo la llamada a 5.4.1 (ipc_addid()) son realizadas mientras se mantiente el spinlock global de cola de mensajes. Después de abrir el spinlock, newque() llama a msg_buildid(), que mapea directamente a 5.4.1 (ipc_buildid()). 5.4.1 (ipc_buildid()) usa el índice del descriptor de cola de mensajes para crear una única ID de cola de mensaje que es entonces retornada al llamante de newque().

freeque() Cuando una cola de mensajes va a ser borrada, la función freeque() es llamada. Esta función asume que el spinlock global de la cola de mensajes ya está cerrado por la función llamante. Libera todos los recursos del núcleo asociados con esta cola de mensajes. Primero, llama a 5.4.1 (ipc_rmid()) (a través de msg_rmid()) para borrar el descriptor de cola de mensajes del array de descriptores de cola de mensajes global. Entonces llama a 5.2.3 (expunge_all) para despertar a todos los receptores durmiendo en esta cola de mensajes. Posteriormente el spinlock global de la cola de mensajes es liberado. Todos los mensajes almacenados en esta cola de mensajes son liberados y la memoria para los descriptores de cola son liberados.

ss_wakeup() ss_wakeup() despierta todas las tareas en la cola de mensajes del emisor dado. Si esta función es llamada por 5.2.3 (freeque()), entonces todos los emisores en la cola son quitados de ella.

ss_add() ss_add() recibe como parámetro un descriptor de cola de mensajes y un mensaje de estructura de datos del emisor. Rellena el campo `tsk` del mensaje de la estructura de datos del emisor con el proceso actual, cambia el estado del proceso actual a TASK_INTERRUPTIBLE, entonces inserta el mensaje de la estructura de datos del emisor a la cabeza de la cola de emisores esperando la cola de mensajes dada.

ss_del() Si el mensaje de la estrutura de datos del emisor dado (`mss`) aún está en la cola de espera del emisor asociado, entonces ss_del() quita `mss` de la cola.

expunge_all() expunge_all() recibe como parámetros un descriptor de la cola de mensajes (`msq`) y un valor entero (`res`) indicando el motivo para despertar a los receptores. Para cada receptor durmiendo asociado con `msq`, el campo `r_msg` es establecido para indicar el motivo para despertar (`res`), y la tarea asociada recibiendo es despertada. Esta función es llamada cuando una cola de mensajes es quitada o un operación de control de mensajes ha sido realizada.

load_msg() Cuando un proceso envía un mensaje, la función 5.2.1 (sys_msgsnd()) primero invoca a la función load_msg() para cargar el mensaje desde al espacio de usuario al espacio del núcleo. El mensaje es representado en la memoria del núcleo como una lista enlazada de bloques de datos. Asociado con el primer bloque de datos está una estructura 5.2.2 (msg_msg) que describe el mensaje completo. El bloque de datos asociado con la estructura msg_msg está limitado por el tamaño de DATA_MSG_LEN. El bloque de datos y la estructura son asignados en un bloque contiguo de memoria que puede ser tan grande como un página en memoria. Si el mensaje total no se ajusta en este primer bloque de datos, entonces bloques de datos adicionales son asignados y son reorganizados en una lista enlazada. Estos bloques de datos están limitados por un tamaño de DATA_SEG_LEN, y cada uno incluye una estructura 5.2.2 (msg_msgseg)). La estructura msg_msgseg y los bloques de datos asociados son asignados en un bloque de memoria contigua que puede ser tan grande como una página en memoria. Esta función retorna la dirección de la nueva estructura 5.2.2 (msg_msg) si es que tiene éxito.

store_msg() La función store_msg() es llamada por 5.2.1 (sys_msgrcv()) para reensamblar un mensaje recibido en la antememoria del espacio de usuario suministrado por el llamante. Los datos descritos por la estructura 5.2.2 (msg_msg) y cualquier estructura 5.2.2 (msg_msgseg) son secuencialmente copiados a la antememoria del espacio de usuario.

free_msg() La función free_msg() libera la memoria para una estructura de datos de mensaje 5.2.2 (msg_msg), y los segmentos del mensaje.

