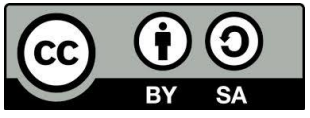
Práctica 2: Llamadas al Sistema

DSO - Curso 2013-2014



J.C. Sáez, M. Prieto
High Performance Computing Division
ArTeCs Group
Universidad Complutense de Madrid

Contenido

1 Introducción

2 Ejercicios

3 Práctica

Práctica 2: Llamadas al sistema

Objetivos

- Familiarizarse con:
  - Implementación de llamadas al sistema en Linux y su procedimiento de invocación
  - Compilación del kernel Linux y creación de parches
  - Exportación de símbolos (funciones) para su uso desde módulos del kernel

Contenido

1 Introducción

2 Ejercicios

3 Práctica

Ejercicios

- La entrada `/proc/cpuinfo` permite obtener información acerca de las CPUs del sistema

Terminal

```
dsouser@debian:~$ cat /proc/cpuinfo
processor   : 0
vendor_id   : GenuineIntel
cpu family  : 6
model       : 23
model name  : Intel(R) Xeon(R) CPU           E5450  @ 3.00GHz
stepping    : 10
cpu MHz     : 2003.000
cache size  : 6144 KB
physical id : 0
siblings    : 4
core id     : 0
cpu cores   : 4
apicid      : 0
initial apicid  : 0
fpu     : yes
fpu_exception   : yes
cpuid level : 13
wp      : yes
flags       : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush dts acpi mmx fxsr sse
bogomips    : 5984.67
clflush size    : 64
cache_alignment : 64
address sizes   : 38 bits physical, 48 bits virtual
power management:
...
```

Ejercicios

Ejercicio 1

- Estudiar la implementación del programa cpuinfo.c
  - Este programa imprime por pantalla el contenido de `/proc/cpuinfo` haciendo uso de las llamadas al sistema `open()` y `close()`, y las funciones `printf()` y `syscall()`.
  - ¿Qué llamada al sistema invoca el programa mediante `syscall()`?
  - Reescribir el programa anterior reemplazando las llamadas a `open()`, `close()` y `printf()` por invocaciones a syscall() que tengan el mismo comportamiento.

Contenido

1 Introducción

2 Ejercicios

3 Práctica

Sistema de ficheros /proc

- El sistema de ficheros /proc de Linux resulta de gran utilidad para distintos subsistemas del kernel
  1. Interfaz para comunicación entre kernel/módulos y el modo usuario
  2. Interfaz extensible
     - Los módulos pueden crear nuevas entradas en cualquier directorio del árbol
- En otros SSOO tipo UNIX /proc no existe o no puede emplearse como mecanismo de interacción de “proposito general” entre modo usuario y modo kernel
  - Inexistente Mac OS X
  - En Solaris, /proc se emplea únicamente para alojar información sobre los procesos
    - Los módulos cargables (también existentes en Solaris) no usan el /proc como interfaz *ad-hoc* con el usuario o las aplicaciones

Especificación de la práctica (I)

- En esta práctica construiremos un mecanismo alternativo a /proc llamado KIFS (*Kernel InterFace System*)

Características de KIFS

- El kernel contiene una lista enlazada de entradas `kifs_entry_t` que tienen asociado:
  1. Un nombre (ej.: *"mi_entrada"*)
     - Identificador único
  2. Una función de escritura (*write callback*)
  3. Una función de lectura (*read callback*)
- Las funciones de lectura/escritura se comportan como las *callbacks* de las entradas de /proc

```
typedef int (write_kifs_t)(const char *user_buffer, unsigned int
    maxchars, void *data);
typedef int (read_kifs_t)(char *user_buffer, unsigned int maxchars
    , void *data);
```

Especificación de la práctica (II)

Características de KIFS (cont.)

- Los procesos de usuario pueden invocar la operación de lectura/escritura de una entrada mediante una nueva *syscall*

```
int kifs(const char* entryname,unsigned int read_write,
                char* user_buffer,unsigned int maxchars);
```

- **Parámetros**
  - `entryname`: nombre de la entrada cuya *callback* queremos invocar
  - `read_write`: 0 → lectura, 1 → escritura
  - `user_buffer` y `maxchars`: parámetros que se pasan a *callback*
- **Valor de retorno**
  - Similar a `read()`/`write()` → Número de caracteres escritos/leidos en el/del buffer o negativo (error).

