Estructura Interna de los Sistemas Operativos

Javier Uruen Val y Aitor Acedo

11th February 2004

Contents

1 Práctica 1. Acceso a la tabla de procesos mediante una nueva llamada al sistema.

1.1 Introducción

Esta práctica consiste en la adición de una nueva llamada al sistema para proporcionar información de los procesos de un usuario, incluyendo las llamadas al sistema que realiza. Para la realización de este trabajo hemos optado por un kernel de desarrollo 2.6.0-TEST11.

Al no existir documentación actualizada sobre los kernels en desarrollo, nuestra principal fuente de información ha sido el código en sí mismo. Para ayudarnos a desplazarnos por las fuentes del kernel hemos usado un *lxr(linux-cross-reference)* , disponible en `http://lxr.linux.no`, se trata de una referencia cruzada de todas las estructuras, variables, funciones... que han sido declaradas y utilizadas en el kernel. Se usa vía web, siendo html y pudiendo así usarlo cómodamente desde consola, con ayuda de un navegador tipo LYNX.

Como sistema de desarrollo hemos usado la distribución DEBIAN en su versión inestable. Cabe reseñar que para utilizar un kernel 2.6.x en Debian se debe tener en cuenta lo siguiente:

- La serie 2.6.x usa un nuevo sistema de ficheros para mostrar información sobre el sistema, véase drivers, módulos, etc... este sistema es conocido como *sysfs* y debe ser añadido al fichero */etc/fstab* para su utilización al arrancar.
- Para poder utilizar módulos del kernel se debe actualizar el paquete *module-init-tools*. Ya que ha habido un cambio notable en la utilización y compilación de los módulos en esta serie.
- El sistema de compilación del kernel ha mejorado notablemente, siendo optimizado para estríctamente recompilar aquellas partes que lo necesitan. Además de eso, ahora ya no se tiene que realizar el típico *make dep && make clean && make bzImage* , ahora tendremos todo listo con un simple *make* en el directorio raíz de las fuentes.

1.2 Detalles de la Implementación

1.2.1 Modificación de entry.S

Como se aprecia en el siguiente código, ya no es necesario utilizar $SYMBOL\_NAME()$.

```
ENTRY(system_call)
pushl %eax              # save orig_eax
SAVE_ALL
GET_THREAD_INFO(%ebp)
cmpl $(nr_syscalls), %eax
jae syscall_badsys
testb $_TIF_SYSCALL_TRACE,TI_FLAGS(%ebp)
jnz syscall_trace_entry
syscall_call:
pushl %eax              # EISO SysCall number in %eax
```

```
call eiso_llama         # EISO Get the number as parameter
popl  %eax              # EISO Restore stack
call *sys_call_table(,%eax,4)
movl %eax,EAX(%esp)     # store the return value
pushl %eax              # EISO Right-Most parameter: return value
movl ORIG_EAX+4(%esp), %eax# EISO Get the SysCall number
pushl %eax              # EISO Pass as parameter
call eiso_salida        # EISO Call the function
popl %eax               # EISO Restore stack
popl %eax               # EISO Restore stack
```

También ha sido preciso en el mismo fichero añadir al vector *ENTRY(sys_call_table)* , nuestra nueva llamada al sistema. Nótese, que ya no es preciso modificar el tamaño del vector, ya que es calculado automáticamente. En */asm/unistd.h* hemos introducido el valor de nuestra nueva llamada, esto es *274*.

```
   ENTRY(sys_call_table)
          ...
          ...
   .long sys_tgkill        /* 270 */
   .long sys_utimes
   .long sys_fadvise64_64
   .long sys_ni_syscall    /* sys_vserver */
   long sys_psplus        /* EISO */
syscall_table_size=(.-sys_call_table)
```

1.2.2 Exportar símbolos

Para exportar símbolos, como por ejemplo la variable *eiso_run* que tiene que ser visible en *vigila.c* y en *psplus.c,* hay que utilizar la directiva EXPORT_SYMBOL(). Como ejemplo, un extracto del fichero *vigila.c* donde *eiso_run* es declarada.

```
volatile int eiso_run = 0;
EXPORT_SYMBOL(eiso_run);
```

1.2.3 Compilación de nuestras modificaciones

Un rápido vistazo a los *Makefiles* de los directorios de las fuentes del kernel, revela que, para que una unidad de compilación de un archivo .c a un objeto .o se lleve a cabo, el fichero objecto debe estar declarado dentro de la variable *obj-y* del *Makefile* del directorio donde se encuentran nuestros nuevos ficheros. He aquí el ejemplo del fichero *Makefile* del directorio */kernel/Makefile*.

```
obj-y= sched.o fork.o exec_domain.o panic.o printk.o profile.o \
    exit.o itimer.o time.o softirq.o resource.o \
    sysctl.o capability.o ptrace.o timer.o user.o \
    signal.o sys.o kmod.o workqueue.o pid.o psplus.o vigila.o\
    rcupdate.o intermodule.o extable.o params.o posix-timers.o
```

1.2.4 Modificaciones de la práctica

Hemos cambiado la declaración de *sysinfo* en *task_struct* para que sea un puntero a estructura. De tal manera que cuando un proceso es creado, *do_fork*(), se comprueba si la variable *eiso_run* está activa, si lo está reservamos espacio para la estructura *sysinfo*. Además cuando se activa la monitorización, la llamada al sistema se encarga de recorrer toda lista de procesos, y todos aquellos que tengan el campo *sysinfo* apuntando a *NULL*, se reserva espacio. También hay que tener en cuenta que cuando un proceso termine, es decir, ejecute do_exit(), habrá que liberar la memoria de *sysinfo* para no desperdiciarla.

La modificación de la práctica que debe conseguir que cuando un proceso acabe se desactive la monitorización, la hemos fusionado con la modificación que permite que varios procesos monitoricen concurrentemente. Lo que hemos hecho es utilizar un semáforo (*list_mutex)* y un vector de PIDs (*list_pids*) donde guardamos qué procesos se están monitorizando, además una variable global *num_procs,* que indica cuantos procesos hay concurrentemente monitorizando. El acceso y modificación tanto a *num_procs* como *list_pids,* se realiza en exclusión mútua usando el semáforo. Básicamente lo que hacemos es: cada vez que un proceso inicia la monitorización buscamos un sitio libre en *list_pids,* esto se denota porque estará puesto a EMPTY, anotamos el PID actual del proceso en el hueco libre y si somos el primer proceso en activar la monitorización, esto es *num_proc* es igual 0, entonces ponemos la variable *eiso_run* a 1. Ahora realizaremos la operación de indicar que un proceso ha dejado de monitorizar, esto es, o ha desactivado la monitorización vía *ioctl*() o el proceso ha acabado. La operación consistirá en buscar el PID del proceso que desea acabar o desactivar la monitorización, en *list_pids,* liberará la posición que ocupaba, decrementará en 1 el número de procesos concurrentes, y si este último número es igual a 0 querrá decir que somos el último proceso que monitoriza. Con lo cual desactivaremos *eiso_run* y resetaremos las estructuras a 0.

Nótese que en esta implementación limitamos el número de procesos concurrentes por el tamaño del vector. Una opción más escalable sería usar una lista genérica, por ejemplo la que se encuentra en *<linux/list.h>*. Aunque consideramos que esa solución es más elegante, pero no la implementamos porque creemos que no es objeto de está práctica, sino la sincronización en sí misma, la cual ya la conseguimos con el semáforo y el vector.

