



Práctica 3: Procesos y Sincronización

LIN - Curso 2015-2016



Contenido



1 Introducción

2 Tuberías

3 Ejercicios

4 Práctica



Contenido



1 Introducción

2 Tuberías

3 Ejercicios

4 Práctica



Práctica 3: Procesos y Sincronización



Objetivos

- Familiarizarse con:
 - Uso de mecanismos de sincronización en el kernel Linux
 - Implementación de tuberías para comunicación entre procesos



Contenido



1 Introducción

2 Tuberías

3 Ejercicios

4 Práctica





Tuberías (I)

- Mecanismo de **comunicación y sincronización** entre procesos
 - Transferencia de datos entre distintos espacios de direcciones
 - Sincronización tipo productor/consumidor
- Dos tipos de tubería:
 - 1 Sin nombre: **pipe**
 - 2 Con nombre: **FIFO**





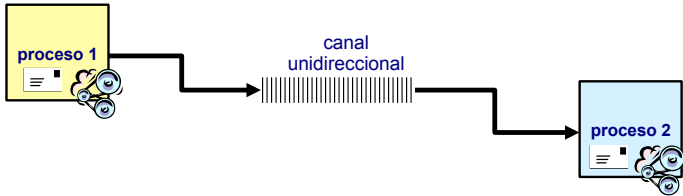
Tuberías (II)

Características

- Mecanismo con capacidad de almacenamiento
- Flujo de datos unidireccional (Proceso A \rightarrow Proceso B)
- Dos extremos:
 - extremo de escritura (envío)
 - extremo de lectura (recepción)
- Los extremos se manejan como si fueran ficheros:
 - 1 Obtener descriptor
 - 2 Enviar con `write()` / Recibir con `read()`
 - 3 Cerrar extremo de lectura o escritura con `close()`



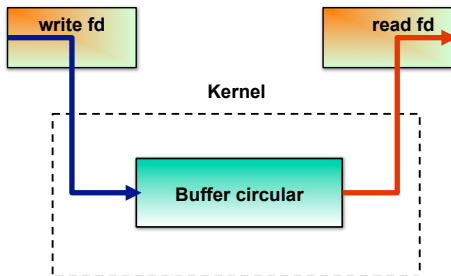
Tuberías (III)



- Los datos escritos en la tubería se conservan hasta el momento en que son leídos, y luego **desaparecen**
- El **orden** de lectura/escritura es *first-in first out* (FIFO)
- No está permitida la operación de posicionamiento aleatorio (**`lseek()`**)

Tuberías (IV)

- A pesar de que los extremos las tuberías se manejan como si fueran ficheros, **los datos no se almacenan en disco**
- El SO emplea habitualmente un **buffer circular** para implementarlo
 - Región de memoria del kernel usada como almacenamiento intermedio





Tuberías sin nombre (*pipes*)

- Un *pipe anónimo* se crea con la llamada al sistema `pipe()` que devuelve un par de descriptores de fichero

```
int pipe(int fildes[2]);
```

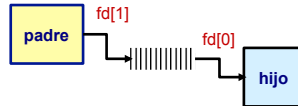
- Identificación: dos descriptores
 - 1 Para lectura: `fildes[0] → read()`
 - 2 Para escritura: `fildes[1] → write()`
- Compartido por el proceso que lo crea y los hijos de éste, gracias al **mecanismo de herencia de ficheros** a través de `fork()`
- El *pipe* se destruye cuando todos los procesos que tienen acceso al mismo cierran su actividad con él (llamada `close()` o finalización del proceso)



Tuberías sin nombre (*pipes*)

```
int fd[2];
pid_t pid;
pipe(fd); ←

if ((pid=fork())>0) {
    close(fd[0]);
    while ( ...haya datos ... ) {
        write(fd[1], ... );
    }
    close(fd[1]);
} else if (pid==0){
    close(fd[1]);
    while (read(fd[0], ... )>0) {
        ... usar los datos...
    }
    close(fd[0]);
    exit (0);
}
```

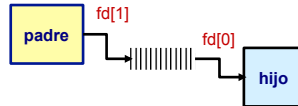


Tuberías sin nombre (*pipes*)

```
int fd[2];
pid_t pid;
pipe(fd);

if ((pid=fork())>0) {
    close(fd[0]);
    while ( ...haya datos ... ) {
        write(fd[1], ... );
    }
    close(fd[1]);
} else if (pid==0){
    close(fd[1]);
    while (read(fd[0], ... )>0) {
        ... usar los datos...
    }
    close(fd[0]);
    exit (0);
}
```

... El padre continua ...