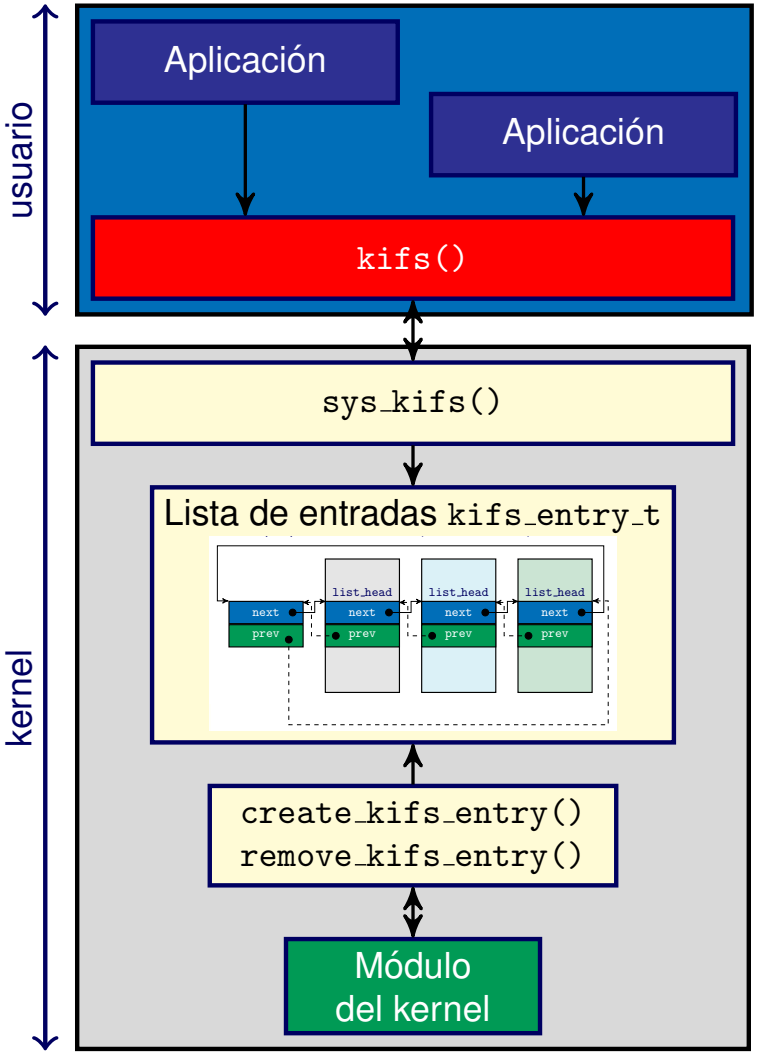
Especificación de la práctica (III)

Características de KIFS (cont.)

- El sistema KIFS, que ha de implementarse en el kernel, exporta las funciones `create_kifs_entry()` y `remove_kifs_entry()`
  - Permiten a módulos y otros subsistemas del kernel añadir/eliminar entradas

```
kifs_entry_t* create_kifs_entry(const char* entryname,
                                read_kifs_t *read_kifs,
                                write_kifs_t *write_kifs,
                                void* data);
int remove_kifs_entry(const char* entry_name);
```

Visión global de KIFS



Componentes del KIFS

KIFS vs. /proc

Propiedad	`/proc`	KIFS
Callbacks de lectura y escritura	Sí	Sí
Interfaz Extensible	Sí	Sí
Uso de `copy_from_user()` y `copy_to_user()`	Solo es necesario invocar `copy_from_user()` desde write *callback*	Necesario usar ambas funciones. Siempre se pasa puntero al espacio de usuario
Organización de entradas	Jerárquica (Árbol)	Lineal (Lista enlazada)
Entradas tienen presencia en sistema de ficheros	Sí	No
Soporte de `lseek()`	Sí	No (No hay fichero asociado → no hay puntero de posición)
Invocación callbacks	Llamadas al sistema `read()` y `write()`	Llamada al sistema `kifs()`

