Sistemas Operativos. Práctica 3.

Pablo Cuesta Sierra y Álvaro Zamanillo Sáez

Índice

Memoria compratida

Ejercicio 1

a) Primero (línea 2), se intenta crear un segmento de memoria compartida, con las opción `O_EXCL`, si existe el segmento, `shm_open` devuelva error (-1). Después (línea 3), se trata este error y, en caso de que el error se deba a que el segmento con ese nombre ya existía, se abre este segmento (esta vez sin usar la flag de crear). Si el error no era porque ya existiera o si al intentar abrir el segmento ya existente hay otro error, se llama a `perror` y termina el programa.

b) Para forzar la inicialización (en la primera llamada), habría que quitar la opción `O_EXCL` (para que si existe, no se devuelva error), y añadir la opción `O_TRUNC`, con la que, si existe, se trunca a tamaño 0. De este modo, se forzaría la inicialización.

Ejercicio 2

a) Para obtener el tamaño basta con:

```
if (fstat(fd, &statbuf) != 0) {
    perror("fstat");
    exit(EXIT_FAILURE);
}
```

Y tras esto, el tamaño se encuentra en `statbuf.st_size`.

b) Para truncar a 5 bytes:

```
if (ftruncate(fd, 5) != 0) {
    perror("ftruncate");
    exit(EXIT_FAILURE);
}
```

El fichero resultante contiene: `Test␣`. Es decir, solamente los 5 primeros carácteres.

Ejercicio 3

a) Cada vez que se ejecuta el programa, el counter incrementa su valor en 1. Esto es porque el programa abre el fichero en caso de que ya exista, y varía el valor que ya existe. Si borramos el fichero antes de ejecutar, el fichero se crea de nuevo y tras la ejecución el valor vuelve a ser 1.

b) El fichero es binario, por lo que las variables no aparecen con los valores en forma de carácteres que se puedan leer con un editor de texto.

Ejercicio 4

a) No, ya que de ese modo, ningún otro lector podría abrir el segmento de memoria compartida.

b) No, porque se supone que es el escritor el primero que accede a la memoria compartida, para escribir en ella, por lo que debe ser el que determine su tamaño.

c) `mmap` crea un mapeo del segmento de memoria compartida dentro de las direcciones virtuales del proceso, por tanto se puede manejar como si esta memoria perteneciera al heap del proceso, con más facilidad que escribir en un fichero.

d) Sí se puede, habría que escribir en el segmento de memoria compartida con `write` tratándolo como un descriptor de fichero.

Ejercicio 5

a) Se envían en orden creciente (1,2,3..) pero se reciben según la prioridad con la que fueron enviados. Y para mensajes de la misma prioridad, siguiendo un criterio FIFO (first in, first out).

b) Si se cambia por `O_RDONLY` la llamada a `mq_send` devuelve error y el proceso termina. Del mismo modo, si se cambia por `O_WRONLY` el error ocurre al intentar recibir los mensajes.

Ejercicio 6

a) El emisor envía el mensaje y el receptor lo recibe e imprime por *stdout*

b) El receptor se bloquea en la orden `mq_receive` hasta que el emisor envía el mensaje.

c) Si se ejecuta primero el emisor, el receptor recibe el mensaje y lo imprime correctamente. Si se ejecuta primero el receptor, este no se bloquea hasta recibir el mensaje, sino que la función `mq_receive` devuelve error el estar la cola de mensajes vacía y se imprime el mensaje por *stderr* (`.Error receiving message"n`).

d) (Asumimos que se crea la cola de mensajes como bloqueante.) No, ya que si un receptor llega antes que otro, el segundo se quedará esperando al siguiente mensaje en caso de que solo haya uno; o recibirá el siguiente mensaje de la cola en caso de que haya más.

Ejercicio 7

a) Memoria compartida

El proceso *stream-ui* tiene que inicializar el segmento de memoria por lo que en la llamada a `shm_open` incluimos las *flags* `O_CREAT` y `O_EXCL` de forma que si ya existe un segmento de memoria con el mismo nombre,la función devuelva error.

En cuanto a los dos procesos que se lanzan *stream-server* y *stream-client*, tras la llamada a `fork` ejecutamos `execl` pasando como argumento el nombre del fichero de entrada o salida dependiendo del caso.

Los procesos *stream-server* y *stream-client* abren la memoria compartida con las *flags* `PROT_READ` y `PROT_WRITE` ya que van a escribir y leer (para acceder a los contadores por ejemplo) en la memoria.

b) Semáforos

Los semáforos, como son anónimos, los inicializa el proceso *stream-ui* dentro de la memoria compartida (con `sem_init`), y los destruye cuando los otros dos procesos ya han terminado. Para las esperas con tiempo máximo (2 segundos), hemos creado una función auxiliar (`st_timed_wait`), que las encapsula. A pesar de que, con las instrucciones del enunciado, no nos quedaba muy claro si había que usar `sem_timedwait` también para el *mutex*, lo hemos usado también ahí. En caso de que se supere el tiempo de espera, los procesos *stream-server* y *stream-client* desechan la operación y vuelven a esperar mensajes.

c) Colas de instrucciones

El proceso *stream-ui* tiene que crear las dos colas de mensajes y como solo va a enviar mensajes, añadimos la *flags* `O_WRONLY` a parte de `O_CREAT` y `O_EXCL`. Por su parte, el proceso cliente y servidor, abren las colas, pero en este caso con la *flag* `O_RDONLY` pues solo van a recibir mensajes.

En lugar de mandar las cadenas `exit`, `post` y `get`, mandamos un *int* que las identifica (definidos en macros), para no estar comparando cadenas en todos los procesos, que es menos conveniente.

Finalmente, si se recibe el mensaje asociado al comando *exit*, ambos procesos, cliente y servidor, salen del bucle y finalizan liberando sus recursos. Del mismo modo, si el servidor llega al final del fichero, o el cliente lee el carácter `'\0'`, saldrán del bucle e ignorarán cualquier orden (mensaje) que les llegue que no sea *exit*.