Práctica Final

LIN - Curso 2021-2022

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

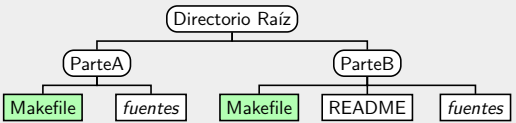
Práctica Final

- Existen 2 variantes de la práctica final
  - Cada estudiante del mismo grupo tiene asociada una variante distinta
  - El **listado con las variantes asignadas** se encuentra **en el Campus Virtual**

Entrega de la práctica

- Entrega a través del Campus Virtual
  - Entrega INDIVIDUAL hasta el 18 de enero a las 12:30 de la mañana
  - No se permiten entregas tardías
- Defensa individual obligatoria el 18 de enero en Lab 3
  - Defensa variante 1 o solape con otro examen posterior (p.ej. CAL): 13:30h
  - Defensa variante 2: 14:45h
- Esta práctica es "opcional" (Máximo 2.5 puntos+)
  - Se permiten "entregas parciales" (p. ej., sólo un apartado)

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



Práctica Final

Parte A (sobre MV de Debian)

- **Variante 1**: Extender módulo del kernel de la Práctica 4B para que gestione múltiples "FIFOs"
- **Variante 2**: Extender módulo del kernel de la Práctica 5 para que gestione múltiples generadores de códigos

Parte B (sobre MV de Android-x86)

1. Hacer funcionar el módulo de la parte A en MV de Android-x86
2. Crear programa multihilo en C que interactúe con el módulo del kernel desarrollado
   - Recomendable depurar el programa de usuario en Linux y luego probarlo sobre Android

Práctica Final

Parte A (Ambas variantes)

- Al cargar el módulo, éste creará un directorio `$PROC_DIRECTORY` en el sistema de ficheros /proc
  - Variante 1: PROC_DIRECTORY=`/proc/multififo`
  - Variante 2: PROC_DIRECTORY=`/proc/multicode`
- También se crearán dos entradas /proc en `$PROC_DIRECTORY`:
  1. `"test"`: sustituye a la entrada que existía en el módulo original (`/proc/fifoproc` o `/proc/codetimer`)
     - Mismo comportamiento aunque distinta implementación
     - Tendrá asociada un FIFO o un generador de códigos
  2. `"admin"`: permite crear/destruir entradas /proc en $PROC_DIRECTORY con misma semántica que la entrada `"test"` pero con sus propias estructuras asociadas (FIFO o generador de códigos)

Práctica Final

- Directorios a crear:
  - Variante 1: PROC_DIRECTORY=/proc/multififo
  - Variante 2: PROC_DIRECTORY=/proc/multicode

Comportamiento entrada `admin`

- `$ echo new <name> > $PROC_DIRECTORY/admin`
  - crear nueva entrada (como "test") pero con nombre `name`
- `$ echo delete <nombre> > $PROC_DIRECTORY/admin`
  - Eliminar entrada existente con nombre `name` (como "test")

Particularidad variante 2 de la práctica

- Habrá una entrada extra `/proc/multicode/codeconfig` que permitirá modificar/-consultar el valor de los parámetros configurables de los generadores de códigos
  - Por simplicidad, puede asumirse que todos los generadores de códigos tienen parámetros compartidos (mismo conjunto de variables globales)

Práctica Final

Requisitos Parte A

- Cada vez que se cree una entrada /proc escribiendo en la entrada `admin`, se creará también una estructura privada asociada a la nueva entrada (FIFO o generador de códigos)
  - La mayor parte de las variables globales existentes en el módulo original serán ahora miembros de una estructura a definir en el módulo
  - La entrada `test` tendrá asociada también su propia estructura privada
  - Las callbacks de la entrada `test` así como aquellas de las entradas creadas dinámicamente se implementarán con las mismas funciones
    - Mismas callbacks pero sobre distintos datos
    - No se permite replicar código
- Al eliminar una entrada, la memoria asociada a la estructura correspondiente debe liberarse

Práctica Final

Requisitos Parte A (Cont.)