convert_mode() convert_mode() es llamada por 5.2.1 (sys_msgrcv()). Recibe como parámetros las direcciones del tipo del mensaje especificado (`msgtyp`) y una bandera (`msgflg`). Devuelve el modo de búsqueda al llamante basado en el valor de `msgtyp` y `msgflg`. Si `msgtyp` es null (cero), entonces SEARCH_ANY es devuelto, Si `msgtyp` es menor que 0, entonces `msgtyp` es establecido a su valor absoluto y SEARCH_LESSEQUAL es retornado. Si MSG_EXCEPT es especificado en `msgflg`, entonces SEARCH_NOTEQUAL es retornado. En otro caso SEARCH_EQUAL es retornado.

testmsg() La función testmsg() chequea cuando un mensaje conoce el criterio especificado por el receptor. Devuelve 1 si una de las siguientes condiciones es verdad:

- El modo de búsqueda indica buscar cualquier mensaje (SEARCH_ANY).
- El modo de búsqueda es SEARCH_LESSEQUAL y el tipo de mensaje es menor o igual que el tipo deseado.
- El modo de búsqueda es SEARCH_EQUAL y el tipo de mensaje es el mismo que el tipo deseado.
- El modo de búsqueda es SEARCH_NOTEQUAL y el tipo de mensajes no es igual al tipo especificado.

pipelined_send() pipelined_send() permite a un proceso enviar directamente un mensaje a la cola de receptores mejor que depositar el mensaje en la cola asociada de mensajes esperando. La función 5.2.3 (testmsg()) es invocada para encontrar el primer receptor que está esperando por el mensaje dado. Si lo encuentra, el receptor esperando es quitado de la cola de receptores esperando, y la tarea receptora asociada es despertada. El mensaje es almacenado en el campo `r_msg` del receptor, y 1 es retornado. En el caso donde no hay un receptor esperando por el mensaje, 0 es devuelto.

En el proceso de búsqueda de un receptor, los receptores potenciales quizás encuentren que han solicitado un tamaño que es muy pequeño para el mensaje dado. Tales receptores son quitados de la cola, y son despertados con un status de error de E2BIG, el cual es almacenado en el campo `r_msg`. La búsqueda entonces continúa hasta que alguno de los receptores válidos es encontrado, o la cola está exausta.

copy_msqid_to_user() copy_msqid_to_user() copia el contenido de una antememoria del núcleo a la antememoria del usuario. Recibe como parámetros una antememoria del usuario, una antememoria del núcleo del tipo 5.2.2 (msqid64_ds), y una bandera versión indicando la nueva versión IPC vs. la vieja versión IPC. Si la bandera de la versión es igual a IPC_64, entonces copy_to_user() es llamado para copiar desde la antememoria del núcleo a la antememoria del usuario directamente. En otro caso una antememoria temporal del tipo struct msqid_ds es inicializada, y los datos del núcleo son trasladados a esta antememoria temporal. Posteriormente copy_to_user() es llamado para copiar el contenido de la antememoria temporal a la antememoria del usuario.

copy_msqid_from_user() La función copy_msqid_from_user() recibe como parámetros un mensaje de la antememoria del núcleo del tipo struct msq_setbuf, una antememoria de usuario y una bandera de la versión indicando la nueva versión IPC vs. la vieja versión IPC. En la caso de la nueva versión IPC, copy_from_user() es llamada para copiar el contenido de la antememoria del usuario a una antememoria temporal del tipo 5.2.2 (msqid64_ds). Entonces, los campos `qbytes`,`uid`, `gid`, y `mode` de la antememoria del núcleo son rellenados con los valores de los campos correspondientes desde la antememoria temporal. En el caso de la vieja versión IPC, una antememoria temporal del tipo struct 5.2.2 (msqid_ds) es usado en su vez.

5.3 Memoria Compartida

5.3.1 Interfaces de las llamadas al sistema de la Memoria Compartida

sys_shmget() La llamada entera a sys_shmget() es protegida por el semáforo global de memoria compartida.

En el caso donde un valor de llave es suministrado para un segmento existente de memoria compartida, el correspondiente índice es buscado en la matriz de descriptores de memoria compartida, y los parámetros y los permisos del llamante son verificados antes de devolver la ID del segmento de memoria compartida. Las operaciones de búsqueda y verificación son realizadas mientras es mantenido el spinlock global de memoria compartida.

sys_shmctl()

IPC_INFO Una antememoria temporal 5.3.2 (shminfo64) es cargada con los parámetros del sistema de memoria compartida y es copiada fuera del espacio de usuario para el acceso de la aplicación llamante.

