5- Sincronización y comunicación basada en variables compartidas

1. Introducción

En la programación concurrente existen dos unidades básicas de ejecución:

- **Procesos**: proporciona un contexto de ejecución completo con sus propios recursos y espacio de memoria para permitir la ejecución de un programa. Pueden contener varios hilos de ejecución que comparten entre sí los recursos del proceso.
- **Hilos**: proporciona también un entorno de ejecución, pero su creación requiere menos recursos.

Cuando varios procesos o hilos se ejecutan de forma concurrente, se hace necesario considerar las interacciones entre las tareas para asegurar la correcta ejecución. Si estos son independientes, se pueden ejecutar en paralelo sin ningún problema, pero es habitual que haya dependencia por compartir información, recursos o por restricciones lógicas entre ellos.

Es importante que haya sincronización entre tareas que satisfagan requisitos en el orden de ejecución, así como comunicación para el paso de información entre tareas y mantener la sincronización (la comunicación requiere sincronización, mientras que la sincronización puede considerarse comunicación sin contenido).

La comunicación entre tareas se realiza habitualmente mediante:

- **Variables compartidas**: son objetos que se comparten entre tareas, por lo que son fáciles de implementar si hay memoria compartida. Es una buena solución para sistemas multiprocesador con memoria compartida.
- **Paso de mensajes**: supone el envío explícito de información entre tareas que no comparten espacio de memoria, lo que lo hace adecuado para sistemas distribuidos.

Mediante estos métodos se pueden resolver los mismos problemas de concurrencia utilizando enfoques diferentes.

2. Variables compartidas

Las variables compartidas son objetos accesibles por varias tareas que pueden leerlas o escribirlas según sus necesidades. De esta manera hay un intercambio explícito de información de modo que hay comunicación entre tareas. Las variables requieren sincronización para asegurar una correcta interacción cuando varios procesos acceden a información compartida mediante variables.

El acceso y modificación de las mismas se debe realizar de forma atómica e indivisible, es decir, ningún proceso puede interrumpir o interactuar con una variable si se está llevando a cabo una operación. En caso contrario, puede haber secuencias de ejecución en las que el resultado sea incorrecto.

Esta dependencia del resultado con el orden de ejecución de los procesos se denomina **condición de carrera**, ya que, en cierta manera, existe una competición entre procesos para obtener el acceso a la variable compartida.

Exclusión mutua

Las secuencias de código que deben ejecutarse de forma indivisible constituyen una **sección crítica**. Además, deben tener garantizada la **exclusión mutua**, es decir, no puede haber dos tareas ejecutándose simultáneamente en la misma sección crítica. Así es necesaria la sincronización entre ellas para proteger una sección crítica.

Si dos tareas no comparten variables, no es necesario usar exclusión mutua, aunque pueden necesitar sincronización (una tarea solo puede ejecutarse cuando otra tarea se encuentra en un determinado estado). Un ejemplo sería el problema del productor-consumidor, donde dos tareas intercambian datos a través de un búfer; un productor no debe introducir un dato si el búfer está lleno, y un consumidor no debe extraer un dato si el búfer está vacío.

3. Espera activa u ocupada

Uno de los mecanismos principales para la implementación de los métodos de sincronización consiste en utilizar una variable que actúa como bandera para controlar el acceso, tomando valores de cierto o falso. Este mecanismo se denomina **espera activa** porque el proceso debe comprobar continuamente el valor de la variable para detectar posibles cambios en su estado.

La condición de sincronización mediante el uso de la variable representa el estado que debe satisfacerse para que el proceso pueda continuar su ejecución, que utiliza su tiempo de ejecución para comprobar el valor de la variable. El proceso que señaliza la condición únicamente modifica el valor de la bandera.

```
task P1;
  ...
  while flag = down do
    null
  end;
  ...
end P1;
```

```
task P2;
  ...
  flag = up;
  ...
end P2;
```

Sin embargo, es un método poco eficiente, ya que:

- El bucle de espera no realiza trabajo útil, gasta todos sus ciclos de computador en comprobar el valor de la variable, en vez de usar éstos en otros procesos.
- En sistemas distribuidos puede provocar un tráfico excesivo.
- Es difícil utilizar colas de espera.
- Existe posibilidad de bloqueo activo o livelock.

