2DO:
TRIMESTRE

4. Sincronización entre procesos

Hay situaciones en las que varios procesos tienen que comunicarse, cooperar o utilizar el mismo recurso, esto implica que tiene que haber cierto sincronismo entre los procesos, si este sincronismo no existiese algunos procesos tendrían que esperar a que otros finalicen algunas acciones.

El SO se encarga de este sincronismo, los lenguajes de programación de alto nivel se encargan de encapsular los mecanismos de sincronismo que proporciona el sistema operativo.

El problema: que sea inconsistente la actualización de un recurso compartido por varios procesos.

En programación concurrente, siempre que accedamos a un recurso compartido, vamos a tener en cuenta las condiciones de nuestro proceso respecto al recurso ¿será de forma exclusiva o no?

En el case de lectura y escritura de fichero, debemos determinar cómo queremos acceder al fichero (como lectura, como escritura o como lectura escritura) y utilizaremos los objetos que nos permitan establecer los mecanismos de sincronización dejando en algunos casos ciertos procesos bloqueados.

Se conoce como el problema de los procesos lectores-escritores. El sistema operativo nos ayudará a resolver los problemas que se plantean.

- Si el **acceso es de solo lectura**, permite que todos los procesos lectores (solo quieren leer información del fichero) puedan acceder simultáneamente a él.
- En el caso de **escritura o lectura-escritura**, el sistema operativo nos permite solicitar un tipo de acceso de forma exclusiva la cual significará que el resto de procesos tendrán que esperar.

La comunicación con el sistema operativo se produce por objetos y métodos proporcionados por el lenguaje de programación.

Hay que tener cuidado con la documentación de las clases que usamos.

Regiones críticas

Conjunto de instrucciones en las que el proceso accede a un recurso compartido. Estas instrucciones que forman la región crítica se ejecutan de forma indivisible y de forma exclusiva respecto a otros procesos que quieren acceder al mismo recurso.

Para identificar y definir las regiones críticas, hay que tener en cuenta:

- Se protegerán con secciones críticas sólo aquellas instrucciones que accedan a un recurso compartido.
- Las instrucciones que forman una sección crítica serán las mínimas posibles incluyendo sólo las imprescindibles.
- Se pueden definir tantas secciones críticas como sean necesarias.
- Cuando un proceso entra en su sección crítica, queda bloqueado para el resto de procesos, que esperan a que este salga de su sección crítica. El proceso que está en su sección crítica es el que bloquea el recurso.
- Al final de cada sección crítica el recurso debe ser liberado, para que puedan utilizarlo otros recursos.

Cada vez que actualicemos los datos de un recurso compartido, se necesitará establecer una región crítica. Esto implicará:

- **Leer el dato que se quiera actualizar**. Pasar el dato a la zona local de memoria del proceso.
- **Realizar el cálculo de actualización**. Modificar el dato en memoria.
- **Escribir el dato actualizado**. Llevar el dato modificado al recurso compartido.

Categoría de proceso cliente-servidor

Clasificación de los procesos dentro de la categoría cliente-suministrador (cliente-servidor):

- **Cliente**: Proceso que solicita, pide o requiere información o servicios a otros procesos que se le proporcionan.
- **Suministrador (Servidor)**: Proceso que da o suministra información o servicios ya sea por memoria compartida, un fichero o una red.
- **Información o servicio**: Es perecedero. La información desaparece cuando es consumida por el cliente. El servicio se presta en el momento en el que el cliente y el suministrador estén sincronizados.

El sincronismo entre cliente y suministrador se establece mediante un intercambio de mensajes o a través de un recurso compartido.

Entre un cliente y un servidor la comunicación se establece por un intercambio de mensajes con sus correspondientes reglas de uso, lo que se conoce como protocolo. Podemos usar protocolos ya establecidos o nuestros propios protocolos.

Los procesos cliente y suministrador se puede extender a los casos en los que tengamos un proceso que lee y otro que escribe en un recurso compartido.