SHM_INFO El semáforo global de memoria compartida y el spinlock global de memoria compartida son mantenidos mientras se obtienen estadísticas de la información del sistema para la memoria compartida. La función 5.3.3 (shm_get_stat()) es llamada para calcular el número de páginas de memoria compartidas que están residentes en memoria y el número de páginas de memoria compartida que han sido intercambiadas (swapped out). Otras estadísticas incluyen el número total de páginas de memoria compartida y el número de segmentos de memoria compartida en uso. Las cuentas de `swap_attempts` y `swap_successes` son codificadas fuertemente a cero. Estas estadísticas son almacenadas en una antememoria temporal 5.3.2 (shm_info) y copiadas fuera del espacio de usuario para la aplicación llamante.

SHM_STAT, IPC_STAT Para SHM_STAT y IPC_STATA, una antememoria temporal del tipo 5.3.2 (struct shmid64_ds) es inicializada, y el spinlock global de memoria compartida es cerrado.

Para el caso SHM_STAT, el parámetro ID del segmento de memoria compartida se espera que sea un índice exacto (esto es, 0 a n donde n es el número de IDs de memoria compartida en el sistema). Después de validar el índice, 5.4.1 (ipc_buildid()) es llamado (a través de shm_buildid()) para convertir el índice en una ID de memoria compartida. En el caso transitorio de SHM_STAT, la ID de la memoria compartida será el valor de retorno. Notar que esta es una característica no documentada, pero es mantenida para el programa ipcs(8).

Para el caso IPC_STAT, el parámetro ID del segmento de memoria compartida se espera que sea una ID que ha sido generada por una llamada a 5.3.1 (shmget()). La ID es validada antes de proceder. En el caso transitorio de IPC_STAT, el valor de retorno será 0.

Para SHM_STAT y IPC_STAT, los permisos de acceso del llamante son verificados. Las estadísticas deseadas son cargadas en la antememoria temporal y entonces copiadas fuera de la aplicación llamante.

SHM_LOCK, SHM_UNLOCK Después de validar los permisos de acceso, el spinlock global de memoria compartida es cerrado, y la ID del segmento de memoria compartida es validado. Para SHM_LOCK y SHM_UNLOCK, 5.3.3 (shmem_lock()) es llamada para realizar la función. Los parámetros para 5.3.3 (shmem_lock()) identifican la función a realizar.

IPC_RMID Durante el IPC_RMID el semáforo global de memoria compartida y el spinlock global de memoria compartida son mantenidos a través de esta función. La ID de la Memoria Compartida es validada, y entonces si no hay conexiones actuales, 5.3.3 (shm_destroy()) es llamada para destruir el segmento de memoria compartida. En otro caso, la bandera SHM_DEST es establecida para marcarlo para destrucción, y la bandera IPC_PRIVATE es establecida para prevenir que otro proceso sea capaz de referenciar la ID de la memoria compartida.

IPC_SET Después de validar la ID del segmento de memoria compartida y los permisos de acceso del usuario, las banderas `uid`, `gid`, y `mode` del segmento de la memoria compartida son actualizadas con los datos del usuario. El campo `shm_ctime` también es actualizado. Estos cambios son realizados mientras se mantiene el semáforo global de memoria compartida global y el spinlock global de memoria compartida.

sys_shmat() sys_shmat() toma como parámetro, una ID de segmento de memoria compartida, una dirección en la cual el segmento de memoria compartida debería de ser conectada (`shmaddr`), y las banderas que serán descritas más adelante.

Si `shmaddr` no es cero, y la bandera SHM_RND es especificada, entonces `shmaddr` es redondeado por abajo a un múltiplo de SHMLBA. Si `shmaddr` no es un múltiplo de SHMLBA y SHM_RND no es especificado, entonces EINVAL es devuelto.

Los permisos de acceso del llamante son validados y el campo `shm_nattch` del segmento de memoria compartida es incrementado. Nótese que este incremento garantiza que la cuenta de enlaces no es cero y previene que el segmento de memoria compartida sea destruido durante el proceso de enlazamiento al segmento. Estas operaciones son realizadas mientras se mantiene el spinlock global de memoria compartida.

La función do_mmap() es llamada para crear un mapeo de memoria virtual de las páginas del segmento de memoria compartida. Esto es realizado mientras se mantiene el semáforo `mmap_sem` de la tarea actual. La bandera MAP_SHARED es pasada a do_mmap(). Si una dirección fue suministrada por el llamante, entonces la bandera MAP_FIXED también es pasada a do_mmap(). En otro caso, do_mmap() seleccionará la dirección virtual en la cual mapear el segmento de memoria compartida.

