Práctica 3: Procesos y Sincronización

DSO - Curso 2013-2014

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

Contenido

ArTeCS

Contenido

Práctica 3: Procesos y Sincronización

Objetivos

- Familiarizarse con:
  - Uso de mecanismos de sincronización en el kernel Linux
  - Implementación de tuberías para comunicación entre procesos
  - Creación de *drivers* de dispositivos de caracteres

Requisitos

- Leer el capítulo 4 de “The Linux Kernel Module Programming Guide”
  - `http://tldp.org/LDP/lkmpg/2.6/html/lkmpg.html`
  - `http://www.tldp.org/LDP/lkmpg/2.6/lkmpg.pdf`

Contenido

ArTeCS

Tuberías (I)

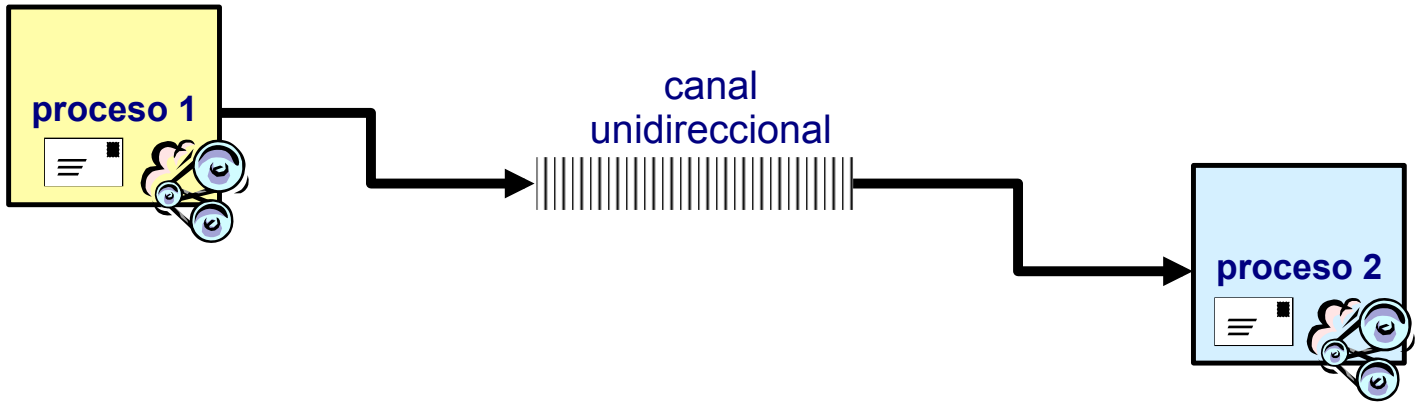
- Mecanismo de **comunicación y sincronización** entre procesos
  - Transferencia de datos entre distintos espacios de direcciones
  - Sincronización tipo productor/consumidor
- Dos tipos de tubería:
  1. Sin nombre: pipe
  2. Con nombre: FIFO

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()`

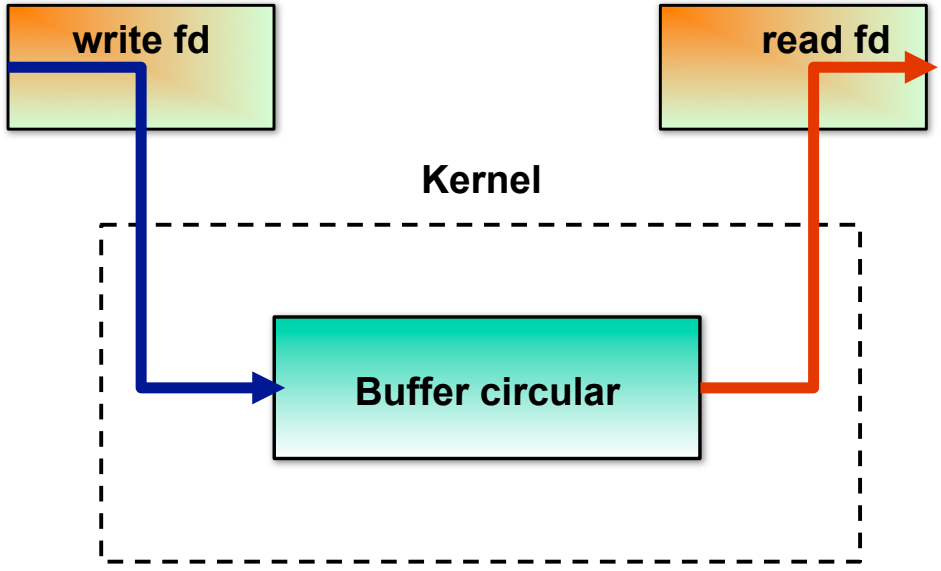
Tuberías (III)



