Prácticas de Arquitectura de Computadores

4 curso de Ingeniería de Telecomunicación. Curso 2000-2001
Tutor de prácticas: Daniel Morató

6 de noviembre de 2000

1 Introducción

El objetivo de estas prácticas es la creación de unas herramientas que permitan un cierto modelo de comunicación entre procesos.

El desarrollo se va a centrar en la creación de lo que correspondería a la parte que da la funcionalidad de comunicación dentro del sistema operativo y de una librería que provea de un interfaz de programación cómodo al usuario para emplear esas nuevas funcionalidades.

Para evitar la programación de un módulo en el espacio del kernel la funcionalidad de sistema se implementará como un proceso de usuario. La librería permitirá acceder a los servicios de ese proceso sin necesidad de conocer su implementación específica sino tan solo los fundamentos del servicio y la especificación del API.

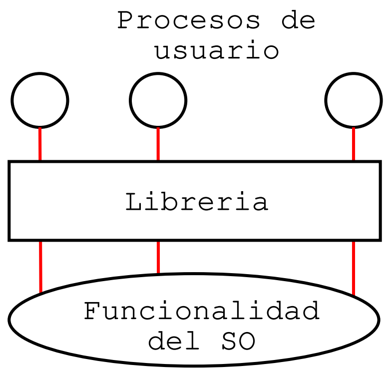

Figura 1: Esquema general

1.1 Esquema de comunicación

1.1.1 Extremo a extremo

Se pretende proveer de canales unidireccionales de comunicación entre procesos dependientes de diferentes *entidades de comunicación*. Estas *entidades* pueden encontrarse en el mismo o distintos ordenadores. Para realizar la comunicación extremo a extremo entre los procesos de usuario los datos pasarán a través de las entidades de comunicación mediante un paradigma de conmutación de paquetes.

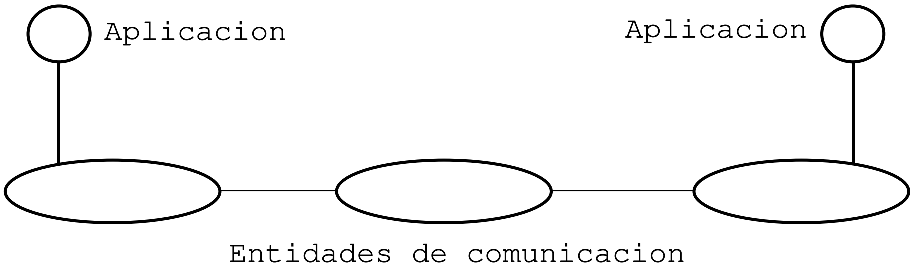

Figura 2: Esquema de comunicación

1.1.2 Paradigma cliente-servidor

Los extremos emplearán un modelo de cliente-servidor. Para establecer un canal de comunicación uno de los extremos deberá colocarse *a la espera* sobre un punto de acceso *conocido* sobre su entidad de comunicación. El flujo de datos se dirigirá desde la otra aplicación extremo (cliente) hacia ese punto de acceso conocido.

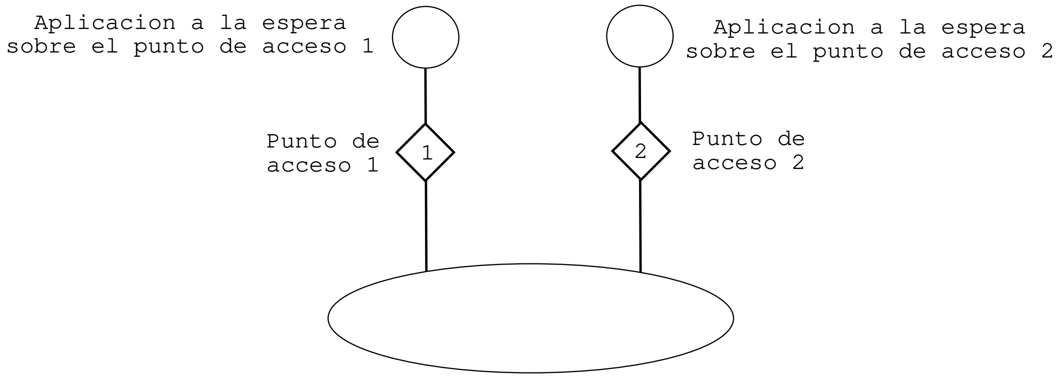

Figura 3: Puntos de acceso

1.1.3 Interconexión de entidades de comunicación

Las entidades de comunicación van a formar lo que se conoce como una topología, definida por el esquema de interconexión. Cada entidad puede tener uno o varios canales bidireccionales de comunicación permanentes con otras entidades. A través de estos canales fluyen los datos que se desean transmitir las aplicaciones, encapsulados con la infomación que precisen las entidades en paquetes de tamaño constante. Dos aplicaciones que deseen comunicarse entre sí pueden depender de entidades que no poseean un canal de datos directo entre ambas. En ese caso la comunicación se realizará a través de otros enlaces, pasando por otras entidades, hasta llegar al destino.

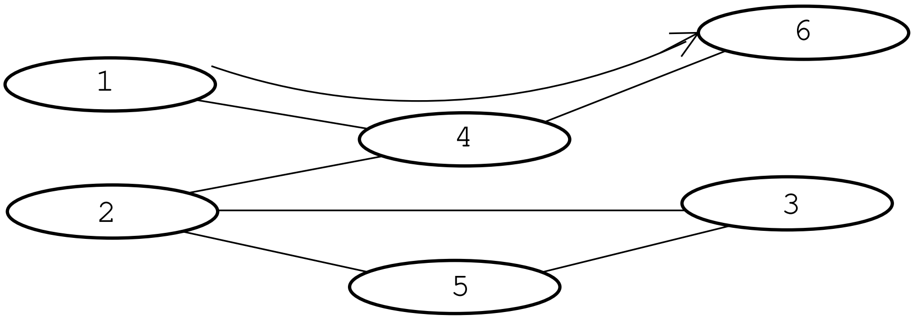

Figura 4: Ejemplo de topologia y de flujo indirecto

1.2 Visión general de la implementación

En este apartado se va a presentar el esquema global de funcionamiento de una *entidad de comunicación*.

En la figura 5 se ve este esquema. Encerrado en un óvalo se encuentran los diversos componentes que forman una entidad de comunicación. Las flechas representan canales de comunicación, que pueden emplear distintas técnicas de comunicación entre procesos. Todos los canales representados son unidireccionales. Así, la conexión entre dos entidades de comunicación viene dada por dos líneas, una para cada sentido de la comunicación. Cada entidad de comunicación está identificada por un número.

El bloque B realiza la función de multiplexar toda la información proviniente de otras entidades. Entrega esta información al bloque A. El bloque A a su vez se encarga de entregar a los servidores dependientes de ella los datos que se reciban para ellos y aquellos datos (de usuarios o de otras entidades) que no sean para usuarios de esta entidad son reenviados a otras entidades.

2 Programas y librerías

En esta sección se va a detallar el funcionamiento de cada una de las partes de la *entidad de comunicación*, así como de la librería y de la topología a emplear.