NÓTESE que 5.3.3 (shm_inc()) será invocado con la llamada a la función do_mmap() a través de la estructura `shm_file_operations`. Esta función es llamada para establecer el PID, para establecer el tiempo actual, y para incrementar el número de enlaces a este segmento de memoria compartida.

Después de la llamada a do_mmap(), son obtenidos el semáforo global de memoria compartida y el spinlock global de la memoria compartida. La cuenta de enlaces es entonces decrementada. El siguiente cambio en la cuenta de enlaces es 1 para una llamada a shmat() por culpa de la llamada a 5.3.3 (shm_inc()). Si, después de decrementar la cuenta de enlaces, la cuenta resultante que se encuentra es cero, y el segmento se marca para la destrucciónn (SHM_DEST), entonces 5.3.3 (shm_destroy()) es llamado para liberar los recursos del segmento de memoria compartida.

Finalmente, la dirección virtual en la cual la memoria compartida es mapeada es devuelta al llamante en la dirección especificada por el usuario. Si un código de error ha sido retornado por do_mmap(), entonces este código de fallo es pasado en el valor de retorno para la llamada al sistema.

sys_shmdt() El semáforo global de la memoria compartida es mantenido mientras se realiza sys_shmdt(). La `mm_struct` del actual proceso es buscado para la `vm_area_struct` asociada con la dirección de memoria compartida. Cuando es encontrada, do_munmap() es llamado para deshacer el mapeo de direcciones virtuales para el segmento de la memoria compartida.

Nótese también que do_munmap() realiza una llamada atrás a 5.3.3 (shm_close()), la cual realiza las funciones manteniendo el libro de memoria compartida, y libera los recursos del segmento de memoria compartida si no hay más enlaces.

sys_shmdt() incondicionalmente devuelve 0.

5.3.2 Estructuras de Soporte de Memoria Compartida

struct shminfo64

```
struct shminfo64 {
        unsigned long   shmmax;
        unsigned long   shmmin;
        unsigned long   shmmni;
        unsigned long   shmseg;
        unsigned long   shmall;
        unsigned long   __unused1;
        unsigned long   __unused2;
        unsigned long   __unused3;
        unsigned long   __unused4;
};
```

struct shm_info

```
struct shm_info {
        int used_ids;
        unsigned long shm_tot;  /* shm asignada total */
        unsigned long shm_rss;  /* shm residente total */
        unsigned long shm_swp;  /* shm intercambiada total */
        unsigned long swap_attempts;
        unsigned long swap_successes;
};
```

struct shmid_kernel

```
struct shmid_kernel /* privadas del núcleo */
{
        struct kern_ipc_perm    shm_perm;
        struct file *           shm_file;
```

```
        int                     id;
        unsigned long           shm_nattch;
        unsigned long           shm_segsz;
        time_t                  shm_atim;
        time_t                  shm_dtim;
        time_t                  shm_ctim;
        pid_t                   shm_cprid;
        pid_t                   shm_lprid;
};
```

struct shmid64_ds

```
struct shmid64_ds {
        struct ipc64_perm       shm_perm;       /* permisos de la operación */
        size_t                  shm_segsz;      /* tamaño del segmento (bytes) */
        __kernel_time_t         shm_atime;      /* último tiempo de enlace */
        unsigned long           __unused1;
        __kernel_time_t         shm_dtime;      /* último tiempo de desenlace */
        unsigned long           __unused2;
        __kernel_time_t         shm_ctime;      /* último tiempo de cambio */
        unsigned long           __unused3;
        __kernel_pid_t          shm_cpid;       /* pid del creador */
        __kernel_pid_t          shm_lpid;       /* pid del último operador */
        unsigned long           shm_nattch;     /* número de enlaces actuales */
        unsigned long           __unused4;
        unsigned long           __unused5;
};
```

struct shmem_inode_info

```
struct shmem_inode_info {
        spinlock_t      lock;
        unsigned long   max_index;
        swp_entry_t     i_direct[SHMEM_NR_DIRECT]; /* para el primer bloque */
        swp_entry_t   **i_indirect; /* doble bloque indirecto */
        unsigned long   swapped;
        int             locked;     /* en la memoria */
        struct list_head        list;
};
```