- 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

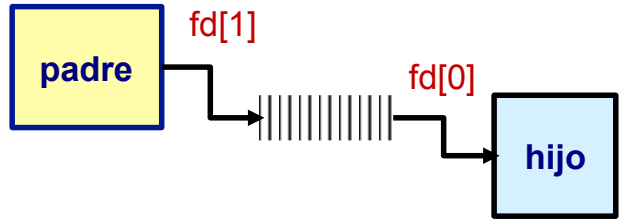
```
int pipe(int fildes[2]);
```

- 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);
}
... El padre continua ...
```



Padre: cierra extremo lectura

Padre: envía datos (escritura)

Padre: cierra extremo escritura

Hijo: cierra extremo escritura

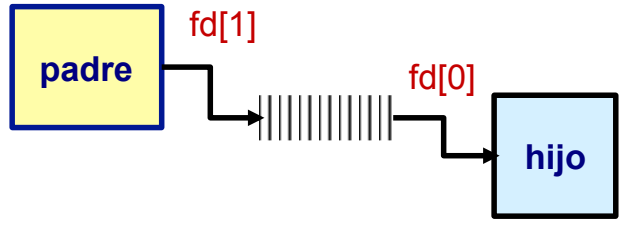
Hijo: recibe datos (lectura)

Hijo: cierra extremo lectura

Tuberías 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

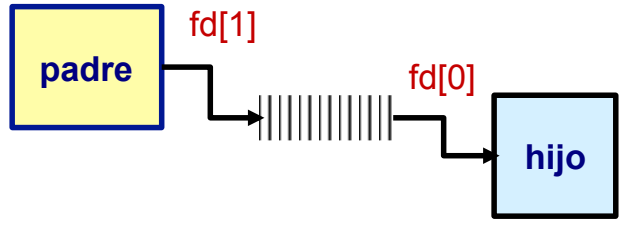
Hijo: recibe datos (lectura)

Hijo: cierra extremo lectura

Tuberías 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