Entre procesos cliente y suministrador tenemos que disponer de mecanismos de sincronización que nos permitan:

Un cliente no debe poder leer un dato hasta que no haya sido completamente suministrado. De esta fama nos aseguraremos de que el dato leído es correcto y consistente.

Un suministrador irá produciendo información que en cada instante no podrá superar en volumen de tamaño el máximo establecido, si hemos alcanzado este máximo el suministrador no debe poder escribir un dato. De esta forma no se desbordará el cliente.

Vamos a pensar en el caso más sencillo, en el que el suministrador sólo produce un dato y el cliente lo consume ¿Qué sincronismo hace falta para esta situación?

1) El cliente tiene que esperar a que el suministrador haya generado todo el dato
2) El suministrador genera este dato y de alguna forma avisa al cliente de que éste ya está listo

Podríamos pensar en solucionar esta situación con programación secuencial. Incluyendo un bucle en el que se pruebe el calor de una variable que nos indique si el dato ha sido producido.

Semáforos (Semaphores)

En programación concurrente se usan los semáforos para bloquear los procesos cuando no pueden acceder al recurso. El semáforo se encarga de ir desbloqueándolos cuando se pueda pasar.

El semáforo es un componente de bajo nivel de abstracción que permite arbitrar a un recurso compartido (en programación concurrente).

El semáforo se va a ver como un tipo de dato que podemos instanciar. Este objeto semáforo va a poder tomar un conjunto de determinar dos valores y se podrá realizar con él, un conjunto determinado de operaciones.

Un semáforo tendrá asociada una lista de procesos suspendidos que estarán esperando para entrar en el mismo.

Clasificación según el conjunto de datos

- **Semáforos binarios**: Sólo podrán tomar los valores 0 y 1 (podemos asociarlo a las luces verde y roja).
- **Semáforos generales**: Pueden tomar cualquier valor natural, incluido cero.

¿Qué representan los valores que toma un semáforo?

- Valor sea igual a cero: El semáforo está cerrado
- Valor sea mayor que cero: El semáforo está abierto

Cualquier semáforo permite dos operaciones seguras:

1) objSemaforo.wait() – Si el semáforo no es nulo, decrementa en uno el valor de un semáforo. Si el semáforo es nulo, el proceso que lo ejecuta se suspende y lo manda al final de la cola.
2) objSemaforo.signal() – Si hay algún proceso en la lista del semáforo, activa uno de ellos para que se ejecute la sentencia que siga al wait. Si no hay ningún proceso, incrementamos en uno el valor del semáforo.

También se puede realizar una operación no segura: la inicialización del valor del semáforo, este valor nos indica cuantos procesos pueden entrar concurrentemente a él. Esta inicialización se hace al crear el semáforo.

Pasos para la utilización de semáforos:

1. Un proceso padre creará o inicializará el semáforo.
2. Este proceso padre creará el resto de procesos y los pasará al semáforo que ha creado. Estos procesos hijos accederán todos al recurso compartido.
3. Cada uno de los procesos hijos hará uso de las operaciones wait y signal siguiendo el esquema:
    - objSemaforo.wait() para consultar si puede acceder a la sección crítica
    - Sección crítica: instrucciones que acceden al recurso que está protegido por el semáforo
    - objSemaforo.signal() indica que un proceso abandona su sección y otro puede entrar en ella
    - El proceso padre creará tantos semáforos como secciones críticas distintas; suele ser una sección crítica por cada recurso compartido

Ventajas: Son fáciles de comprender, proporcionan una gran capacidad funcional.

Desventajas: Son peligrosos de manejar, pueden causar muchos errores como el interbloqueo. Cualquier olvido o cambio de orden puede conducir a bloqueos. La gestión de un semáforo se distribuye por todo el código lo que hace que la depuración de errores sea difícil.

En Java existe la clase Semaphore, ésta se aplica a los hilos de un mismo proceso, para arbitrar el acceso de estos hilos a una misma región de memoria.