1.3 Relación de los ficheros generados y de los ficheros modificados

1.3.1 Ficheros generados

- kernel/psplus.c
- kernel/vigila.c
- include/linux/psplus.h

1.3.2 Ficheros modificados

- arch/i386/kernel/entry.S
- include/asm/unistd.h

- include/linux/sched.h[1]
- kernel/Makefile
- kernel/exit.c[2]
- kernel/fork.c[3]

1.4 Fuente de los ficheros generados

1.4.1 kernel/psplus.c

```
/*
 * psplus.c
 *     Implementación de una nueva llamada al sistema
 *     para el proyecto 1. EISO
 *
 *
 *
 * Version:     0.1     5/01/2004
 *
 * Authors:     Aitor Acedo, <460829@celes.unizar.es>
 *              Javier Uruen Val, <460821@celes.unizar.es>
 *
 *
 * This program is free software; you can redistribute it and/or
 * modify it under the terms of the GNU General Public License
 * as published by the Free Software Foundation; either version
 * 2 of the License, or (at your option) any later version.
 */
#include <linux/config.h>
#include <linux/sched.h>
#include <linux/stddef.h>
#include <asm/uaccess.h>
#include <asm/unistd.h>
#include <asm/semaphore.h>
#include <linux/fs.h>
#include <linux/fs_struct.h>
#include <linux/psplus.h>
extern int eiso_run;
/*
 * Distingimos si estamos en la ampliación
 * de la práctica o en la primera implementación.
 * Utilizando para ello el flag AMPLIACION
 */
#ifdef AMPLIACION
#define SYSINFO (void *)p->sysinfo
DECLARE_MUTEX(list_mutex); /* Semáforo para mutex */
```

[1] Modificado también para parte opcional 3
[2] Parte opcional 1
[3] Parte opcional 2

```
int list_pids[MAXPROC];    /* Lista de procesos
                            * que están monitorizando
                            */
int num_proc = 0;
#else
#define SYSINFO (void *)&p->sysinfo
#endif
/*
 * Rellena una estructura t_syscall_all y la copia
 * a espacio de usuario.
 * Variables globales accedidas:
 *   - current
 * Variables globales modificadas:
 * Funciones del núcleo utilizadas:
 *   - memcpy()
 *   - copy_to_user()
 */
static int make_syscall_all(unsigned int uid, int max, void *dst){
  struct t_syscall_all syscall_all;
  struct task_struct *p;
  int byte_cp; /* Bytes copiados a user space */
  int times = 0;
  for_each_process(p){
    if ( uid == p->uid  && (SYSINFO != NULL) ) {

      times++;
      /*
       * Rellenamos la estructura
       * syscall_all con los datos
       * del proceso.
       */
      syscall_all.basica_all.basica_ident.pid = p->pid;
      syscall_all.basica_all.basica_ident.ppid = p->parent->pid;
      syscall_all.basica_all.basica_ident.uid = p->uid;
      syscall_all.basica_all.basica_ident.gid = p->gid;
      syscall_all.basica_all.prioridad = p->prio;
      syscall_all.basica_all.estado = p->state;
      syscall_all.basica_all.tiempo = p->start_time;
      syscall_all.basica_all.root_inode =
       p->fs->root->d_inode->i_ino;
     syscall_all.basica_all.cwd_inode =
       p->fs->pwd->d_inode->i_ino;
      strcpy(syscall_all.basica_all.comando, p->comm);

      /*
       * Hacemos esto o rellenamos campo a campo
       *
       */
      memcpy( (void *)&syscall_all.open, SYSINFO,
        sizeof(t_syscall_kernel) );
```

```
      /*
       * Una vez rellena la estructura la copiamos
       * a espacio de usuario
       */

      byte_cp = copy_to_user( dst, &syscall_all,
            sizeof(struct t_syscall_all));

      /*
       * Si la copia ha sido correcta actualizamos
       * el puntero al espacio de usuario
       */
      if ( !byte_cp )

        dst+= sizeof(struct t_syscall_all);
      else
        return -1;
      /*
       * Comprobamos que no rebasamos el número
       * máximo de procesos a copiar
       */
      if(times >= max)
        break;
    }
  }

  return times;

}
/*
 * Rellena una estructura t_basica_indent y la copia
 * a espacio de usuario.
 * Variables globales accedidas:
 *   - current
 * Variables globales modificadas:
 * Funciones del núcleo utilizadas:
 *   - memcpy()
 *   - copy_to_user()
 */
static int make_syscall_ident(unsigned int uid, int max, void *dst){
  struct t_basica_ident syscall_ident;
  struct task_struct *p;
  int byte_cp; /* Bytes copiados a user space */
  int times = 0;
  for_each_process(p){
    if ( uid == p->uid  && (SYSINFO != NULL) ) {

      times++;
      /*
```

```
        * Rellenamos la estructura
        * syscall_ident con los datos
        * del proceso.
        */
       syscall_ident.pid = p->pid;
       syscall_ident.ppid = p->parent->pid;
       syscall_ident.uid = p->uid;
       syscall_ident.gid = p->gid;

       /*
        * Hacemos esto o rellenamos campo a campo
        *
        */
       memcpy( (void *)&syscall_ident, SYSINFO,
         sizeof(struct t_basica_ident) );

       /*
        * Una vez rellena la estructura la copiamos
        * a espacio de usuario
        */

       byte_cp = copy_to_user( dst, &syscall_ident,
             sizeof(struct t_basica_ident));

       /*
        * Si la copia ha sido correcta actualizamos
        * el puntero al espacio de usuario
        */
       if ( !byte_cp )

         dst+= sizeof(struct t_basica_ident);
       else
         return -1;
       /*
        * Comprobamos que no rebasamos el número
        * máximo de procesos a copiar
        */
       if(times >= max)
         break;
     }
   }

   return times;

}
/*
 * Realiza la llamada al sistema sys_psplus.
 * Variables globales accedidas:
 *   - current
 *   - eiso_run
```

```
 *   - list_pids[] ( Ampliación)
 *   - list_mutex  ( Ampliación)
 *   - num_proc    ( Ampliación)
 * Variables globales modificadas:
 * Funciones del núcleo utilizadas:
 *   - printk()
 *   - down_interruptible() (Ampliación)
 *   - up() (Ampliación(
 */
asmlinkage long sys_psplus(unsigned int uid, int max, void *ptr, int CMD)
{
struct task_struct *p;
int i;

  switch(CMD){
    case OFF:
      printk("Desactivando eiso_run\n");

      /*
       * Acceso en mutex a la lista de procesos
       * que están monitorizando.
       */
      if ( down_interruptible(&list_mutex) != 0 )
        return -1;
      /*
       * Buscamos el pid en la lista
       */
      for ( i = 0; i < MAXPROC; i++)
        if ( list_pids[i] == current->pid ){
          /*
           * Lo liberamos
           */
          list_pids[i] = EMPTYPID;
          printk("Liberamos pos %i del pid %i\n",
            i, current->pid);
          num_proc--;
          break;
        }
      /* Si se cumple la siguiente condición quiere decir
       * que un proceso que no ha activado la monitorización
       * está intentando desactivarla
       */
      if ( i == MAXPROC ){
        printk("Proceso %i no habia activado monitor.",
          current->pid);
        up(&list_mutex);
        return -1;
      }

      if ( !num_proc ){ /* Somos el último proceso */
```

```
    eiso_run = 0;
    for_each_process(p){ / * Reiniciamos contadores */
      if ( p->sysinfo != NULL ){
        memset(p->sysinfo, 0, sizeof(t_syscall_kernel));
      }
    }
  }

  /* Soltamos el semáforo */
   up(&list_mutex);
return 0;

case ON:
  printk("Activando eiso_run\n");

  /*
   * Acceso en mutex a la lista de procesos
   * que están monitorizando.
   */
  if ( down_interruptible(&list_mutex) != 0 )
    return -1;

  /*
   * Si es la primera llamada a sysplus,
   * inicializo el vector que almacena
   * los pids de los procesos monitorizando
   * a la vez
   */
  if ( eiso_run == 0 )
    for ( i = 0; i < MAXPROC; i++)
      list_pids[i] = EMPTYPID;


  /*
   * Colocamos el pid del proceso actual
   * en la primera posición libre
   */
  if ( num_proc < MAXPROC ) {
    for ( i = 0; i < MAXPROC; i++)
      if ( list_pids[i] == EMPTYPID )
        break;

    printk("Añadiendo pid %i a la posición %i\n",
      current->pid, i);
    list_pids[i] = current->pid;
     num_proc++;
  }

  else {
    up(&list_mutex);
```

```
        return -1;
      }

      /*
       * Debemos recorrer todos los procesos
       * y reservar memoria para aquellos procesos
       * que tenga sysinfo apuntando a NULL
       */
      for_each_process(p){
        if ( p->sysinfo == NULL ){
           p->sysinfo = kmalloc(sizeof(t_syscall_kernel),
                                GFP_KERNEL);
           if ( p->sysinfo)
               memset(p->sysinfo, 0, sizeof(t_syscall_kernel));
        }
      }

       /* Soltamos el semaforo */
       up(&list_mutex);
      return 0;
    case IDENT:
      if(!eiso_run)
        return EINVAL;
      return make_syscall_ident(uid, max, ptr);
    case ALL:
      if(!eiso_run)
        return EINVAL;
      return make_syscall_all(uid, max, ptr);
    default:
      return EINVAL;
  }

}
```