```
dsouser@debian:~$ mkfifo /var/tmp/ficheroFIFO
dsouser@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/ dsouser)   Gid: ( 1000/ dsouser)
Access: 2013-11-17 14:35:53.000000000 +0100
Modify: 2013-11-17 14:35:53.000000000 +0100
Change: 2013-11-17 14:35:53.000000000 +0100
```

Tuberías con nombre (*FIFOs*)

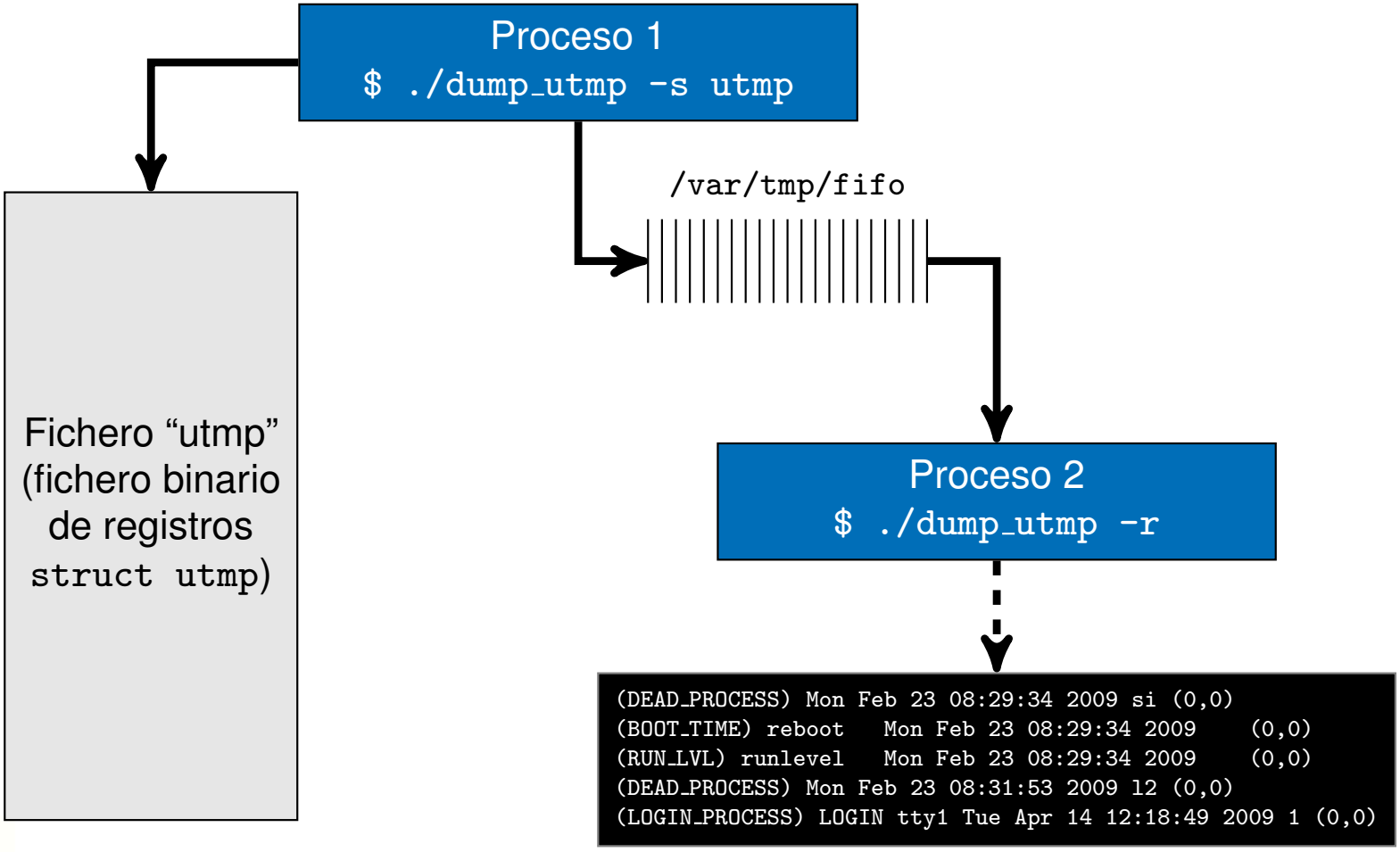
- Cualquier proceso con los permisos apropiados puede solicitar la apertura mediante `open()`
  - Los procesos que se comunican no tienen por qué estar relacionados jerárquicamente entre sí
- **Comportamiento** `open()`
  - El modo de apertura al invocar `open()` (lectura o escritura) determina el extremo 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)

ArTeCS

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 *dump_utmp*



Ejemplo: programa *dump_utmp*

```
/* Lee registros del fichero UTMP y los envía por un FIFO, uno a uno */
static void utmp_send (char *fichero) {
  int n_entradas,i,fd_fifo,bytes;
  struct utmp *utmp_buf;
  const struct utmp *cur;
  const int size=sizeof(struct utmp);

  /* Read utmp file */
  if (read_utmp(fichero, &n_entradas, &utmp_buf) != 0)
    err (EXIT_FAILURE, "%s", fichero);
   ...
   fd_fifo=open(PATH_FIFO,O_WRONLY);
   ...
   /* Bucle de envío de datos a través del FIFO */
   for (i=0;i<n_entradas;i++) {
      cur=&(utmp_buf[i]);
      bytes=write(fd_fifo,cur,size);

      if (bytes < size) {
        .. Tratar error..
    }
  }
    ...
  close(fd_fifo);
}
```

Ejemplo: programa *dump_utmp*

```
/* Recibe un conjunto de registros UTMP a través de un FIFO y e imprime su
     contenido por pantalla */
static void utmp_receive (void) {
  struct utmp received;
  char strtype[50];
  char strexit[50];
  char* strtime;
  int fd_fifo=0;
  int bytes=0;
  const int size=sizeof(struct utmp);

  fd_fifo=open(PATH_FIFO,O_RDONLY);
  ...
  while((bytes=read(fd_fifo,&received,size))==size)
  {
     ... Imprime contenido del registro por pantalla ...
  }
    ...
   close(fd_fifo);
}
```

Ejemplo: programa *dump_utmp*

terminal 1

```
dsouser@debian:~/DumpUTMP$ mkfifo /var/tmp/fifo
dsouser@debian:~/DumpUTMP$ ./dump_utmp -s utmp
dsouser@debian:~/DumpUTMP$
```

terminal 2

