SOPER: Práctica 2

Adrián Lorenzo Mateo, Ari Handler Gamboa
Grupo 221, Equipo 07

Ejercicios de memoria compartida y semáforos

1. Modifícalo en un programa1.c para que cree la memoria compartida sólo en caso de que no exista previamente. Si ya existiese sólo debe mostrar un mensaje indicando que ya está creada.

Se ha modificado el argumento shmflg de la función shmget, siendo ahora 'IPC_CREAT | IPC_EXCL' para asegurarse el fallo si el segmento ya existe.

2. Modifícalo de nuevo en un programa2.c para que libere la memoria compartida al salir sólo si fue él quien la creó. En caso contrario debe mostrar un mensaje en pantalla indicando el identificador de la memoria.

Se ha añadido la directiva **shmctl** con el argumento cmd a 'IPC_RMID' para la eliminación de la memoria compartida.

Hacemos una llamada al comando de shell **ipcs** antes y después de la eliminación con el fin de que sea apreciable dicha operación.

3. Modifica el código anterior en un programa3.c para engancharse a la memoria compartida aunque esta exista previamente. Controla todos los errores posibles y realiza una salida limpia del programa liberando todos los recursos utilizados.

La modificación del código ha consistido en:

- Controlar que el segmento de memoria haya sido creado previamente, y si es así, incluir los permisos de lectura y escritura.
- Vincular el proceso a la memoria compartida, controlando la captura de la dirección de la memoria compartida.
- Desvincular el proceso a la memoria compartida y eliminarla.

4. Realiza un programa sencillo (programa4.c) que demuestre esto y explica como lo hace.

Se crea un nuevo espacio de memoria compartida y se vincula el proceso a ella. Éste proceso crea un hijo que lo marca cómo borrado. Se observa en la ejecución que dicha memoria no es borrada por el SO hasta que el padre (único proceso vinculado a ella) se desvincula.

5. Modifícalo en un programa5.c para crear 10 semáforos e inicializarlos con valores desde 1 hasta 10.

Se ha modificado el argumento **nsems** de la llamada a la función **semget** a 10 para que cree un grupo de 10 semáforos.

Una vez hecho ésto se rellena el campo **array** de una variable del tipo union **semun** (arg) con valores entre 1 y 10. Con ésta variable realizamos una llamada a la función **semctl** con los siguientes argumentos:

```
semctl(semid, 0, SETALL, arg)
// El 2° argumento es innecesario con el flag SETALL
```

6. Modifícalo de nuevo en un programa6.c para liberar correctamente los semáforos antes de salir.

Llamando a la función **semctl** con el flag IPC_RMID, se liberan todos los grupos de semáforos indicados por **semid**.

7. Escribe una función de prototipo: int down (int id, int num_sem) que llame a semop para bloquear el semáforo en la posición num_sem de un array de semáforos con identificador id.
Si todo es correcto devolverá 0 y otro valor en caso de error.

Éste es el código de la función:

```
void down(int id, int num_sem){

     struct sembuf operdown = {0, -1, 0};
     operdown.sem_num = num_sem;
     return semop(id, &operdown, 1);
}
```

8. Escribe una función de prototipo: int up (int id, int num_sem) que llame a semop para liberar el semáforo en la posición num_sem del array de semáforos con identificador id. Si todo es correcto devolverá 0 y otro valor en caso de error.

Éste es el código de la función:

```
void up(int id, int num_sem){

     struct sembuf operdown = {0, 1, 0};
     operdown.sem_num = num_sem;
     return semop(id, &operdown, 1);
}
```

9. Completa este código en un programa7.c y no olvides controlar posibles errores y realizar una correcta liberación de recursos.

Se ha completado el programa7.c con las funciones correspondientes a los comentarios indicados junto con sus respectivos controles de errores.

10. ¿Qué problema hay si falla la función down?

Si falla la función **down** podría darse el caso de que dos o más procesos modificarán a la vez la misma dirección de memoria compartida, originando errores en dichos datos.