Partes de la práctica

(Parte A.) Implementación de KIFS en el kernel

- Durante el proceso de arranque del SO, se crearán dos entradas:
  - `list`: (Sólo lectura) Imprime un listado de las entradas KIFS registradas en el sistema (recorrido de la lista)
  - `clipboard`: (Lectura/Escritura) Comportamiento similar al ejemplo 'clipboard.c' de la práctica anterior.
    - Diferencia: Sólo puede leerse el contenido del `clipboard` con una sola llamada (truncar cadena con `maxchars` caracteres)

(Parte B.) Crear un módulo que haga uso de KIFS

- Al cargar el módulo, éste creará una entrada kifs llamada `counter` (`create_kifs_entry()`)
  - Escritura → Incremento de un contador (inicialmente a 0)
  - Lectura → se escribirá en el buffer del usuario el valor del contador (Usar `sprintf()`)
- La entrada `counter` se eliminará al descargar el módulo (`remove_kifs_entry()`)

Implementación (I)

Se proporcionan 2 ficheros fuente

- `kifs.h`: Contiene definiciones de funciones y tipos de datos de KIFS
  - Se ha de copiar en `$LINUX_SOURCE/include/linux`
- `kifs_invoke.c`: Programa de modo usuario para invocar `kifs()`
  - Modo de uso:
    - (Lectura): `./kifs_invoke -r <nombre_entrada>`
    - (Escritura): `./kifs_invoke -w <nombre_entrada> <cadena>`

Implementación (II)

Definiciones <include/linux/kifs.h>

```
typedef struct
{
    char entryname[MAX_KIFS_ENTRY_NAME_SIZE]; /* name of the entry */
    read_kifs_t *read_kifs;                    /* read callback */
    write_kifs_t *write_kifs;                  /* write callback */
    void *data;                    /* argument passed to the callbacks */
    struct list_head links;   /* prev and next links (linked list) */
}kifs_entry_t;

/* OPCODES for sys_kifs() */
enum {
    KIFS_READ_OP=0,
    KIFS_WRITE_OP=1,
    KIFS_NR_OPS};
```

Implementación (III)

Interfaz de operaciones <include/linux/kifs.h>

```
/* Global initialization function (invoked in the boot process) */
void init_kifs_entry_set(void);

/* Syscall */
asmlinkage long sys_kifs(const char* entry_name,unsigned int op_mode,
     char* user_buffer,unsigned int maxchars);

/* == API for kernel modules and other kernel components == */
kifs_entry_t* create_kifs_entry(const char* entryname,
                                read_kifs_t *read_kifs,
                                write_kifs_t *write_kifs,
                                void* data);
int remove_kifs_entry(const char* entry_name);
```

Implementación (IV)

Pasos a seguir

- Copiar `kifs.h` en `$KERNEL_SOURCE/include/linux`
- Crear un fichero `kifs.c` en `$KERNEL_SOURCE/kernel`
  - Este fichero contiene la implementación de las funciones declaradas en `kifs.h` y las estructuras (variables globales) necesarias
  - Exportar las dos funciones que constituyen el API de los módulos
    - Incluir sentencias `EXPORT_SYMBOL()` al final de cada función.
    - Ej: `EXPORT_SYMBOL(create_kifs_entry);`
- Modificar el fichero `$KERNEL_SOURCE/kernel/Makefile` para que `kifs.c` se compile durante el proceso de construcción del kernel

Implementación (V)

Pasos a seguir (cont.)

- Modificar los ficheros necesarios del kernel para incluir la llamada `sys_kifs` en la tabla de llamadas al sistema
  - Para IA32 `sys_kifs` $\rightarrow$ número 345
- Invocar `init_kifs_entry_set()` desde `sched_init()` [kernel/sched.c]
  - Necesario para inicializar las estructuras de datos de kifs durante el proceso de arranque del SO
  - Esta función crea también las entradas 'list' y 'clipboard'
- Compilar el kernel
- Instalar los paquetes de *kernel* y *headers* en la máquina virtual
- Reiniciar el sistema

Ficheros a añadir/modificar en el kernel

- kernel/kifs.c
- include/linux/kifs.h
- kernel/Makefile
- kernel/sched.c
- arch/x86/include/asm/unistd_32.h
- arch/x86/kernel/syscall_table_32.S
- (opcional) arch/x86/include/asm/unistd_64.h

