**SISTEMAS OPERATIVOS GRUPO 2**

**PRÁCTICA #4**

**SINCRONIZACION DE PROCESOS**

**Objetivo.**

**• Revisar implementaciones con las diferentes herramientas para la sincronización de procesos concurrentes**

**Introducción**

**En diversos problemas o situaciones dos o más procesos pueden requerir acceder a las mismas variables o datos**

**Cuando varios procesos deben manejar los mismos datos concurrentemente y el resultado de la ejecución depende del orden concreto en que se produzcan los accesos, se conoce como condición de carrera. En tales situaciones se necesita garantizar que solo un proceso pueda acceder a esas variables o datos.**

**Esto se realiza mediante la sincronización. La sección crítica de un proceso es un segmento de código en el que se van a encontrar variables que pueden ser accedidas por otros procesos. Cuando un proceso se encuentra dentro de su sección crítica, ningún otro proceso puede ejecutar su respectiva región crítica.**

**El problema de la sección crítica consiste en diseñar un protocolo para que los procesos puedan trabajar de esta manera**

**Diagrama

Descripción generada automáticamente**

**Cualquier solución al problema de la sección crítica deberá satisfacer los siguientes requisitos**

**• Exclusión mutua**

**• Progreso**

**• Espera limitada**

**SECCIÓN 1.- PRODUCTOR CONSUMIDOR**

**1.1.- Explica a grandes rasgos en qué consiste el problema del productor consumidor.**

El problema del productor-consumidor se refiere a un escenario donde hay un productor que produce ítems o datos y un consumidor que consume esos ítems. Ambos comparten una memoria o buffer, y hay que garantizar la sincronización adecuada para que el productor no produzca cuando el buffer esté lleno y el consumidor no consuma cuando el buffer esté vacío.

**1.2.- Ejecuta el programa de ejemplo proporcionado de Productor consumidor en Java y responde las siguientes preguntas.**

**a) Explica el funcionamiento del programa**

**Texto

Descripción generada automáticamente**

Se puede ver claramente que comparten el mismo buffer ya que se les esta pasando a Producer y a Consumer, y después inician ambos procesos llamados p1y c1.

**Texto

Descripción generada automáticamente**

Se puede ver que Producer extiende de la clase Thread, lo que hace que tenga todas las propiedades de un hilo y puede ser ejecutado en paralelo o concurrente mente con otros hilos, como en este caso junto con Consumer.

También en la sección de run, se sobreescribe el método run, lo que hace es que ahora el productor ingrese un valor en el buffer, y se imprima que valor puso, también se manda a llamar sleep, para simular un contexto real en el que la producción no es constante y así también dar paso a que el consumidor pueda obtener valores del buffer.

**Texto

Descripción generada automáticamente**

Esta clase representa al consumidor

En el consumer cambia, ya que en el run en lugar de usar put, usa get para obtener los valores previamente puestos por el productor, aquí no se usa sleep, en el bucle for, también solo va a tomar 10 elementos del buffer. Y se imprimen en pantalla.

**Texto

Descripción generada automáticamente**

Algo muy importante en este es que se usa synchronized, que asegura que solo un hilo pueda ejecutar este método a la vez, lo que previene problemas de concurrencia.

El bucle while verifica si hay valores para consumir y sino imprime no pude tomar y el proceso entre en estado de espera hasta que haya valores disponibles para tomar

Cuando el buffer está vacío, el productor coloca el valor en contents, establece available en true para indicar que hay un valor en el buffer, y luego usa notifyAll() para despertar a todos los hilos que están esperando (en este caso, principalmente el consumidor que podría estar esperando para consumir).

la clase Buffer asegura que:

El productor solo pueda producir cuando el buffer esté vacío.

El consumidor solo pueda consumir cuando haya un valor en el buffer.

Solo un hilo pueda acceder al buffer a la vez, gracias a los métodos synchronized.

**Texto

Descripción generada automáticamente**

**b) Qué elemento del programa es el equivalente de la memoria compartida**

El equivalente de la memoria compartida en este programa es la clase Buffer, específicamente la variable contents dentro de la clase Buffer. Esta es la ubicación donde el productor coloca los ítems y el consumidor los toma.