- El módulo del kernel exportará dos parámetros, a establecer en tiempo de carga:
  1. `max_entries`: número máximo de entradas `/proc` que puedan existir de forma simultánea (contando la entrada `test`).
     - El módulo debe denegar la creación de un número de entradas por encima de ese máximo, devolviendo un código de error
     - Valor por defecto: 5
  2. `max_size`: tamaño máximo de buffers circulares usados en el código
     - Tamaño especificado en bytes (ha de ser potencia de 2). Valor por defecto=32
- El módulo no debe poder cargarse (error en `init_module()`) si el usuario establece valores no permitidos en los parámetros
- Para más detalles sobre parámetros en módulos, repasar Ejercicio 3 de la Práctica 1 o la información de este enlace

Práctica Final

Requisitos Parte A (Cont.)

- Se ha de mantener una estructura de datos (p. ej., lista enlazada) para llevar la cuenta de las entradas /proc y estructuras privadas creadas dinámicamente
- Al descargar el módulo, todas las entradas /proc creadas (incluido el directorio) deben destruirse y la memoria asociada a las estructuras asociadas debe liberarse
- La implementación del módulo del kernel debe ser SMP-safe
  - Se valorará la eficiencia/calidad de la implementación
  - Nota: `remove_proc_entry()` es una función bloqueante
    - Bloquea al flujo del kernel invocador hasta que la entrada /proc deja de estar en uso (en el caso de que lo esté)

Parte A (V1): Ejemplo de ejecución

terminal 1

```
kernel@debian:~/Final/ParteAV1$ sudo insmod multififo.ko max_entries=3
kernel@debian:~/Final/ParteAV1$ ls /proc/multififo/
admin  test
kernel@debian:~/Final/ParteAV1$ cd ~/FifoTest/
kernel@debian:~/FifoTest$ ./fifotest -f /proc/multififo/test -s < test.txt
kernel@debian:~/FifoTest$
```

terminal 2

```
kernel@debian:~/FifoTest$ ./fifotest -f /proc/multififo/test -r
En un lugar de la Mancha, de cuyo nombre no quiero acordarme, no ha mucho tiempo
que vivía un hidalgo de los de lanza en astillero, adarga antigua, rocín flaco y
galgo corredor. Una olla de algo más vaca que carnero, salpicón las más noches,
duelos y quebrantos los sábados, lentejas los viernes, algún palomino de añadidura
los domingos, consumían las tres partes de su hacienda. El resto della concluían
sayo de velarte, calzas de velludo para las fiestas con sus pantuflos de lo mismo,
los días de entre semana se honraba con su vellori de lo más fino. Tenía en su casa
una ama que pasaba de los cuarenta, y una sobrina que no llegaba a los veinte, y
un mozo de campo y plaza, que así ensillaba el rocín como tomaba la podadera...
kernel@debian:~/FifoTest$
```

Parte A (V1): Ejemplo de ejecución (cont.)

terminal 1

```
kernel@debian:~/FifoTest$ echo new fifoA > /proc/multififo/admin
kernel@debian:~/FifoTest$ echo new fifoB > /proc/multififo/admin
kernel@debian:~/FifoTest$ ls /proc/multififo
admin  test  fifoA  fifoB
kernel@debian:~/FifoTest$ ./fifotest -f /proc/multififo/fifoA -s
Esto va por fifoA
kernel@debian:~/FifoTest$
```

terminal 2

```
kernel@debian:~/FifoTest$ ./fifotest -f /proc/multififo/fifoA -r
Esto va por fifoA
kernel@debian:~/FifoTest$
```

terminal 3

```
kernel@debian:~/FifoTest$ ./fifotest -f /proc/multififo/fifoB -s
Esto va por fifoB
kernel@debian:~/FifoTest$
```

terminal 4

```
kernel@debian:~/FifoTest$ ./fifotest -f /proc/multififo/fifoB -r
Esto va por fifoB
kernel@debian:~/FifoTest$
```

Parte A (V1): Ejemplo de ejecución (cont.)