Comenzaremos por dar unas hipótesis sobre las características de los programas que implementarán los bloques A y B de la figura 5. Con ello se diseñará un programa que cree una sencilla topología de entidades interconectadas (práctica 1).

Tras esto pasaremos a la implementación del bloque B (práctica 2), cuya función es simplemente de multiplexación.

El bloque A contiene la mayor cantidad de funcionalidades y en su desarrollo (práctica 3) se verá cuál debe ser el esquema de la librería (práctica 4) para un correcto funcionamiento en conjunción con él.

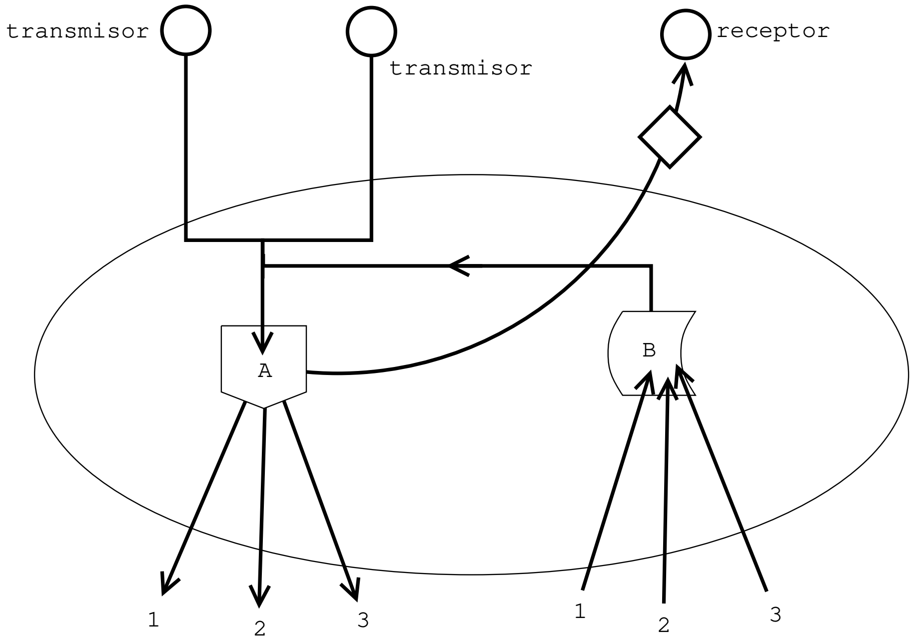

Figura 5: Implementación

Finalmente se propone la realización de unos programas que empleen el sistema de comunicación recién diseñado.

2.1 Especificación externa de los bloques A y B

En este apartado vamos a presentar el funcionamiento de los bloques A y B a alto nivel, lo necesario para poder lanzarlos creando una topología de entidades interconectadas.

Para simplificar se supondrá que cada entidad tiene 3 enlaces a otras entidades. Así pues, cada bloque de tipo A podrá enviar datos hasta a 3 entidades y cada bloque B recibir de hasta 3 entidades.

2.1.1 Bloque A

El programa que implementa el bloque A se va a llamar *nucleo*. La sintaxis de opciones del mismo es la siguiente:

nucleo IDesteServ IDservDesc3 IDservDesc4 IDservDesc5 clave

Donde *IDesteServ* es el indentificador de la entidad de comunicación de la que va a formar parte. *IDservDesc3*, *IDservDesc4* y *IDservDesc5* son los identificadores de las 3 entidades a las que está conectado y *clave* es un valor numérico que permite reconocer a este bloque A de otros dentro de una misma máquina y facilita la comunicación con el bloque B de la misma entidad mediante una cola de mensajes. A la entidad identificada con *IDservDesc5* se la conocerá también como *enlace o ruta por defecto*. Si se desea conectar la entidad a menos de 3 destinos bastará con realizar más de un enlace con el mismo, es decir, si solo se quiere conectar con otras dos entidades se podría hacer con *IDservDesc4=IDservDesc5*.

Para su correcto funcionamiento, la aplicación *nucleo* supone que sobre sus descriptores de fichero 3, 4 y 5 se encuentran los canales de comunicación (de escritura) que le enlazan con las entidades *IDservDesc3*, *IDservDesc4* y *IDservDesc5* respectivamente.

2.1.2 Bloque B

El programa que implementa el bloque B se va a llamar *muxin*. La sintaxis de opciones del mismo es la siguiente:

muxin clave desc1 desc2 desc3

Donde *clave* tiene el mismo valor que el del mismo nombre en el programa *nucleo* que junto con éste forma una entidad de comunicación.

Los valores *desc1*, *desc2* y *desc3* son los descriptores de fichero sobre los que se encuentras los canales de lectura cuyos datos de entrada debe multiplexar esta aplicación. Si no hay tantos canales de entrada se indica poniendo un -1 como descriptor que no se desee utilizar, por ejemplo una entidad con dos enlaces de entrada sobre los descriptores 4 y 5 y que emplee como clave 2100 se indicaría con *muxin 2100 4 5 -1*.

2.2 Creación de topología (práctica 1: 2 ptos)

El programa a crear en esta práctica debe disponer en una máquina la topología de entidades de comunicación que se ve en la figura 6

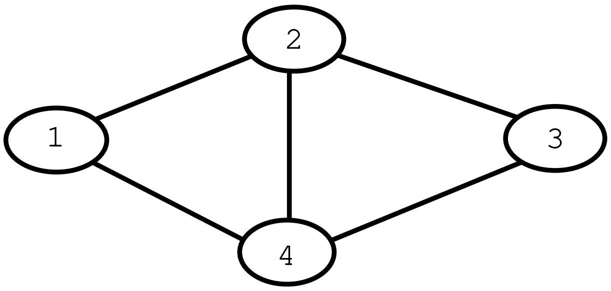

Figura 6: Topología

Dado que los programas *nucleo* y *muxin* necesitan descriptores de fichero como puntos de acceso a los canales de comunicación una solución lógica es el empleo de *pipes* o tuberías como técnica de comunicación entre procesos.

La aplicación que va a crear la topología se va a llamar *lanzador* y su cometido será crear el número de *pipes* y procesos necesarios, colocar las pipes sobre los descriptores adecuados de cada proceso y ejecutar los programas *nucleo* y *muxin* con las opciones que corresponda. Tras crear la topología, *lanzador* termina su ejecución.

2.2.1 Requisitos

Para realizar esta práctica se dispone de unas versiones de los programas *nucleo* y *muxin*.

Se necesitará el empleo al menos de las siguientes llamadas al sistema de UNIX:

fork(): Para la creación de los nuevos procesos.

execvp(): Para la ejecución de los programas dentro de los nuevos procesos creados.

pipe(): Para crear las pipes.

dup2(): Para colocar los descriptores de las pipes en el lugar deseado.

close(): Para eliminar los descriptores que no se necesiten.

Nota: Tenga en cuenta que cada vez que emplee la llamada *pipe()* el sistema reservará dos nuevos descriptores de la tabla de descriptores de fichero del proceso. Dado que inicialmente es probable que los descriptores 3, 4 y 5 estén libres pueden ser empleados por las pipes. Un método simple de evitar esto es ocuparlos antes de crear las pipes, tal vez duplicando sobre ellos otro ya existente.