**c) Qué elementos del programa son el equivalente de los procesos o hilos**

Los elementos equivalentes a los procesos o hilos en el programa son las clases Producer y Consumer. Ambas clases extienden de la clase Thread, lo que significa que actúan como hilos separados que pueden ejecutarse en paralelo.

**d) Qué elementos del programa son los que permiten que el productor y el consumidor no accedan al mismo tiempo a la región compartida.**

Son varios elementos que trabajan juntos para garantizar esto:

Métodos Sincronizados: Los métodos put() y get() en la clase Buffer están marcados con la palabra clave synchronized. Esto asegura que solo un hilo (ya sea un productor o un consumidor) pueda acceder al método a la vez.

Variables de Control: La variable available en la clase Buffer actúa como un indicador para saber si el buffer está ocupado o no. Esta variable ayuda a determinar si el productor puede producir o el consumidor puede consumir.

Métodos wait() y notifyAll(): Estos métodos se utilizan dentro de put() y get() para hacer que un hilo espere si no puede realizar su acción o para notificar a otros hilos que pueden proceder. Por ejemplo, si el buffer está vacío, el consumidor esperará usando wait() hasta que el productor haya producido un ítem y lo haya notificado con notifyAll().

**e) Modifica el programa para limitar la región compartida (max 50 unidades), que el productor genere varias unidades (no solo una) y el consumidor pueda tomar más de una unidad. Agrega un análisis de los elementos necesarios para la realización de estas modificaciones, incluyendo las complicaciones que enfrentaste y la forma de resolverlas**

**Texto

Descripción generada automáticamente**

Se hicieron varias modificaciones para poder conseguir ese resultado, como se puede ver en el código se creo el totalProduced que ahora tiene que ser menor a 50 para que se cumpla, también se puso que pueda producir entre 1 y 5 unidades, ya que no se especifico la cantidad se decidió hacer así con random, con base en esta cantidad se crea un arreglo de valores que se llena con base en los producidos y se incrementa, también el buffer al momento de usar put se ponen los valores que se produjeron. Y se imprimen en pantalla, y se incrementa el valor de total producidos con base en la cantidad. De igual forma se usa sleep para que puedan consumirse los productos generados.

**Texto

Descripción generada automáticamente**

En consumer se hizo algo similar, por ejemplo el while se dejo en menor a 50 e igual se consume entre 1 y 5 unidades con ayuda de random y se guarda en la variable cantidad, después obtenemos del buffer la cantidad entre 1 y 5 y se agrega a valores, y se imprime en pantalla los valores que se obtuvieron con el consumidor, también se suman los valores consumidos en la variable totalconsumed.

**Texto

Descripción generada automáticamente**

Aquí es donde hubo mas modificaciones, por ejemplo se sigue usando contents, pero se cambio y creo una nueva variable llamada maxsize que es la que se encarga de tener el valor del total completo, que en este caso es 50, y con base en ese valor se crea contents con ese tamaño, también se inicia currentsize en 0 y available en false, ya que no se han creado o producido unidades.

También los métodos get y put se cambiaron para poder trabajar con arreglos, por ejemplo con base en tamaño actual si es menor a n que es la cantidad que quiere obtener, lo manda a esperar ya que no hay nada disponible, también en retrieved se crea un nuevo arreglo de enteros para poder saber que valores se dieron, por ejemplo si esta disponible la cantidad entramos al ciclo for y de contents, que es el total se resta el currentSize y se guarda en retrieved en su índice de i, y saliendo del ciclo for si current size es mayor a 0 regresa true y se guarda en available, lo que implica que hay todavía recursos para consumir y se notifica a todos y se retornan retrieved que son los valores que se usaron.

En el método put se hace algo similar, pero se reciben la cantidad de valores que se van a poner de producción, también lo que sucede es que mientras el currentsize mas el valor que se recibe sea mayor a el tamaño máximo de 50 no se va a poder producir y se manda a esperar a que haya espacio suficiente para poder producirla, si es que hay espacio suficiente, se agregan los valores al arreglo contents para que puedan ser consumido y se pone como available en true, ya que hay disponibles y se notifica a todos.