La implementación de la espera activa tampoco es sencilla, utilizándose soluciones como el *algoritmo de Peterson*. Además, existen una serie de dificultades en el uso de la espera activa, como:

- Los protocolos son difíciles de entender, programar y demostrar su corrección.
- Las pruebas no examinan todas las posibles secuencias de ejecución.
- Es posible que existan secuencias de ejecución que lleven a una violación de la exclusión mutua o a un bloque activo.
- Los bucles de espera activa son ineficientes.
- Una tarea que haga un mal uso de las variables compartidas puede corromper todo el sistema.

Debido a esto, ningún lenguaje concurrente se basa por completo en la espera activa, acudiendo a otros mecanismos de sincronización como los semáforos o los monitores.

4. Suspender y reanudar

Para solucionar la exclusión mutua evitando la ineficiencia en uso de CPU de la espera activa, se utiliza un enfoque diferente como el uso de la **suspensión y reanudación**. Cuando una tarea se encuentra en una situación en la que debe esperar una condición externa para continuar su ejecución, pasa al estado de suspensión, dejando de estar en la lista de tareas ejecutables y añadiéndose a la de tareas suspendidas y cuando la condición se cumple, la tarea es reanudada.

Uno de los principales problemas de este mecanismo es la condición de carrera, que hace que el resultado depende del orden o la temporización y que además puede producirse un deadlock, llegando a una situación en la que se impide la evolución de los procesos en ejecución.

5. Semáforos

El **semáforo** es un mecanismo simple utilizado para controlar el acceso a recursos compartidos programando la exclusión mutua y la sincronización de condición. Aporta principalmente dos beneficios: simplifica los protocolos de sincronización y eliminación la necesidad de espera activa. Su funcionamiento se basa en una variable entera sobre la que se definen tres operaciones atómicas, lo que garantiza el correcto funcionamiento:

- **Inicialización** (init): asigna un valor inicial al semáforo pudiendo ser luego modificado únicamente por las operaciones de espera (wait) y señalización (signal), pero sin poder leer su valor actual.

- **Espera** (wait): disminuye en uno el valor del semáforo. Si el resultado es negativo, se bloquea la tarea hasta que el valor del semáforo sea positivo. De esta forma, se elimina temporalmente del planificador y no se le asigna tiempo de procesador hasta que se satisfagan las condiciones necesarias para que la tarea pueda reanudarse, lo que hace la espera más eficiente.
- **Señalización** (signal): incrementa en uno el valor del semáforo.

El semáforo es útil para solucionar problemas comunes de sincronización:

- **Tiempo de espera de una notificación**: las notificaciones son necesarias cuando un proceso o hilo tiene que esperar a un evento o a que se ejecute una sección de código en otro proceso, antes de seguir con su ejecución. Cuando un proceso A en espera (wait) de un evento de un proceso B, el evento B, llegado el momento, despierta al proceso A tras haber aumentado (signal) el valor del semáforo.
- **Encuentro entre procesos**: es una generalización de la notificación. Dos procesos se esperarán mutuamente antes de continuar con su ejecución. El primero que llegue al punto de encuentro, debe esperar al otro para que ambos puedan proseguir con su ejecución.
- **Mutex**: son semáforos usados para resolver el problema de la exclusión mutua. El semáforo se inicializa a 1 y cada proceso que quiera entrar en la sección crítica debe esperar, así como señalizar al salir de la misma. El primer proceso que encuentre un valor positivo podrá seguir ejecutándose entrando en la sección crítica, y los demás procesos encuentran un valor cero manteniéndose a la espera sin ejecutar su código.
  Este mecanismo puede generalizarse con múltiples procesos simultáneos inicializando el semáforo con dicho número de procesos, menos uno. De esta forma, a medida que los procesos van entrando en la sección critica, el valor del semáforo va decreciendo hasta llegar al último proceso premitido.
- **Barrera**: Es una generalización del encuentro entre procesos. Varios procesos deben esperar a reunirse en un punto de la ejecución antes de poder continuar.

Notificación

```
P1:
// Código previo al evento.
Evento.wait()
// Código posterior al evento.
P2:
// Código previo al evento.
Evento.signal()
// Código posterior al evento.
```

Encuentro

```
P1:
// Código previo al encuentro.
p1ok.signal()
p2ok.wait()
// Código posterior al encuentro.
P2:
// Código previo al encuentro.
P2ok.signal()
P1ok.wait()
// Código posterior al encuentro.
```

Mutex

```
Inicializar(mutex, 1)
P1:
   Esperar(mutex)
      // Sección crítica.
   Señalizar(mutex)
      // Código no sincronizado.
P2:
   Esperar(mutex)
      // Sección crítica.
   Señalizar(mutex)
      // Código no sincronizado.
```

