Grupo 8
SOPER P3

Gregorio Blázquez Martínez
Eduardo Terrés Caballero

PRÁCTICA 3

Ejercicio 1: Creación de Memoria Compartida. Dado el siguiente código en C:

a) El código consiste en intentar abrir un segmento de memoria compartida, en modo lectura y escritura, con *flags* que indican a la función *fd_shm* que debe comprobar la exclusividad del nombre del segmento. Así pues, en caso de ya existir el espacio de memoria con el nombre indicado por *SHM_NAME* – comprobación mediante *errno* -, se tratará de abrir, esta vez eliminando los *flags* que indicaban exclusividad. El primer *else* actúa a modo de control de errores; el segundo *else* notifica que la primera llamada a la apertura del segmento se ha realizado con éxito.

b) Añadiendo el flag *O_TRUNC* y eliminando *O_EXCL* se podrá forzar la apertura del segmento, con el inconveniente de que se borrará la totalidad de los datos que almacene en ese momento.

Ejercicio 2: Tamaño de Ficheros

a) Mediante el siguiente código se consigue el tamaño del fichero, que resulta ser 12 bytes.

```
/* Get size of the file. */
if (fstat(fd, &statbuf) == -1){
    perror("fstat");
}
fprintf(stdout,"Tamaño de fichero: %d\n", (int) statbuf.st_size);
fflush(stdout);
```

b) Mediante el siguiente código se cambia el tamaño del fichero a 5 bytes. Al abrirlo, la información que contiene es: “Test “, incluyendo el espacio.

```
/* Truncate the file to size 5. */
ftruncate(fd,5);
```

Ejercicio 3: Mapeado de Ficheros

a) La primera ejecución crea el fichero y establece el valor del entero a 1. Posteriores llamadas incrementan el valor en una unidad, ya que el flag *create* únicamente se activa – vale 1 – en la primera llamada.

b) No se puede leer el contenido con un editor de texto, dado que se trata de un archivo binario. Así pues, se almacena el valor del número en representación binaria, en vez del valor de los caracteres *ASCII* que representarían este número.

Ejercicio 4: Memoria Compartida

a) No, porque los requisitos que se le otorgan al lector son de tipo *read only*, para poder mermar la capacidad de acción que tiene sobre la memoria compartida. Permitir que elimine la referencia, y por ende forzando la eliminación de la memoria compartida, impide que posteriores lectores accedan a la memoria, ya que el nombre a la zona de memoria queda eliminado. En otras palabras, si se incluye el *unlink*, una segunda ejecución del programa lector fallaría por no encontrar el segmento de memoria.

b) De nuevo, es inherente a la decisión de que el programa lector únicamente tenga posibilidad de lectura el hecho de no permitir que modifique la información. Cambiar el tamaño del fichero puede dar lugar a la eliminación de información, y esta decisión no sería consistente con limitar el alcance del lector sobre la memoria compartida.

c) La llamada a *shm_open* permite obtener el descriptor de fichero, mientras que *mmap* permite una mayor flexibilidad en el uso que le da el programa a la memoria compartida, a través de diferentes modos de acceso a la misma – *MAP_PRIVATE, MAP_SHARED, MAP_ANONYMOUS* -. Además, se puede manejar la memoria, a través de la referencia obtenida mediante *shm_open*, sin necesidad de mapearla - f*truncate* -, así como llamar a *mmap* sin que el descriptor de fichero sea obtenido mediante *shm_open*.

d) Como se ha explicado, en Linux se puede tener acceso a memoria compartida que empleen procesos accediendo a los directorios */dev/shm* o */proc/<pid>/maps*. Como ya se ha explicado, otra forma de usar la memoria sin mapearla se da mediante llamadas a *ftruncate*.

Ejercicio 5: Envío y Recepción de Mensajes en Colas

a) Los mensajes se envían en orden ascendente – primero el 1, luego el 2, y así consecutivamente -. Se recibe primero el mensaje con prioridad 3, luego los de prioridad 2 y finalmente los de prioridad 1. El orden de recibo de mensajes para una prioridad concreta queda determinado por el orden de envío.

b) Al cambiar *O_RDWR* por *O_RDONLY*, la salida es la siguiente:

Sending: Message 1
mq_send: Bad file descriptor

En este caso, no el proceso no tiene permiso para el envío de mensajes, y se produce un error.

Al cambiarlo por *O_WRONLY*, la salida es la siguiente:

Sending: Message 1
[…]
Sending: Message 6

mq_receive: Bad file descriptor

El error se encuentra ahora en el recibo de mensajes: el programa no tiene privilegios para recibirlos.

Gregorio Blázquez Martínez
Eduardo Terrés Caballero

Ejercicio 6: Colas de Mensajes

a) El programa *mq_sender* envía el mensaje y entra en espera del *getchar* para su finalización. Tras ejecutar *mq_receptor*, se imprime el mensaje, que ya se encontraba en la cola.

b) El programa *mq_receptor* se queda bloqueado – ya que no se ha especificado la señal *O_NONBLOCK* -, en espera del mensaje. Una vez se ejecuta el programa *mq_sender*, el receptor recibe e imprime el mensaje.

c) Al ejecutar primero el *sender*, el resultado es el mismo. No obstante, al ejecutar primero el *receptor*, este proceso no se bloquea, y la llamada a *mq_receive* devuelve error (*mqd_t* -1). La salida de *mq_receptor* es la siguiente:

Error receiving message

d) No. Si se pretende enviar mensajes dirigidos a ciertos procesos, sería pertinente crear diferentes colas de mensajes. El uso de semáforos implica dos esperas, una en la cola del propio semáforo, y otra en la cola de mensajes. Esto puede dar lugar a fugas, por ejemplo, en caso de que falle el envío de un mensaje. Si por el contrario lo que se pretende es asegurar que no se produzca condición de carrera, el *kernel* del *SO* se encarga de evitarlo.