5.3.3 Funciones de Soporte De Memoria Compartida

newseg() La función newseg() es llamada cuando un nuevo segmento de memoria compartida tiene que ser creado. Actúa con tres parámetros para el nuevo segmento: la llave, la bandera y el tamaño, Después de validar que el tamaño del segmento de la memoria compartida que va a ser creado está entre SHMMIN y SHMMAX y que el número total de segmentos de memoria compartida no excede de SHMALL, asigna un nuevo descriptor de memoria compartida. La función 5.3.3 (shmem_file_setup()) es invocada posteriormente a crear un archivo no enlazado del tipo tmpfs. El puntero del archivo retornado es guardado en el campo `shm_file` del descriptor del segmento de memoria compartida asociado. El nuevo descriptor de memoria compartida es inicializado e insertado en la matriz global de IPC de descriptores de segmentos de memoria compartida. La ID del segmento de memoria compartida es creado por shm_buildid() (a través de 5.4.1 (ipc_buildid())). La ID de este segmento es guardada en el campo `id` del descriptor de memoria compartida, al igual que en el campo `i_ino` del inodo asociado. En adicción, la dirección de las operaciones de memoria compartida definidas en la estructura `shm_file_operation` son almacenadas en el fichero asociado. El valor de la variable global `shm_tot`, que indica el número total de segmentos de memoria compartida a lo largo del sistema, es también incrementado para reflejar este cambio. Si tiene éxito, la ID del segmento es retornada a la aplicación llamante.

shm_get_stat() Los ciclos de shm_get_stat() van a través de todas las estructuras de memoria compartida, y calcula el número total de páginas de memoria en uso por la memoria compartida y el número total de páginas de memoria compartida que están intercambiadas. Hay una estructura de fichero y una estructura de inodo para cada segmento de memoria compartida. Como los datos requeridos son obtenidos a través del inodo, el spinlock para cada estructura inodo que es accedido es cerrado y abierto en secuencia.

shmem_lock() shmem_lock() recibe como parámetros un puntero al descriptor del segmento de memoria compartida y una bandera indicando cerrado vs. abierto. El estado de bloqueo del segmento de memoria compartida es almacenado en el inodo asociado. Este estado es comparado con el estado de bloqueo deseado: shmem_lock() simplemente retorna si ellos se corresponden.

Mientras se está manteniendo el semáforo del inodo asociado, el estado de bloqueo del inodo es establecido. La siguiente lista de puntos ocurren en cada página en el segmento de memoria compartida:

- find_lock_page() es llamado para cerrar la página (estableciendo PG_locked) y para incrementar la cuenta de referencia de la página. Incrementando la cuenta de referencia se asegura que el segmento de memoria compartida permanece bloqueado en memoria durante esta operación.
- Si el estado deseado es cerrado, entonces PG_locked es limpiado, pero la cuenta de referencia permanece incrementada.
- Si el estado deseado es abierto, entonces la cuenta de referencia es decrementada dos veces durante la actual referencia, y una vez para la referencia existente que causó que la página permanezca bloqueada en memoria. Entonces PG_locked es limpiado.

shm_destroy() Durante shm_destroy() el número total de páginas de memoria compartida es ajustada para que cuente el borrado del segmento de memoria compartida. 5.4.1 (ipc_rmid()) es llamado (a través de shm_rmid()) para borrar la ID de Memoria Compartida. 5.3.3 (shmem_lock) es llamado para abrir las páginas de memoria compartida, efectivamente, decrementando la cuenta de referencia a cero para cada página. fput() es llamado para decrementar el contador de uso para `f_count` para el objeto fichero deseado, y si es necesario, para liberar los recursos del objeto fichero. kfree() es llamado para liberar el descriptor de segmento de memoria compartida.

shm_inc() shm_inc() establece el PID, establece el tiempo actual, e incrementa el número de enlaces para el segmento de memoria compartida dado. Estas operaciones son realizadas mientras se mantiene el spinlock global de memoria compartida.

shm_close() shm_close() actualiza los campos `shm_lprid` y `shm_dtim` y decrementa el número de segmentos enlazados de memoria compartida. Si no hay otros enlaces al segmento de memoria compartida, entonces 5.3.3 (shm_destroy()) es llamado para liberar los recursos de la memoria compartida. Estas operaciones son todas realizadas mientras se mantienen el semáforo global de memoria compartida y el spinlock global de memoria compartida.