Monitores

El problema que tienen los semáforos es que es el programador el que se encarga de implementar el uso correcto de los mismos para proteger el uso compartido.

Los monitores son como guardaespaldas, se encargan de proteger uno o varios recursos específicos, encierran esos recursos de tal forma que el proceso sólo puede acceder a ellos a través de los métodos que el monitor expone.

- **Monitor**: Componentes de alto nivel de abstracción destinado a gestionar recursos que van a ser accedidos de forma recurrente.

Los monitores encierran en su interior las variables y recursos compartidos como componentes privados y garantizan el acceso a ellos en exclusión mutua.

¿Cómo se declara un monitor?

- Declaración de las constantes, variables, procedimientos y funciones que son privados del monitor (sólo el monitor tiene visibilidad de ellos).
- Declaración de los procedimientos y funciones que expone, es decir, lo que es público. Es la interfaz a través de que los procesos acceden al monitor.
- Cuerpo del monitor construido por el bloque de código que se ejecuta al ser instanciado o inicializado el monitor. Su finalidad es inicializar las variables y funciones internas para poder usarlas.
- El monitor garantiza el acceso al código interno de forma exclusiva (régimen de exclusión mutua).
- Tiene asociada una lista de procesos que tratan de acceder al recurso mientras quedan suspendidos.

Los paquetes Java no proporcionan una implementación para la clase monitor, hay que implementar un monitor para cada variable o recurso compartido.

Se puede crear una clase monitor usando semáforos.

Cuando realizamos lectura o escritura de un fichero, los procesos quedan bloqueados hasta que el proceso que lo está usando lo libere (cuando haya finalizado la operación).

Nosotros inicializamos el acceso a un recurso indicando su ruta al crear el objeto (por ejemplo cuando usamos FileReader) y utilizamos métodos expuestos por ese objeto para realizar las operaciones que queramos.

Sin embargo el código que realmente realiza estas operaciones es el implementado en la clase FileReader. Por ejemplo, esta clase no te garantiza que el acceso al recurso sea en régimen de exclusión mutua, o por lo menos no en todos sus métodos.

Utilizamos objetos de tipo monitor, aunque no se llamen monitor para acceder a los recursos del sistema.

Ventajas

- **Uniformidad**, el monitor nos provee de una única capacidad, la exclusión mutua, con lo cual no existe la confusión de los semáforos.
- **Modularidad**, el código que se ejecuta en exclusión mutua está separado no entre mezclado con el resto del programa.
- **Simplicidad**, el programador no necesita preocuparse de las herramientas de exclusión mutua.
- **Eficiencia de la implementación**, la implementación no tiene por qué se controlada o revisada.

Desventajas

- Interacción de múltiples condiciones de sincronización. Cuando el número de condiciones crece y se hacen complicadas la complejidad del código crece extraordinariamente.

Monitores: Lecturas y escritura bloqueantes en recursos compartidos

Funcionamiento de los procesos cliente y suministrador:

- Utilizan un recurso compartido (del sistema), el suministrador introduce elementos y el cliente los extrae.
- Se sincronizan utilizando una variable compartida que indica el número de elementos que tiene el recurso cuyo tamaño máximo será N.
- El proceso suministrador siempre comprueba, antes de introducir un elemento, que esa variable tenga un valor menor que N. Al introducir un elemento, el valor de esta variable se incrementa en 1.
- El proceso cliente comprueba que haya elementos que pueda extraer, en ese caso los extrae y va decrementando el valor de la variable compartida.

Los mecanismos de sincronismo que permiten lo anterior son las lecturas y escrituras bloqueantes de recursos compartidos.

Para Java son:

Arquitectura java.io

- **Implementación de clientes**: Las clases derivadas de Reader (FileReader, InputStream, InputStreamReader) y los métodos read(Buffer) y read(Buffer, desplazamiento, tamaño).
- **Implementación de suministradores**: Con sus semejantes y derivados de Writer y los métodos write(info) y write(info, desplazamiento, tamaño).