Barrera

```
Inicializar(mutex, 1)
Inicializar(barrera, 0)
Pn:
  Esperar(mutex)
     Procesos ++
  Señalizar(mutex)
  if (procesos == n) Señalizar(barrera)
  Esperar(barrera)
  Señalizar(barrera)
```

6. Monitores

Regiones críticas condicionales

Una **región crítica** es una sección de código que tiene garantizada la ejecución en exclusión mutua mediante un mecanismo proporcionado por el lenguaje de programación, solventando así los problemas asociados con los semáforos. Una **región crítica condicional** es un bloque de código del que se garantiza su ejecución en exclusión mutua en relación con una variable de condición. Antes de entrar en dicha región, se comprueba la condición y si es cierta, se permite la entrada, y en caso contrario, la tarea queda bloqueada.

Monitores en JAVA

Para la implementación de monitores, es necesario definir una clase cuyos métodos públicos sean todos etiquetados con la palabra clave `synchronized`, lo que permite que el bloque proporcionado por el objeto controle el acceso exclusivo. La sincronización de los accesos se completa con el uso de los métodos:

- `wait`: su invocación provoca la suspensión del hilo actual, que libera el bloqueo sobre el objeto. Por tanto, debe ejecutarse sólo por el hilo que posea el bloqueo. La espera se realiza hasta que otro hilo realiza una llamada a `notify` o `notifyAll` sobre el objeto.
- `notify`: despierta a un solo hilo de los que se encuentran esperando para que pueda obtener de nuevo el bloqueo sobre el objeto y proseguir su ejecución. El hilo es elegido arbitrariamente entre los hilos que se encuentran esperando el bloqueo del objeto.
- `notifyAll`, despierta a todos los hilos esperando en el objeto.

Al ser notificados los hilos, éstos no obtienen el bloqueo, por lo que cuando se despiertan, deben competir por su acceso. Estos métodos deben ser llamados por un hilo que haya obtenido el bloqueo sobre una instancia de objeto, pudiéndolo hacer de tres formas:

- Ejecutando un método sincronizado del objeto.
- Ejecutando un bloque de código que sincroniza con el objeto.
- Para objetos de tipo `Class`, ejecutando un método sincronizado estático de la clase.

```
class ContadorSincronizado {
    private int contador = 0;
    public synchronized void incrementar() {
      contador ++;
    }
}
```

- **Problema del productor/consumidor**

Método depositar

```
public synchronized void depositar(int item)
  throws InterruptedException {
    while (lleno) {
        wait();
    }
    buffer[index++] = item;
    lleno = (index == size);
    vacio = false;
    notifyAll();
}
```

Método extraer

```
public synchronized int extraer()
  throws InterruptedException {
    while (vacio) {
        wait();
    }
    int valor = buffer[--index];
    vacio = (index == 0);
    lleno = false;
    notifyAll();
    return valor;
}
```

Productor

```
class Productor extends Thread {
    ProductorConsumidor buffer;
    public Productor(ProductorConsumidor buffer) {
        this.buffer = buffer;
    }
    public void run() {
        while (true) {
            try {
                int n = (int)(100*Math.random());
                int retardo = (int)(100*Math.random());
                Thread.sleep(retardo);
                buffer.depositar(n);
                System.out.println("Productor genera: " + n);
            } catch(InterruptedException e) {}
        }
    }
}
```

Consumidor

```
class Consumidor extends Thread {
    ProductorConsumidor buffer;
    public Consumidor(ProductorConsumidor buffer) {
        this.buffer = buffer;
    }
    public void run() {
        try {
            while (true) {
                int n = buffer.extraer();
                int retardo = (int)(100*Math.random());
                Thread.sleep(retardo);
                System.out.println("Consumidor procesa: " + n);
            }
        } catch(InterruptedException e) {}
    }
}
```

- **Problema de los lectores/escritores**

Empezar lectura

```
public synchronized void empezarLectura()
    throws InterruptedException {
        while (escribiendo || quiereEscribir > 0) {
            wait();
        }
        lectores ++;
    }
```

Terminar lectura

```
public synchronized void terminarLectura()
    throws InterruptedException {
        lectores --;
        if (lectores == 0) {
            notifyAll();
        }
    }
```

Empezar **escritura**

```
public synchronized void empezarEscritura()
    throws InterruptedException {
        quiereEscribir ++;
        while (lectores > 0 || escribiendo) {
            wait();
        }
        quiereEscribir --;
        escribiendo = true;
    }
```

Terminar **escritura**

```
public synchronized void terminarEscritura()
    throws InterruptedException {
        escribiendo = false;
        notifyAll();
    }
```