Padre: **cierra extremo lectura**

Padre: **envía datos (escritura)**

Padre: **cierra extremo escritura**

Tuberías sin nombre (*pipes*)

```
int fd[2];
pid_t pid;
pipe(fd);

if ((pid=fork())>0) {
    close(fd[0]);
    while ( ...haya datos ... ) {
        write(fd[1], ... );
    }
    close(fd[1]);
} else if (pid==0){
    close(fd[1]);
    while (read(fd[0], ... )>0) {
        ... usar los datos...
    }
    close(fd[0]);
    exit (0);
}

... El padre continua ...
```



Padre: **cierra extremo lectura**

Padre: **envía datos (escritura)**

Padre: **cierra extremo escritura**

Hijo: **cierra extremo escritura**

Hijo: **recibe datos (lectura)**

Hijo: **cierra extremo lectura**



Tuberías con nombre (*FIFOs*)

- Un FIFO es un **fichero especial** en UNIX:
 - Tubería que tiene presencia en el sistema de ficheros
- Tres mecanismos para crear un FIFO
 - 1 Comando **mkfifo**
 - 2 Llamada al sistema **mknod()** con argumento **S_IFIFO**
 - 3 Función de librería **mkfifo()**

terminal

```
kernel@debian:~$ mkfifo /var/tmp/ficheroFIFO
kernel@debian:~$ stat /var/tmp/ficheroFIFO
  File: «/var/tmp/ficheroFIFO»
  Size: 0          Blocks: 0          IO Block: 4096   `fifo'
Device: 801h/2049d Inode: 91502       Links: 1
Access: (0644/prw-r--r--)  Uid: ( 1000/ kernel)   Gid: ( 1000/ kernel)
Access: 2014-11-17 14:35:53.000000000 +0100
Modify: 2014-11-17 14:35:53.000000000 +0100
Change: 2014-11-17 14:35:53.000000000 +0100
```





Tuberías con nombre (*FIFOs*)

- Cualquier proceso con los permisos apropiados puede solicitar la apertura mediante `open()`
 - Los procesos que se comunican no tienen por qué estar relacionados jerárquicamente entre sí
- **Comportamiento `open()`**
 - El modo de apertura al invocar `open()` (lectura o escritura) determina el extremo de la tubería al que se accede
 - La apertura de un FIFO en modo lectura bloquea al proceso hasta otro proceso haya abierto su extremo de escritura (y viceversa)
 - Sincronización tipo *rendezvous*





Tuberías con nombre (*FIFOs*)

■ Comportamiento `read()`

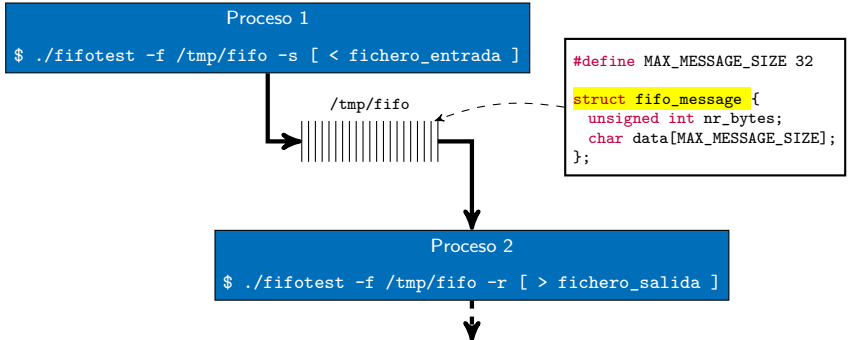
- La lectura de un FIFO sin datos ocasiona el bloqueo del proceso que la solicita
- El intento de leer de un FIFO vacío cuyo extremo de escritura ha sido cerrado devuelve el valor 0 (EOF)

■ Comportamiento `write()`

- La escritura en un FIFO cuya capacidad está completa ocasiona el bloqueo del proceso que la solicita.
- Al escribir en un FIFO cuyo extremo de lectura ha sido cerrado:
 - 1 `write()` devuelve un error (valor -1)
 - 2 El SO envía una señal (SIGPIPE) al proceso cuya acción por defecto es terminar su ejecución



Ejemplo: programa fifotest



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...