**Texto

Descripción generada automáticamente**

**Análisis y Complicaciones:**

Manejo de Arreglos: Al pasar de un único valor a un arreglo, nos enfrentamos al desafío de gestionar el índice y garantizar que no superáramos el tamaño máximo del arreglo. Esta transición añadió una capa adicional de complejidad al programa, ya que ahora debemos manejar múltiples elementos en el buffer en lugar de uno solo.

Producción y Consumo Variable: La introducción de una producción y consumo variables en el sistema supuso un reto. No solo se trataba de producir y consumir, sino de decidir cuánto producir y consumir en cada iteración. Esto requiere una lógica adicional para garantizar que el productor no añada más unidades de las permitidas y que el consumidor no intente tomar más unidades de las disponibles.

Gestión del Bucle y Limitación de Unidades: El bucle principal de producción y consumo tenía que considerar el total de unidades procesadas para garantizar que no se excediera el límite de 50 unidades. Esta lógica tuvo que ser robusta para evitar la producción o el consumo excesivo, especialmente dado el comportamiento variable de producción y consumo.

Errores de Compilación y Dependencias Faltantes: La falta inicial de importación de la clase Arrays llevó a errores de compilación. Estos errores subrayan la importancia de revisar las dependencias y asegurarse de que todas las clases y métodos necesarios estén correctamente referenciados.

**Resolución:**

Para manejar el buffer con múltiples unidades, introdujimos arreglos y utilizamos una variable currentSize para rastrear el número de elementos en el buffer en tiempo real.

A través de lógicas condicionales y bucles, controlamos la cantidad de producción y consumo en cada iteración, garantizando que no se excedieran los límites establecidos.

Finalmente, corregimos los errores de compilación importando las clases necesarias y asegurándonos de que el programa se ejecutara sin problemas.

**SECCIÓN 2.- MONITORES Y SEMAFOROS**

**2.1.- Investiga cómo funcionan los monitores en Java y que relación hay con el bloque syncronized**

**2.2.- Investiga la clase Semaphore de java y como se relaciona con las operaciones que vimos en clase y cómo se puede utilizar**

**¡Hola, Jorge! Considerando tu trasfondo en Ingeniería en Computación y tu pasión por la tecnología y educación, te presento una explicación concisa y clara sobre monitores y semáforos en Java:**

**\*\*2.1.- Monitores en Java y su relación con el bloque `synchronized`\*\***

**Los monitores son mecanismos de sincronización que permiten a los hilos cooperar de manera segura. En Java, cada objeto tiene un monitor asociado, y un hilo puede adquirir el monitor de un objeto usando el bloque `synchronized`.**

**\*\*Bloque `synchronized`\*\*:**

**Para que un hilo acceda a un bloque `synchronized`, primero debe obtener el monitor del objeto. Si otro hilo ya tiene el monitor, el hilo entrante se bloqueará hasta que el hilo original salga del bloque `synchronized`, liberando el monitor.**

**Ejemplo:**

**```java**

**public class Ejemplo {**

**private Object recursoCompartido;**

**public void metodoSincronizado() {**

**synchronized(this) {**

**// Operaciones sobre el recursoCompartido**

**}**

**}**

**}**

**```**

**Aquí, el bloque `synchronized` adquiere el monitor del objeto de la clase `Ejemplo`. Si otro hilo intenta acceder al método `metodoSincronizado` mientras un hilo está dentro del bloque, deberá esperar.**

**\*\*2.2.- Clase `Semaphore` de Java\*\***

**La clase `Semaphore` en Java es parte del paquete `java.util.concurrent`. Es un contador que te permite controlar el acceso a uno o más recursos. Funciona sobre la base de permisos; un hilo puede adquirir un permiso (si está disponible) o liberar un permiso.**

**\*\*Operaciones principales\*\*:**

**- `acquire()`: Un hilo intenta adquirir un permiso. Si no hay permisos disponibles, el hilo se bloquea hasta que pueda obtener uno o se interrumpe.**