```
dsouser@debian:~/DumpUTMP$ ./dump_utmp -r
(DEAD_PROCESS)              Mon Feb 23 08:29:34 2009    si  (0,0)
(BOOT_TIME) reboot      ~   Mon Feb 23 08:29:34 2009    ~~  (0,0)
(RUN_LVL)   runlevel        ~   Mon Feb 23 08:29:34 2009    ~~      (0,0)
(DEAD_PROCESS)              Mon Feb 23 08:31:53 2009    l2  (0,0)
(LOGIN_PROCESS) LOGIN       tty1    Tue Apr 14 12:18:49 2009    1   (0,0)
(USER_PROCESS)  root        tty2    Mon Feb 23 08:36:12 2009    2   (0,0)
(LOGIN_PROCESS) LOGIN       tty3    Mon Feb 23 08:38:07 2009    3   (0,0)
(LOGIN_PROCESS) LOGIN       tty4    Mon Feb 23 08:31:53 2009    4   (0,0)
(LOGIN_PROCESS) LOGIN       tty5    Mon Feb 23 08:31:53 2009    5   (0,0)
(INIT_PROCESS)              Mon Feb 23 08:31:53 2009    6   (0,0)
(LOGIN_PROCESS) LOGIN       ttyS0   Mon Feb 23 08:31:53 2009    T0  (0,0)
(USER_PROCESS)  dario       pts
```

Contenido

Dispositivos de caracteres

- Un dispositivo de caracteres es otro tipo de fichero especial en UNIX
  - Abstracción software que proporciona el SO
  - Los programas de usuario pueden acceder a ellos como si fueran ficheros convencionales
    - Tienen presencia en el sistema de ficheros (ej: `/dev/ttyS0`)
    - `open()`, `read()`, `write()`, `close()`
    - *No necesariamente han de estar asociados a dispositivos HW*
  - El SO expone algunos dispositivos HW a los programas mediante dispositivos de caracteres
    - Ejemplo: terminales, puertos serie, ...

Dispositivos de caracteres

- Para cada tipo de dispositivo de caracteres hay asociado un *driver* de dispositivo
  - Cada *driver* tiene un identificador numérico único (*major number*) que sirve para identificarlo
  - Si el *driver* gestiona varios dispositivos, cada dispositivo tiene asociado un identificador secundario de dispositivo (*minor number*)
  - El par (*major number*, *minor number*) identifica de forma únivoca a cada dispositivo de caracteres del sistema

terminal

```
dsouser@debian:~$ stat /dev/tty1
  File: ≪/dev/tty1≫
  Size: 0       Blocks: 0    IO Block: 4096   fichero especial de caracteres
Device: 5h/5d   Inode: 3788        Links: 1     Device type: 4,1
Access: (0600/crw-------)  Uid: (    0/    root)   Gid: (    0/    root)
Access: 2013-11-12 09:25:53.108121554 +0100
Modify: 2013-11-12 09:25:58.236121553 +0100
Change: 2013-11-12 09:25:57.236121553 +0100
```

Dispositivos de caracteres

terminal

```
dsouser@debian:~$ cat /proc/devices
Character devices:
  1 mem
  4 /dev/vc/0
  4 tty
  4 ttyS
  5 /dev/tty
  5 /dev/console
  5 /dev/ptmx
  7 vcs
 10 misc
 13 input
 29 fb
128 ptm
136 pts
180 usb
189 usb_device
252 hidraw
253 bsg
254 rtc

Block devices:
  2 fd
  3 ide0
259 blkext
  7 loop
  8 sd
 22 ide1
 65 sd
 66 sd
```

- La asociación entre el *driver* del dispositivo y el número de versión mayor asignado puede consultarse en `/proc/devices`
- La mayor parte de los *drivers* de dispositivo se implementan como módulos cargables del kernel
  1. Implementan un interfaz especial
  2. Se registran como *driver* de dispositivo de caracteres

Interfaz *Driver*

Interfaz de Operaciones

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

Implementación de un driver

- Crear un módulo del kernel con funciones `init()` y `cleanup()`
- Declarar estructura global `struct file_operations`
  - Especifica qué operaciones de dispositivo de caracteres se implementan y su asociación con las funciones del módulo

```
struct file_operations fops = {
    .read = device_read,      //read()
    .write = device_write,    //write()
    .open = device_open,      //open()
    .release = device_release    //close()
};
```

- En la función de inicialización, registrar el módulo como *driver* de dispositivo de caracteres mediante `register_chrdev()`
  - Si pasamos un 0 como primer parámetro, nos devuelve el *major number* asignado automáticamente