1.4.2 kernel/vigila.c

```
/*
 * vigila.c
 *     Funciones que actualizan las estadisísticas
 *     de acceso a las llamada de sistema, fallos,
 *     aciertos, etc...
 *
 *
 *
 * Version:     0.1     5/01/2004
 *
 * Authors:     Aitor Acedo, <460829@celes.unizar.es>
 *              Javier Uruen Val, <460821@celes.unizar.es>
 *
 *
```

```
 * This program is free software; you can redistribute it and/or
 * modify it under the terms of the GNU General Public License
 * as published by the Free Software Foundation; either version
 * 2 of the License, or (at your option) any later version.
 */
#include <linux/config.h>
#include <linux/sched.h>
#include <linux/module.h>
#include <asm/uaccess.h>
#include <asm/unistd.h>
/*
 * Macros para actualizar las estructuras de información
 * que almacenan los resultados de las llamadas al sistema.
 * Dependiendo si estamos en la ampliación o no, el acceso
 * será por puntero o por estructura estática.
 */
#ifndef AMPLIACION
#define UPDATE_ENTRADA(call) \
    current->sysinfo.call.entradas++
#define UPDATE_OK(call) \
    current->sysinfo.call.salidas_ok++
#define UPDATE_ERROR(call) \
    current->sysinfo.call.salidas_error++
#else
#define UPDATE_ENTRADA(call) do {\
  if (current->sysinfo) current->sysinfo->call.entradas++;  }while(0)

#define UPDATE_OK(call)  do{ \
  if (current->sysinfo) current->sysinfo->call.salidas_ok++; }while(0)

#define UPDATE_ERROR(call) do {\
  if (current->sysinfo) current->sysinfo->call.salidas_error++;}while(0)

#endif

int eiso_run = 0;
/*
 * Hacemos visible la variable al resto del kernel
 */
EXPORT_SYMBOL(eiso_run);
/*
 *
 * Variables globales accedidas:
 *   - current
 *   - eiso_run
 *   - current->sysinfo
 * Variables globales modificadas:
 *   - current->sysinfo
 * Funciones del núcleo utilizadas:
 */
```

```
asmlinkage long eiso_llamada(int num_syscall)
{
  if (!eiso_run)
    return 0;

  switch(num_syscall)
  {
    case __NR_open:
      UPDATE_ENTRADA(open);
      break;
    case __NR_close:
      UPDATE_ENTRADA(close);
      break;
    case __NR_read:
      UPDATE_ENTRADA(read);
      break;
    case __NR_write:
      UPDATE_ENTRADA(write);
      break;
    case __NR_exit:
      UPDATE_ENTRADA(exit);
      break;
    case __NR_fork:
      UPDATE_ENTRADA(fork);
      break;
    default:
      return 0;
  }
  return 0;
}
/*
 *
 * Variables globales accedidas:
 *   - current
 *   - eiso_run
 *   - current->sysinfo
 * Variables globales modificadas:
 *   - current->sysinfo
 * Funciones del núcleo utilizadas:
 */
asmlinkage long eiso_salida(int num_syscall, int ret)
{
  if (!eiso_run)
    return 0;

  switch(num_syscall)
  {
    case __NR_open:
      if ( ret < 0 )  UPDATE_ERROR(open);
      else     UPDATE_OK(open);
```

```
        break;
      case __NR_close:
        if ( ret < 0 )   UPDATE_ERROR(close);
               else      UPDATE_OK(close);
        break;
      case __NR_read:
        if (ret < 0)     UPDATE_ERROR(read);
        else             UPDATE_OK(read);
        break;
      case __NR_write:
        if (ret < 0)     UPDATE_ERROR(write);
        else             UPDATE_OK(write);
        break;
      case __NR_exit:
        if (ret < 0)     UPDATE_ERROR(exit);
        else             UPDATE_OK(exit);
        break;
      case __NR_fork:
        if (ret < 0)     UPDATE_ERROR(fork);
        else             UPDATE_OK(fork);
        break;
      default:
        return 0;
    }

    return 0;
  }
```

1.4.3 include/linux/psplus.h

```
#ifndef _LINUX_PSPLUS
  #define _LINUX_PSPLUS
  #define AMPLIACION
  #define MAXPROC 10 //Máximo número de procesos monitorizando concurrentemente
  #define EMPTYPID -1 //Indica que hay un hueco en el vector de pids
  /*
  * EISO estructuras
  *
  */
  #define OFF 0
  #define ON 1
  #define ALL 2
  #define IDENT 3
  struct t_info_syscall{
  int entradas;
  int salidas_ok;
  int salidas_error;
  };
  #ifdef __KERNEL__
```

```
typedef struct {
struct t_info_syscall open;
struct t_info_syscall close;
struct t_info_syscall read;
struct t_info_syscall write;
struct t_info_syscall exit;
struct t_info_syscall fork;
} t_syscall_kernel;
#endif
struct t_basica_ident{
int pid;
int ppid;
int uid;
int gid;
};
struct t_basica_all{
struct t_basica_ident basica_ident;
int prioridad;
int estado;
unsigned long long int tiempo;
int root_inode;
int cwd_inode;
char comando[16];
};
struct t_syscall_all{
struct t_basica_all basica_all;
struct t_info_syscall open;
struct t_info_syscall close;
struct t_info_syscall read;
struct t_info_syscall write;
struct t_info_syscall exit;
struct t_info_syscall fork;
};
#endif
```

1.5 Fuente de los ficheros modificados

1.5.1 arch/i386/kernel/entry.S

```
ENTRY(system_call)
pushl %eax              # save orig_eax
SAVE_ALL
GET_THREAD_INFO(%ebp)
cmpl $(nr_syscalls), %eax
jae syscall_badsys
testb $_TIF_SYSCALL_TRACE,TI_FLAGS(%ebp)
jnz syscall_trace_entry
syscall_call:
pushl %eax              # EISO SysCall number in %eax
call eiso_llama         # EISO Get the number as parameter
```

```
popl  %eax              # EISO Restore stack
call *sys_call_table(,%eax,4)
movl %eax,EAX(%esp)     # store the return value
pushl %eax              # EISO Right-Most parameter: return value
movl ORIG_EAX+4(%esp), %eax# EISO Get the SysCall number
pushl %eax              # EISO Pass as parameter
call eiso_salida        # EISO Call the function
popl %eax               # EISO Restore stack
popl %eax               # EISO Restore stack
       ...
       ...
ENTRY(sys_call_table)
       ...
       ...
.long sys_tgkill        /* 270 */
.long sys_utimes
.long sys_fadvise64_64
.long sys_ni_syscall    /* sys_vserver */
.long sys_psplus        /* EISO */
syscall_table_size=(.-sys_call_table)
```

1.5.2 include/asm/unistd.h

```
#define __NR_statfs64           268
#define __NR_fstatfs64          269
#define __NR_tgkill             270
#define __NR_utimes             271
#define __NR_fadvise64_64       272
#define __NR_vserver            273
#define __NR_psplus             274 /* Practica EISO */
#define NR_syscalls 275
```

1.5.3 include/linux/sched.h

```
struct task_struct{
     ...
     ...
        struct reclaim_state *reclaim_state;
        struct dentry *proc_dentry;
        struct backing_dev_info *backing_dev_info;

        struct io_context *io_context;

        unsigned long ptrace_message;
        siginfo_t *last_siginfo; /* For ptrace use.  */
#ifndef AMPLIACION
        t_syscall_kernel sysinfo; /* Static struct EISO */
#else
        t_syscall_kernel *sysinfo; /* Dinamic struct EISO */
#endif
```

```
};
```

1.5.4 kernel/Makefile

```
 Makefile for the linux kernel.
#
obj-y     = sched.o fork.o exec_domain.o panic.o printk.o profile.o \
            exit.o itimer.o time.o softirq.o resource.o \
            sysctl.o capability.o ptrace.o timer.o user.o \
            signal.o sys.o kmod.o workqueue.o pid.o psplus.o vigila.o\
            rcupdate.o intermodule.o extable.o params.o posix-timers.o
...
...
...
```