terminal

```
kernel@debian:~/Final/ParteAV1$ echo delete no_existe > /proc/multififo/admin
bash: echo: error de escritura: No existe el fichero o el directorio
kernel@debian:~/Final/ParteAV1$ echo delete test > /proc/multififo/admin
kernel@debian:~/Final/ParteAV1$ ls /proc/multififo
admin  fifoA  fifoB
kernel@debian:~/Final/ParteAV1$ echo delete fifoB > /proc/multififo/admin
kernel@debian:~/Final/ParteAV1$ ls /proc/multififo
admin  fifoA
kernel@debian:~/Final/ParteAV1$ sudo rmmod multififo
```

Parte A (V2): Ejemplo de ejecución

Terminal

```
kernel@debian:~/Final/ParteAV2$ sudo insmod multicode.ko max_entries=4
kernel@debian:~/Final/ParteAV2$ ls /proc/multicode
admin  codeconfig  test
kernel@debian:~/Final/ParteAV2$ cat /proc/multicode/codeconfig
timer_period_ms=500
emergency_threshold=75
codeFormat=aA00
kernel@debian:~/Final/ParteAV2$ cat /proc/multicode/test
bH41
hE63
aY52
zX02
yZ97
zD38
gT55
nJ09
jJ53
hY21
^C
```

Parte A (V2): Ejemplo de ejecución (cont.)

Terminal

```
kernel@debian:~/Final/ParteAV2$ echo new random0 > /proc/multicode/admin
kernel@debian:~/Final/ParteAV2$ echo new random1 > /proc/multicode/admin
kernel@debian:~/Final/ParteAV2$ cat /proc/multicode/random0
zD38
gT55
nJ09
jJ53
hY21
...
```

terminal 2

```
kernel@debian:~/Final/ParteAV2$ cat /proc/multicode/random1
aB21
zT42
hE27
aA07
bU79
hG24
...
```

Parte A (V2): Ejemplo de ejecución (cont.)

Terminal

```
kernel@debian:~/Final/ParteAV2$ echo delete no_existe > /proc/multicode/codeconfig
bash: echo: error de escritura: No existe el fichero o el directorio
kernel@debian:~/Final/ParteAV2$ echo delete test > /proc/multicode/admin
kernel@debian:~/Final/ParteAV2$ ls /proc/multicode/
admin  codeconfig random0  random1
kernel@debian:~/Final/ParteAV2$ echo delete random0 > /proc/multicode/admin
kernel@debian:~/Final/ParteAV2$ ls /proc/multicode/
admin  codeconfig random1
kernel@debian:~/Final/ParteAV2$ sudo rmmod multicode
kernel@debian:~/Final/ParteAV2$
```

Parte A: Pistas

Creación y destrucción de directorios en /proc

- La función `proc_mkdir()` permite crear un directorio `name` en el sistema de ficheros `/proc`. Retorna descriptor del directorio creado

```
struct proc_dir_entry *proc_mkdir(const char *name, struct proc_dir_entry *parent)
```

- El descriptor devuelto se pasa como tercer parámetro de la función `proc_create()` o `proc_create_data()` para crear una entrada /proc que "cuelge" del directorio creado
- Para destruir un directorio dentro de /proc es preciso utilizar `remove_proc_entry()`
  - El directorio ha de estar vacío para que la eliminación tenga éxito
- Se proporciona módulo de ejemplo `clipboard_dir.c` que crea entrada `/proc/test/clipboard`

Parte A: Pistas (cont.)

Asociar datos de uso privado (`private_data`) a entrada /proc

- Al crear una entrada /proc con la función `proc_create_data()` es posible asociar cualquier tipo de datos privados a la entrada /proc
  - `private_data` puede ser un puntero a un entero, un puntero a una estructura, una cadena de caracteres,...

```
struct proc_dir_entry *proc_create_data(const char *name,
                umode_t mode,
                struct proc_dir_entry *parent,
                const struct file_operations *ops,
                void *private_data);
```

- Desde el código de las callbacks de una entrada /proc podemos recuperar el campo `private_data` a partir del parámetro `struct file*` de la siguiente forma:

```
ssize_t my_read_callback (struct file *filp, char __user *buf, size_t len, loff_t *off) {
  mi_tipo_de_datos* private_data=(mi_tipo_de_datos*)PDE_DATA(filp->f_inode);
   ....
}
```

Parte A: Pistas (cont.)