```
int register_chrdev(unsigned int major, const char *name,
                                    struct file_operations *fops);
```

Implementación de un driver (cont.)

- En la función *cleanup()* del módulo, desregistrar el módulo como *driver* de dispositivo de caracteres mediante `unregister_chrdev()`
  - Pasar el *major number* asignado por `register_chrdev()` como primer parámetro

```
int unregister_chrdev(unsigned int major, const char *name);
```

- Implementar las operaciones del interfaz para conseguir la funcionalidad deseada
  - Semántica similar a la llamada al sistema pertinente
    - P. ej., `device_read()` devolverá el número de bytes escritos por el *driver* en el buffer que pasa el usuario o un valor negativo si se produjo un error

Invocar funciones del driver

- Para usar las funciones del driver:
  1. Crear un dispositivo de caracteres con `mknod` que tenga el mismo *major number* con el que se registró el driver

     ```
     $ sudo mknod <ruta_dispositivo> -m 666 c <major> <minor>
     ```

  2. Acceder al dispositivo creado:
     - Desde un programa de usuario: `open()`, `read()`, `write()`, `close()`
     - Desde el shell: `cat`, `echo`

Contenido

ArTeCS

Ejercicios

Ejercicio 1

- Consultar y probar el ejemplo `chardev.c` del capítulo 4 de *Linux Kernel Module Programming Guide*
  - Módulo que implementa un driver de dispositivos de caracteres ficticios
  - El driver ejecutará las operaciones pertinentes cuando un programa de usuario abra, lea o escriba en los dispositivos de caracteres asociados al driver
    - Deben tener el número de versión mayor con el que el driver se registra
    - Ejemplo: `$ cat /dev/chardev`
  - Advertencia!: Código adaptado a la versión del kernel 2.6.39.4 en el Campus Virtual

Ejercicio 1: chardev

terminal

```
dsouser@debian:~/Chardev# ls
Makefile     chardev.c
dsouser@debian:~/Chardev$ make
make -C /lib/modules/2.6.39.4.dso/build M=/home/dsouser/Chardev modules
make[1]: se ingresa al directorio ‘/usr/src/linux-headers-2.6.39.4.dso’
  CC [M]  /home/dsouser/Chardev/chardev.o
  Building modules, stage 2.
  MODPOST 1 modules
  CC      /home/dsouser/Chardev/chardev.mod.o
  LD [M]  /home/dsouser/Chardev/chardev.ko
make[1]: se sale del directorio ‘/usr/src/linux-headers-2.6.39.4.dso’
dsouser@debian:~/Chardev# sudo insmod ./chardev.ko
dsouser@debian:~/Chardev# lsmod | grep chardev
chardev                 1636  0
dsouser@debian:~/Chardev# cat /proc/devices | grep chardev
251 chardev
```

Ejercicio 1: chardev

terminal

```
dsouser@debian:~/Chardev# dmesg
...
[  519.238393] chardev: module license 'unspecified' taints kernel.
[  519.238397] Disabling lock debugging due to kernel taint
[  519.243271] I was assigned major number 251. To talk to
[  519.243274] the driver, create a dev file with
[  519.243276] 'mknod /dev/chardev c 251 0'.
[  519.243277] Try various minor numbers. Try to cat and echo to
[  519.243279] the device file.
[  519.243280] Remove the device file and module when done.
dsouser@debian:~/Chardev# sudo mknod /dev/chardev -m 666 c 251 0
dsouser@debian:~/Chardev# ls -l /dev/chardev
crw-rw-rw- 1 root root 251, 0 may  9 18:29 /dev/chardev
dsouser@debian:~/Chardev# stat /dev/chardev
  File: ≪/dev/chardev≫
  Size: 0     Blocks: 0      IO Block: 4096   fichero especial de caracteres
Device: 5h/5d   Inode: 17143       Links: 1     Device type: fb,0
Access: (0666/crw-rw-rw-)  Uid: (    0/    root)   Gid: (    0/    root)
...
dsouser@debian:~/Chardev# cat /dev/chardev
I already told you 0 times Hello world!
dsouser@debian:~/Chardev# cat /dev/chardev
I already told you 1 times Hello world!
```

Ejercicios (cont.)

Ejercicio 2

- 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 2: ProdCons1

terminal

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

Ejercicios (cont.)

Ejercicio 3

- 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

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()`
- **(Parte B)** Implementación de un FIFO mediante un dispositivo de caracteres
  - Módulo de kernel que actúa como driver de dispositivo de caracteres
  - Sincronización gestionada mediante semáforos

Parte B: Implementación

- El módulo gestionará un único FIFO implementado como dispositivo de caracteres
  - Responderá peticiones emitidas sobre cualquier dispositivo de caracteres cuyo *major number* coincida con el valor devuelto por `register_chrdev()` al cargar el módulo
    - Suponer que hay sólo un dispositivo de caracteres con ese *major number*
  - Las funciones del interfaz del dispositivo de caracteres se implementaran para emular la semántica de un fichero FIFO
    - Advertencia: Las operaciones `read()` y `write()` reciben punteros al espacio de usuario $\rightarrow$ usar `copy_to_user()` y `copy_from_user()`

Parte B: Implementación

- La implementación debe llevarse a cabo empleando
  - Buffer circular de bytes (ya implementado)
    - Almacenamiento temporal asociado al FIFO
  - Tres semáforos y otras variables compartidas

Variables globales (`fifodev.c`)