Para aprender el manejo de estas llamadas al sistema se sugiere la realización antes de los ejercicios 1-8.

2.2.2 Pruebas sugeridas

Se ha dejado en el directorio $(HOME)/../ficheros dos programas llamados *nucleog* y *muxing*. Son versiones de los programas *nucleo* y *muxin* que dan cierta información por la salida estándar cada vez que reciben un paquete. Pueden emplearse para comprobar que el programa *lanzador* ejecuta los programas correctamente. Igualmente en ese directorio hay dos programas *cliente_test1* y *servidor_test1* que son dos simples programas que emplean la librería *libcommarq.a* para comunicarse. Se pueden emplear para comprobar que la interconexión entre los programas nucleo y muxin es correcta. Su sintaxis de empleo es la siguiente:

servidor_test1 IDcolaSuEntidad PuntoDeAcceso

Donde *IDcolaSuEntidad* es la clave empleada en los programas nucleo y muxin que forman la entidad que va a emplear este programa y *PuntoDeAcceso* es el valor de punto de acceso al servicio que va emplear.

cliente_test1 IDcolaSuEntidad IDEntidadRemota PuntoDeAccesoRemoto

Donde *IDcolaSuEntidad* es la clave empleada en los programas nucleo y muxin que forman la entidad que va a emplear este programa, *IDEntidadRemota* es el identificador de la entidad sobre la que trabaja el servidor (el identificador, no la clave de la cola) y *PuntoDeAccesoRemoto* es el valor de punto de acceso que está empleando el servidor con el que se quiere comunicar.

El programa servidor espera leer un paquete, imprime los primeros 4 bytes de datos del mismo como un número entero sin signo y termina la ejecución. El programa cliente envía un paquete con un número entero aleatorio al servidor, valor que también saca por la salida estándar para que se compruebe la llegada correcta del valor.

Está disponible también el código fuente de estos dos programas para que se compruebe la sencillez de uso de la librería que se va a desarrollar posteriormente.

2.3 Encapsulado

Cada aplicación que vaya a emplear este método de comunicación, para especificar el destino, debe dar el ID de la entidad destino y el punto de acceso de la aplicación destino sobre esa entidad. Ambos son identificadores numéricos enteros.

Como ya se ha comentado, el flujo de datos entre los dos extremos se va a producir en paquetes de tamaño constante, aunque la cantidad de datos dentro de cada paquete puede variar. La *entidad de comunicación* o la librería especificará el destino y origen de un paquete añadiendo una cabecera al mismo antes de reenviarlo por uno de sus enlaces (figura 7). En el paquete hay además un campo para indicar cuántos bytes de datos reales hay en el paquete y otro para indicar el número máximo de entidades de comunicación por las que puede pasar el paquete. Cada entidad reducirá en 1 este valor y si alcanza el valor 0 el paquete es descartado. Esto evita que se mantenga indefinidamente en un bucle que pueda existir en la topología.

El formato del paquete de datos puede verse en la figura 8

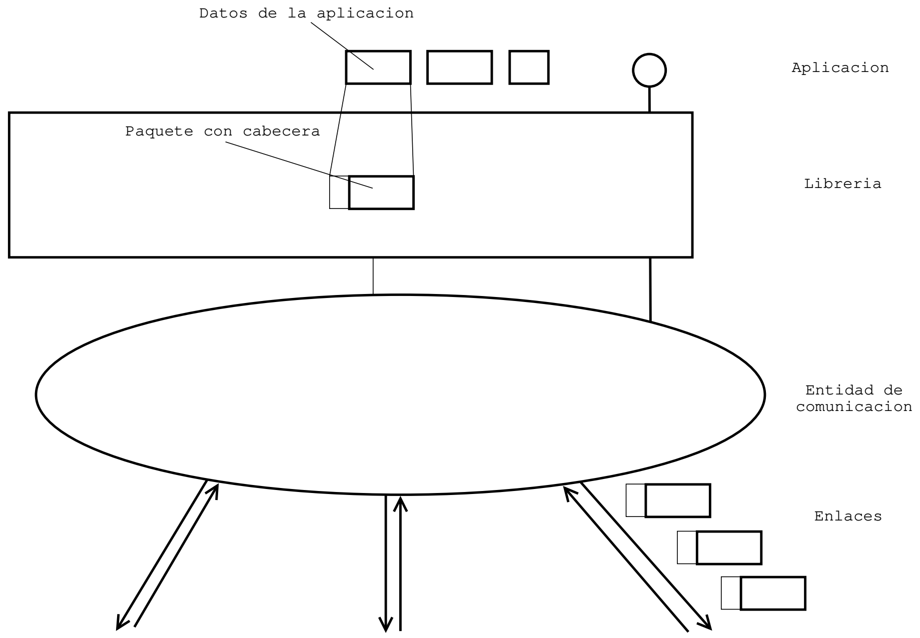

Figura 7: Encapsulado

2.4 Bloque de multiplexación o bloque *B* (práctica 2: 2 ptos)

En esta práctica se va a crear el programa *muxin*, cuyo interfaz con el usuario ya ha sido descrito.

muxin debe leer de 3 descriptores. Al recibir un paquete por uno cualquiera de ellos decrementa el campo *maxsaltos* del mismo. Si el nuevo valor de *maxsaltos* es mayor que cero transfiere el paquete al bloque A de su entidad. Esta transferencia se realiza a través de una cola de mensajes, la cual transportará el tráfico multiplexado de los tres descriptores entrantes. Si el valor resultante de *maxsaltos* es menor o igual que cero descarta silenciosamente el paquete.

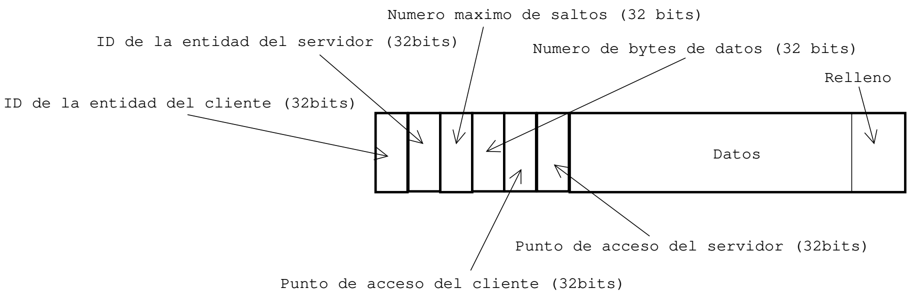

Figura 8: Formato de la cabecera

La cola de mensajes para la comunicación entre los bloques A y B de una entidad viene determinada por la clave entregada como primer parámetro del programa *muxin* y como último parámetro de *nucleo*. Esta cola es creada por el programa *nucleo*.

La lectura simultánea por los 3 enlaces entrantes se hará bloqueante mediante la llamada al sistema *select*.

Así pues el programa *muxin* debe:

1.- Acceder a la cola de comunicación con el bloque A.

2.- Bloquearse mediante un *select* a la espera de que llegue algún paquete por alguno de los 3 enlaces entrantes.

3.- Decrementar el campo *maxsaltos* del paquete.

4.- Enviar dicho paquete al bloque A de su entidad a través de la cola de mensajes si maxsaltos es estrictamente positivo.

