INFORME PRÁCTICA 3
SOPER

Álvaro Castillo García

Pablo Ernesto Soëtard

Universidad Autónoma de Madrid

Escuela Politécnica Superior

Ingeniería Informática bilingüe, grupo 2292

Sistemas Operativos

ÍNDICE

EJERCICIO 1

a)

Este programa intentará crear un segmento de memoria compartida con el nombre “/shared” con los permisos de escritura y lectura. En el caso de que ya existiese un segmento con ese nombre simplemente lo abre. El programa comprueba si la función shm_open se ejecuta corretamente y en su defecto imprime en la salida de errores un mensaje.

b)

shm_open() crea o abre un descriptor de fichero de memoria compartida, mientras que mmap() enlaza al espacio de direcciones del proceso, el segmento de memoria compartida creado.

El hecho de que existan dos funciones diferentes, genera la posibilidad de que un proceso cree un segmento de memoria compartida y no lo utilice. De esta manera otros procesos podrán acceder a ese segmento sin necesidad de crear otro nuevo.

c)

Para forzar dicha inicialización fuera del código, podríamos dirigirnos al directorio */dev/shm* y borrar el segmento de memoria compartido, que tuviera el nombre “/shared”.

Por otro lado, dentro del programa, podríamos llamar a la función shm_unlink(), para que borre tanto la memoria, como el nombre del segmento cuando no quede ningún proceso con el descriptor de fichero abierto o el segmento enlazado al espacio de direcciones.

EJERCICIO 2

a)

Este programa intentará crear un segmento de memoria compartida con el nombre “/shared” con los permisos de escritura y lectura. En el caso de que ya existiese un segmento con ese nombre simplemente lo abre. El programa comprueba si la función shm_open se ejecuta correctamente y en su defecto imprime en la salida de errores un mensaje.

b)

Para inicializar el tamaño de memoria a 1000B podríamos usar la función ftruncate(int *fd,* off_t *length*), tomando como fd la variable fd_shm y como length 1000.

Por otra parte, si se quiere estar seguro de que futuras ejecuciones no vuelven a inicializar o destruir lo ya inicializado, deberíamos comprobar si la variable errno es igual a la flag EEXIST y en ese

caso simplemente truncar el segmento con un tamaño concreto. En caso contrario y si shm_open no ha devuelto -1, entonces es que se ha creado un segmento de memoria nuevo.

c)

Para forzar dicha inicialización fuera del código, podríamos dirigirnos al directorio */dev/shm* y borrar el segmento de memoria compartido, que tuviera el nombre “/shared”.

Por otro lado, dentro del programa, podríamos llamar a la función shm_unlink(), para que borre tanto la memoria, como el nombre del segmento cuando no quede ningún proceso con el descriptor de fichero abierto o el segmento enlazado al espacio de direcciones.

EJERCICIO 3

a)

El programa que describe este ejercicio se encuentra en el archivo *shm_concurrence.c*

Se usan las librerias standard: <stdlib.h> y <stdio.h>, asi como la de manejo de errores <errno.h> , funciones matemáticas <math.h> y strings <string.h>. También se incluyen <unistd.h>, <sys/types.h> y <sys/wait.h> para poder crear procesos hijos y coordinarlos con el padre. La librería <signal.h> es necesaria para todas las funciones que manejan las señales. Por otra parte encontramos librerias para el manejo de secciones de memoria compartida <fcntl.h>, <sys/mman.h> y <sys/stat.h>. Finalmente están las librerias para controlar la hora <sys/time.h> y <time.h>.

Hemos decidido crear el número aleatorio en función del id del proceso con srand(getpid()). El número aleatorio se generará de esta manera: ((801.0)*rand()/(RAND_MAX + 1.0)+ 100)/1000). Así nos aseguramos que sea completamente aleatorio y se encuentre en el rango pedido (100ms-900ms).

b)

El planteamiento del apartado anterior, tiene un problema, y es que no garantiza la exclusión mutua. Todos los procesos hijos deberán escribir en la estructura, y el padre tendrá que leer en esta. Cómo los procesos se ejecutan concurrentemente y no hay un método de controlar el acceso a los campos de la estructura, lo más probable es que se sobreescriban resultados y los datos no sean correctos al leerlos.

c)

El programa que describe este ejercicio se encuentra en el archivo *shm_concurrence_solved.c*

Ademas de las mismas librerias descritas en el apartado a), incluye <semaphore.h> para poder usar las funciones de semáforos.

Para resolver este problema hemos creado un semáforo (mutex) dentro de la estructura. También son necesarias las variables globales n y m.

El proceso padre quedará en sigsuspend hasta que se active la flag sigusr1f. Tras esto esperará a los hijos.

Por otro lado, el manejador de SIGUSR1, cada vez que lea en la estructura comparará el id del log más 1 con la multiplicación de n por m , para saber si se ha leído el contenido de todos los hijos (n) todas y cada una de las veces (m). Si ambos valores coinciden, activará la bandera sigusr1f.

Los procesos hijos realizan un down() al semáforo sin nombre (mutex) tras dormir un número aleatorio de milisegundos. Como el semáforo se ha inicializado a 1, sólo un proceso podrá entrar en la sección crítica al principio.

Todos los hijos escribirán en la estructura, mandaran la señal SIGUSR1 al padre y si no han escrito m veces, volverán a dormirse y harán un down al semáforo.