Gestión de memoria en KIFS

- Por motivos de robustez de la implementación no usaremos `vmalloc()` y `vfree()` para gestionar la memoria
  - `vmalloc()` es bloqueante
  - Algunos puntos del código del núcleo no permiten invocar funciones bloqueantes
- KIFS soportará hasta un máximo de 10 entradas cuya memoria se reservará de forma estática
  - *Pool* de entradas (*array* definido estáticamente)
  - Lista doblemente enlazada de entradas "ocupadas"

```
#define MAX_KIFS_ENTRIES 10

kifs_entry_t pool[MAX_KIFS_ENTRIES];
struct list_head entry_list; /* Cabeza de la lista */
```

Gestión de memoria en KIFS (Cont.)

- Varias alternativas para la gestión de las entradas libres
  1. Mapa de bits
     - Un bit por cada elemento del array de entradas (*pool*)
     - *unsigned int* $\rightarrow$ Cada bit valdrá 0 (libre) ó 1 (ocupada)
  2. Lista enlazada de entradas libres
     - Inicialmente lista de “libres” (10 elementos) y lista de “ocupadas” vacía

```
struct list_head free_list; /* Lista de entradas libres */
```

sys_kifs

Pseudocódigo

```
asmlinkage long sys_kifs(const char* entry_name,unsigned int op_mode,
     char* user_buffer,unsigned int maxchars)
{
    copiar entryname al espacio de kernel (copy_from_user())
    y añadir '\0' al final de la cadena;

    entry=Buscar entrada entry_name en la lista enlazada;

    Si la entrada no existe -> return -EINVAL;

    Si la operación solicitada no está implementada (read_kifs ó
         write_kifs == NULL) -> return -EINVAL;

    Ejecutar la operación solicitada mediante el puntero a función
         correspondiente y devolver el resultado como valor de retorno
         de sys_kifs();
}
```

Parte A: Ejemplo de ejecución

terminal

```
dsouser@debian$ gcc -g kifs_invoke.c -o kifs_invoke
dsouser@debian$ ./kifs_invoke
Usage: ./kifs_invoke [-r|-w] <entry_name> [value]
dsouser@debian$ ./kifs_invoke -r list
*** READING list ENTRY **
clipboard
list

*****
dsouser@debian$ ./kifs_invoke -w clipboard Hoooola
dsouser@debian$ ./kifs_invoke -r clipboard
*** READING clipboard ENTRY **
Hooooola
*****
dsouser@debian$
```