4.- Volver al punto 2.

El formato que se sugiere para el mensaje con el paquete de datos es el siguiente:

```
typedef struct msg_data
{
     int tipo; // Tipo de mensaje MSG_DATA
     union
     {
      char buf[sizeof(paquete)]; // El paquete en si
      paquete paq;
     } data;
} msg_data;
```

Donde la estructura del paquete es la misma que se transfiere por las *pipes* y como se ve en la figura 8 es:

```
typedef struct paquete
{
     unsigned int id_serv_origen;
     unsigned int id_serv_destino;
     unsigned int maxsaltos;
     unsigned int numbytesdatos;
     unsigned int id_proc_origen;
     unsigned int id_proc_destino;
     char buf[TAM_MDU];
} paquete;
```

2.4.1 Requisitos

Se necesitará el empleo al menos de las siguientes llamadas al sistema de UNIX:

msgget() msgsnd() msgctl(): para controlar la cola de mensajes

select(): para esperar simultáneamente a la llegada de información a través de varios descriptores de fichero

read(): para tomar los datos que vienen del enlace

Para aprender el manejo de estas llamadas al sistema se sugiere la realización antes de los ejercicios 9-11.

Tenga en cuenta que las colas de mensajes, así como los semaforos o las zonas de memoria compartida, siguen existiendo aunque no haya ningún programa empleandolas. Hay que destruirlas explícitamente. Si en alguna ocasión necesita borrar una de éstas desde un terminal consulte las utilidades *ipcs* e *ipcrm*.

2.4.2 Pruebas sugeridas

Tomando el programa $(HOME)/../ficheros/nucleog y el programa *muxin* desarrollado se sugiere crear la topología de la práctica 1 (u otra cualquiera) y comprobar el correcto funcionamiento de la comunicación con los programas de prueba *cliente_test1* y *servidor_test1*. Igualmente se deja disponible una version del programa que crea la topología llamado *lanzadorg*.

2.5 Bloque de control y reenvío o bloque *A* (práctica 3: 4 ptos)

En esta práctica se va a crear el programa *nucleo*, cuyo interfaz con el usuario ya ha sido descrito.

Las tareas de este programa son múltiples y se resumen a continuación:

- Recibir los paquetes de datos que desean enviar los clientes que trabajan sobre esta entidad
- Recibir los paquetes provinientes de otras entidades y que han sido recogidos de los enlaces por el programa *muxin*

o Reenviar los paquetes que no sean para esta entidad

o Atender a las solicitudes de los servidores sobre esta entidad para reservar un punto de acceso específico

o Entregar a los servidores los paquetes destinados a sus puntos de acceso

Tanto paquetes de datos como solicitudes de reserva se reciben por el mismo canal de comunicación. Este canal es la cola de mensajes que tiene como clave el último parámetro de la invocación del programa *nucleo* y es creada por éste. El ciclo de ejecución de este programa se resume en:

1.- Crear y acceder a cola de mensajes

2.- Recibir mensaje, que será un paquete de datos o una solicitud

3.- Procesar el mensaje

4.- Volver al punto 2

Veremos cuáles son las acciones que debe emprender el programa ante cada tipo de mensaje que reciba por esta cola.

2.5.1 Paquetes de datos hacia el exterior

Los paquetes de datos vienen identificados por ser mensajes de la cola de tipo MSG_DATA. Todos los paquetes de datos siguen el mismo esquema (figura 8) vengan de un cliente sobre esta entidad o del bloque B correspondiente, se dirijan al exterior o a un servidor en esta entidad. Los paquetes son de tamaño constante y contienen toda la información necesaria para indicar cuál es su cliente de origen y su servidor destino.

Cuando el bloque A recibe un paquete de datos tendrá que comprobar primero si la entidad destino es él o es otra diferente. El caso en que es él el destino se resuelve en otra sección. Si el paquete es para otra entidad comprobará si es una con la que tenga comunicación a través de los enlaces que tiene establecidos. De ser así se lo envía directamente. Si el destino no es ninguna de esas entidades ni es para él entonces lo envía por el enlace que se ha definido como *enlace o ruta por defecto*. Estos reenvíos se realizan escribiendo el paquete sobre el descritor correspondiente. Para evitar bloqueos del programa ante saturación de los enlaces la escritura sobre estos descriptores se hará NO bloqueante.

Para simplificar la realización posterior de la librería, el programa núcleo hará una pequeña modificación en los paquetes: si el identificador de la entidad origen es 0 lo sustituye por su identificador de entidad. Se considera pues que 0 no es un identificador válido. Los paquetes que reciba con este valor serán los provinientes de los clientes sobre él a través de la librería.

2.5.2 Solicitud de punto de servicio

Cuando una aplicación quiere actuar como servidor debe eligir un número que le sirva de *punto de acceso al servicio*. Este valor, junto con el que identifica a la entidad de comunicación es lo que necesita conocer el cliente para ponerse en comunicación con este servidor. Este valor le identifica entre todos los posibles servidores sobre la misma entidad de comunicación, así pues no puede haber dos servidores en la misma entidad que empleen el mismo punto de acceso al servicio. Dado que una aplicación no tiene por qué tener conocimiento de los puntos de acceso que se están empleando resolveremos esto pidiendo permiso al bloque A para emplear un valor de punto de acceso al servicio. El bloque A guardará un registro de los puntos de acceso que se están empleando en un momento dado para autorizar a un servidor el uso de uno o rechazárselo.

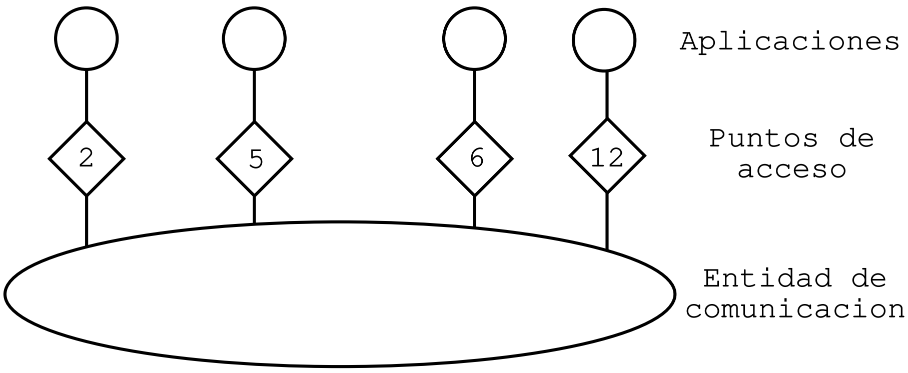

Figura 9: Puntos de acceso al servicio

Para implementar esta funcionalidad emplearemos un mensaje de control que la librería enviará en favor de la aplicación a la entidad de comunicación. Este mensaje irá por la cola de mensajes de la cual espera datos el bloque A y lo reconocerá por no tener el mismo tipo que los datos sino otro diferente. En este mensaje, cuya estructura se incluye a continuación, se especificará que se desea emplear un cierto valor como punto de acceso.

```
typedef struct msg_ctrl_accept
{
     int tipo; // Tipo de mensaje MSG_ACCEPT
     int id_cli; // Valor de punto de acceso que se desea emplear
     key_t clavecola; // clave de la cola de mensajes por la que se
                              espera la confirmacion/rechazo
} msg_ctrl_accept;
```