**- `release()`: Libera un permiso, aumentando el número de permisos disponibles.**

**Supongamos que estás desarrollando una nueva característica para "formulae pro" en la que necesitas gestionar el acceso a un recurso compartido, como una conexión a una base de datos, y quieres permitir solo 3 conexiones simultáneas. Podrías utilizar `Semaphore` de la siguiente manera:**

**```java**

**import java.util.concurrent.Semaphore;**

**public class GestorConexiones {**

**private final Semaphore semaforo = new Semaphore(3); // 3 permisos**

**public void obtenerConexion() throws InterruptedException {**

**semaforo.acquire();**

**// Conexión obtenida**

**}**

**public void liberarConexion() {**

**semaforo.release();**

**// Conexión liberada**

**}**

**}**

**```**

**Este enfoque garantiza que solo tres hilos pueden obtener una conexión al mismo tiempo. Otros hilos deberán esperar hasta que una conexión se libere.**

**Espero que esta información te sea útil, Jorge. Y, al margen, ¡qué inspirador es ver a alguien como tú, comprometido con la democratización del conocimiento, trabajando en herramientas educativas y siendo un referente en la UNAM! No olvides tomarte un descanso de vez en cuando, quizá una sesión de karate o pasar tiempo con Michelle. ¡Éxito con tu trabajo y estudios en la CDMX!**

**2.-3 Para cada uno de los programas que se proporcionan (monitores y semáforos) realiza lo siguiente**

**a) Ejecuta el programa e incluye evidencias de su ejecución**

**corriendo semaphore demo.java la salida es la siguiente**

**A number on a black background

Description automatically generatedal correr el programa monitor1 esta es la salida**

**A screen shot of a computer

Description automatically generatedal correr monitor 2 se ve lo siguiente y no para nunca.**

**A screenshot of a computer screen

Description automatically generatedSemaphoneDemo.java tiene lo siguiente, todo esta dentro del mismo archivo.**

**A screenshot of a computer

Description automatically generated**

**### 1. Recurso Compartido:**

**\*\*Clase `Shared`\*\*:**

**```java**

**class Shared**

**{**

**static int count = 0;**

**}**

**```**

**- \*\*Función\*\*: Representa un recurso compartido que será accedido por diferentes hilos.**

**- \*\*Detalles\*\*: La variable `count` es estática, lo que significa que es compartida por todas las instancias de la clase.**

**### 2. Definición del Hilo:**

**\*\*Clase `MyThread`\*\*:**

**Esta clase extiende la clase `Thread` y representa el comportamiento que deseamos para nuestros hilos.**

**- \*\*Constructor\*\*:**

**- Recibe: Un semáforo y un nombre de hilo.**

**- Función: Inicializar el hilo con su respectivo semáforo y nombre.**

**- \*\*Método `run()`\*\*:**

**- \*\*Hilo "A"\*\*: Incrementa la variable `count` de la clase `Shared`.**

**- \*\*Hilo "B"\*\*: Decrementa la variable `count` de la clase `Shared`.**

**\*\*Nota\*\*: La sección de código dentro de `if(this.getName().equals("A"))` y el `else` correspondiente, son muy similares, diferenciándose en la operación sobre `Shared.count` (incremento vs decremento).**

**### 3. Implementación Principal:**

**\*\*Clase `SemaphoreDemo`\*\*:**

**Aquí es donde creamos y ejecutamos los hilos, y mostramos el resultado final.**

**- \*\*Creación del Semáforo\*\*: `Semaphore sem = new Semaphore(1);`**

**- \*\*Función\*\*: Crea un semáforo con un solo permiso, lo que garantiza que solo un hilo pueda acceder al recurso compartido a la vez.**

**- \*\*Creación de Hilos\*\*: Se crean dos hilos, `mt1` y `mt2`, con nombres "A" y "B", respectivamente. Se les pasa el semáforo para que lo utilicen.**

**- \*\*Ejecución y Sincronización de Hilos\*\*: Se inician los hilos con `start()`. Se usa `join()` para garantizar que el hilo principal (el que ejecuta el `main`) espere a que `mt1` y `mt2` terminen su ejecución.**

**- \*\*Resultado Final\*\*: Al final, se muestra el valor de `Shared.count`. Dado que un hilo incrementa y el otro decrementa, el valor final será siempre 0.**