```
cbuffer_t* cbuffer;   /* Buffer circular */
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 */
int prod_count = 0;       /* Número de procesos que abrieron el
    dispositivo para escritura (productores) */
int cons_count = 0;       /* Número de procesos que abrieron el
    dispositivo para lectura (consumidores) */
```

Parte B: Implementación

- Módulo consta de los siguientes ficheros fuente
  1. fifodev.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
  - Modificar `Makefile` de los ejemplos `ProdCons1` o `ProdCons2`

Parte B: Implementación

Funciones a implementar (`fifodev.c`)

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

/* Se invoca al hacer open() del dispositivo de caracteres */
static int fifodev_open(struct inode *, struct file *);

/* Se invoca al hacer close() del dispositivo de caracteres */
static int fifodev_release(struct inode *, struct file *);

/* Se invoca al hacer read() al dispositivo de caracteres */
static ssize_t fifodev_read(struct file *, char *, size_t, loff_t *);

/* Se invoca al hacer write() al dispositivo de caracteres */
static ssize_t fifodev_write(struct file *, const char *, size_t,
    loff_t *);
```

Parte B: Implementación

Implementación interfaz *driver* (`fifodev.c`)

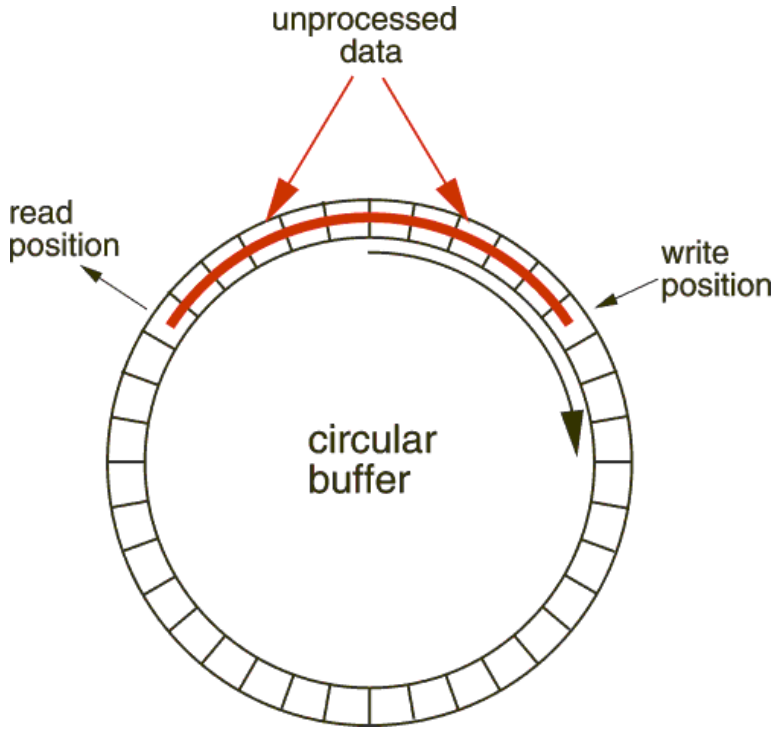
```
#define DEVICE_NAME "fifodev"
...
struct file_operations fops = {
    .read = fifodev_read,
    .write = fifodev_write,
    .open = fifodev_open,
    .release = fifodev_release
};

int major=0;

int init_module(void) {
    ...
    major = register_chrdev(0, DEVICE_NAME, &fops);
    ...
}
```

