Sistemas Operativos y Redes (E0224)
Año 2021

Trabajo Práctico Integrador

Grupo N°4:

Ignacio Hamann - 68410/3
Juan Pablo Elisei - 68380/5
Tomás Tavella - 68371/4

Resumen

En este informe se desarrollan los detalles de diseño e implementación de un juego de naipes españoles "Escoba de 15" en el lenguaje de programación C (y bibliotecas estándar de Linux). Con este fin, se utiliza un servidor concurrente a base de *sockets* de red que acepta hasta 4 conexiones (jugadores) simultáneos conectados bajo el protocolo TCP.

Facultad de Ingeniería
Universidad Nacional de La Plata

Índice

1. Enunciado

Queremos un programa ***servidor concurrente*** que permita a un cliente que se conecta utilizando el protocolo TCP enviar un archivo para que el servidor lo almacene.

- Cuando un cliente se conecta, el servidor enviará el mensaje: "listo" y esperará recibir la palabra "archivo".
- A continuación el servidor espera un espacio y luego el nombre con que se quiere guardar el archivo (solo letras, números y el carácter ".") finalizado con un espacio.
- Luego se quedará esperando un número codificado en `ascii` que indica el tamaño en bytes del archivo, también finalizado con un espacio.
- A continuación comenzará la recepción de los datos, que serán almacenados en un archivo cuyo nombre es el recibido en primer término, hasta completar la cantidad de bytes correspondiente.
- El servidor debe permitir la recepción de archivos binarios.
- Una vez recibida la totalidad de los datos, el servidor contestará con el siguiente mensaje:
  - "Archivo xx completo, tamaño declarado yy bytes, tamaño real zz bytes."
    - xx: nombre del archivo.
    - yy: tamaño enviado en el mensaje del cliente.
    - zz: total de bytes recibidos por el servidor.
- Luego el servidor cerrará la conexión con el cliente.
- El servidor debe llevar un archivo de registro de todas las conexiones entrantes, con fecha y hora de inicio y de finalización, tamaño del archivo recibido, cantidad de bytes enviados y cantidad de bytes recibidos en formato `csv`.
- Defina explícitamente cómo será el manejo de errores.
- Para probar el programa puede utilizar el cliente TCP que se vio como ejemplo, o el programa `nc`.
- Recuerde verificar que no queden procesos *zombie* cuando finalizan los hijos generados.

2. Interpretación del problema

Para este programa, se debe crear dos espacios de memoria compartida utilizando las funciones de la biblioteca `<sys/shm.h>`, en los que existirán las dos mitades de un *buffer ping-pong*.

Los datos leídos del archivo `datos.dat` se almacenan en una estructura con los siguientes elementos:

- Una variable de tipo `int` que almacena un identificador menor a 50000.
- Una etiqueta con el tiempo en el que fue escrito el dato, con precisión de micro segundos.
- El dato que se leyó del archivo, de tipo `float`.

Como no se aclara el tamaño del *buffer* en el enunciado, se elige que este tenga un tamaño total de 100 estructuras de las mencionadas previamente, dividido en dos mitades de 50 estructuras.

El proceso productor debe leer datos del archivo `datos.dat` y almacenar las estructuras en su mitad del *buffer*, mientras que el proceso consumidor debe leerlas en la otra mitad del *buffer* a medida que estén disponibles, para imprimirlas en pantalla y escribirlas en un nuevo archivo `datos.csv`.

Para administrar el *buffer* de manera que no surjan condiciones de carrera y tanto el productor como el consumidor puedan trabajar en secciones críticas sin interrupciones, se van a crear dos programas que utilizan métodos distintos para este fin:

- **Semáforos:** se utilizan los semáforos para sincronizar con las llamadas a sistema contenidas en la biblioteca `<sys/sem.h>` y memoria compartida mediante `<sys/ipc.h>`.
- **Colas de mensajes:** se utilizan las colas de mensajes para sincronizar mediante las llamadas a sistema de la biblioteca `<sys/msg.h>`.

