Sistemas Operativos y Redes (E0224)
Año 2021

Trabajo Práctico N°3

Grupo N°4:

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

Resumen

Para este trabajo se desarrolla la implementación en C de un *buffer ping-pong* utilizando memoria compartida, semáforos y colas de mensajes; como solución de un problema de productor-consumidor.

Facultad de Ingeniería
Universidad Nacional de La Plata

Índice

1. Enunciado

Se desea implementar un *buffer ping-pong* para que un proceso genere datos y otro los consuma. La implementación debe utilizar memoria compartida para los datos producidos y leídos.

El proceso productor debe leer datos desde el archivo `datos.dat` y almacenarlos en un *buffer ping-pong* de estructuras, cuyos campos son:

- Un identificador entero menor que 50000.
- Una etiqueta de tiempo con precisión de microsegundos.
- El dato leído desde el archivo `datos.dat`.

El proceso consumidor debe leer desde el *buffer ping-pong* e imprimir los valores leídos en pantalla a medida que estén disponibles. También debe almacenarlos en un archivo llamado `datos.csv`, a razón de una estructura por línea, con los valores separados por comas (CSV).

Deben implementarse dos variantes del problema, una que administre el *buffer ping-pong* usando semáforos y variables compartidas, y otro que utilice colas de mensajes. En ambos casos decida y justifique que se va a hacer cuando el productor llene ambos *buffers* y el consumidor no consumió los datos.

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 textitbuffer.

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

```
INICIO
    Declaracion y asignacion de variables, macros y estructuras;
    Se obtiene la clave de las dos memorias compartidas y el semaforo (en el caso de que no las obtenga imprime error);
    Se llama al sistema para obtener el ID de las memorias compartidas (en el caso de que no las obtenga imprime error);
    Se asocia el espacio de memoria compartida con un puntero(si no puede asociar imprime error);
    Creacion de semaforos (si no los puede crear imprime error);
    Inicializacion de semaforos;
    Verificacion de la existencia del archivo datos.dat;
    Se obtiene el tiempo de UNIX inicial;
    Inicializo variables auxiliares para los buffers;
    Mientras(1){
        Se bloquea el semaforo del Buffer1;
        Se lee una linea del archivo .dat;
        Mientras(No se llegue al fin del buffer1){
            Se copian los datos en el buffer1;
            Se lee una linea del archivo .dat;
            Si(Se llego al final del archivo){
                Se sale del bucle;
            }
            Sino{
                Sigue en el bucle;
            }
            Se incrementan la variable para recorrer el buffer y la del id;
        }
        Si(Se llego al final del archivo){
            Se sale del bucle;
        }
        Se resetea la variable que recorre el buffer;
        Se desbloquea el semaforo del Buffer1;
        Se bloquea el semaforo del Buffer2;
        Se bloquea el semaforo de sincronizacion;
        Se lee una linea del archivo.dat;
        Mientras(No se llegue al fin del buffer2){
            Se copian los datos en el buffer2;
            Se lee una linea del archivo .dat;
            Si(Se llego al final del archivo){
                Se sale del bucle;
            }
            Sino{
                Sigue en el bucle;
            }
            Se incrementan la variable para recorrer el buffer y la del ID;
        }
        Si(Se llego al final del archivo){
            Se sale del bucle;
        }
        Se resetea la variable que recorre el buffer;
        Se desbloquea el semaforo del Buffer2;
        Se bloquea el semaforo de sincronizacion;
    }
    Se pone un ID Null despues de llegar al ultimo elemento para avisarle al consumidor que se llego al EOF;
    Se cierra el archivo .dat;
    Se libera la memoria compartida y se remueven los semaforos;
FIN

Consumidor:
INICIO
    Declaracion y asignacion de variables, macros y estructuras;
    Se obtiene la clave de las dos memorias compartidas y el semaforo (en el caso de que no las obtenga imprime error);
    Se llama al sistema para obtener el ID de las memorias compartidas (en el caso de que no las obtenga imprime error);
    Se asocia el espacio de memoria compartida con un puntero(si no puede asociar imprime error);
    Creacion de semaforos (si no los puede crear imprime error);
    Inicializacion de semaforos;
    Verificacion de la existencia del archivo datos.dat;
    Inicializo variables auxiliares para los buffers;
    Mientras(1){
        Se bloquea el semaforo del Buffer1;
        Mientras(No se llegue al fin del buffer1 y el ID del mismo no sea -1){
            Se copian los datos del buffer1 en el archivo csv y se imprimen en pantalla;
            Se incrementan la variable para recorrer el buffer;
            Si(Se encontro el ID -1){
                Se sale del bucle;
            }
        }
        Se resetea la variable que recorre el buffer;
        Si(Se encontro el ID -1){
            Se sale del bucle principal;
        }
        Se desbloquea el semaforo del Buffer1;
        Se desbloquea el semaforo de sincronizacion;
        Se bloquea el semaforo del Buffer2;
```

```
            Mientras(No se llegue al fin del buffer2 y el ID del mismo no sea -1){
                Se copian los datos del buffer2 en el archivo csv y se imprimen en pantalla;
                Se incrementan la variable para recorrer el buffer;
                Si(Se encontro el ID -1){
                    Se sale del bucle;
                }
            }
            Se resetea la variable que recorre el buffer;
            Si(Se encontro el ID -1){
                Se sale del bucle principal;
            }
            Se desbloquea el semaforo del Buffer2;
            Se desbloquea el semaforo de sincronizacion;
            Si(Se encontro el ID -1){
                Se sale del bucle principal;
            }
        }
        Se cierra el archivo .csv;
        Se libera la memoria compartida y se remueven 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, el 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

