Práctica 1.3: Comunicación y sincronización entre procesos

Índice

Objetivos

Esta práctica persigue que los estudiantes se familiaricen con los dos principales mecanismos de comunicación entre procesos disponibles en Linux: regiones de memoria compartida y tuberías. Además de utilizar estos dos mecanismos (en su variante con nombre) se hará uso de semáforos POSIX con nombre para establecer sincronización entre procesos.

Ejercicios

Ejercicio 1: El problema del productor-consumidor con procesos, semáforos y memoria compartida

En esta práctica se realizará la modificación de una variante de la implementación del problema productor-consumidor analizada en clase que emplea semáforos con nombre y memoria compartida. Como punto de partida se proporciona el código de dicha solución, que constituye una versión más robusta de la implementación disponible en las transparencias. Esencialmente, después de invocar llamadas al sistema esenciales, esta implementación consulta los valores de retorno de las llamadas para comprobar la existencia de errores y, en su caso, llevar a cabo el tratamiento correspondiente.

En este ejercicio se han de realizar los siguientes cambios en la implementación proporcionada:

1. Incluir la sincronización necesaria en el código para que el productor no comience a insertar números en el buffer circular compartido hasta que el proceso consumidor haya iniciado su ejecución y esté preparado para recibir los números (es decir, cuando haya invocado la función `Consumer()`). **Sugerencia**: Se puede utilizar un semáforo adicional inicializado por el productor, cuyo contador se inicializa a cero.
2. Modificar el mecanismo de comunicación de información entre el productor y el consumidor: en lugar de utilizar un fichero real para respaldar la región de memoria compartida (es decir, un fichero proyectado en memoria), se debe hacer uso de un objeto de memoria compartida POSIX, que se creará y al que se accederá mediante la llamada al sistema `shm_open()`. Consultar la página de manual asociada para más detalles y ejemplos sobre este tipo de zonas de memoria compartida: `man shm_open`

Ejercicio 2: Programa `fifotest`

En este ejercicio se ha de estudiar la implementación y el comportamiento del programa `fifotest` introducido en clase. Este programa tiene dos modos de uso que corresponden a las opciones `-s` (*send*) y `-r` (*receive*) de la línea de

comando. Para poder usar este programa es necesario lanzar dos instancias del mismo, cada una en un terminal diferente. Los dos procesos (emisor y receptor) se comunicaran entre sí a traves de un fichero FIFO creado previamente, cuya ruta se ha de pasar con la opción -f del programa.

En el ejemplo mostrado a continuación se ilustra cómo utilizar este programa de forma interactiva, ejecutando las instancias correspondientes de envío y recepción desde distintos terminales:

Terminal 1:

```
## Creamos fichero FIFO
osuser@debian:~/ejercicio2$ mkfifo myfifo

## Lanzamos instancia de envío, e insertamos cadenas de caracteres para enviar al "consumidor"
## Teclear CTRL + D para introducir el fin de fichero en la entrada estándar y acabar la comunicación
osuser@debian:~/ejercicio2$ ./fifotest -f myfifo -s
Hello
How ya doin
...
```

Terminal 2:

```
## Lanzamos instancia de recepción, y recibiremos las cadenas desde el otro extremo
osuser@debian:~/ejercicio2$ ./fifotest -f myfifo -r
Hello
How ya doin
...
```

Este programa puede ejecutarse para realizar envío de forma no interactiva, leyendo datos de un fichero de entrada en lugar de la información introducida con el teclado. Para usar este modo basta utilizar la redirección de entrada del shell como se muestra en el siguiente ejemplo, donde usaremos un fichero de texto ya existente en el directorio:

Terminal 1:

```
## Suponemos fichero fifo ya creado
## Lanzamos instancia de envío usando fichero existente como entrada al programa
osuser@debian:~/ejercicio2$ ./fifotest -f myfifo -s < test.txt
osuser@debian:~/ejercicio2$
```

Terminal 2:

```
## Lanzamos instancia de recepción, y recibiremos el contenido del fichero por el otro extremo
osuser@debian:~/ejercicio2$ ./fifotest -f myfifo -r
En un lugar de la Mancha, de cuyo nombre no quiero acordarme, no ha mucho tiempo que vivía un hidalgo de
osuser@debian:~/ejercicio2$
```

Tras estudiar la implementación del programa y ejecutarlo como se muestra en los ejemplos especificados, contesta a las siguientes preguntas:

- Ejecuta solamente la instancia de envío del programa en una terminal y de forma interactiva (sin redirección de entrada). Como se puede observar, este programa se quedará bloqueado en un punto concreto de su ejecución. ¿Qué llamada al sistema provoca el bloqueo? **Pista**: Para averiguarlo, se puede ejecutar el programa paso a paso con el depurador `gdb`, o bien utilizar el comando `strace` para extraer una traza de las llamadas al sistema que ejecuta el programa en tiempo real: `strace ./fifotest -f myfifo -s`. Consultar página de manual de `strace`.
- Habiendo ejecutado ya la instancia de envío, ejecuta ahora la de recepción en otra terminal. Ahora ambos procesos (emisor y receptor) se quedan bloqueados al invocar llamadas al sistema. ¿Cuáles son dichas llamadas? **Pista**: Usar herramienta `strace` para averiguarlo.
- Ahora procede al contrario que en el primer punto, es decir, lanza solamente la instancia de recepción del programa en una terminal sin lanzar la de envío. ¿En qué llamada al sistema se queda bloqueado ahora el programa?