1.5.5 kernel/exit.c

```
...
include <asm/pgtable.h>
#include <asm/mmu_context.h>
#include <linux/psplus.h> /* EISO */
extern void sem_exit (void);
extern struct task_struct *child_reaper;
extern int eiso_run;
#ifdef AMPLIACION
extern int list_pids[MAXPROC];
extern struct semaphore list_mutex;
extern int num_proc;
#endif
...
...
NORET_TYPE void do_exit(long code)
{
  struct task_struct   *tsk = current;
  struct task_struct *p; /* EISO */
  int i;
  if (unlikely(in_interrupt()))
    panic("Aiee, killing interrupt handler!");
  if (unlikely(!tsk->pid))
    panic("Attempted to kill the idle task!");
  if (unlikely(tsk->pid == 1))
    panic("Attempted to kill init!");
  if (tsk->io_context)
    exit_io_context();
  tsk->flags |= PF_EXITING;
  del_timer_sync(&tsk->real_timer);
  if (unlikely(in_atomic()))
    printk(KERN_INFO "note: %s[%d] exited with preempt_count %d\n",
        current->comm, current->pid,
```

```
        preempt_count());
#ifdef AMPLIACION
  /*
   * EISO
   * Si un proceso se acaba antes de que se haya puesto
   * eiso_run a 0, entonces debemos liberar el espacio
   * de sysinfo
   */
  if (  tsk->sysinfo ) {
    kfree(tsk->sysinfo);
    tsk->sysinfo = NULL;
  }
  /*
   * Acceso en mutex a la lista de procesos
   * que están monitorizando.
   */
  down_interruptible(&list_mutex);
  /*
   * Buscamos el pid en la lista
   */
  for ( i = 0; i < MAXPROC; i++)
    if ( list_pids[i] == current->pid ){
      /*
       * Lo liberamos
       */
      list_pids[i] = EMPTYPID;
      printk("Liberamos pos %i del pid %i\n",
        i, current->pid);
      num_proc--;
      break;
    }
  if ( !num_proc) {
 if ( !num_proc) {
    /*
     * Si somos el proceso que ha activado la monitorización
     * entonces ponemos eiso_run a 0 y liberamos espacio
     */
    if ( tsk->pid == eiso_run ){
      for_each_process(p){
        if ( p->sysinfo != NULL ){
           memset(p->sysinfo, 0, sizeof(t_syscall_kernel));
           //p = NULL;
        }
      }
      printk("Forzando desactivación de eiso_run\n");
      eiso_run = 0;
    }
  }
  up(&list_mutex);
#endif
```

```
/*
 * Fin EISO
 */
  profile_exit_task(tsk);
  if (unlikely(current->ptrace & PT_TRACE_EXIT)) {
    current->ptrace_message = code;
    ptrace_notify((PTRACE_EVENT_EXIT << 8) | SIGTRAP);
  }
...
...
...
```

1.5.6 kernel/fork.c

```
...
...
...
extern int eiso_run;
...
...
...
long do_fork(unsigned long clone_flags,
        unsigned long stack_start,
        struct pt_regs *regs,
        unsigned long stack_size,
        int __user *parent_tidptr,
        int __user *child_tidptr)
{
  struct task_struct *p;
  int trace = 0;
  long pid;
  if (unlikely(current->ptrace)) {
    trace = fork_traceflag (clone_flags);
    if (trace)
      clone_flags |= CLONE_PTRACE;
  }
  p = copy_process(clone_flags, stack_start,
      regs, stack_size, parent_tidptr, child_tidptr);
  /*
   * Do this prior waking up the new thread - the thread pointer
   * might get invalid after that point, if the thread exits   quickly.
   */
  pid = IS_ERR(p) ? PTR_ERR(p) : p->pid;
  if (!IS_ERR(p)) {
    struct completion vfork;
#ifdef AMPLIACION
    /* EISO
     * Si eiso_run está activada, reservamos espacio
     * para nuestra estructura.
     */
```

```
    if ( eiso_run ) {
      p->sysinfo = kmalloc(sizeof(t_syscall_kernel), GFP_KERNEL);
      if ( p->sysinfo)
        memset(p->sysinfo, 0, sizeof(t_syscall_kernel));
    }
    else
      p->sysinfo = NULL;
#endif
...
...
...
```

1.6 Fuente de los programas de prueba

```
/*
 * testit.c
 *     Code to test project 1.
 *
 *
 *
 *
 * Version:     0.1     5/01/2004
 *
 * Authors:     Aitor Acedo, <460829@celes.unizar.es>
 *              Javier Uruen Val, <460821@celes.unizar.es>
 *
 *
 * This program is free software; you can redistribute it and/or
 * modify it under the terms of the GNU General Public License
 * as published by the Free Software Foundation; either version
 * 2 of the License, or (at your option) any later version.
 */
#include <asm/unistd.h>
#include <pwd.h> /* getpwnam() */
#include <sys/types.h>
#include <linux/psplus.h>
#include <errno.h>
#include <stdio.h>
#define MAXPROC 20
#define PRINT_SYSCALL(syscall_all,field) \
  fprintf(stdout, #field ": Entradas %i\t\tOk %i\t\tError %i\n",\
  syscall_all.field.entradas, syscall_all.field.salidas_ok,\
  syscall_all.field.salidas_error)
#define PRINT_BASICA_IDENT(syscall_all,fmt,field) \
  fprintf(stdout, #field ": "  #fmt "\n" , syscall_all.basica_all.basica_ident.field)
#define PRINT_BASICA(syscall_all,fmt,field) \
  fprintf(stdout, #field ": "  #fmt "\n" , syscall_all.basica_all.field)
#define SHOWINFO(syscall_all) do {\
  PRINT_BASICA_IDENT(syscall_all,%i,pid);\
  PRINT_BASICA_IDENT(syscall_all,%i,ppid);\
```

```
  PRINT_BASICA_IDENT(syscall_all,%i,uid);\
  PRINT_BASICA_IDENT(syscall_all,%i,gid);\
  PRINT_BASICA(syscall_all,%i,prioridad);\
  PRINT_BASICA(syscall_all,%i,estado);\
  PRINT_BASICA(syscall_all,%lli,tiempo);\
  PRINT_BASICA(syscall_all,%i,root_inode);\
  PRINT_BASICA(syscall_all,%s,comando);\
  PRINT_SYSCALL(syscall_all,open);\
  PRINT_SYSCALL(syscall_all,close);\
  PRINT_SYSCALL(syscall_all,read);\
  PRINT_SYSCALL(syscall_all,write);\
  PRINT_SYSCALL(syscall_all,exit);\
  PRINT_SYSCALL(syscall_all,fork);\
  } while(0)
/*
 * Macro for a 4 arguments system call
 */
_syscall4(int, psplus , int,  a, int, b, void*, foo, int,  c)
int main(int argc, char *argv[]){
  struct passwd *pass;
  struct t_syscall_all info[MAXPROC];
  int num, i;

  if ( argc < 2 ) {
    fprintf(stderr, "Usage: %s <name>\n", argv[0] );
    return -1;
  }
  /*
   * Getting the uid
   */
  if ( ! ( pass = getpwnam(argv[1])) ) {
    fprintf(stderr, "No such username %s\n", argv[0] );
                return -1;

  }
  fprintf(stdout, "Username %s has uid %i\n", argv[1], pass->pw_uid);
  /*
   * Setting monitoring mode on
   *
   */
  psplus(0, 0, NULL, ON);

  /*
   * We let 5 seconds for system running
   */
  sleep(5);
  /*
   * Getting info
   *
   */
```

```
   num = psplus(pass->pw_uid, MAXPROC, &info, ALL);
   if ( num < 0 ){
    fprintf(stderr, "Couldnt get info \n" );
    return -1;
   }

   else if ( num == 0 ) {
    fprintf(stdout, "No process for this uid\n");

   }
   else {
     for ( i = 0; i < num ; i++){
       fprintf(stdout, "\n\n==========\n");
       SHOWINFO(info[i]);

     }
   }

  /*
   * Setting monitoring mode
   * We let the kernel auto-off monitoring after
   * last process exits.
   */
//  psplus(0, 0, NULL, OFF);

  return 1;
}
```

2 Práctica 2. Programación de un módulo para monitorizar el Slab Allocator

2.1 Introducción

Esta práctica consiste en la creación de un módulo de kernel que utilizará un dispositivo de tipo carácter para dar a conocer al usuario información sobre la utilización de las caches genéricas del Slab Allocator. Para ello hemos optado por implementarlo para la versión 2.6.2 del kernel de linux.