El bloque A deberá comprobar si existe ya alguna aplicación empleando este punto de acceso y mandará un mensaje de vuelta a la aplicación autorizando su uso o rechazándolo. Dado que las colas de mensajes son unidireccionales el bloque A no podrá contestar por el mismo canal por el cual ha recibido la solicitud. Para resolver esto la librería, antes de enviar este mensaje de solicitud, creará una nueva cola de mensajes y enviará su clave al bloque A dentro del mensaje de solicitud de reserva de punto de acceso (campo *clavecola*). Para enviar la respuesta, el bloque A accederá a la cola cuya clave ha obtenido en el mensaje y enviará por ella la respuesta. Tras enviarla se deshace de esta cola auxiliar y el servidor destruye la cola al recibir por ella la respuesta. Este esquema se puede ver en la figura 10 y la estructura del mensaje de respuesta asignando ese punto de acceso es la siguiente:

```
typedef struct msg_ctrl_accept_ack
{
     int tipo; // Tipo de mensaje MSG_ACCEPT_ACK
     key_t clavemem; // clave de la zona de memoria compartida
     key_t clavesem1_paquetes; // clave del semaforo 1
     key_t clavesem2_huecos; // clave del semaforo 2
} msg_ctrl_accept_ack;
```

En caso de que ese valor de punto de acceso ya se esté empleando enviará un mensaje diferente para indicar tal contingencia:

```
typedef struct msg_fin
{
     int tipo; // Tipo de mensaje MSG_FIN
     int id_cli; // El valor que no se ha podido reservar
} msg_fin;
```

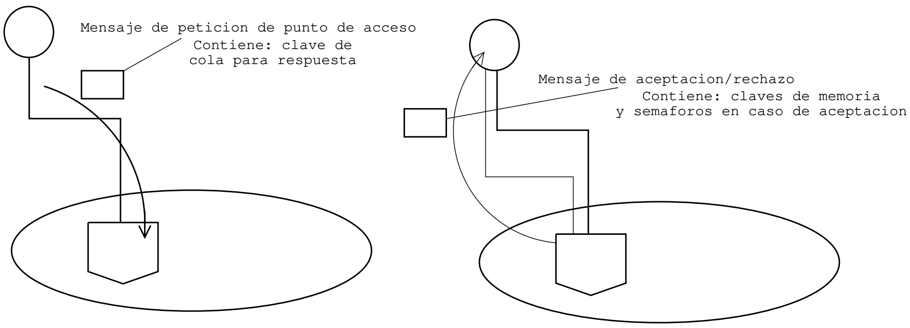

Figura 10: Peticion de punto de acceso al servicio

Es evidente que el bloque A (implementado en el programa *nucleo*) deberá recordar todos los valores de puntos de acceso al servicio que están siendo empleados. Para ello se recomienda emplear una lista dado que este bloque no sabe cuántos servidores van a existir y le conviene ir reservando y liberando puntos de acceso con tan solo las limitaciones de memoria y sin desperdiciarla.

2.5.3 Paquetes de datos para un servidor

Con este esquema el servidor consigue reservar un punto de acceso. Sin embargo hemos visto que no podemos emplear la cola de mensajes principal para enviar los datos al servidor dado. Una posible solución es emplear la cola auxiliar que se crea en el proceso de reserva para enviar los datos desde la entidad al programa. Sin embargo destruiremos esta cola al recibir el mensaje de aceptación y adoptaremos una solución diferente. Para la comunicación desde la entidad de comunicación al servidor vamos a emplear una zona de memoria compartida entre el proceso que implementa las funciones del bloque A y el proceso servidor (figura 11).

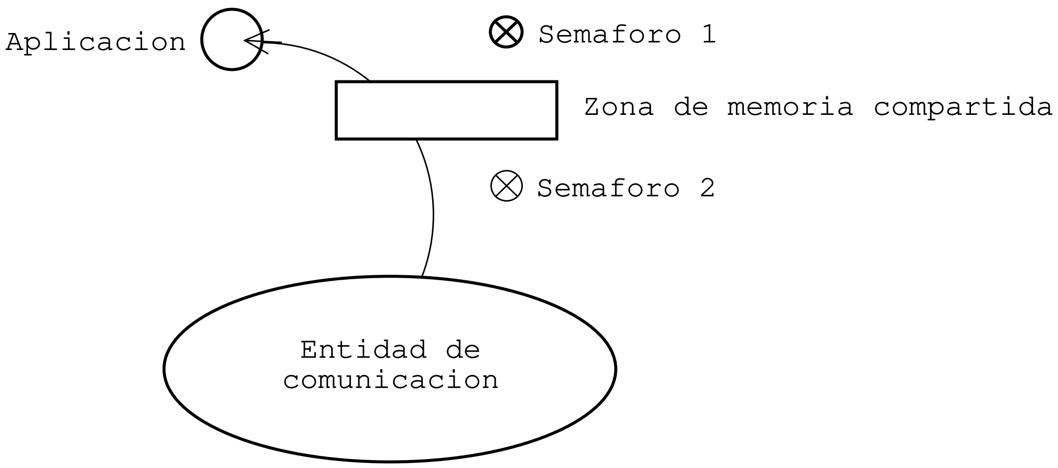

Figura 11: Comunicación a través de memoria compartida

En esta zona de memoria podremos almacenar varios paquetes simultáneamente y emplearemos una estrategia de *buffer circular* para ir colocando y extrayendo los paquetes. Englobaremos esta zona de memoria compartida en una sección crítica mediante el empleo de semáforos. Este implementación nos ahorrará las copias de datos que se realizarían en el caso de emplear una cola de mensajes, donde el mensaje ha de pasar al kernel (una copia) y del espacio del kernel al del otro proceso (segunda copia), por lo que obtendremos mejores prestaciones.

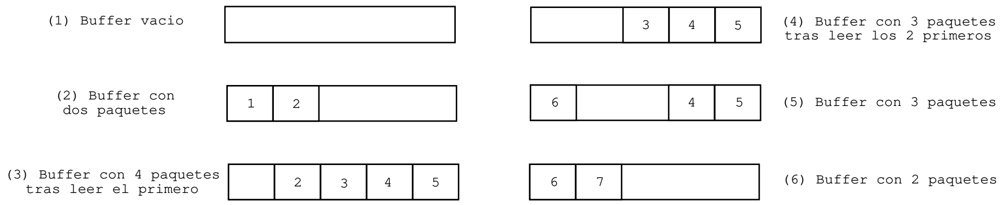

Figura 12: Empleo de un buffer circular

Para el control de esta zona de memoria (en realidad para cada zona de memoria por separado, una por cada punto de acceso) se emplearán dos semáforos multivaluados. Uno marcará el número de paquetes sin procesar (sem1_paquetes) que existen en el buffer y el otro (sem2_huecos) el número de huecos libres en el mismo. Así, para colocar un nuevo paquete, el bloque A deberá realizar los siguientes pasos:

1.- Comprobar primero que quede algún hueco libre realizando una función P sobre *sem2_huecos*. Dado que los paquetes son de tamaño fijo el espacio libre se puede medir en unidades de paquetes.