- Una vez lanzadas ambas instancias del programa, este utiliza paquetes de datos de longitud fija para enviar información desde el proceso emisor al receptor. ¿Qué sucede al introducir una cadena acabada en `'\n'` en la terminal del emisor (lanzado de forma interactiva)? ¿Se espera el programa a construir un paquete de datos "completo" o envía la cadena tecleada directamente a pesar de ser más corta? ¿De qué tamaño es ese paquete enviado? **Pista**: Usar herramienta `strace` para averiguarlo.

Ejercicio 3: Desarrollo de un chat multihilo con FIFOs

Implementar el programa de usuario `chat.c`, un sencillo chat que hará uso de ficheros FIFO y múltiples hilos POSIX para la comunicación. El modo de uso del programa será el siguiente:

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

donde `usuario` es una cadena de caracteres arbitraria que representa el nombre del usuario que se "conecta" al chat. El programa usará un FIFO para enviar mensajes al otro usuario del chat y otro FIFO para recibir los mensajes. Las rutas de ambos ficheros FIFO –que han de estar ya creados– se ha de pasar como argumento, como consta en el modo de uso.

Para poder usar el chat será necesario lanzar dos instancias de dicho programa en distintos terminales del siguiente modo:

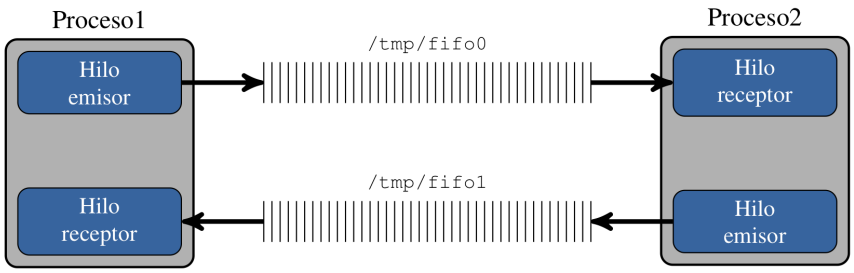
- Terminal 1: `$ ./chat <usuario1> <ruta-fifo1> <ruta-fifo2>`
- Terminal 2: `$ ./chat <usuario2> <ruta-fifo2> <ruta-fifo1>`

Es decir, en ambas instancias del programa se pasará la ruta del mismo par de ficheros FIFOs, pero en distinto orden. Al fin y al cabo, el FIFO de envío en una de las instancias del programa será el FIFO de recepción de mensajes para la otra instancia del programa.

Para la correcta utilización de los FIFOs en el modo de uso habitual de estos (modo bloqueante), el programa `chat.c` creará dos hilos (emisor y receptor) usando `pthread_create()`:

1. **Hilo emisor:** Este hilo leerá líneas por la entrada estandar (esperando la entrada del usuario). Al leer una línea, envíará un mensaje por el primer FIFO especificado en línea de comando
2. **Hilo receptor:** Este hilo permanecerá bloqueado hasta que se reciba un mensaje por el segundo FIFO especificado en línea de comando. Al recibir el mensaje, lo procesará y si procede mostrará información por pantalla

En la siguiente figura se muestra un ejemplo de ejecución del programa `chat.c`, así como un diagrama que ilustra la interacción entre los dos hilos (emisor y receptor) presentes en las dos instancias del programa. En el ejemplo, estos hilos se comunican usando los ficheros FIFO `/tmp/fifo0` y `/tmp/fifo1` previamente creados:




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

Menganito dice:  Cargando un dataframe de 10GB :(

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

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

Fulanito dice:  ¿Estas por ahí?

> Sí, Menganito, aquí ando
> Cargando un dataframe de 10GB :(
>
Fulanito dice:  bueno, hasta luego entonces
>ciao
>Conexión finalizada por Fulanito!!
$
```

Por simplicidad en la implementación, los datos transferidos a través de cada FIFO se representarán mediante el tipo de datos `struct chat_message`, donde se recogen distintos tipos de mensajes:

```c
#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;
};
```

El comportamiento del hilo emisor y receptor de mensajes se resume a continuación, indicando también el uso de los tres tipos de mensajes (`message_type_t`) durante el protocolo de comunicación:

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. Cierra FIFO de envío y sale

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