3. Resolución

Se utilizaron las siguientes bibliotecas de C para poder llevar a cabo la resolución del problema planteado:

- Para ambas implementaciones:
  - `<sys/ipc.h>`: biblioteca de *System V* para la comunicación entre procesos.
  - `<sys/time.h>`: biblioteca para obtener el tiempo de la *timestamp*.
  - `<sys/shm.h>`: biblioteca de *System V* para la memoria compartida.
- Para la resolución con semáforos:
  - `<sys/sem.h>`: implementación de *System V* para semáforos.
- Para la resolución por colas de mensajes:
  - `<sys/msg.h>`: implementación de *System V* para colas de mensajes.

En ambos casos se eligió reservar dos segmentos de memoria compartida, uno para cada mitad del *buffer*, con capacidad para almacenar 100 estructuras de datos en total. Esto se logró mediante la función `shmget()` (contenida en la biblioteca `<sys/shm.h>`), a la cual se le pasan como argumentos el tamaño del segmento y una clave única (obtenida con la función `ftok()`) que debe ser la misma para el productor y el consumidor, para así poder compartir los segmentos creados.

Además, dado que los requerimientos no establecen que se debe hacer en el caso que el productor quiera escribir un *buffer* que aun no fue consumido, se decidió priorizar la integridad de los datos a detrimento de la velocidad de respuesta, por lo que el productor espera a que el consumidor finalize de leer los datos del *buffer*.

3.1. Realización con semáforos

Para la implementación con semáforos, se utilizaron las funciones `semget()`, `semctl()`, `semop()` incluidas en la biblioteca apropiada para implementar 3 semáforos distintos:

- **Semáforo del *buffer* 1**: Este semáforo es bloqueado por alguno de los dos procesos al comenzar operaciones sobre el primer *buffer*, y desbloqueado al terminarlas.
- **Semáforo del *buffer* 2**: De manera similar al semaforo anterior, es bloqueado por alguno de los dos procesos al comenzar operaciones sobre el segundo buffer, y desbloqueado al terminarlas.
- **Semáforo de sincronización**: Se encarga de sincronizar los dos programas, de manera que el productor no sobrescriba datos aún no consumidos.

Primeramente, se crean los semáforos y espacios de memoria compartida (con `shmget()` y `semget()`), utilizando claves únicas obtenidas con la función `ftok()`. En este caso, tanto la memoria compartida como los semáforos se crean en el programa productor, por lo que si se llama al consumidor previo a este, se va a arrojar en pantalla un error indicando que no se pudieron obtener los recursos.

Una vez creados los semáforos, se inicializan dentro del productor mediante `semctl()`, y una vez que comienza la lectura y escritura de datos, se opera sobre los mismos mediante la función `semop` (para incrementar la legibilidad del código, las tres instrucciones requeridas para operar un

semáforo se sintetizaron en un macro definido en el *header* del programa).

Finalmente, al terminar de correr el productor, se llaman a `shmctl()` y `semctl()`, para indicar al sistema que destruya los semáforos y las memorias compartidas que se crearon una vez que el ultimo programa haya dejado de utilizarlos (en este caso el consumidor).