Ejemplo: programa fifotest

```
/* Lee datos de la entrada estandar y los envía por un FIFO encapsulados en
   estructura "fifo_message" */
static void fifo_send (const char* path_fifo) {
    struct fifo_message message;
    int fd_fifo=0;
    int bytes=0,wbytes=0;
    const int size=sizeof(struct fifo_message);

    fd_fifo=open(path_fifo,O_WRONLY);

    ...

    while((bytes=read(0,message.data,MAX_MESSAGE_SIZE))>0) {
        message.nr_bytes=bytes;
        wbytes=write(fd_fifo,&message,size);

        ... Tratamiento errores write ...
    }

    ...

    close(fd_fifo);
}
```





Ejemplo: programa fifotest

```
/* Recibe un conjunto de registros "fifo_message" a través de un FIFO y e
   imprime su contenido por pantalla */
static void fifo_receive (const char* path_fifo) {
    struct fifo_message message;
    int fd_fifo=0;
    int bytes=0,wbytes=0;
    const int size=sizeof(struct fifo_message);

    fd_fifo=open(path_fifo,O_RDONLY);

    ...

    while((bytes=read(fd_fifo,&message,size))==size) {
        /* Write to stdout */
        wbytes=write(1,message.data,message.nr_bytes);

        ... Tratamiento errores write ...
    }

    ...

    close(fd_fifo);
}
```





Ejemplo: programa fifotest

terminal 1

```
kernel@debian:~/FifoTest$ mkfifo /tmp/fifo
kernel@debian:~/FifoTest$ ./fifotest -f /tmp/fifo -s < test.txt
kernel@debian:~/FifoTest$
```

terminal 2

```
kernel@debian:~/FifoTest$ ./fifotest -f /tmp/fifo -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$
```



Contenido



1 Introducción

2 Tuberías

3 Ejercicios

4 Práctica



Ejercicio 1

- Estudiar la implementación del módulo ProdCons1
 - Módulo del kernel que gestiona un buffer circular acotado de punteros a enteros
 - El módulo exporta entrada `/proc/prodcons`
 - Insertar al final del buffer: `$ echo 7 > /proc/prodcons`
 - Extraer primer elemento del buffer: `$ cat /proc/prodcons`
 - Las operaciones de inserción/eliminación del buffer tienen la semántica productor/consumidor
 - 1 Un proceso que inserta en buffer lleno se bloquea
 - 2 Proceso que consume de buffer vacío se queda bloqueado



Ejercicio 1: ProdCons1

terminal

```
kernel@debian:~/ProdCons1$ echo 4 > /proc/prodcons
kernel@debian:~/ProdCons1$ echo 5 > /proc/prodcons
kernel@debian:~/ProdCons1$ echo 6 > /proc/prodcons
kernel@debian:~/ProdCons1$ cat /proc/prodcons
4
kernel@debian:~/ProdCons1$ cat /proc/prodcons
5
kernel@debian:~/ProdCons1$ cat /proc/prodcons
6
kernel@debian:~/ProdCons1$ cat /proc/prodcons
<<proceso se queda bloqueado>>
```



Ejercicio 2: ProdCons2



Ejercicio 2

- Estudiar la implementación del módulo ProdCons2
 - Variante de ProdCons1 donde los semáforos se utilizan como colas de espera
 - Los semáforos se deben usar de esta forma en la parte B de la práctica



Contenido



1 Introducción

2 Tuberías

3 Ejercicios

4 Práctica



Dos partes:

- **(Parte A)** Implementación *SMP-safe* de la Práctica 1 usando *spin locks*
 - Se ha de garantizar **exclusión mutua** entre las distintas regiones de código que **acceden a la lista enlazada** de enteros (estructura compartida)
 - **No es posible invocar funciones bloqueantes** como `vmalloc()` dentro de `spin_lock()` y `spin_unlock()`
 - **Hacer pruebas de acceso concurrente a la lista enlazada mediante un script lanzado desde varios terminales**
 - Entregar script(s) usado(s)
- **(Parte B)** Implementación de un FIFO mediante una entrada `/proc`
 - El módulo implementará 4 operaciones de la entrada `/proc`: `read()`, `write()`, `open()`, **`release()`**
 - Sincronización gestionada mediante **semáforos**