shmem_file_setup() La función shmem_file_setup() configura un archivo sin enlazar que vive en el sistema de ficheros tmpfs con el nombre y tamaño dados. Si hay suficientes recursos de memoria para este fichero, crea una nueva dentry bajo la raiz montada de tmpfs, y asigna un nuevo descriptor de fichero y un nuevo objeto inodo del tipo tmpfs. Entonces asocia el nuevo objeto dentry con el nuevo objeto inodo llamando a d_instantiate() y guarda la dirección del objeto dentry en el descriptor de fichero. El campo `i_size` del objeto inodo es establecido para ser del tamaño del fichero y el campo `i_nlink` es establecido para ser 0 en orden a marcar el inodo no enlazado. También, shmem_file_setup() almacena la dirección de la estructura `shmem_file_operations` en el campo `f_op`, e inicializa los campos `f_mode` y `f_vfsmnt` del descriptor de fichero propio. La función shmem_truncate() es llamada para completar la inicialización del objeto inodo. Si tiene éxito, shmem_file_setup() devuelve el nuevo descriptor de fichero.

5.4 Primitivas IPC de Linux

5.4.1 Primitivas IPC de Linux Genéricas usadas con Semáforos, Mensajes y Memoria Compartida

Los semáforos, mensajes, y mecanismos de memoria compartida de Linux son construidos con un conjunto de primitivas comunes. Estas primitivas son descritas en las secciones posteriores.

ipc_alloc() Si el asignamiento de memoria es mayor que PAGE_SIZE, entonces vmalloc() es usado para asignar memoria. En otro caso, kmalloc() es llamado con GFP_KERNEL para asignar la memoria.

ipc_addid() Cuando un nuevo conjunto de semáforos, cola de mensajes, o segmento de memoria compartido es añadido, ipc_addid() primero llama a 5.4.1 (grow_ary()) para asegurarse que el tamaño de la correspondiente matriz de descriptores es suficientemente grande para el máximo del sistema. La matriz de descriptores es buscada para el primer elemento sin usar. Si un elemento sin usar es encontrado, la cuenta de descriptores que están en uso es incrementada. La estructura 5.4.2 (kern_ipc_perm) para el nuevo recurso descriptor es entonces inicializado, y el índice de la matriz para el nuevo descriptor es devuelto. Cuando ipc_addid() tiene éxito, retorna con el spinlock global cerrado para el tipo IPC dado.

ipc_rmid() ipc_rmid() borra el descriptor IPC desde la matriz de descriptores global del tipo IPC, actualiza la cuenta de IDs que están en uso, y ajusta la máxima ID en la matriz de descriptores correspondiente si es necesario. Un puntero al descriptor asociado IPC con la ID del IPC dado es devuelto.

ipc_buildid() ipc_buildid() crea una única ID para ser asociada con cada descriptor con el tipo IPC dado. Esta ID es creada a la vez que el nuevo elemento IPC es añadido (ej. un nuevo segmento de memoria compartido o un nuevo conjunto de semáforos). La ID del IPC se convierte fácilmente en el índice de la correspondiente matriz de descriptores. Cada tipo IPC mantiene una secuencia de números la cual es incrementada cada vez que un descriptor es añadido. Una ID es creada multiplicando el número de secuencia con SEQ_MULTIPLIER y añadiendo el producto al índice de la matriz de descriptores. La secuencia de números usados en crear una ID de un IPC particular es entonces almacenada en el descriptor correspondiente. La existencia de una secuencia de números hace posible detectar el uso de una ID de IPC sin uso.

ipc_checkid() ipc_checkid() divide la ID del IPC dado por el SEQ_MULTIPLIER y compara el cociente con el valor seq guardado en el descriptor correspondiente. Si son iguales, entonces la ID del IPC se considera válida y 1 es devuelto. En otro caso, 0 es devuelto.

grow_ary() grow_ary() maneja la posibilidad de que el número máximo (ajustable) de IDs para un tipo IPC dado pueda ser dinámicamente cambiado. Fuerza al actual límite máximo para que no sea mayor que el limite del sistema permanente (IPCMNI) y lo baja si es necesario. También se asegura de que la matriz de descriptores existente es suficientemente grande. Si el tamaño de la matriz existente es suficientemente grande, entonces el límite máximo actual es devuelto. En otro caso, una nueva matriz más grande es asignada, la matriz vieja es copiada en la nueva, y la vieja es liberada. El correspondiente spinlock global es mantenido mientras se actualiza la matriz de descriptores para el tipo IPC dado.