```
    INICIO
        Declarar y asignar variables, macros y estructuras;
        Obtener la clave de las dos memorias compartidas y colas de mensaje (en el caso de que
    no las obtenga imprime error);
        Llamar al sistema para obtener el ID de las memorias compartidas (en el caso de que no
    las obtenga imprime error);
        Asociar el espacio de memoria compartida con un puntero(si no puede asociar imprime
    error);
        Crear colas de mensajes (si no los puede crear imprime error);
        Verificar de la existencia del archivo datos.dat;
        Obtener el tiempo de UNIX inicial;
        Inicializar variables auxiliares para los buffers;
        Mientras(no es el fin del archivo){
            Esperar mensaje que se puede escribir el buffer 1;
            Llenar buffer con datos;
            Preparar mensaje de "listo para leer";
            Enviar mensaje en cola 1;

            Esperar mensaje que se puede escribir el buffer 2;
            Llenar buffer con datos;
            Preparar mensaje de "listo para leer";
            Enviar mensaje en cola 2;
        }
        Cerrar archivo;
        Liberar memoria compartida;
    FIN

    Consumidor:
    INICIO
        Declarar y asignar variables, macros y estructuras;
        Obtener la clave de las dos memorias compartidas y colas de mensaje (en el caso de que
    no las obtenga imprime error);
        Llamar al sistema para obtener el ID de las memorias compartidas (en el caso de que no
    las obtenga imprime error);
        Asociar el espacio de memoria compartida con un puntero(si no puede asociar imprime
    error);
        Crear colas de mensajes (si no los puede crear imprime error);
        Enviar mensaje de "ambos buffers listos para escribir";
        Abrir archivo de destino en modo escritura;
        Mientras(1){
            Esperar mensaje que se puede leer el buffer 1;
            Si el mensaje indicaba fin del archivo, salir;
            Leer buffer, imprimir en pantalla y en archivo de destino;
            Preparar mensaje de "listo para escribir";
            Enviar mensaje en cola 1;

            Esperar mensaje que se puede leer el buffer 2;
            Si el mensaje indicaba fin del archivo, salir;
```

```
147             Leer buffer, imprimir en pantalla y en archivo de destino;
148             Preparar mensaje de "listo para escribir";
149             Enviar mensaje en cola 2;
150         }
151         Cerrar archivo;
152         Liberar memoria compartida;
153         Liberar colas de mensajes;
154     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

Luego de realizar el trabajo se pudo rescatar que los semaforos tiene una relacion con la rama de comunicaciones ya que no son una opción.

Referencias

[1] *Thread Arguments and Return Values*. URL: `https://w3.cs.jmu.edu/kirkpams/OpenCSF/Books/csf/html/ThreadArgs.html`.