Parte B: Implementación

Recursos para la implementación

- Buffer circular de bytes (**ya implementado**)
 - Almacenamiento temporal asociado al FIFO
 - Al crear el buffer, establecer que tamaño máximo sea 50 bytes
- Dos contadores para registrar el número de procesos que han abierto el FIFO para lectura y escritura, respectivamente
- Un “mutex” para proteger el buffer y los contadores
 - Usaremos un semáforo inicializado a 1 para emular el mutex
 - Mutex también asociado a las “variables condición”
- Dos “variables condición”
 - Una para bloquear al productor y otra para bloquear al consumidor
 - Para emular cada variable condición se usará:
 - 1 Semáforo inicializado a 0 (cola de espera)
 - 2 Contador que registra número de procesos esperando (0 ó 1)





Parte B: Implementación

Variables globales (fifoproc.c)

```
cbuffer_t* cbuffer; /* Buffer circular */
int prod_count = 0; /* Número de procesos que abrieron la entrada
                    /proc para escritura (productores) */
int cons_count = 0; /* Número de procesos que abrieron la entrada
                    /proc para lectura (consumidores) */
struct semaphore mtx; /* para garantizar Exclusión Mutua */
struct semaphore sem_prod; /* cola de espera para productor(es) */
struct semaphore sem_cons; /* cola de espera para consumidor(es) */
int nr_prod_waiting=0; /* Número de procesos productores esperando */
int nr_cons_waiting=0; /* Número de procesos consumidores esperando */
```



Parte B: Implementación

- Módulo consta de los siguientes ficheros fuente:
 - 1 **fifoproc.c**: Fichero principal
 - 2 **cbuffer.h**: Declaración del tipo `cbuffer_t` y operaciones sobre el mismo
 - 3 **cbuffer.c**: Implementación de las operaciones del tipo `cbuffer_t`
- Necesario crear Makefile para compilar módulo que consta de varios ficheros .c

Makefile

```
obj-m = fifomod.o #fifomod.c no ha de existir
fifomod-objs = fifoproc.o cbuffer.o

all:
    make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules

clean:
    make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
```





Parte B: Implementación

Funciones a implementar (fifoproc.c)

```
/* Funciones de inicialización y descarga del módulo */
int init_module(void);
void cleanup_module(void);

/* Se invoca al hacer open() de entrada /proc */
static int fifoproc_open(struct inode *, struct file *);

/* Se invoca al hacer close() de entrada /proc */
static int fifoproc_release(struct inode *, struct file *);

/* Se invoca al hacer read() de entrada /proc */
static ssize_t fifoproc_read(struct file *, char *, size_t, loff_t *);

/* Se invoca al hacer write() de entrada /proc */
static ssize_t fifoproc_write(struct file *, const char *, size_t,
    loff_t *);
```



Parte B: Productor y consumidor

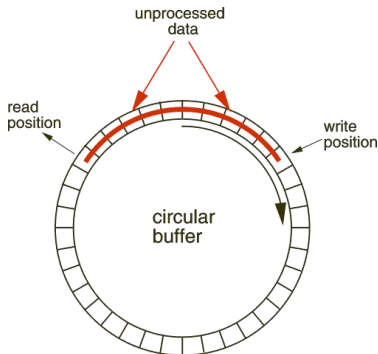
- Se puede distinguir entre *productor* y *consumidor* consultando el campo `f_mode` de la estructura `struct file`

```
static int fifoproc_open(struct inode *inode, struct file *file)
{
    ...
    if (file->f_mode & FMODE_READ)
    {
        /* Un consumidor abrió el FIFO */
        ...
    } else{
        /* Un productor abrió el FIFO */
        ...
    }
    ...
}
```



Parte B: Buffer circular

- El tipo de datos `cbuffer_t` implementa **buffer circular de bytes**
 - En este contexto `char` = byte
 - Las **inserciones** en el buffer circular se realizan **siempre al final**
 - La **cabeza del buffer (*head*)** es el **extremo de lectura**, por donde se extraen los elementos