2.- Colocar el paquete en el siguiente hueco libre (para recorrer este buffer de forma circular se puede emplear aritmética *modulo* el tamaño del buffer)

3.- Ejecutar una función V sobre *sem1_paquetes* para indicar que hay un nuevo paquete sin procesar.

Puede suceder que la aplicación no atienda a la llegada de nuevos paquetes por un punto de acceso y se llene el buffer circular del mismo. En tal caso la operación P ejecutada en el paso 1 podría bloquear al proceso que implementa el bloque A de forma indefinida. Para evitar esto se realizará la operación P sobre sem1 de forma *no bloqueante*. Esto quiere decir que si no puede decrementar en 1 el valor del semáforo, en vez de detener al proceso a la espera de poder devolverá un error en la operación. Ante este error, debido al llenado del buffer, el bloque A procederá a descartar silenciosamente el paquete de datos que intentaba entregar a la aplicación.

Como vemos, al reservar un nuevo punto de acceso hace falta crear una zona de memoria compartida y 2 semáforos. Esta tarea la realiza el bloque A al comprobar que el valor de punto de acceso no está siendo utilizado por otra aplicación. Para que el proceso servidor sepa qué zona de memoria y semáforos se empleará en la comunicación se envían sus claves dentro del mensaje de respuesta de aceptación de reserva del punto de acceso.

2.5.4 Liberación de un punto de acceso

Una vez que un servidor haya terminado de utilizar un punto de acceso debe notificarlo a su entidad de comunicación para que ésta libere la memoria y semáforos y marque ese valor de punto de acceso como *libre*. Para ello se enviará un mensaje al bloque A indicándole dicha finalización. Este mensaje tan solo contiene el valor del punto de acceso a liberar y no requiere confirmación. El formato del mismo es el siguiente:

```
typedef struct msg_fin
{
     int tipo; // Tipo de mensaje MSG_FIN
     int id_cli; // El valor del punto de acceso que se pretende
                              liberar
} msg_fin;
```

2.5.5 Resumen del funcionamiento del bloque A

1.- Crear la cola de mensajes por la que leerá paquetes de datos y comandos.

2.- Bloquearse hasta recibir un mensaje por esa cola.

3.- Si el mensaje es un paquete de datos:

.1.- Si no es para esta entidad reenviarlo por el canal que conecte con su destino o por el enlace por defecto de no existir camino directo.

.2.- Si es para esta entidad y exite un servidor con el punto de acceso indicado en el paquete colocárselo en su zona de memoria compartida (si hay espacio) y actualizar los semáforos de la misma.

.3.- Volver al punto 2

4.- Si el mensaje es una petición de reserva de punto de acceso:

.1.- Buscar si este punto de acceso está siendo utilizado

.2.- Si ya está en uso devolver el mensaje de error por la nueva cola y volver al punto 2.

.3.- Si no está en uso crear la zona de memoria y semaforos, apuntar el valor del punto como reservado, devolver el mensaje de aceptación por la nueva cola y volver al punto 2.

5.- Si el mensaje es una indicación de liberación de punto de acceso indicarlo como liberado, eliminar la zona de memoria compartida y los semáforos y volver al punto 2.

2.5.6 Requisitos

Se necesitará el empleo al menos de las siguientes llamadas al sistema de UNIX:

msgget() msgsnd() msgctl() msgrcv(): para controlar las cola de mensajes

shmget() shmmat() shmdt() shmctl(): para controlar la zona de memoria compartida

semget() semop() semctl(): para controlar los semáforos. Para hacer las funciones no bloqueantes basta con activar el flag IPC_NOWAIT.

write(): para reenviar paquetes a otras entidades de comunicación

fcntl(): para hacer los descriptores no bloqueantes (O_NONBLOCK)

Para aprender el manejo de estas llamadas al sistema se sugiere la realización antes de los ejercicios 9-13.

Los tipos de datos están definidos en el fichero $(HOME)/../ficheros/tiposcomm.h

2.5.7 Pruebas sugeridas

De nuevo se pueden emplear todos los programas entregados cambiando *nucleog* por el programa *nucleo* desarrollado en esta práctica para comprobar su correcto funcionamiento.

2.6 Librería (práctica 4: 2 ptos)

Esta librería va a dar a las aplicaciones un método sencillo de uso de este sistema de comunicación mediante un interfaz de programación de aplicaciones (API). El fichero de la librería se llamara *libcommarq.a* y vendrá acompañada del fichero de cabeceras *commarq.h*.

Se define el concepto de *endpoint* como el punto de acceso al canal de comunicación, el cual estará configurado para lectura o para escritura. Descritas brevemente las funciones del API van a ser:

endpoint *newendpoint(endpoint_info *info);

Crea un nuevo *endpoint* con las características que se especifican en el parámetro *info*. Entre las características se encuentra que se vaya a emplear para lectura (servidor) o para escritura (cliente).

int envia(endpoint* desc, const void* data, int tam);

Envía por un *endpoint* de escritura un paquete con número de bytes de datos igual a *tam*, los cuales se encuentran en la zona de memoria apuntada por *data*.

int recibe(endpoint* desc, void* data);

Recibe de un *endpoint* de lectura un paquete de datos, los cuales se colocan en la dirección apuntada por *data*.

int closendpoint(endpoint *desc);

Destruye un *endopint*.

int putserverentry(int clave);

Especifica cuál es la clave para acceder a la entidad de comunicación local.

2.6.1 int putserverentry(int clave);

Esta es la primera función que se debe emplear dado que sirve para que la librería sepa cómo ponerse en contacto con la entidad de comunicación que va a emplear. Se le da la clave que ha de servir para acceder a la cola de mensajes que permite enviar comandos y paquetes de datos al bloque A de la entidad.

Si tiene éxito devuelve 0, en caso de error -1.

2.6.2 endpoint *newendpoint(endpoint_info *info);

Esta función crea un nuevo *endpoint* que sirve para poder emplear las funciones de enviar y recibir.

```
struct info_r
{
     int id_cli; // valor del punto de acceso
};
```

```
struct info_w
{
     int id_serv; // id del entidad de comunicaci\'on destino
     int id_proc; // valor del punto de acceso de la aplicaci\'on
                            destino
};

typedef struct endpoint_info
{
     char tipo; // 'r' o 'w'
     union
     {
          struct info_r data_r;
          struct info_w data_w;
     } data;
} endpoint_info;
```

El *endpoint* puede ser para lectura o para escritura. Esto viene determinado por el contenido del campo *tipo* que será 'r' para lectura o 'w' para escritura. El resto del contenido de la estructura será de un tipo u otro según sea para especificar un punto de lectura o de escritura. En el caso de punto de escritura se indica la entidad de comunicación destino sobre la cual trabaja el servidor y el valor del punto de acceso del servidor. En el caso de lectura se indica el valor del punto de acceso que se desea emplear.