Arquitectura java.nio (disponible para la versión 1.4 dentro de java.nio.channels)

- **Implementación de clientes**: Sus clases FileChannel y SocketChannel y los métodos read(Buffer) y read(Buffer, desplazamiento, tamaño).
- **Implementación de suministradores**: Sus clases FileChannel y SocketChannel y los métodos write(info, desplazamiento, tamaño).

Utilizamos el método lock() de FileChannel para implementar las secciones críticas de forma correcta. Tanto para clientes como para suministradores.

Memoria Compartida

En la comunicación ente procesos existe la posibilidad de disponer de zonas de memoria compartida (variables, estructuras…). Los mecanismo de sincronización en programación concurrente, las regiones críticas, semáforos y monitores; tienen su razón de ser en la existencia de recursos compartidos, incluyendo la memoria compartida.

Cuando se crea un proceso, el sistema operativo le asigna los recursos iniciales que va a necesitar, el principal recurso que necesita es la zona de memoria en la que se guardan las instrucciones, datos y la pila de ejecución.

Los sistemas operativos modernos permiten proteger la zona de memoria de cada proceso, siendo privada para cada proceso, de tal forma que otros no puedan acceder a ella.

A pesar de esto, hay posibilidad de tener comunicación entre procesos por medio de memoria compartida, gracias a la programación multi-hilo (teniendo varios flujos de ejecución de un mismo proceso, se comparte entre ellos la memoria asignada al proceso).

Problemas que nos surgen si lo resolvemos con un solo procesador.

Ordenar los elementos de una matriz: ordenar una matriz pequeña no supone ningún problema, pero si se hace muy grande, y si disponemos de varios procesadores y somos capaces de partir la matriz en trozos de forma que cada procesador se encargue de ordenar cada parte de la matriz, conseguiremos resolver el problema en menos tiempo, aunque hay que tener en cuenta la complejidad de dividir el problema y asignar a cada procesador un conjunto de datos (la zona de memoria) que tiene que manejar y la tarea a proceso a desarrollar y finalizar con la tarea de combinar todos los resultados. Esto es el caso de los sistemas multiproceso, como los actuales microprocesadores de varios núcleos.

OpenMP. Api para la programación multiproceso de memoria compartida en múltiples plataforma. Se compone de un conjunto de directivas de compilador, rutinas de biblioteca y variables de entorno que influencian el comportamiento en tiempo de ejecución. OpenMP es un modelo de programación portable y escalable que proporciona a los programadores una interfaz simple y flexible para el desarrollo de aplicaciones paralelas para las plataformas que van desde la computadora de escritorio hasta las supercomputadores.

COLA DE MENSAJES

El paso de mensajes es una técnica de programación concurrente, se usa para apartar sincronización entre procesos y permitir la exclusión mutua, similar a la que sucede con los semáforos y monitores, su principal diferencia es que no necesita memoria compartida.

Elementos principales que intervienen en el paso de mensajes:

Proceso que envía.

Proceso que recibe.

Mensaje.

Dependiendo si el proceso que envía el mensaje tiene que esperar a que le mensaje sea recibido para continuar su ejecución, el paso de mensajes podrá ser síncrono o asíncrono.

En el paso de mensajes asíncronos el proceso que envía el mensaje no espera a que éste sea recibido y continúa su ejecución, siendo posible que vuelva a generar un nuevo mensaje y a enviarlo antes de que se haya recibido el anterior. Por este motivo se suelen enviar buzones o colas, para almacenar mensajes a espera de que un proceso los reciba.

Generalmente, cuando usamos este sistema, proceso que envía mensajes solo se bloquea o para cuando finaliza su ejecución o si el buzón está lleno. Para conseguir esto, se establece un protocolo entre emisor y receptor, de forma que el receptor puede indicar al emisor qué capacidad restante queda en su cola.

En el paso de mensajes síncronos el proceso que envía mensajes espera a que un proceso lo reciba, de esta forma podrá continuar su ejecución. A esta técnica se le suele llamar encuentro o rendezvous.