Parte B: *cbuffer_t*

```
typedef struct
{
    char* data;          /* raw byte vector */
    unsigned int head;    /* Index of the first element in [0 .. max_size-1] */
    unsigned int size;    /* Current Buffer size // size in [0 .. max_size] */
    unsigned int max_size; /* Buffer max capacity */
}cbuffer_t;

/* Operations supported by cbuffer_t */

/* Creates a new cbuffer (takes care of allocating memory) */
cbuffer_t* create_cbuffer_t (unsigned int max_size);

/* Release memory from circular buffer */
void destroy_cbuffer_t ( cbuffer_t* cbuffer );

/* Returns the number of elements in the buffer */
int size_cbuffer_t ( cbuffer_t* cbuffer );

/* Returns the number of free gaps in the buffer */
int nr_gaps_cbuffer_t ( cbuffer_t* cbuffer );

/* Returns a non-zero value when buffer is full */
int is_full_cbuffer_t ( cbuffer_t* cbuffer );
```





Parte B: *cbuffer_t*

```
...
/* Returns a non-zero value when buffer is empty */
int is_empty_cbuffer_t ( cbuffer_t* cbuffer );

/* Inserts an item at the end of the buffer */
void insert_cbuffer_t ( cbuffer_t* cbuffer, char new_item );

/* Inserts nr_items into the buffer */
void insert_items_cbuffer_t ( cbuffer_t* cbuffer, const char* items, int nr_items);

/* Removes the first element in the buffer and returns a copy of it */
char remove_cbuffer_t ( cbuffer_t* cbuffer);

/* Removes nr_items from the buffer and returns a copy of them */
void remove_items_cbuffer_t ( cbuffer_t* cbuffer, char* items, int nr_items);

/* Removes all items in the buffer */
void clear_cbuffer_t (cbuffer_t* cbuffer);

/* Returns a pointer to the first element in the buffer */
char* head_cbuffer_t ( cbuffer_t* cbuffer );
```

ArTeCS



Desarrollo de la parte B

- Antes de comenzar con la implementación, escribir pseudocódigo de la solución usando mutexes y variables condición
 - El pseudocódigo se debe entregar con la práctica (fichero aparte)

Plantilla para el pseudocódigo

```
mutex mtx;
condvar prod, cons;
int prod_count=0, cons_count=0;
cbuffer_t* cbuffer;

void fifoproc_open(bool abre_para_lectura) {
    /* Completar */
}

int fifoproc_write(char* buff, int len) {
    /* Completar */
}

int fifoproc_read(const char* buff, int len) {
    /* Completar */
}

void fifoproc_release(bool lectura) {
    /* Completar */
}
```





Parte B: FIFO (Pseudocódigo)

```
mutex mtx;
condvar prod, cons;
int prod_count=0, cons_count=0;
cbuffer_t* cbuffer;

int fifoproc_write(char* buff, int len) {
    char kbuffer[MAX_KBUF];

    if (len > MAX_CBUFFER_LEN || len > MAX_KBUF) { return Error; }
    if (copy_from_user(kbuffer, buff, len)) { return Error; }

    lock(mtx);

    /* Esperar hasta que haya hueco para insertar (debe haber consumidores) */
    while (nr_gaps_cbuffer_t(cbuffer) < len && cons_count > 0) {
        cond_wait(prod, mtx);
    }

    /* Detectar fin de comunicación por error (consumidor cierra FIFO antes) */
    if (cons_count == 0) { unlock(mtx); return -EPIPE; }

    insert_items_cbuffer_t(cbuffer, kbuffer, len);

    /* Despertar a posible consumidor bloqueado */
    cond_signal(cons);

    unlock(mtx);
    return len;
}
```



Desarrollo de la parte B (Cont.)

- Una vez escrito el pseudocódigo, obtener implementación con semáforos usando la técnica de traducción descrita en el tema anterior
 - Añadir semáforos y contadores al código del módulo del kernel
 - En la implementación no habrá ningún mutex ni variables condición
- Este FIFO NO puede usarse con echo y cat
 - Usar programa `fifo_test` para depurar el código

Comportamiento del FIFO

- Si se intenta realizar una **lectura o escritura de un número de bytes superior al tamaño máximo del buffer**, el módulo devolverá un error
 - El semáforo `sem_prod` se usa para *bloquear al productor*
 - El semáforo `sem_cons` se usa para *bloquear al consumidor*



Desarrollo de la parte B (Cont.)