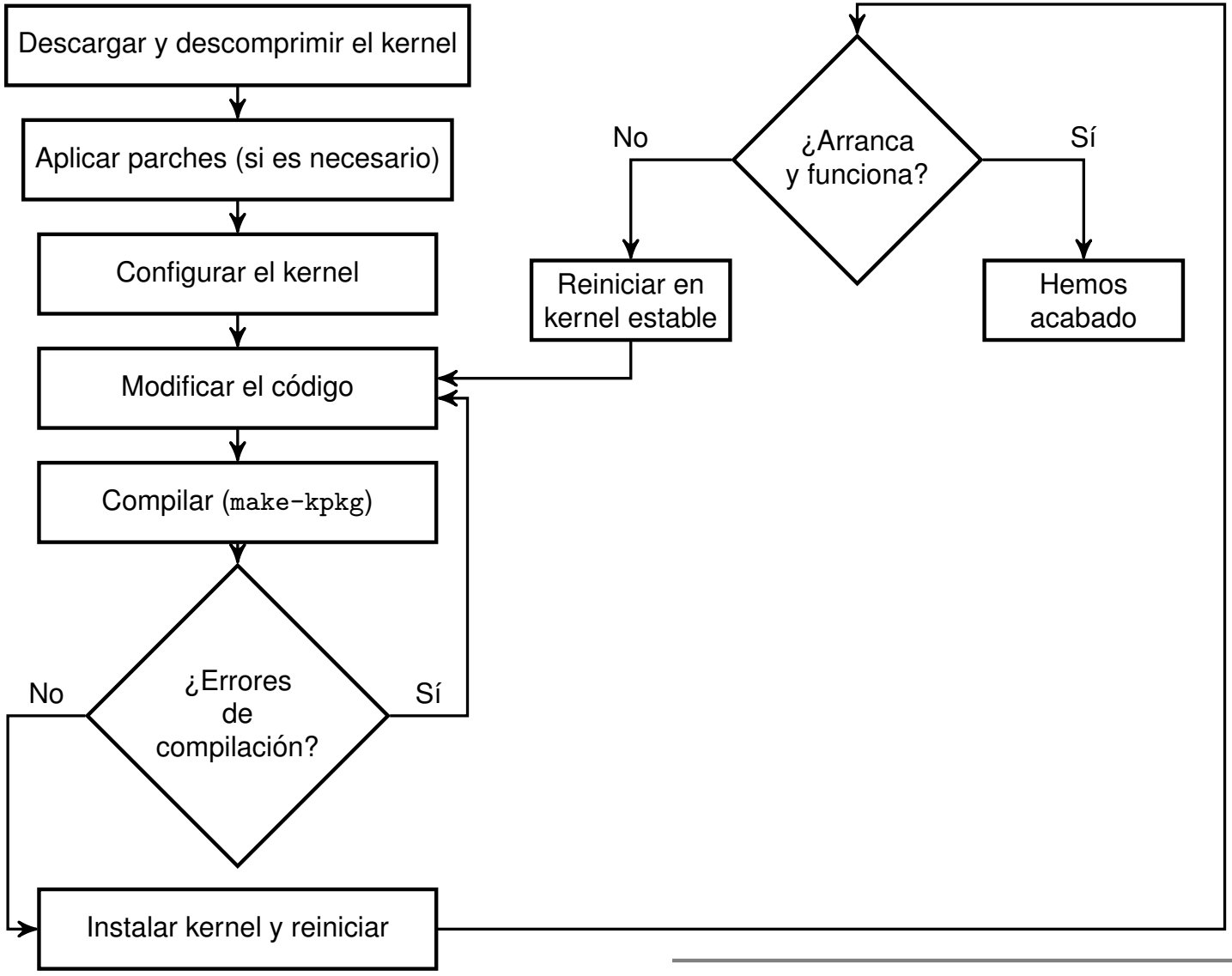
Parte B: Ejemplo de ejecución

terminal

```
dsouser@debian$ sudo insmod kifsmodule.ko
dsouser@debian$ ./kifs_invoke -r list
*** READING list ENTRY **
clipboard
list
counter

*****
dsouser@debian$ ./kifs_invoke -r counter
*** READING counter ENTRY **
0
*****
dsouser@debian$ ./kifs_invoke -w clipboard foo;  ./kifs_invoke -w clipboard foo
dsouser@debian$ ./kifs_invoke -r counter
*** READING counter ENTRY **
2
*****
dsouser@debian$ sudo rmmod kifsmodule
dsouser@debian$ ./kifs_invoke -r list
*** READING list ENTRY **
clipboard
list
*****
```

Flujo de trabajo con el kernel



Algunos consejos ...

1. Crear primero una versión “Hola Mundo” de la llamada al sistema kifs() para verificar que el procedimiento de llamada es correcto
   - Código: `printk(KERN_DEBUG "Hola DSO\n"); return 0;`
2. Depurar KIFS usando un módulo del kernel
   - **Ejemplo:** crear un módulo del kernel que exporte una entrada /proc/kifs
     - Al escribir en /proc/kifs se selecciona una entrada KIFS específica

       `echo mi_entrada > /proc/kifs`
     - Al leer desde /proc/kifs se ejecuta la función de lectura de la entrada KIFS seleccionada anteriormente

       `cat /proc/kifs`
3. Una vez depurado $\rightarrow$ introducir los cambios en el kernel y recompilar
   - No olvidar incluir la invocación de la función de inicialización `init_kifs_entry_set()` en `sched_init()` [kernel/sched.c]

Partes opcionales (I)

- **(Opcional 1)** Extender KIFS del siguiente modo:
  - Añadir un campo `description` de tipo cadena de caracteres a `kifs_entry_t`
    - Este campo almacenará una descripción de la entrada kifs
    - El nuevo campo se inicializará adecuadamente pasándolo como parámetro a `create_kifs_entry()`
  - Modificar KIFS de tal forma que el campo `description` de una entrada específica pueda leerse desde el programa 'kifs_invoke'
    - Ej: `kifs_invoke -d <nombre_entrada>`
- **(Opcional 2)** Ampliar el módulo de la parte B para que cree una entrada KIFS llamada 'modlist' que tenga el mismo comportamiento que la entrada /proc de la Práctica 1