Al igual que en la práctica anterior nuestra principal fuente de información ha sido las fuentes del kernel. Pero además, para conocer el funcionamiento del Slab Alloctor hemos utilizado el libro *UNIX Internals: The New Frontier by Uresh Vahalia,* y una explicación más detallada en el trabajo de *Jeff Bonwick (Sun Mcirosystems)* para el *USENIX Summer 1994 Technical Conference.* También cabe destacar una interesante fuente de información para el *porting* de drivers a 2.6.x en `http://lwn.net`.

Como sistema de desarrollo hemos usado la distribución DEBIAN en su versión inestable. Cabe reseñar que para utilizar un kernel 2.6.x en Debian se debe tener en cuenta lo siguiente:

- La serie 2.6.x usa un nuevo sistema de ficheros para mostrar información sobre el sistema, véase drivers, módulos, etc... este sistema es conocido como *sysfs* y debe ser añadido al fichero */etc/fstab* para su utilización al arrancar.

- Para poder utilizar módulos del kernel se debe actualizar el paquete *module-init-tools.* Ya que ha habido un cambio notable en la utilización y compilación de los módulos en esta serie. Sin este paquete actualizado será imposible cargar nuestro módulo.

- El sistema de compilación del kernel ha mejorado notablemente, siendo optimizado para estrictamente recompilar aquellas partes que lo necesitan. Además de eso, ahora ya no se tiene que realizar el típico *make dep && make clean && make bzImage* , ahora tendremos todo listo con un simple *make* en el directorio raíz de las fuentes.

- Este anterior punto concierne directamente a la compilación de nuestro módulo. Si bien en los núcleos inferiores a 2.5.x, compilábamos un módulo donde la fase de linkado se obviaba hasta que este se cargaba en el kernel. Es decir, sólo necesitábamos las fuentes del kernel y no necesitábamos ningún otro fichero objeto a parte de nuestros módulos para compilar nuestro módulo. Ahora la cosa cambia, el cargador de módulo necesita conocer la localización de ciertos símbolos a la hora de cargar el módulo. Así pues a partir de los kernels 2.6.x es preciso este paso de pre-linkado contra las librerías que incluye el kernel. Haciendo, en principio, más complicado la compilación de los módulos externos. Decimos en principio más complicado, porque como se verá en la siguiente sección, la complejidad del proceso aumenta, pero desde el punto de vista del programador de módulos se hace más transparente.

2.2 Detalles de la implementación

2.2.1 Detalles de la compilación

Como hemos explicado en la anterior sección, el paradigma de compilación de módulos cambia sensiblemente en la última rama del kernel. Para facilitar el trabajo al programador, lo que hacemos es usar la opción *-C,* del progama *make,* este flag recibe como parámetro un directorio, en este caso será el directorio raíz de nuestras fuentes del kernel además le pasaremos a *make* la variable *SUBDIRS* que contendrá el directorio donde se encuentre las fuentes de nuestros módulos. Lo que hará entonces el programa make será trasladarse al directorio que le hemos pasado tras la opción *-C*, es decir, las fuentes del kernel, leerá entonces la información de los *Makefiles*, configs, etc.. necesarios. Posteriormente, con la configuración del kernel, pasará a ejecutar los *Makefiles* que se encuentran en el directorio de *SUBDIRS*, estos *Makefiles* simplemente dirán al *Makefile* principal del kernel que queremos compilar nuestro módulo, y éste así lo hará, pero utilizando la configuración de kernel tal y como si hubiéramos añadido un directorio nuevo a las fuentes del kernel, con nuestro módulo, hubiéramos modificado los *Makefiles* para que leyera el de nuestro nuevo directorio, etc... En síntesis lo que deberemos añadir al directorio donde se encuentre nuestro módulo será un *Makefile* que contendrá simplemente:

```
obj-y := eiso.o
```

Posteriormente si nos situamos dentro del directorio donde tenemos las fuentes de nuestro módulo, simplemente tendremos que ejecutar:

```
make -C /path/to/kernelsource SUBDIRS=$PWD modules
```

Tras la compilación obtendremos un bonito *eiso.ko* (nótese el cambio de extensión, muy zaragozana ella), listo para ser insertado en el kernel.

2.2.2 Cambios en el registro de char devices

En esta nueva versión del kernel se ha trabajado en aumentar el número de posibles dispositivos, limitado anteriormente por el tamaño que podía tener el número *major* y *minor*. Por supuesto hay compatibilidad hacia atrás, pero si un dispositivo quiere usar un mayor rango de valores deberá usar el nuevo API para el registro de dispositivos de carácter. Como hemos mencionado, el antiguo *register_ chrdev*(), sigue existiendo. Ahora podemos usar, si conocemos el número major de antemano:

```
int register_chrdev_region(dev_t from, uint32 count, char*name);
```

Aunque seguimos teniendo compatibilidad hacia atrás, ahora no registramos un par de major/minor, sino un rango de dispositivos. Desde *from* hasta *from + count,* si hay disponibles. Sino se pueden intercalar.

Sin embargo, nuestra decisión no es utilizar un *major* preestablecido, sino permitir al kernel que nos asigne uno dinámicamente. Así que usamos la función:

```
int alloc_chrdev_region(dev_t *dev, uint32 baseminor,
                        unint32 count, char* name);
```

Como se aprecia aquí, hasta ahora no hemos registrado ningún *file_ ops,* como hubiéramos hecho en otras versiones inferiores del kernel. Para ello tenemos una nueva estructura *struct cdev*, la cual es responsable de ello además de hacer visible al driver en el sistema de archivos. Esta nueva estructura además, es la encargada de aumentar en 1 el contador de uso del módulo cada vez que sea hace un *open()* y decremetnar en los *close().* Esto es útil, ya que las macros MODULE_INC_USE() y MODULE_DEC_USE() se considerán *deprecated* en esta nueva versión de kernel. *A* su vez, dentro de esta estructura contiene un *kobject,* este nuevo tipo de estructura ha sido añadido en esta versión de kernel y no es trivial de entender. Viene a ser una tipo de apaño para conseguir algo de orientación a objetos en el kernel, reusable en muchas partes... Para intentar aprender a utilizar esta nueva API nos hemos basado en los escasos ejemplos que podemos encontrar en el kernel, en este caso la inicialización de una *tty,* correspondiente al fichero: *drivers/char/tty_ io.c*

```
int tty_register_driver(struct tty_driver *driver)
{
  int error;
        int i;
  dev_t dev;
  char *s;
  void **p;
  if (driver->flags & TTY_DRIVER_INSTALLED)
    return 0;
  p = kmalloc(driver->num * 3 * sizeof(void *), GFP_KERNEL);
  if (!p)
    return -ENOMEM;
  memset(p, 0, driver->num * 3 * sizeof(void *));
  if (!driver->major) {
    error = alloc_chrdev_region(&dev, driver->minor_start, driver->num,
            (char*)driver->name);
    if (!error) {
      driver->major = MAJOR(dev);
      driver->minor_start = MINOR(dev);
    }
  } else {
    dev = MKDEV(driver->major, driver->minor_start);
    error = register_chrdev_region(dev, driver->num,
            (char*)driver->name);
  }
  if (error < 0) {
    kfree(p);
    return error;
  }
  driver->ttys = (struct tty_struct **)p;
  driver->termios = (struct termios **)(p + driver->num);
  driver->termios_locked = (struct termios **)(p + driver->num * 2);
  driver->cdev.kobj.parent = &tty_kobj;
  strcpy(driver->cdev.kobj.name, driver->name);
  for (s = strchr(driver->cdev.kobj.name, '/'); s; s = strchr(s, '/'))
```

```
    *s = '!';
  cdev_init(&driver->cdev, &tty_fops);
  driver->cdev.owner = driver->owner;
  error = cdev_add(&driver->cdev, dev, driver->num);
  if (error) {
    kobject_del(&driver->cdev.kobj);
    unregister_chrdev_region(dev, driver->num);
    driver->ttys = NULL;
    driver->termios = driver->termios_locked = NULL;
    kfree(p);
    return error;
  }
  if (!driver->put_char)
    driver->put_char = tty_default_put_char;
  list_add(&driver->tty_drivers, &tty_drivers);
 if ( !(driver->flags & TTY_DRIVER_NO_DEVFS) ) {
    for(i = 0; i < driver->num; i++)
        tty_register_device(driver, i, NULL);
  }
  proc_tty_register_driver(driver);
  return 0;
}
```