La aplicación que emplee este API no necesita conocer el contenido de la estructura *endpoint* dado que no la empleará directamente sino a través de funciones del API. Sin embargo esta es la estructura que se sugiere para el *endpoint*:

```
typedef struct endpoint
{
     endpoint_info info; // El parametro de la funcion newendpoint()
     int idnucleo; // ID de la entidad sobre la que trabaja
     // Informacion para un endpoint de lectura
     int idmem; // Identificador de la memoria compartida empleada
     void *mem; // Puntero al comienzo de esa zona de memoria
     int idsem1_paquetes,idsem2_huecos; // Identificadores de los
                    semaforos. 1=numero paquetes, 2=numero huecos
     int paqleyendo; // Numero del siguiente paquete de la zona de
                               memoria (de 0 al numero maximo-1)
     // Infomracion para un endpoint para escritura
     int id_local; // Identificador de aplicacion origen a emplear
} endpoint;
```

En esta estructura la librería puede guardar toda la información necesaria para gestionar el *endpoint*. En el caso de un *endpoint* para escritura contiene la información necesaria del destino y del origen, donde como *id_local* se empleará en PID del proceso. En el caso de un *endpoint* para lectura contiene además los identificadores para emplear la memoria compartida y semáforos de comunicación entre el bloque A y la aplicación.

En el caso de la creación de un *endpoint* de escritura basta con almacenar en la nueva estructura la información necesaria. En el caso de uno de lectura además hay que reservar el valor de punto de acceso. Para ello la función *newendpoint()* envía un mensaje al bloque A indicando el valor que se desea emplear. El mensaje puede ser la estructura *msg_ctrl_accept* presentada a continuación:

```
typedef struct msg_ctrl_accept
{
     int tipo; // Tipo de mensaje MSG_ACCEPT
     int id_cli; // Valor de punto de acceso que se desea emplear
     key_t clavecola; // clave de la cola de mensajes por la que se es-
pera la confirmacion/rechazo
} msg_ctrl_accept;
```

El mensaje de respuesta que envía el bloque A como aceptación es:

```
typedef struct msg_ctrl_accept_ack
{
     int tipo; // Tipo de mensaje MSG_ACCEPT_ACK
     key_t clavemem; // clave de la zona de memoria compartida
     key_t clavesem1; // clave del semaforo 1
     key_t clavesem2; // clave del semaforo 2
} msg_ctrl_accept_ack;
```

O en caso de rechazo envía:

```
typedef struct msg_fin
{
     int tipo; // Tipo de mensaje MSG_FIN
     int id_cli; // El valor que no se ha podido reservar
} msg_fin;
```

Si no se ha podido reservar ese valor de punto de acceso la función *newendpoint* devuelve un indicador de error. Una vez tiene las claves de la memoria y los semáforos la librería puede acceder a ellos obtienendo los identificadores que guardará en la estructura *endpoint*.

Su valor de retorno es el nuevo *endpoint* o NULL en caso de error.

2.6.3 int envia(endpoint* desc, const void* data, int tam);

Esta función necesita un *endpoint* de escritura. Envía *tam* bytes que se encuentran a partir de la dirección apuntada por *data*. La cantidad de bytes no puede superar el tamaño máximo de datos en un paquete TAM_MDU y si son menos esta función incluye más hasta completar el paquete, teniendo en cuenta que no importa el valor de estos bytes de relleno dado que con el campo *numbytesdatos* se sabe perfectamente dónde acaban los bytes útiles del paquete.

Para enviar los datos la función construye un *paquete* que engloba dentro de un mensaje de tipo *msg_data* y lo envía a la entidad de comunicación a través de la cola de mensajes. El campo *id_serv_origen* lo rellena con 0 y el campo *id_proc_origen* con el PID de la aplicación cliente.

Su valor de retorno es el número de bytes de datos que se entregan a la entidad de comunicación. Un error se indica con un -1.

2.6.4 int recibe(endpoint* desc, void* data);

Esta función necesita un *endpoint* de lectura. Intenta leer un paquete de datos que haya llegado a ese punto de acceso y colocar los datos en la zona de memoria apuntada por *data*. Para obtener un nuevo paquete los pasos que sigue son:

1.- Realizar una operación P sobre el semáforo sem1_paquetes

2.- Tomar el siguiente paquete de la zona de memoria teniendo en cuenta que se emplea de forma circular

3.- Realizar una operación V sobre el semáforo sem2_huecos para indicar que hay un nuevo hueco libre

La operación P es bloqueante, de forma que si no hay datos en la memoria la función se bloquea hasta que lleguen. Una vez obtenido un paquete copia *los datos* del mismo a la zona de memoria *data* y da como valor de retorno la cantidad de bytes de datos que había en el paquete.

2.6.5 int closendpoint(endpoint *desc);

Esta función destruye un endpoint liberando toda la información y memoria asociada a él. En el caso de un endpoint de escritura basta con liberar la estructura *endpoint*. Para un endpoint de lectura hace falta notificar a la entidad de comunicación que ya no se va a emplear más ese punto de acceso. Para ello se envía un mensaje de tipo MSG_FIN al bloque A.

```
typedef struct msg_fin
{
     int tipo; // Tipo de mensaje MSG_FIN
     int id_cli; // El valor del punto de acceso a liberar
} msg_fin;
```

A continuación se deja de emplear la zona de memoria y los semáforos, así como se libera la estructura *endpoint*. El bloque A se encarga de destruir la zona de memoria compartida y los semáforos.

2.6.6 Pruebas sugeridas

Dado que se dispone de las fuentes de los programas de ejemplo *servidor_test1* y *cliente_test1* se pueden linkar sus objetos con la librería recién creada y comprobar que la comunicación funciona correctamente.

2.7 Programas cliente y servidor (trabajo opcional)

En el código ejemplo de los programas *cliente_test1* y *servidor_test1* se puede ver cómo es el procedimiento de empleo del API proporcionado por la librería.

Para completar las pruebas de comunicación se lanzarán entidades de comunicación en varias máquinas del laboratorio interconectadas a través de la Ethernet empleando protocolos de Internet (TCP/IP). Dado que la implementación en UNIX de TCP/IP permite el acceso a las conexiones mediante descriptores de fichero se puede reutilizar los programas realizados sin más que reemplazar las pipes que interconectan las entidades de comunicación por conexiones de red. Para conocer el estado de este entorno de pruebas consulte la pagina web de la asignatura o hable con el tutor de prácticas.

2.7.1 Talk

En esta práctica se va a crear una pareja de programas de comunicacion textual. La sintaxis de los programas que se sugiere es la siguiente:

servidor_talk claveEntidadServidor puntodeacceso

cliente_talk claveEntidadCliente IDEntidadServidor puntodeacceso

Donde *claveEntidadCliente* y *claveEntidadServidor* son las claves de las entidades sobre las que trabajan cada uno de los dos programas y *IDEntidadServidor* y *puntodeacceso* son los valores que hacen falta para contactar con el servidor.

Ambos programas tienen el mismo funcionamiento de cara al usuario: permiten que escriba líneas de texto, las cuales mandan al otro extremo y cada vez que llega un línea de texto del otro usuario la imprimen por pantalla.