Comportamiento del FIFO (cont.)

- Al **abrir FIFO en modo lectura** (consumidor) se **bloquea al proceso** hasta que productor haya abierto su extremo de escritura
- Al **abrir FIFO en modo escritura** (productor) se **bloquea al proceso** hasta que consumidor haya abierto su extremo de lectura
- El **productor se bloquea si no hay hueco en el buffer** para insertar el número de bytes solicitados mediante `write()`
- El **consumidor se bloquea si el buffer contiene menos bytes** que los solicitados mediante `read()`

Desarrollo de la parte B (Cont.)

Comportamiento del FIFO (cont.)

- Si cualquier proceso bloqueado en un semáforo **se despierta por la recepción de una señal**, la operación en cuestión (`open()`, `read()` o `write()`) devolverá un error
- Cuando todos los procesos (productores y consumidores) **finalicen su actividad** con el FIFO, **el buffer circular ha de vaciarse**
- Si se intenta hacer una lectura del FIFO cuando el buffer circular esté vacío y no haya productores, el módulo devolverá el valor 0 (EOF)
- Si se intenta escribir en el FIFO cuando no hay consumidores (extremo de lectura cerrado), el módulo devolverá un error



Parte B: Ejemplo de ejecución

terminal 1

```
kernel@debian:~/FifoTest$ sudo insmod ../ProcFifo/fifomod.ko
kernel@debian:~/FifoTest$ ./fifotest -f /proc/modfifo -s < test.txt
kernel@debian:~/FifoTest$
```

terminal 2

```
kernel@debian:~/FifoTest$ ./fifotest -f /proc/modfifo -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$
```





Partes opcionales

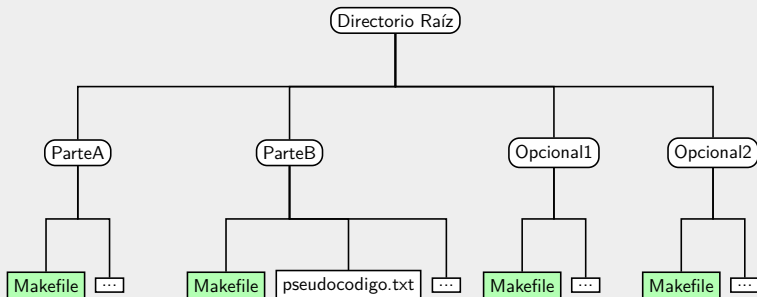
- **(Opcional 1)** Reimplementar el FIFO usando la estructura de buffer circular de bytes disponible en Linux (`struct kfifo`) en lugar del tipo de datos `cbuffer_t`
 - Operaciones de `struct kfifo` declaradas en `<linux/kfifo.h>`
 - Más información en el capítulo 6 (“Kernel Data Structures”) del libro “Linux Kernel Development”
- **(Opcional 2)** Modificar el módulo de la parte B de la práctica para que exponga el FIFO al usuario como un dispositivo de caracteres en lugar de como una entrada `/proc`
 - El módulo se comportará como un driver de dispositivo de caracteres
 - El fichero especial de caracteres se creará explícitamente con el comando `mknod`



Entrega de la práctica

- A través del Campus Virtual
 - Hasta el 11 de diciembre
- Obligatorio mostrar el funcionamiento después de hacer la entrega

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





LIN - Práctica 3: Procesos y Sincronización Versión 0.2

©J.C. Sáez

*This work is licensed under the Creative Commons **Attribution-Share Alike 3.0 Spain License**. To view a copy of this license, visit <http://creativecommons.org/licenses/by-sa/3.0/es/> or send a letter to Creative Commons, 171 Second Street, Suite 300, San Francisco, California, 94105, USA.*

*Esta obra está bajo una licencia **Reconocimiento-Compartir Bajo La Misma Licencia 3.0 España de Creative Commons**. Para ver una copia de esta licencia, visite <http://creativecommons.org/licenses/by-sa/3.0/es/> o envíe una carta a Creative Commons, 171 Second Street, Suite 300, San Francisco, California 94105, USA.*

Este documento (o uno muy similar) está disponible en <https://cv4.ucm.es/moodle/course/view.php?id=62472>