2.2.3 Obtención de la información del Slab Allocator

Pará obtener información de los aciertos/fallos del Slab Allocator, hemos modificado la función:

```
void * __cache_alloc (kmem_cache_t *cachep, int flags)
```

Para comprobar si ha habido acierto o fallo, miramos dentro de esa función si se ha podido devolver el espacio solicitado. Además de eso, tenemos que validar que se trate de nuestras caches a estudiar, es decir las comprendidas entre 2^5B-2^{17}B. Para ello debemos comprobar que el tamaño de la cache solicitada es alguno de los que queremos monitorizar, *cachep->cachesize*. Además tenemos que ser capaces de distinguir si estamos ante el tamaño de una cache general o el de alguna estructura. Para esta última comprobación bastará con saber si el nombre de la misma, *cachep->name,* comienza con la cadena "*size-*", si es así deberemos actualizar el contador de aciertos/fallos de la cache.

2.2.4 Detallar uso de driver concurrente por varios procesos

Para que varios procesos pudieran monitorizar el Slab Allocator con nuestro driver concurrentemente, deberíamos llevar a cabo la misma solución que la primera práctica. Es decir, mantener una lista con los procesos que estén monitorizando el Slab Allocator, accediendo a ella en *mutex* usando un semáforo. Cada vez que un proceso abriera el dispositivo deberíamos añadir el *pid* del proceso a esa lista. Si somos el primer proceso en monitorizar, *setearíamos* la variable *eisopeep_run* a 1, y claro está reservaríamos espacio para las estructuras donde almacenamos las estadísticas. Los restantes procesos que abrieran el dispositivo harían lo mismo pero sin modificar *esiopeep_run.* El otro aspecto

a controlar es la salida del driver, es decir la ejecución de la función *release()*. El proceso sería parecido: acceder a la lista en mutex, borrarnos de la lista, y si somos el último proceso desactivar la monitorización y liberar el espacio de las estructuras usadas para almacenar el conteo.

2.2.5 Uso de flags aditivos para los comandos

Nuestra solución es numerar los comandos de manera que no compartan ningún bit entre ellos, es decir:

```
#define CHMON   _IOW(EISOPEEP_IOC_MAGIC, 1, int)
#define CHPID   _IOW(EISOPEEP_IOC_MAGIC, 2, int)
#define RESET   _IOW(EISOPEEP_IOC_MAGIC, 4, int)
#define STOP    _IOW(EISOPEEP_IOC_MAGIC, 8, int)
#define RESUME  _IOW(EISOPEEP_IOC_MAGIC, 16, int)
```

De esta manera en un sólo entero podemos pasar más de un comando, incluso todos.