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);

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

Desarrollo de la parte B

- Aconsejable abordar la implementación del módulo que gestiona el FIFO en dos partes
  1. FIFOv1: Versión NO *SMP-safe* del módulo (sin semáforos)
  2. FIFOv2: Versión a entregar (con semáforos)

Parte B : FIFOv1

- **FIFOv1:**
  - Partir del código del módulo `chardev`
  - **No hay bloqueos** en `open()`, `read()` ni `write()`
  - **Comportamiento lectura** de `nbytes`
    - Si buffer cicrular está vacío $\rightarrow$ Se devuelve 0 (**EOF** - *fin de fichero*)
    - Si `nbytes` $\geq$ `tam_buffer` $\rightarrow$ Se devuelve `tam_buffer` bytes al usuario
    - Si `nbytes` $<$ `tam_buffer` $\rightarrow$ Se devuelve `nbytes` bytes al usuario
  - **Comportamiento escritura**
    - Se **permite escribir en el dispositivo** de caracteres cuando el **buffer está lleno** (potencial sobreescritura de posiciones)
    - Si se intenta realizar una **escritura de un número de bytes superior al tamaño máximo** del buffer el módulo **devolverá un error**
  - Este FIFO puede usarse con `echo` y `cat`

Parte B: FIFOv1

terminal

```
dsouser@debian:~/DumpUTMP$ sudo rm -f /var/tmp/fifo
dsouser@debian:~/DumpUTMP$ sudo mknod /var/tmp/fifo -m 666 c 251 0
dsouser@debian:~/DumpUTMP$ cat /var/tmp/fifo
dsouser@debian:~/DumpUTMP$ echo Hooola > /var/tmp/fifo
dsouser@debian:~/DumpUTMP$ cat /var/tmp/fifo
Hooola
dsouser@debian:~/DumpUTMP$ cat /var/tmp/fifo
dsouser@debian:~/DumpUTMP$ echo AAA  > /var/tmp/fifo
dsouser@debian:~/DumpUTMP$ echo BBB  > /var/tmp/fifo
dsouser@debian:~/DumpUTMP$ cat /var/tmp/fifo
AAA
BBB
```

Parte B: FIFOv2

- **FIFOv2:**
  - **Añadir semáforos y contadores** a la implementación de FIFOv1
    - Uso de los semáforos muy similar al del ejemplo `ProdCons2`
    - **Aconsejable escribir primero pseudocódigo con mutex y variables condición** y traducir a representación con semáforos del kernel
  - Este FIFO NO puede usarse con `echo` y `cat`
    - Usar programa `dump_utmp` para depurar el código
    - Hacer que capacidad máxima del buffer circular sea 512 bytes
  - Si se intenta realizar una **lectura o escritura de un número de bytes superior al tamaño máximo del buffer**, el módulo devolverá un error
  - 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

Parte B: FIFOv2

- **FIFOv2:** (cont.)
  - 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()`
  - El semáforo `sem_prod` se usa para *bloquear al productor*
  - El semáforo `sem_cons` se usa para *bloquear al consumidor*
  - Si cualquier proceso bloqueado en un semáforo **se despierta por la recepción de una señal**, la operacion 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: FIFOv2 (Pseudocódigo)

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

int fifodev_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);

    if (cons_count==0) {unlock(mtx); return Error;}

    /* Esperar hasta que haya hueco para insertar */
    while (nr_gaps_cbuffer_t(cbuffer)<len){
        cond_wait(prod,mtx);
    }

    insert_items_cbuffer_t(cbuffer,kbuffer,len);
    cond_signal(cons);

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

Parte B: FIFOv2

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

```
static int fifodev_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: FIFOv2

terminal 1

```
dsouser@debian:~/DumpUTMP$ sudo rm -f /var/tmp/fifo
dsouser@debian:~/DumpUTMP$ sudo mknod /var/tmp/fifo -m 666 c 251 0
dsouser@debian:~/DumpUTMP$ ./dump_utmp -s utmp
dsouser@debian:~/DumpUTMP$
```

terminal 2