3.1.1. Pseudocódigo

```
Productor:
INICIO
    Declarar y asignar variables, macros y estructuras;
    Obtener la clave de las dos memorias compartidas y el semaforo (en el caso de no obtenerlas imprimir error);
    Llamar al sistema para obtener el ID de las memorias compartidas (en el caso de que no obtenerlas imprimir error);
    Asociar el espacio de memoria compartida con un puntero (si no puede asociar imprimir error);
    Crear semaforos (si no puede crear imprimir error);
    Inicializar semaforos;
    Verificar la existencia del archivo datos.dat;
    Obtener el tiempo de UNIX inicial;
    Inicializar variables auxiliares para los buffers;
    Mientras(1){
        Bloquear el semaforo del buffer1;
        Leer una linea del archivo .dat;
        Mientras(No se llegue al fin del buffer1){
            Copiar los datos en el buffer1;
            Leer una linea del archivo .dat;
            Si(Se llega al final del archivo){
                Salir del bucle y avisar que el ultimo elemento se escribio en el buffer1;
            }
            Incrementar la variable para recorrer el buffer y la del id;
        }
        Resetear la variable que recorre el buffer;
        Desbloquear el semaforo del buffer1;
        Bloquear el semaforo del buffer2;
        Bloquear el semaforo de sincronizacion;
        Leer una linea del archivo.dat;
        Mientras(No se llegue al fin del buffer2){
            Copiar los datos en el buffer2;
            Leer una linea del archivo .dat;
            Si(Se llega al final del archivo){
                Se sale del bucle;
            }
            Incrementar la variable para recorrer el buffer y la del ID;
        }
        Si(Se llega al final del archivo){
            Se sale del bucle y avisar que el ultimo elemento se escribio en el buffer2;
        }
        Resetear la variable que recorre el buffer;
        Desbloquear el semaforo del buffer2;
        Bloquear el semaforo de sincronizacion;
    }
    Poner un ID Null despues de llegar al ultimo elemento del buffer correspondiente para avisarle al consumidor que se llego al EOF;
    Cerrar el archivo .dat;
    Liberar la memoria compartida y se remover los semaforos;
FIN

Consumidor:
INICIO
    Declarar y asignar variables, macros y estructuras;
    Obtener la clave de las dos memorias compartidas y el semaforo (en el caso de no obtenerlas imprimir error);
    Llamar al sistema para obtener el ID de las memorias compartidas (en el caso de que no obtenerlas imprimir error);
    Asociar el espacio de memoria compartida con un puntero (si no puede asociar imprimir error);
    Crear semaforos (si no puede crear imprimir error);
    Inicializar de semaforos;
    Verificar la existencia del archivo datos.dat;
    Inicializar variables auxiliares para los buffers;
    Mientras(1){
        Bloquear el semaforo del Buffer1;
        Mientras(No se llega al fin del buffer1 y el ID del mismo sea distinto a -1){
            Copiar los datos del buffer1 en el archivo csv e imprimirlos en pantalla;
            Incrementar la variable para recorrer el buffer;
            Si(Se encontro el ID -1){
                Salir del bucle;
            }
        }
        Resetear la variable que recorre el buffer;
        Si(Se encontro el ID -1){
            Salir del bucle principal;
        }
        Desbloquear el semaforo del buffer1;
```

```
            Desbloquear el semaforo de sincronizacion;
            Bloquear el semaforo del buffer2;
            Mientras(No se llegue al fin del buffer2 y el ID del mismo sea distinto a -1){
                Copiar los datos del buffer2 en el archivo csv e imprimirlos en pantalla;
                Incrementar la variable para recorrer el buffer;
                Si(Se encontro el ID -1){
                    Salir del bucle;
                }
            }
            Resetear la variable que recorre el buffer;
            Si(Se encontro el ID -1){
                Salir del bucle principal;
            }
            Desbloquear el semaforo del buffer2;
            Desbloquear el semaforo de sincronizacion;
            Si(Se encontro el ID -1){
                Salir del bucle principal;
            }
        }
        Cerrar el archivo .csv;
        Liberar la memoria compartida y remover los semaforos;
    FIN
```

3.2. Realización con colas de mensajes

La implementación por cola de mensajes emplea dos colas con dos tipos de mensajes. Se utiliza una *queue* por buffer, y los tipos de mensaje indican si el buffer está listo para ser escrito, o para ser leído. El productor siempre envía mensajes de un tipo, y el consumidor envía mensajes del otro. Para la sincronización, ambos programas esperan primero a recibir el mensaje del tipo que corresponda antes de operar con la memoria compartida, y una vez que terminan envían el mensaje recíproco. Al iniciar, el consumidor es el primero en enviar mensajes de que los buffers están disponibles, con lo cual ningún programa empieza por separado, sino que el trabajo sobre la memoria compartida se realiza solo con los dos programas en ejecución.