El manejador de la señal SIGUSR1 tras leer en la estructura, hace un up del sémaforo mutex, permitiendo así que otro proceso hijo pueda acceder a la sección crítica.

EJERCICIO 4

a)

Los programas que describe este ejercicio se encuentran en los archivos archivo *shm_producer.c, shm_consumer.c, cola_prod_consum.c y cola_prod_consum.h*

Se usan las librerias standard: <stdlib.h> y <stdio.h>, asi como la de manejo de errores <errno.h> y strings <string.h>. Por otra parte encontramos librerias para el manejo de secciones de memoria compartida <fcntl.h>, <sys/mman.h> y <sys/stat.h>. Y aquellas necesarias para el manejo de semáforos <semaphore.h>.

Este programa implementa el problema del productor consumidor mediante semáforos y el uso de memoria compartida como “almacén”, que en este caso es una cola circular que al igual que los programas principales hemos tenido que implementar.

b)

Los programas que describe este ejercicio se encuentran en los archivos archivo *shm_producer_file.c, shm_consumer_file.c, cola_prod_consum.c y cola_prod_consum.h*

Las librerias son las mismas que las del apartado anterior.

El único cambio a realizar para este apartado es conseguir un descriptor de fichero de un archivo en vez de de una memoria compartida, para ello sustituimos las llamadas a shm_open() por open(), pasándole el nombre del archivo donde guardar la cola y los parámetro adecuados.

EJERCICIO 5

a)

Si ejecutamos primero el emisor y tras él el receptor, observamos que el primero se ejecuta correctamente y el segundo recibe el mensaje *29: Hola a todos*. Esto se debe a que el primer proceso crea un mensaje y lo envía a la cola de mensajes. Cuando el segundo proceso llama a mq_recieve leerá el mensaje que el emisor a escrito. Tras ejecutar así ambos podremos presionar cualquier tecla para que acaben.

b)

En este caso el receptor quedará a la espera de la llegada de un mensaje, que será enviado posteriormente por el emisor. En pantalla aparece la frase *29: Hola a todos*.

Podemos comprobar que mq_receive y mq_send son bloqueantes.

Si la cola de mensajes está vacía, el proceso que llama a mq_recieve se quedará bloqueado esperando a que llegue algún mensaje, o bien a que alguna señal interrumpa la ejecución llamando al manejador.

Por otra parte si la cola de mensajes está llena el emisor al llamar a mq_send, quedará bloqueado. Como este no es el caso, se observa que el receptor se bloquea hasta que el emisor llama a mq_send y el proceso receptor puede leer el nuevo mensaje de la cola.

c)

Al crear la cola de mensajes como no bloqueante, si ejecutamos primero el receptor y luego el emisor, aparecerá el mensaje *Error receiving message*, ya que la llamada a mq_recieve devuelve -1 con errno igual a EAGAIN. Esta bandera indica que la cola de mensajes está vacía y que se ha establecido la bandera O_NONBLOCK. Como el receptor no se bloquea hasta que le llegue algún mensaje, intentará leer de la cola vacía en ese momento, retornando error.

En caso de ejecutar primero el emisor y luego el receptor, este último imprime por pantalla el mensaje *29: Hola a todos*. Ocurrirá lo mismo que en el apartado a). El emisor enviará el mensaje y el receptor lo leerá de la cola, ya que esta no se encuentra vacía.

EJERCICIO 6

a)

El ejercicio descrito se encuentra en los archivos *mq_injector.c* y *mq_workers_pool.c.*

Se usan las librerias standard: <stdlib.h> y <stdio.h>, asi como la de manejo de errores <errno.h> y strings <string.h>. Por otra parte encontramos la libreria para el manejo de colas de mensajes <mqueue.h> y para constantes <fcntl.h>y <sys/stat.h>. Y aquellas necesarias para el manejo de señales <signal.h> y de procesos <unistd.h> <sys/types.h> <sys/wait.h>

En ambos programas, se abre una cola con un nombre pasado por parámetro y hemos asumido que el usuario va a introducir un nombre correcto. No hemos comprobado que empieze por "/" y que no aparezca ese carácter más veces.

Por otra parte el tamañomáximo de los mensajes es de 2KB+1, ya que si se leen 2KB del fichero de una vez, tendrá que caber el carácter '\0' para indicar que termina la string del mensaje.

El mensaje que indica que el proceso mq_injector ha finalizado de leer y enviar el archivo es "He acabado".

b)

El ejercicio descrito se encuentra en el archivo *mq_workers_pool_timed.c.*

Incluye las mismas librerias que el apartado a).

Para realizar este apartado, se nos ocurrió al principio usar la función ualarm() y establecer el manejador para la señal SIGALRM. Pero luego nos dimos cuenta de que existe la función mq_timedreceive, en la que se puede definir un tiempo límite para el cuál el proceso quedará bloqueado intentando leer un mensaje de la cola.

Por tanto, simplemente hemos cambiado la llamada a mq_recieve por mq_timedreceive, creando una estructura timespec y estableciendo dentro tv_nsec a 100000000 (100ms). También hemos añadido la comprobación de errno = ETIMEDOUT en el caso de que la llamada a la función anterior devuelva -1. Esto significará que ha expirado el tiempo indicado y el proceso no ha leído nada de la cola mientras que estaba bloqueado en el mq_timedreceive. Si esto ocurre el proceso imprime lo que ha hecho hasta el momento y expresa que no se le necesita más, liberando sus recursos y finalizando.