2.3 Relación de los ficheros generados y de los ficheros modificados

2.3.1 Ficheros generados

- eiso.c
- include/linux/eisopeep.h
- Makefile

2.3.2 Ficheros modifcados

- mm/slab.c

2.4 Fuente de los ficheros generados

2.4.1 eiso.c

```
/*
 * EISOPEEP   An implmentation for project 3, for the subject
 *    EISO at University of Zaragoza
 *
 *
 *
 *
 * Version:  0.1  5/01/2004
 *
 * Authors:  Aitor Acedo <460829@celes.unizar.es>
 *    Javier Uruen Val, <460821@celes.unizar.es>
 *
 *
 *This program is free software; you can redistribute it and/or
```

```
 *modify it under the terms of the GNU General Public License
 *as published by the Free Software Foundation; either version
 *2 of the License, or (at your option) any later version.
 */
#include <linux/config.h>
#include <linux/module.h>
#include <linux/kernel.h>  /* printk() */
#include <linux/version.h>
#include <linux/init.h>
#include <asm/uaccess.h>
#include <linux/kobject.h>
#include <linux/sysfs.h>
#include <linux/fs.h>      /* everything... */
#include <linux/cdev.h>    /* char dev estuff */
#include <linux/slab.h>    /* kmalloc() */
#include <linux/errno.h>   /* error codes */
#include <linux/types.h>   /* size_t */
#include <linux/proc_fs.h> /* proc system */
#include <linux/fcntl.h>  /* O_ACCMODE */
#include <linux/string.h> /* strcpy() ..*/
#include <asm/system.h>   /* cli(), *_flags */
#include "eisopeep.h"     /* local definitions */
#include <linux/eisopeep.h>
/*
 * Global var to check if the monitoring is activated
 */
extern unsigned int eisopeep_run ;
/*
 * Global var to store the old value before stopping
 * to be able to restore it
 */
unsigned int old_eisopeep;
/*
 * These pointers to structs have been declared
 * in mm/slab.c.
 */
extern t_eisopeep_ctrl *p_eisopeep_ctrl;
extern t_slabstats *p_slabstats;
dev_t dev; /* Kdev is no longer used */
struct cdev  cdev; /* Char device */
int eisopeep_major =   EISOPEEP_MAJOR;
MODULE_LICENSE("GPL");
#define EISOPEEP_USE_PROC
#ifdef EISOPEEP_USE_PROC /* don't waste space if unused */
/*
 * The proc filesystem: function to read and entry
 */
int eisopeep_read_procmem(char *buf, char **start, off_t offset,
                   int count, int *eof, void *data)
{
```

```
   int len = 0 , i;
  if (p_slabstats)
            for ( i = CACHE_POS(MIN_CACHE_SIZE);
                i < CACHE_POS(MAX_CACHE_SIZE)+1 ; i++ )

           len += sprintf(buf + len, "Cache\t%uB\t%u\t %u\n",
                        MIN_CACHE_SIZE << i, p_slabstats[i].misses,
                        p_slabstats[i].hits );
  else
    len = sprintf(buf, "Monitoring not active\n");

  return len;

}
#ifdef USE_PROC_REGISTER
static int eisopeep_get_info (char *buf, char **start, off_t offset,
                int len, int unused)
{
    int eof = 0;
    return eisopeep_read_procmem(buf, start, offset, len, &eof, NULL);
}
struct proc_dir_entry eisopeep_proc_entry = {
        0,                 /* low_ino: the inode -- dynamic */
        9, "eisopeepmem",     /* len of name and name */
        S_IFREG | S_IRUGO, /* mode */
        1, 0, 0,           /* nlinks, owner, group */
        0, NULL,           /* size - unused; operations -- use default */
        eisopeep_get_info,   /* function used to read data */
        /* nothing more */
    };
static inline void create_proc_read_entry (const char *name, mode_t mode,
                struct proc_dir_entry *base, void *read_func, void *data)
{
    proc_register_dynamic (&proc_root, &eisopeep_proc_entry);
}
static inline void remove_proc_entry (char *name, void *parent)
{
    proc_unregister (&proc_root, eisopeep_proc_entry.low_ino);
}
#endif /* USE_PROC_REGISTER */
#endif /* EISOPEEP_USE_PROC */
/*
 * Open and close
 */
int eisopeep_open (struct inode *inode, struct file *filp)
{

   if ( eisopeep_run )
     return -1; /* Busy */
```

```
   /*
    * Allocate memory to collect the stats in mm/slab.c
    */
   p_slabstats = kmalloc(sizeof(t_slabstats) * NUM_CACHES, GFP_KERNEL);
   p_eisopeep_ctrl = kmalloc(sizeof(t_eisopeep_ctrl), GFP_KERNEL);
   memset(p_slabstats, 0, sizeof(t_slabstats) * NUM_CACHES);
   memset(p_eisopeep_ctrl, 0, sizeof(t_eisopeep_ctrl));

   p_eisopeep_ctrl->pid  = 0;

   if ( !p_slabstats || !p_eisopeep_ctrl ){
       printk(KERN_ERR "Couldnt allocate space for p_slabstats\n");
    eisopeep_run = 0;
    return -ENOMEM;

   }

   eisopeep_run = 0x02;
   return 0;          /* success */

}
int eisopeep_release (struct inode *inode, struct file *filp)
{

    eisopeep_run = 0;

    /*
     * We must free p_slabstats
     */
    eisopeep_run = 0;
    kfree(p_slabstats);
    kfree(p_eisopeep_ctrl);

    p_slabstats = NULL;
    p_eisopeep_ctrl = NULL;

    return 0;

}
/*
 * Data management: read
 */
ssize_t eisopeep_read (struct file *filp, char *buf, size_t count,
                loff_t *f_pos)
{

    int retval = -EFAULT;
    printk("Read device \n");
```

```
    if ( p_slabstats == NULL )
      goto nothing;

    printk("Count is %i f_pos %i tolta size %i\n", count, (int)*f_pos, (NUM_CACHES *

    if ( count < (NUM_CACHES * sizeof(t_slabstats)) )
      goto nothing;

    printk("Here\n");

    if ( *f_pos >= (NUM_CACHES * sizeof(t_slabstats)) )
      goto nothing;

    if ( *f_pos + count >  (NUM_CACHES * sizeof(t_slabstats)) )
      count =  (NUM_CACHES * sizeof(t_slabstats))  - *f_pos;

    if (copy_to_user (buf, p_slabstats, count)) {
        goto nothing;
    }

    *f_pos += count;

    if ( *f_pos == (NUM_CACHES * sizeof(t_slabstats)) )
      *f_pos = 0;

    retval = count;
  nothing:
    return retval;
}
/*
 * The ioctl() implementation
 */
int eisopeep_ioctl (struct inode *inode, struct file *filp,
                 unsigned int cmd, unsigned long arg)
{
    int ret = 0;
    /* don't even decode wrong cmds: better returning  ENOTTY than EFAULT */
    //if (_IOC_TYPE(cmd) != EISOPEEP_IOC_MAGIC) return -ENOTTY;
    //if (_IOC_NR(cmd) > EISOPEEP_IOC_MAXNR) return -ENOTTY;

    printk("cmd %i y %i\n", _IOC_NR(cmd), _IOC_NR(CHMON));

    if ( (_IOC_NR(cmd)  & _IOC_NR(CHPID)) ) {
  p_eisopeep_ctrl->pid = arg;
  printk("EISOPEEP: Changing PID to %i \n", ( unsigned int )arg);
    }
    if ( _IOC_NR(cmd) & _IOC_NR(RESET) ) {
   // Initialize monitoring stats
  memset(p_slabstats, 0, sizeof(t_slabstats) * NUM_CACHES);
  printk("EISOPEEP: Reset stats \n");
```

```
    }
    if ( _IOC_NR(cmd) & _IOC_NR(STOP) ) {
  old_eisopeep = eisopeep_run ;
  eisopeep_run = 0;
  printk("EISOPEEP: Stop monitoring \n");
    }
    if ( _IOC_NR(cmd) & _IOC_NR(RESUME) ){
  eisopeep_run = old_eisopeep;
  printk("EISOPEEP: Resume monitoring \n");
    }

    return ret;
}

/*
 * The fops struct intialized in C99 fashion
 * This should be standard in kernels > 2.6.x
 * to be C99-compliant
 */
struct file_operations eisopeep_fops = {
    .read    =  eisopeep_read,
    .ioctl   =  eisopeep_ioctl,
    .open    =  eisopeep_open,
    .release = eisopeep_release,
};
/*
 * Finally, the module stuff
 */
int __init eisopeep_init(void)
{
    int error;

    /*
     * Register your major, and accept a dynamic number
     */
    error = alloc_chrdev_region(&dev, 0, 1, "eisopeepnew");

    if (error){
      printk("Error allocating chardev region\n");
          return error;
    }
    else {
     printk("Chardev allocated with major %i \n", MAJOR(dev));
    }

    /*
     * Cdev stuff
     */
    cdev.kobj.parent = NULL; // Hasn't got parent in sysfs
    strcpy(cdev.kobj.name, "eisopeepnew");
```

```
    /* Here we init the char device and assign  its fops.
     * Recall, In kernels < 2.6.x we did this at the function
     * register_chrdev()
     */
     cdev_init(&cdev, &eisopeep_fops);
     cdev.owner = THIS_MODULE;
     error = cdev_add(&cdev, dev, 1);
     if (error){
    kobject_put(&cdev.kobj);
        unregister_chrdev_region(dev, 1);
    return error;
     }

#ifdef EISOPEEP_USE_PROC /* only when available */
    create_proc_read_entry("eisopeepmem", 0, NULL, eisopeep_read_procmem, NULL);
#endif
    return 0; /* succeed */
}
void eisopeep_cleanup(void)
{
    unregister_chrdev_region(dev, 1);
     /* This function is responsible for deleting
        the kobject which is contained in the struct
        cdev. If it's no called it'll be quite possible
        to crash the system, for example at the moment the
         sysfs is unmounted
     */
    cdev_del(&cdev);
#ifdef EISOPEEP_USE_PROC
    remove_proc_entry("eisopeepmem", 0);
#endif
}
/*
 * We've to explict which functions are used as
 * entry point and exit point for the module
 */
module_init(eisopeep_init);
module_exit(eisopeep_cleanup);
```

2.4.2 include/linux/eisopeep.h

```
/*
 * EISOPEEP   An implmentation for project 3, for the subject
 *    EISO at University of Zaragoza
 *
 *
 *
 *
 * Version:  0.1  5/01/2004
 *
```

```
 * Authors:  Aitor Acedo <460829@celes.unizar.es>
 *    Javier Uruen Val, <460821@celes.unizar.es>
 *
 *
 *This program is free software; you can redistribute it and/or
 *modify it under the terms of the GNU General Public License
 *as published by the Free Software Foundation; either version
 *2 of the License, or (at your option) any later version.
 */
#ifndef _LINUX_EISOPEEP_H
#define _LINUX_EISOPEEP_H
#include <linux/ioctl.h>
/*
 * IOCTL commands for our char device
 */
/* Use 'j' as magic number */
#define EISOPEEP_IOC_MAGIC 'j'
#define CHMON   _IOW(EISOPEEP_IOC_MAGIC, 1, int)
#define CHPID   _IOW(EISOPEEP_IOC_MAGIC, 2, int)
#define RESET   _IOW(EISOPEEP_IOC_MAGIC, 4, int)
#define STOP    _IOW(EISOPEEP_IOC_MAGIC, 8, int)
#define RESUME  _IOW(EISOPEEP_IOC_MAGIC, 16, int)
#define ESISOPEEP_IOC_MAXNR 17
/*
 * Flags for the var eisopeep_run. Show us if
 * want to collect stats from every proccess requesting
 * memory from the Slab Allocator. Or on the other hand
 * we want to know about a unique PID.
 */
#define COLLECT_SLAB_ALL 0x1
#define COLLECT_SLAB_PID 0x2
/*
 * Caches to monitor
 */
#define MIN_CACHE_SIZE 32 //2^5B
#define MAX_CACHE_SIZE 131072 //2^17B
/*
 * Inline function used to get the index in the slabs stats
 * array from the cache size. Sizes are 32, 64, 512, 1024,
 * 2048...,131072. So they are power of two, so our
 * first position in the array [0] will be for the 32B, second one
 * [1] for the 64B, third one [2] for the 128B and so on.
 * To get the position, we need to know which bit in 'n',
 * i.e: the cache size, is set to 1. So that's what
 * this function is for. It returns the position of the
 * first 1 found in number 'n'. Note that the caller
 * has to make sure he's no passing value '0' to n.
 * Anyway, we'll never do it. Because there's no cache size 0
 * at all :)
 */
```

```
static inline unsigned int bit_pos(unsigned int n){
        short i;
        unsigned int a = 0x1;
        for( i = 0; i < 32; i++){
                if ( a & n)
                        break;
                a<<= 0x1;
        }
        return i;
}
/*
 * This macro is complementary to the above inline
 * function. We just substract 5 to get the position
 * in the array.
 */
#define CACHE_POS(X) (bit_pos(X)-5)
#define NUM_CACHES  (CACHE_POS(MAX_CACHE_SIZE) - CACHE_POS(MIN_CACHE_SIZE) + 1)
typedef struct {
        unsigned int csize;
        unsigned int hits;
        unsigned int misses;
} t_slabstats;
typedef struct {
  int bread;
  int brelse;
  int ext2_new_inodee;
  int ext2_free_inode;
  int namei;
  int open_namei;
  int bufcache_hit;
  int bufcache_miss;
}t_bufcache;
typedef struct {
  int pid;
  int type;
  void *ptr;
}t_eisopeep_ctrl;
#endif /* LINUX_EISOPEEP_H */
```