3.2.1. Pseudocódigo

```
1   Productor:
2   INICIO
3       Declarar y asignar variables, macros y estructuras;
4       Obtener la clave de las dos memorias compartidas y colas de mensaje (en el caso de que
    no las obtenga imprime error);
5       Llamar al sistema para obtener el ID de las memorias compartidas (en el caso de que no
    las obtenga imprime error);
6       Asociar el espacio de memoria compartida con un puntero(si no puede asociar imprime
    error);
7       Crear colas de mensajes (si no los puede crear imprime error);
8       Verificar de la existencia del archivo datos.dat;
9       Obtener el tiempo de UNIX inicial;
10      Inicializar variables auxiliares para los buffers;
11      Mientras(no es el fin del archivo){
12          Esperar mensaje que se puede escribir el buffer 1;
13          Llenar buffer con datos;
14          Preparar mensaje de "listo para leer";
15          Enviar mensaje en cola 1;
16
17          Esperar mensaje que se puede escribir el buffer 2;
18          Llenar buffer con datos;
19          Preparar mensaje de "listo para leer";
20          Enviar mensaje en cola 2;
21      }
22      Cerrar archivo;
23      Liberar memoria compartida;
24  FIN
25
26  Consumidor:
27  INICIO
28      Declarar y asignar variables, macros y estructuras;
29      Obtener la clave de las dos memorias compartidas y colas de mensaje (en el caso de que
    no las obtenga imprime error);
30      Llamar al sistema para obtener el ID de las memorias compartidas (en el caso de que no
    las obtenga imprime error);
31      Asociar el espacio de memoria compartida con un puntero(si no puede asociar imprime
    error);
32      Crear colas de mensajes (si no los puede crear imprime error);
33      Enviar mensaje de "ambos buffers listos para escribir";
34      Abrir archivo de destino en modo escritura;
35      Mientras(1){
36          Esperar mensaje que se puede leer el buffer 1;
37          Si el mensaje indicaba fin del archivo, salir;
38          Leer buffer, imprimir en pantalla y en archivo de destino;
39          Preparar mensaje de "listo para escribir";
40          Enviar mensaje en cola 1;
```

```
41
42              Esperar mensaje que se puede leer el buffer 2;
43              Si el mensaje indicaba fin del archivo, salir;
44              Leer buffer, imprimir en pantalla y en archivo de destino;
45              Preparar mensaje de "listo para escribir";
46              Enviar mensaje en cola 2;
47          }
48          Cerrar archivo;
49          Liberar memoria compartida;
50          Liberar colas de mensajes;
51      FIN
```

3.3. Compilación y ejecución de los programas

Con el código completo, para compilar los programas a un archivo binario ejecutable se llama al comando `gcc` (*GNU C Compiler*):

```
 $ gcc productor_sem.c -o prod_sem
$ gcc consumidor_sem.c -o cons_sem
 $ gcc productor_col.c -o prod_col
$ gcc consumidor_col.c -o cons_col
```

Con lo que se obtienen cuatro archivos binarios ejecutables, los cuales, estando situados en la carpeta en la que se encuentran, se ejecutan desde la terminal de la siguiente manera:

```
$ ./prod_sem
$ ./cons_sem
$ ./prod_col
$ ./cons_col
```

Es importante ejecutar los productores antes que los consumidores, para cualquiera de las dos implementaciones, por la forma en que se maneja la sincronización.

4. Conclusiones

La implementación del *buffer ping-pong* con distintos métodos de sincronización para la memoria compartida demostró que si bien ambos métodos son efectivos para evitar la pérdida de datos o las condiciones de carrera, las distintas opciones tienen sus ventajas y desventajas. Las colas de mensaje permiten compartir mucha mas información entre procesos para generar distintos comportamientos deseados, aunque tienen la desventaja que al eliminarse el mensaje de la cola cuando se lee, comunicar más de dos procesos se vuelve más complicado. Los semáforos tienen un funcionamiento más sencillo, que permite sincronizar más procesos con múltiples juegos de recursos compartidos, aunque su implementación no resulta tan simple en la práctica.