Programación de Sistemas (PrSist)

Departamento de Informática de Sistemas y Computadoras (DISCA)
Universitat Politècnica de València

Práctica 4
Mutex y semáforos

Contenido

1 Objetivos

Los objetivos que se plantean en la practica 4 se centran en el planteamiento de una solución global al problema del acceso concurrente de threads a una variable compartida tipo buffer circular, así como el manejo de las condiciones de buffer lleno o vacío dentro de las funciones get_item y put_item.

2 Introducción

Durante las sesiones de practicas anteriores se ha evolucionado en el manejo de una estructura de datos desde el código C de la siguiente forma

- Primero se ha definido una estructura de datos que respondía a una política FIFO para el almacenamiento en la misma, y que era usada por un único proceso
- Posteriormente se ha declarado una instancia accesible varios threads, haciendo que todos ellos puedan insertar o extraer un dato.
- Partiendo del caso del punto anterior, puede comprobarse como dicho acceso concurrente a la misma variable puede ser susceptible de provocar condiciones de carrera.

Cuestión 1:

Determina un escenario de uso (tamaño de buffer, número de iteraciones para insertar o sacar datos, así como número de threads productores y consumidores) para el cual se compruebe la ocurrencia de una condiciona de carrera. Especifica porque motivo es una condición de carrera.

3 Solución con mutex a la condición de carrera

Como ya se ha visto en las sesiones de teoría, un mutex es un recurso que proporciona la librería POSIX para la sincronización, por tanto nos permiten gestionar el acceso exclusivo a sección critica.
Como progreso hacia el desarrollo del trabajo realizaremos los siguientes desarrollos del código:

- Separar la implementación del buffer circular del código general de la aplicación. Para ello Desarrollar un fichero buffer_circ.h con todas las declaraciones necesarias (variables, constantes, tipos y prototipos de funciones) y un fichero buffer_circ.c con la implementación de las funciones.
- Comprobar con dicha organización del código que se sigue dando la condición de carrera, desarrollada en la introducción cuando se lanzan varios threads productores y consumidores. Todos ellos deben acceder a la variable compartida por medi del puntero de parámetros iniciales arg.
- Modificar el código de buffer_circ.h y buffer_circ.c para que el acceso a la sección critica de las funciones se realice en modo exclusivo mediante el uso de un mutex de nombre buffer_lock.
- En esta implementación el comportamiento de las funciones get_item y put_item cuando no sea posible completarlas porque no exista un dato, o tengamos espacio será la misma que en ejercicios anteriores: devolver un código de error.
- Comprobar que ya no se da la condición de carrera con el mismo caso de ejecución.

4 Solución con semáforos para la gestión de espacio en el buffer

En la sección anterior hemos visto como resolver el problema del acceso a la sección crítica por parte de threads que comparten una variable común. Aun así, el comportamiento del buffer_circ es correcto desde el punto de vista formal de acceso concurrente, pero plantea un reto desde el punto de vista de gestión de los datos ¿qué hacemos cuando un thread intenta insertar/sacar un datos y la función devuelve un código de error?.

Cuestión 2:

Imaginemos una situación real: supongamos que el buffer_circ es una estructura de datos que comparten un thread (T_read) que inserta los paquetes provenientes de la red, con otro thread (T_aplic) que distribuye los paquetes de red entre las aplicaciones en marcha. Razona las siguientes cuestiones:
¿que hace el T_red cuando no puede insertar un dato? ¿desestimarlo? ¿intentarlo otra vez? Y si ese el caso ¿cuándo?
¿que hace el T_aplic cuando quiere obtener un dato nuevo del buffer para una aplicación que se lo ha demandado y no hay ninguno? ¿retorna un código de error? ¿intentarlo otra vez? Y si ese el caso ¿cuándo?

Parece claro que este tipo de operaciones deben proporcionar un mecanismo síncrono de uso del mismo: es decir si no es posible completar la operación (no hay espacio, o no hay datos) se debe esperar a que se den las condiciones adecuadas para hacerlo. El concepto "se debe esperar" se entiende como que el thread invocante de la operación (get_item o put_item) debe quedar suspendido hasta que pueda progresar en la misma (ya hay espacio o hay dato) de forma la finaliza. En este punto es donde los semáforos nos ofrecen el mecanismo adecuado para controlar dicha lógica.

Cuestión 3
Identificar en las operaciones get_item y put_item las condiciones (en forma lógica) que deben darse par poder realizar cada una de las mismas. Realizar de acuerdo a las mismas las declaraciones de semáforos que las sustenten.

De acuerdo con la lógica descrita en la cuestión anterior, modificar la implememntación de buffer_circ (tanto en sus definiciones contenidas en el fichero .h como en la implementación en el fichero .c) de forma que controle que:

- El acceso a la variable de tipo buffer_circ se realiza de modo exclusivo en las secciones criticas por parte de los threads (esto debía estar ya implementado por el apartado 3)
- Cuando un thread productor invoca una operación put_item, esta no retorna el control al código del thread, mientras no se ha completado la misma
- Cuando un thread consumidor invoca una operación get_item, esta no retorna el control al código del thread, mientras no se ha completado la misma

5 Ejercicio opcional 1

Incluir entre los threads en ejecución un hilo monitor, el cual reciba como parámetro el puntero a una estructura buffer_circ, y muestre cada segundo su contenido.

6 Ejercicio opcional 2

En el apartado anterior se ha resuelto el problema de la gestión de las condiciones de ejecución de las operaciones get_item y put_item mediante el uso de semáforos. Una forma alternativa de resolverlo sería mediante el uso de variables condición. Se propone como ejercicio opcional realizar la misma implementación de la practica, sustituyendo los semáforos que rigen la lógica de inserción/extracción de datos por una variable condición.