2.4.3 Makefile

```
obj-y := eiso.o
```

2.5 Fuente de los ficheros modificados

2.5.1 mm/slab.c

```
/*
 * EISO: Var used to point out if hits/misses checking
 * has to be done it. Depending on flags contained on it
```

```
 * we are able to know if we have to collect stats for an
 * unique PID or for every PID.
 * It has to be visible from the rest of kernel, even
 * our module. That's why we export it;
 */
unsigned int eisopeep_run = 0;
EXPORT_SYMBOL(eisopeep_run);
/*
 * The pointer to eisopeep_ctrl struct is declared here. Nevertheless,
 * the memory allocation/dellocation is carried out from the module.
 * This makes sense, because if there is no module loaded, there must not
 * be any waste of memory for a structure which is not going to be used
 * at all. Same with the struct t_slabstats. This structure is used to
 * store the statistics for the slab. Of course, as the symbol eisopeep_run
 * above, we have to export them.
 */
t_eisopeep_ctrl *p_eisopeep_ctrl;
t_slabstats *p_slabstats;
EXPORT_SYMBOL(p_eisopeep_ctrl);
EXPORT_SYMBOL(p_slabstats);
...
...
...
static inline void * __cache_alloc (kmem_cache_t *cachep, int flags)
{
  unsigned int hit = 0;    // EISO: We use this var to check if we get a hit or miss
  unsigned int cache_size; // EISO: Which cache size we are in.
  unsigned long save_flags;
  void* objp;
  struct array_cache *ac;
  cache_alloc_debugcheck_before(cachep, flags);
  local_irq_save(save_flags);
  ac = ac_data(cachep);
  if (likely(ac->avail)) {
    STATS_INC_ALLOCHIT(cachep);
    /*
     * EISO!!
     * We mark we are in a hit
     */
    hit = 1;
    ac->touched = 1;
    objp = ac_entry(ac)[--ac->avail];
  } else {
    STATS_INC_ALLOCMISS(cachep);
    if ( eisopeep_run & (COLLECT_SLAB_ALL | COLLECT_SLAB_PID) ) {
    }
    objp = cache_alloc_refill(cachep, flags);
  }
  local_irq_restore(save_flags);
  objp = cache_alloc_debugcheck_after(cachep, flags, objp, __builtin_return_address(0
```

```
  /*
   * EISO !!
   * Here we know if we have a  hit or miss in the slab allocator.
   * Note that the macro aboves, STATS_INC_ALLOCHIT and STATS_INC_ALLOCMISS,
   * are already used it for our purposes. However, they  get actived
   * only when the DEBUG mode for the slab is selected. Well,
   * we made the decision of not overload the Slab Alloctor with
   * all that debug and implent only what we need.
   * The first step is to check if we have to collect any stats, if
   * we do then we have to make sure if the cache which is being used
   * belongs to any of those we are interested in. After finding out
   * that, we have to index in the p_slabstats array to update
   * the right cache size.
   */
  if ( eisopeep_run & (COLLECT_SLAB_ALL | COLLECT_SLAB_PID) ) {
    //Get the index
    cache_size = CACHE_POS(cachep->objsize);
    //Check if index belongs to our caches
    if ( (cache_size > CACHE_POS(MAX_CACHE_SIZE)) ||
         (cache_size < CACHE_POS(MIN_CACHE_SIZE))  )
           goto out;
 /*
     * Check if we have screwed it up. If the
     * next condition is true, we are in troubles
     * because our module does not exit any longer or
     * something bad is happening.
     */
    if ( p_slabstats == NULL || p_eisopeep_ctrl == NULL ) {
      printk(KERN_ERR "HOUSTON WE'VE GOT A PROBLEM!!!\n");
      goto out;
    }
    /*
     * Check if pid has to match
     */
    if ( eisopeep_run == COLLECT_SLAB_PID  &&
         p_eisopeep_ctrl->pid != current->pid)
      goto out;
    if ( eisopeep_run == COLLECT_SLAB_PID  &&
         p_eisopeep_ctrl->pid == current->pid){
      printk("Hit: %i, size %i, index %i,  pid %i\n", hit, cachep->objsize, cache_siz
    }
    /*
     * Check what we have to update
     */
    if(hit)
      p_slabstats[cache_size].hits++;
    else
      p_slabstats[cache_size].misses++;
  }
out:
```

```
  return objp;
}
...
...
...
```

2.6 Fuente de los programas de prueba

```
/*
 * TESIT   An implmentation for project 3, for the subject
 *    EISO at (C.P.S) University of Zaragoza
 *    Code to test this project, a kernel module to monitor
 *    hits/missess on the slab allocator, user space code
 *    using ptrace().
 *
 * Version:  0.1  5/01/2004
 *
 * Authors:  Aitor Acedo
 *    Javier Uruen Val, <460821@celes.unizar.es>
 *
 *
 *  This program is free software; you can redistribute it and/or
 *  modify it under the terms of the GNU General Public License
 *  as published by the Free Software Foundation; either version
 *  2 of the License, or (at your option) any later version.
 */
#include <sys/types.h>   /* fork */
#include <unistd.h>      /* fork */
#include <sys/ptrace.h> /* ptrace */
#include <errno.h>  /* perror */
#include <stdio.h>  /* error, fprintf */
#include <sys/wait.h>  /* wait */
#include <sys/types.h>  /* open */
#include <sys/stat.h>
#include <fcntl.h>
#include <sys/ioctl.h>
#include <linux/eisopeep.h>
#define EISODEV "/dev/eisopeep"
/*
 * Macro to compute the number of bytes that should
 * be read to get the whole statistic.
 */
#define BYTESTOREAD (NUM_CACHES * sizeof(t_slabstats))
#define MAXARGS 5
int main( int argc,  char *argv[]){
  int pid, fd, status, i;
  char *args[MAXARGS];
  t_slabstats stats[NUM_CACHES];

  if ( argc < 2 ){
```

```
  fprintf(stderr, "Usage: %s <file> [parameters] \n", argv[0] );
  return -1;
}

//Fork
if ( !(pid = fork()) ){
  //Children process is set for tracing
  ptrace(PTRACE_TRACEME, 0, 0, 0);
  execvp(argv[1], &argv[1]);
//  perror("exit"); return -1;
}
/*
 * Parent is the tracer process.
 * Wait for traced process to stop
 */
wait(&status);

fd = open(EISODEV, 0);
if ( fd < 0 ) {
  fprintf(stderr, "Damn it! Cannot open device: %s\n", EISODEV);
  return -1;
}

/*
 * Ioctl() call, we change the pid for monitoring and reset the stats
 */
if ( ioctl(fd, CHPID | RESET, pid) ) {
  fprintf(stderr, "Ough! There's something funny in ioctl():(\n");
  return -1;
}
/*
 * The children process is stopped until executing the next call,
 * where we allow the traced process to keep running
 */
ptrace(PTRACE_CONT, pid, 0, 0);
/*
 * We are gonna wait until traced process stops. We use the
 * flag WNOHANG to point out we have to stop waitting if
 * there is no children process to wait.
 */
wait4(pid, &status, 0, NULL);
/*
 * Now we are going to read the device to gather statistics.
 * We are supposed to read NUM_CACHES * sizeof(t_slabstats) bytes.
 * If we dont read this amount of bytes we consider as
 * something went wrong.
 */
//getchar();
if ( read(fd, stats, BYTESTOREAD) != BYTESTOREAD ){
  fprintf(stderr, "I'm really  sorry, couldnt read the stats\n");
```

```
    return -1;
  }

   for (i = 0; i < NUM_CACHES; i++ )
      printf("Cache %i: %i - %i\n", MIN_CACHE_SIZE << i,
                  stats[i].misses, stats[i].hits);

  close(fd);

  return 1;
}
```