```
dsouser@debian:~/DumpUTMP$ ./dump_utmp -r
(DEAD_PROCESS)              Mon Feb 23 08:29:34 2009    si  (0,0)
(BOOT_TIME) reboot      ~   Mon Feb 23 08:29:34 2009    ~~  (0,0)
(RUN_LVL)   runlevel        ~   Mon Feb 23 08:29:34 2009    ~~      (0,0)
(DEAD_PROCESS)              Mon Feb 23 08:31:53 2009    l2  (0,0)
(LOGIN_PROCESS) LOGIN       tty1    Tue Apr 14 12:18:49 2009    1   (0,0)
(USER_PROCESS)  root        tty2    Mon Feb 23 08:36:12 2009    2   (0,0)
(LOGIN_PROCESS) LOGIN       tty3    Mon Feb 23 08:38:07 2009    3   (0,0)
(LOGIN_PROCESS) LOGIN       tty4    Mon Feb 23 08:31:53 2009    4   (0,0)
(LOGIN_PROCESS) LOGIN       tty5    Mon Feb 23 08:31:53 2009    5   (0,0)
(INIT_PROCESS)              Mon Feb 23 08:31:53 2009    6   (0,0)
(LOGIN_PROCESS) LOGIN       ttyS0   Mon Feb 23 08:31:53 2009    T0  (0,0)
(USER_PROCESS)  dario       pts
```

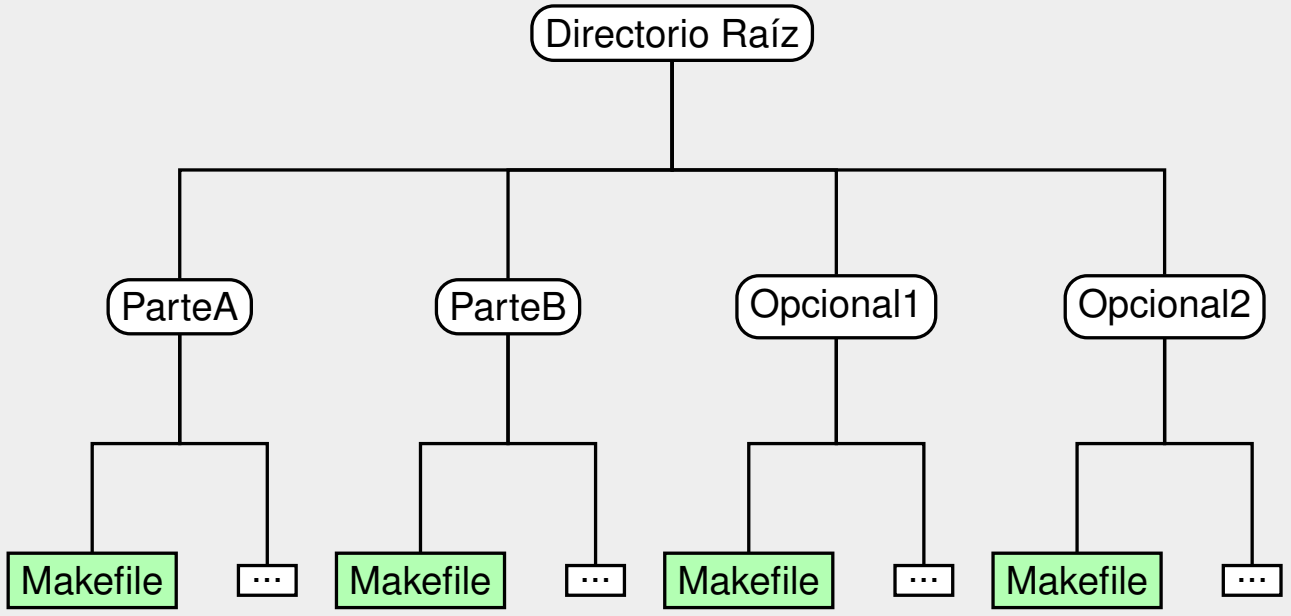
Partes opcionales

- **(Opcional 1)** Realizar una implementación *SMP-safe* de la Práctica 1 empleando *reader writer spin locks* (`rwlock_t`)
- **(Opcional 2)** Realizar una implementación *SMP-safe* de la parte A de la Práctica 2 empleando *spin locks* (`spinlock_t`)
  - La implementación debe garantizar protección en el acceso a la lista de entradas KIFS

Entrega de la práctica

- A través del Campus Virtual
  - Hasta el ~~9 de Enero~~ 10 de Enero
- Obligatorio mostrar el funcionamiento después de hacer la entrega

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



Licencia

DSO - Práctica 3: Procesos y Sincronización
Versión 0.1

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