2.7.2 Transferencia de ficheros

Para finalizar se creará una pareja de programas que permita transferir ficheros. La sintaxis puede ser la siguiente:

receptor_file claveEntidadServidor puntodeacceso

emisor_file claveEntidadCliente IDEntidadServidor puntodeacceso nombrefile

El programa *emisor* emplea la entidad identificada por *claveEntidadCliente* y contacta con el programa *receptor* en la entidad *IDEntidadServidor* y el punto de acceso de valor *puntodeacceso*. El *emisor* transfiere el fichero de nombre *nombrefile*.

El programa *receptor* espera sobre su entidad y ese punto de acceso. Al recibir un nuevo fichero lo guarda en su directorio de trabajo con un nombre que bien puede ser escogido por él o ser el nombre original del mismo. Una vez finalizada la transferencia continua su ejecución a la espera de otra transferencia que guardará en un fichero diferente.

2.8 Retoques y temas avanzados (trabajo opcional)

La librería y los programas así construidos tendrán un funcionamiento correcto en el caso de que nada falle en la parte de usuario. Si se plantea la posibilidad de que el programa de usuario finalice sin cerrar los *endpoints* de lectura que está empleando hace falta buscar un método para que la entidad de comunicación reconozca este suceso y libere los puntos de acceso que estuviera empleando. Una posibilidad es que el bloque A haga comprobaciones de cuántos procesos están empleando una zona de memoria compartida, si solo hay uno es él debido a que haya desaparecido el proceso lector. Esto se puede comprobar mediante una llamada *shmctl* del tipo IPC_STAT.

Por otro lado tambien hace falta una forma cómoda de detener los programas *nucleo* y *muxin* que se encargue de eliminar con ellos los posibles semáforos, colas de mensajes y zonas de memoria compartida que estuvieran empleando. Una posibilidad es poner una función que se ejecute al recibir las señales SIGINT o SIGTERM y que antes de terminar el proceso elimine todas las zonas de memoria compartida, los semáforos y la cola de mensajes.

3 Ejercicios

1.- Escriba un programa que cree un nuevo proceso y a partir de ahí padre (proceso original) e hijo (nuevo proceso) continúen indefinidamente escribiendo cada uno una línea de texto diferente por pantalla cada segundo.

2.- Escriba un programa que cree un nuevo proceso. El proceso padre debe esperar a que el proceso hijo finalice. El proceso hijo escribe una cuenta atrás desde 10 a ritmo de 1 por segundo y termina. Cuando el padre detecta la finalización del hijo imprime una despedida por pantalla y termina su ejecución.

3.- Escriba un programa que emplee alguna llamada *exec* para ejecutar el comando *ls*.

4.- Escriba un programa que lance un nuevo proceso. El hijo ejecuta un *ls*. El padre espera a que el hijo termine y a continuación crear un nuevo proceso. El nuevo hijo ejecuta un *date*. El padre espera su finalización y al detectarla imprime una despedida por la pantalla y termina.

5.- Escriba un programa que cree una pipe, a continuación escriba un array de *char* conteniendo una cadena de texto en esa pipe e inmediatamente después lea ese array por el extremo de lectura de la pipe y lo imprima por la pantalla.

6.- Escriba un programa que cree una pipe y a continuación lance un nuevo proceso. Despues el proceso padre escribe una cadena de texto en la pipe, espera 5 segundos y termina. El hijo lee de la pipe caracteres de uno en uno y los va sacando por la pantalla (emplee fflush(stdout) para que salgan inmediatamente) hasta que la función *read* le indique que no hay más datos para leer.

7.- Escriba un programa que abra un fichero nuevo y duplique su descriptor sobre el de la salida estándar (STDOUT_FILENO). A continuación escriba un mensaje de texto por la salida estándar y compruebe que al finalizar el proceso el mensaje se encuentra en el fichero.

8.- Escriba un programa que abra un fichero nuevo y duplique su descriptor sobre el de la salida estándar (STDOUT_FILENO). A continuación cree un nuevo proceso. El padre puede terminar su ejecución pero el hijo ejecutará el comando *ls*. Compruebe que al terminar todos los procesos la salida del *ls* se ha almacenado en el fichero.

9.- Escriba un programa que cree una cola de mensajes y acceda a ella. A continuación lanza un nuevo proceso. El padre crea un mensaje y lo coloca en la cola. El hijo espera hasta recibir el mensaje por la cola.

10.- Escriba un programa que lance un nuevo proceso. A continuación el hijo realiza una espera (*sleep*) de 5 segundos. Mientras, el padre crea una cola de mensajes mediante una clave concreta y manda un mensaje a la misma. Al transcurrir los 5 segundos el hijo intenta acceder a la cola con la misma clave que se creó y recibir el mensaje. Finalmente el hijo destruye la cola de mensajes.

11.- Escriba un programa que cree dos pipes y lance un proceso. El proceso padre se dedicará a mandar un bytes por la primera pipe cada segundo y otro por la segunda cada 2 segundos. El proceso hijo emplea la llamada select para saber cuándo hay algo que recibir por alguna de las pipes y va mandando los caracteres a pantalla.

12.- Escriba un programa que cree un semáforo y lo inicialice a 0. A continuación lanza un nuevo proceso. El proceso padre irá ejecutando una función V sobre el semáforo cada segundo. Mientras, el proceso hijo está continuamente intentando decrementar el semáforo con una función P. Cada vez que uno de ellos ejecuta una funcion P o V con éxito sacan un mensaje de texto por la pantalla.

13.- Escriba un programa que cree una zona de memoria compartida y la incluya en su mapa de memoria. A continuación lanza un nuevo proceso. El hijo realizará una espera de 10 segundos, mientras tanto el padre copia una cadena de texto a esa zona de memoria. Al terminar la espera de 10 segundos el hijo imprimer por pantalla el contenido de la zona de memoria compartida.

14.- Escriba un programa que realice una pausa infinita pero cada vez que reciba una señal de tipo SIGTERM o SIGINT saque un mensaje por pantalla y continúe esperando otra.

4 Fechas de entrega y formato de las prácticas

El último día para entregar las prácticas es el **Martes 24 de Abril a las 21:30 horas**. A partir de ese momento se bloquearán las cuentas.

Para la fecha mencionada debe haber en el directorio de cada grupo de prácticas un directorio correspondiente a cada práctica. Los directorios se llamarán *practica1*, *practica2*, *practica3* y *practica4*. Dentro de cada directorio deben estar los ficheros *.c* y *.h* necesarios para compilar el programa o librería que se pida en esa práctica, así como un fichero *Makefile* de forma que con hacer *make* en cada directorio se obtenga en los mismos el resultado requerido (programa ejecutable o librería en formato *.a*). También debe haber un fichero $(HOME)/componentes con los nombres de los miembros del grupo de prácticas en dos líneas de texto, las dos únicas líneas que debe contener el fichero.

Para la corrección de las prácticas se borrarán todos los programas ejecutables y códigos objeto, por lo que se deberá poder obtener las soluciones a partir solo de las fuentes.

Para cualquier duda consulte primero la pagina web de la asignatura [1], donde irán apareciendo las aclaraciones a las prácticas que se consideren necesarias.

Podrán recibir información respecto a aparición de notas y otros temas si se subscriben a la lista de correo del Laboratorio de Telemática, para lo cual existen instrucciones en el mismo laboratorio.

[1] http://www.tlm.unavarra.es/asignaturas