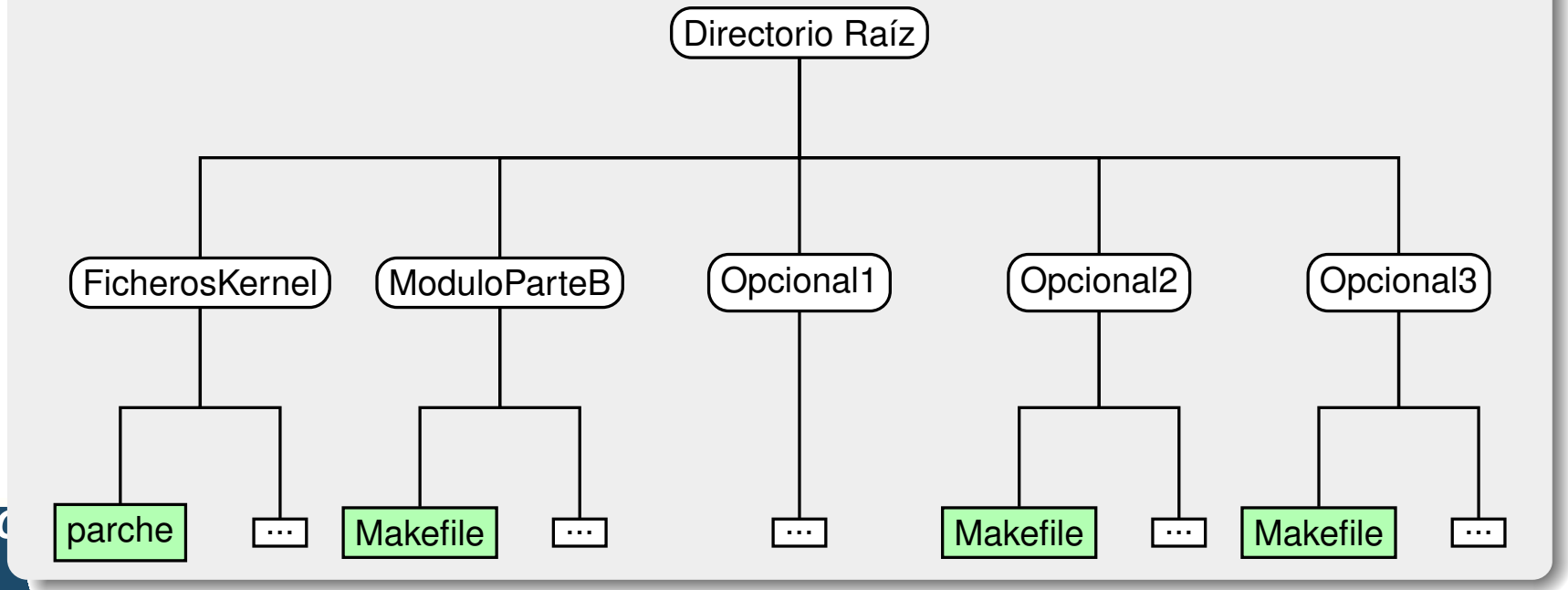
Partes opcionales (II)

- **(Opcional 3)** Implementar un módulo que cree *k* entradas KIFS con nombre <prefijo>0, <prefijo>1, . . . ,<prefijo>k-1
  - Tanto el prefijo (cadena de caracteres) como el número de entradas a crear *k* se pasarán como parámetros al módulo
  - Cada entrada exhibirá el mismo comportamiento que la entrada 'counter' de la Parte B de la práctica y tendrá asociado un contador privado
  - Si no es posible crear *k* entradas por falta de espacio, el módulo eliminará las entradas creadas previamente antes de devolver un error en tiempo de carga

Entrega de la práctica

- A través del Campus Virtual
  - Hasta el 22 de noviembre
- Obligatorio mostrar el funcionamiento después de hacer la entrega

Estructura entrega (en un fichero comprimido .tar.gz o .zip)



Licencia

DSO - Práctica 2: Llamadas al Sistema
Versión 0.1

Este documento (o uno muy similar) está disponible en
`https://cv2.sim.ucm.es/moodle/course/view.php?id=37161`