Asociar datos de uso privado (`private_data`) a entrada /proc

- Este mecanismo permite usar una misma función callback para distintas entradas /proc
  - Misma operación, pero sobre distintos datos ...

Parte A: Pistas (cont.)

¿Y qué ocurre con las funciones del timer y tarea diferida? (variante 2)

- Se puede recuperar fácilmente a partir del parámetro y usando macro `containerof()`

```
typedef struct{
  struct timer_list t;
  struct work_struct ws;
  struct kfifo cbuffer;
  spinlock_t lock;
  ...
} mi_tipo_de_datos;

void fire_timer (struct timer_list *timer) {
  mi_tipo_de_datos* data=containerof(timer,mi_tipo_de_datos,t);
   ....
}

void copy_items_into_list (struct work_struct *work) {
  mi_tipo_de_datos* data=containerof(work,mi_tipo_de_datos,ws);
   ....
}
```

Práctica Final: Parte B

Parte B

1. Mostrar al profesor en el laboratorio el funcionamiento del módulo de la parte A de la práctica sobre la máquina virtual de Android-x86
   - Puede ser necesario alterar el código introduciendo compilación condicional para añadir compatibilidad con kernel v4.9x de la MV de Android-x86
   - Adaptaciones necesarias en API de /proc y kernel timers (ver ejemplos en (FicherosAndroid.tgz)
2. Desarrollar un programa de usuario multihilo en C para Android-x86 que interactúe mediante llamadas al sistema con el módulo del kernel desarrollado en la parte A
   - Compilación del programa para Android-x86 (p.ej., `user.c`)

```
$ gcc -g -Wall -static -lpthread user.c -o user
```

Práctica Final: Parte B

- Como vimos en la clase del 1 de diciembre, se precisa tener el kernel Linux de la MV Android-x86 ya compilado en el *host* de desarrollo para poder compilar módulos del kernel compatibles con el sistema de la MV

Compilación módulos del kernel para Android desde MV Debian

- 2 alternativas
  1. Compilar kernel de Android dentro de la MV de Debian (siguiendo las instrucciones del tutorial "Compilación del kernel de Android") y usar ese kernel para generar fichero .ko del módulo
     - Puede requerir incrementar los recursos de la MV (Discos extra, Cores y RAM disponible)
     - Se valorará positivamente usar esta alternativa
  2. Utilizar el kernel ya compilado disponible en este enlace (tar.gz a extraer en el HOME de la MV)

```
$ cd ; tar xzvf ~/Descargas/android-oreo-kernel-built.tar.gz
```

Variante 1 - programa parte B: Chat

- Implementar el programa de usuario `chat.c` que se comporte como un chat sencillo
- Modo de uso

  `./chat <usuario> <ruta-fifo-envío> <ruta-fifo-recepción>`

  - `usuario`: Cadena de caracteres arbitraria que identifica usuario que se "conecta" al chat
  - El programa usa un FIFO para enviar mensajes al otro usuario del chat y otro FIFO para recibir los mensajes
- Para poder usar el chat será necesario lanzar dos instancias de dicho programa en distintos terminales del siguiente modo
  - Terminal 1: `$ ./chat <usuario1> <ruta-fifo1> <ruta-fifo2>`
  - Terminal 2: `$ ./chat <usuario2> <ruta-fifo2> <ruta-fifo1>`

Variante 1 - programa parte B: Chat

- El programa chat creará dos hilos (emisor y receptor) usando `pthread_create()`
  1. Hilo emisor:
     - Lee líneas por la entrada estandar (espera entrada usuario)
     - Al leer una línea, envía un mensaje por el primer FIFO especificado en línea de comando
  2. Hilo receptor:
     - Permanece bloqueado hasta que se reciba un mensaje por el segundo FIFO especificado en línea de comando
     - Al recibir mensaje, lo procesa y si procede muestra información por pantalla

