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

En el `main()` del programa, se utiliza la función `malloc()` (definida en las bibliotecas estándar) para reservar el espacio de memoria para el vector de N elementos.

Para el manejo de señales, se utiliza la función `signal()` (parte de `<signal.h>`), que se encarga de instalar *signal handlers* para las distintas señales que el programa va a recibir. Esta función toma dos parámetros: el número o nombre de la señal a manejar; y la función que se va a encargar del manejo de la señal.

Para el manejo de procesos padres e hijos, fueron de utilidad las siguientes funciones (parte de las bibliotecas `<unistd.h>` y `<sys/wait.h>`):

- `fork()`: Genera una copia de la imagen del proceso padre y se la asigna al proceso hijo, al cual se le asignan nuevos identificadores. Para distinguir al padre del hijo, esta función devuelve 0 al ser llamada en el hijo y el PID del hijo al ser llamada en el padre.
- `getpid()`: Devuelve el PID del proceso que hace la llamada al sistema.
- `getppid()`: Devuelve el PID del proceso padre del proceso que hace la llamada al sistema.
- `waitpid()`: Suspende la ejecución del proceso que llama a la función hasta que el hijo con el PID especificado termine.
- `wait()`: Suspende la ejecución del proceso que llama a la función hasta que algún proceso hijo termine.

En tanto al manejo de los *threads*, se utilizaron las siguientes funciones:

- `pthread_create()`: Crea un nuevo *thread*, le asigna la función que este va a ejecutar y le pasa los argumentos.
- `pthread_exit()`: Finaliza el *thread* en el que se llama, y devuelve un valor que se le especifica a la función que creó el *thread*.
- `pthread_join()`: Detiene el thread en ejecución hasta que otro hilo termina, y recupera el valor que este devolvió.

Para compilar el código a un archivo binario ejecutable, se debe utilizar el comando `gcc` (*GNU C Compiler*) en la terminal, agregando el argumento `-pthread` para permitir la utilización de *threads*:

```
$ gcc entregable2.c -o vector_procesos_hilos -pthread
```

Donde `vector_procesos_hilos` es el archivo compilado ejecutable. Para correrlo, se debe llamar al archivo seguido del número de elementos que se quiera el vector:

```
$ ./vector_procesos_hilos <N>
```

Si no se pasa la cantidad de argumentos requerida, el programa sugiere un ejemplo con la sintaxis correcta.

Para el envío de señales al programa, se debe utilizar el comando `kill`. Una forma más amigable con el usuario para no tener que recordar el número de la señal ni el PID del proceso es utilizar los siguientes argumentos:

```
$ kill -s <nombre_señal> $(pidof vector_procesos_hilos)
```

Donde `nombre_señal` se corresponde con `SIGUSR1`, `SIGUSR2` o `SIGTERM`, y el comando `pidof` devuelve el PID del programa en ejecución sin necesidad de buscarlo manualmente.

3.1. Pseudocódigo

A continuación se muestra un pseudocódigo correspondiente con el código en C.

```
INICIO
    Declaracion y asignacin 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){

    }
```

4. Conclusiones

a 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`.