11. ¿Y si falla la función up?

En el caso de que falle ésta función, dejaría la memoria compartida que regula dicho semáforo inutilizable, ya que ningún proceso podría acceder a ella.

12. Modifica la función de down anterior en un programa8.c para tener esto en cuenta y volver a realizar la espera en caso de haber recibido una señal.

Si **semop** es interrumpida por una señal cualquiera, su retorno vale -1 y a **errno** se le asigna el valor EINTR. La manera de evitar este error es repetir dicha operación en el caso de que se de dicha situación.

13. Si habíamos realizado un down y recibimos una señal que nos haga finalizar el proceso, SIG-KILL por ejemplo, dejaríamos inutilizado el semáforo. Modifica el código del down y del up para evitar esto en un programa9.c.

Éste problema se solucionaría añadiendo el flag **SEM_UNDO** en el campo **sem_flg** de la estructura **sem_buf** que se pasará cómo segundo argumento a **semop**.

Éste flag marca dicha operación sobre el semáforo (guarda un contador de operaciones con el flag **SEM_UNDO**) de manera que si el proceso termina de cualquier forma, deshace todas las operaciones marcadas con este flag.

14. ¿Cómo se puede saber cuántos procesos están esperando a coger un semáforo? Escribe un programa10.c que demuestre tu respuesta.

Añadiendo el flag **GETNCNT** a la llamada a **semctl** que devuelve el número de procesos que están esperando a que se incremente el valor del semáforo.

15. ¿Cómo se puede saber el valor de un semáforo? Escribe un programa11.c que demuestre tu respuesta.

Añadiendo el flag **GETVAL** a la llamada a **semctl** que devuelve el valor del semáforo especificado.

Ejercicio final

Estructura Peticion:

Está constituida por:

- Un entero que representará la id de la petición/respuesta.
- Una cadena de máximo 100 caracteres (100 + '\0'=101) que contendrá el texto a codificar y posteriormente la codificación del mismo.

Monitor:

Este programa reserva e inicializa los recursos utilizados por los demás programas. Crea las dos zonas de memoria de tamaño el valor pasado por línea de comandos y genera los semáforos que se utilizarán para controlar el acceso a las zonas.

Un semáforo se utilizará para controlar el número de peticiones que se introducirán en la memoria (SEM_ZONA1_NPETICIONES, SEM_ZONA2_NPETICIONES).

Un semáforo mutex para evitar que se lea y se escriba a la vez en las zonas de memoria.

Además se ha creado un semáforo auxiliar que guardará el tamaño de la memoria y los programas no podrán incrementarlo ni decrementarlo.

Finalmente captura la señal SIGINT mediante el signal handler “termina” y libera los recursos, terminando la ejecución.

Codificador:

Este programa lee cada petición de la zona 1 de memoria y la procesa con Vigènere mediante un código de cifrado por la línea de comandos. Una vez codificada la petición la borra de la zona 1 y añade la nueva “respuesta” a la zona 2.

Para evitar que el programa acabe cuando se esta realizando el procesamiento de las peticiones se ha diseñado un signal handler que se llame “termina” cuya función es llamar a todos los procesos hijos creados (guardados en una array dinámico global) y mandarles la señal SIGUSR1.

Una vez recibida SIGUSR1, si el hijo está en el fragmento de código del procesamiento desactivará las señales mediante la función sigfillset y si no, se pondrá en funcionamiento el signal handler correspondiente a esa señal, llamado “terminaHijo” que terminará con su ejecución.

Cliente:

Este programa recibe por línea de comandos el fichero de lectura y el fichero de lectura.

Lee una a una las líneas del fichero (máximo 2000 líneas) y genera las peticiones que introducirá en la zona 1.

Finalmente revisa la zona 2 cada 0.5 segundos en busca de las respuestas es decir, las peticiones procesadas, si encuentra una la imprime en el fichero de escritura. Si se han procesado todas las líneas finaliza su ejecución.