Chat: Ejemplo de ejecución


```
$ ./chat Fulanito /proc/multififo/fifo0 /proc/multififo/fifo1
Conexión de recepción establecida!!
Conexión de envío establecida!!
>Hola Menganito
>¿Estas por ahí?
>
Menganito dice: Sí, Menganito, aquí ando

Menganito dice: compilando el kernel. Tengo para rato...

>bueno, hasta luego entonces
> Menganito dice: ciao
[Ctrl+D]
$
```

```
$ ./chat Menganito /proc/multififo/fifo1 /proc/multififo/fifo0
Conexion de envío establecida!!
Conexion de recepción establecida!!
>
Fulanito dice: Hola Menganito

Fulanito dice: ¿Estas por ahí?

> Sí, Menganito, aquí ando
> compilando el kernel. Tengo para rato...
>
Fulanito dice: bueno, hasta luego entonces
>ciao
>Conexión finalizada por Fulanito!!
$
```

Variante 1 - programa parte B: Chat (Implementación)

- Los datos transferidos a través de cada "FIFO" se representarán mediante el tipo de datos `struct chat_message`

```
#define MAX_CHARS_MSG 128

typedef enum {
   NORMAL_MSG, /* Mensaje para transferir lineas de la conversacion entre
        ambos usuarios del chat */
   USERNAME_MSG, /* Tipo de mensaje reservado para enviar el nombre de
        usuario al otro extremo*/
   END_MSG /* Tipo de mensaje que se envía por el FIFO cuando un extremo      finaliza la
        comunicación */
}message_type_t;

struct chat_message{
    char contenido[MAX_CHARS_MSG]; //Cadena de caracteres (acabada en '\0)
    message_type_t type;
};
```

Variante 1 - programa parte B: Chat (Implementación)

Comportamiento hilo emisor de mensajes

1. Abre FIFO de envío en modo escritura
2. Envía nombre de usuario (especificado en línea de comandos) al otro extremo a través de FIFO de envío usando un mensaje de tipo USERNAME_MSG
3. Lee líneas de la entrada estándar una a una y las envía encapsuladas en el campo contenido del mensaje (NORMAL_MESSAGE) al otro extremo por el FIFO
   - El procesamiento finaliza cuando usuario teclea CTRL+D (EOF - *fin de fichero*)
   - Al detectar EOF en entrada estándar, enviará `END_MESSAGE` al otro extremo
4. Cerrar FIFO de envío y salir

Variante 1 - programa parte B: Chat (Implementación)

Comportamiento hilo receptor de mensajes

1. Abre FIFO de recepción en modo lectura
2. Procesa mensaje de nombre de usuario (`USERNAME_MSG`) leyendo del FIFO de recepción
3. Procesa resto de mensajes hasta fin de fichero en el FIFO de recepción, error de lectura o recepción de `END_MESSAGE`
   - Al recibir un mensaje normal, imprime el campo `content` por pantalla con el siguiente formato:

     `<nombre_usuario_emisor> dice: <contenido>`
4. Cierra FIFO de recepción y sale

Variante 2: programa parte B

Objetivo

- El objetivo del programa multihilo es demostrar que los servicios que exporta el módulo del kernel (p. ej.: entradas `/proc`) funcionan correctamente
  - Queda a elección del alumno el tipo de procesamiento que realice el programa de usuario
    - Adjuntar fichero `README` en la entrega con descripción de alto nivel sobre funcionalidad del programa desarrollado
  - No se permite usar `system()` ni `exec()`

En esta parte se valorarán los siguientes aspectos:

1. Efectividad y completitud de los casos de uso/prueba del módulo llevados a cabo por el programa
2. Originalidad de la propuesta
3. Nivel de complejidad del programa C para Android-x86

Práctica Final: Parte B

Consideraciones adicionales

- La máquina virtual de Android-x86 lleva un kernel de 64 bits
  - El código del kernel (fork de Linux v4.9.109) puede consultarse a través de https://srcxref.dacya.ucm.es
- Recomendable depurar el programa de usuario en Linux y luego llevarlo a Android

Licencia

LIN - Práctica Final
Versión 2.5

Este documento (o uno muy similar) está disponible en https://cvmdp.ucm.es/moodle/course/view.php?id=20152