ipc_findkey() ipc_findkey() busca a través de la matriz de descriptores del objeto especificado 5.4.2 (ipc_ids), y busca la llave especificada. Una vez encontrada, el índice del descriptor correspondiente es devuelto. Si la llave no es encontrada, entonces es devuelto -1.

ipcperms() ipcperms() chequa el usuario, grupo, y otros permisos para el acceso de los recursos IPC. Devuelve 0 si el permiso está garantizado y -1 en otro caso.

ipc_lock() ipc_lock() coge una ID de IPC como uno de sus parámetros. Cierra el spinlock global para el tipo IPC dado, y devuelve un puntero al descriptor correspondiente a la ID IPC especificada.

ipc_unlock() ipc_unlock() libera el spinlock global para el tipo IPC indicado.

ipc_lockall() ipc_lockall() cierra el spinlock global para el mecanismo IPC dado (esto es: memoria compartida, semáforos, y mensajes).

ipc_unlockall() ipc_unlockall() abre el spinlock global para el mecanismo IPC dado (esto es: memoria compartida, semáforos, y mensajes).

ipc_get() ipc_get() coge un puntero al tipo particular IPC (memoria compartida, semáforos o colas de mensajes) y una ID de un descriptor, y devuelve un puntero al descriptor IPC correspondiente. Nótese que aunque los descriptores para cada tipo IPC son tipos de datos diferentes, el tipo de estructura común 5.4.2 (kern_ipc_perm) está embebida como la primera entidad en todos los casos. La función ipc_get() devuelve este tipo de datos común. El modelo esperado es que ipc_get() es llamado a través de la función envoltorio (ej. shm_get()) la cual arroja el tipo de datos al tipo de datos correcto del descriptor.

ipc_parse_version() ipc_parse_version() borra la bandera IPC_64 desde el comando si está presente y devuelve IPC_64 o IPC_OLD.

5.4.2 Estructuras Genéricas IPC usadas con Semáforos, Mensajes, y Memoria Compartida

Todos los semáforos, mensajes, y mecanismos de memoria compartida hacen un uso de las siguientes estructuras comunes:

struct kern_ipc_perm Cada descriptor IPC tiene un objeto de datos de este tipo como primer elemento. Esto hace posible acceder a cualquier descriptor desde cualquier función genérica IPC usando un puntero de este tipo de datos.

```
/* usados por la estructuras de datos en el núcleo */
struct kern_ipc_perm {
    key_t key;
    uid_t uid;
    gid_t gid;
```

```
    uid_t cuid;
    gid_t cgid;
    mode_t mode;
    unsigned long seq;
};
```

struct ipc_ids La estructura ipc_ids describe los datos comunes para los semáforos, colas de mensajes, y memoria compartida. Hay tres instancias globales de esta estructura de datos –`semid_ds`, `msgid_ds` y `shmid_ds`– para los semáforos, mensajes y memoria compartida respectivamente. En cada instancia, el semáforo `sem` es usado para proteger el acceso a la estructura. El campo `entries` apunta a una matriz de descriptores de IPC, y el spinlock `ary` protege el acceso a esta matriz. El campo `seq` es una secuencia de números global la cual será incrementada cuando un nuevo recurso IPC es creado.

```
struct ipc_ids {
    int size;
    int in_use;
    int max_id;
    unsigned short seq;
    unsigned short seq_max;
    struct semaphore sem;
    spinlock_t ary;
    struct ipc_id* entries;
};
```

struct ipc_id Una matriz de estructuras ipc_id existe en cada instancia de la estructura 5.4.2 (ipc_ids). La matriz es dinámicamente asignada y quizás sea reemplazada con una matriz más grande por 5.4.1 (grow_ary()) tal como requiere. La matriz es a veces referida por la matriz de descriptores, desde que el tipo de datos 5.4.2 (kern_ipc_perm) es usado como tipo de datos de descriptores comunes por las funciones genéricas IPC.

```
struct ipc_id {
    struct kern_ipc_perm* p;
};
```

6 Sobre la traducción

Este libro ha sido traducido por Rubén Melcón <melkon@terra.es>.

Revisión de la traducción: Beta 0.03 (2 de Marzo de 2002).

Revisado el 27 de Octubre de 2002 por Manuel Canales Esparcia <macana@macana-es.con>

Publicado por *TLDP-ES*