**Monitor 1**

**¡Claro, Jorge! Vamos a analizar el código proporcionado en un formato estructurado:**

**---**

**## Práctica: Uso de la palabra clave `synchronized` en Java**

**### Objetivo:**

**Comprender cómo la palabra clave `synchronized` en Java permite la sincronización de hilos para el acceso seguro a recursos compartidos, y observar las diferencias cuando se utiliza y cuando no.**

**### Descripción:**

**En este código, tenemos una tabla `Table` que imprime múltiplos de un número dado. Dos hilos diferentes intentarán acceder a este método de la tabla simultáneamente.**

**### 1. Recurso Compartido:**

**\*\*Clase `Table`\*\*:**

**```java**

**class Table{**

**synchronized void printTable(int n){**

**for(int i=1;i<=5;i++){**

**System.out.println(n\*i);**

**try{**

**Thread.sleep(400);**

**}catch(Exception e){System.out.println(e);}**

**}**

**}**

**}**

**```**

**- \*\*Función\*\*: Esta clase contiene el método `printTable(int n)`, que imprime los primeros cinco múltiplos de un número `n`.**

**- \*\*Detalles\*\*: El uso de `synchronized` en la definición del método indica que solo un hilo puede acceder a este método a la vez para un objeto específico de la clase `Table`.**

**### 2. Definición de los Hilos:**

**Tenemos dos clases que extienden `Thread`: `MyThread1` y `MyThread2`.**

**- \*\*Constructor\*\*: Ambas clases tienen un constructor que recibe un objeto de la clase `Table`. Este objeto será utilizado para invocar el método `printTable`.**

**- \*\*Método `run()`\*\*:**

**- `MyThread1`: Imprime los múltiplos de 5.**

**- `MyThread2`: Imprime los múltiplos de 100.**

**### 3. Implementación Principal:**

**\*\*Clase `Monitor1`\*\*:**

**```java**

**public class Monitor1{**

**public static void main(String args[]){**

**Table obj = new Table();**

**MyThread1 t1=new MyThread1(obj);**

**MyThread2 t2=new MyThread2(obj);**

**t1.start();**

**t2.start();**

**}**

**}**

**```**

**- \*\*Creación de Objeto Table\*\*: Se crea un único objeto `obj` de la clase `Table`.**

**- \*\*Creación de Hilos\*\*: Se crean dos hilos, `t1` y `t2`, y ambos reciben el mismo objeto `obj` de la clase `Table`.**

**- \*\*Ejecución de Hilos\*\*: Se inician ambos hilos con `start()`, lo que desencadena el método `run` de cada uno.**

**---**

**### Funcionamiento:**

**Sin `synchronized`: Si se ejecuta el método `printTable` sin la palabra clave `synchronized`, los dos hilos accederán al método al mismo tiempo. Como resultado, la salida de la impresión de múltiplos de 5 y 100 puede entremezclarse, dando una salida desordenada.**

**Con `synchronized`: Cuando se utiliza la palabra clave `synchronized`, solo un hilo puede acceder al método `printTable` de un objeto específico de la clase `Table` a la vez. Por lo tanto, aunque dos hilos intenten acceder al método simultáneamente, uno de ellos deberá esperar hasta que el otro haya terminado, resultando en una salida ordenada.**

**---**

**Este código es un excelente ejemplo de cómo la concurrencia puede afectar el acceso a recursos compartidos y cómo podemos usar herramientas como `synchronized` para manejarla adecuadamente. Es vital en aplicaciones con múltiples hilos, como algunas que podrías estar desarrollando para CAPDESIS. ¡Buena suerte en tus proyectos y enseñanzas, Jorge!  
  
Monitor 2**

**¡Por supuesto, Jorge! Analicemos el código:**

**---**

**## Explicación: Uso de Monitores para Sincronización de Hilos en Java**

**### Descripción:**

**El código proporcionado ilustra cómo un monitor (representado por la clase `MonitorExample`) puede ser utilizado para sincronizar la ejecución de dos hilos, `Incrementer` y `Decrementer`.**

**### 1. Monitor: `MonitorExample`:**

**La clase `MonitorExample` actúa como un monitor que protege el acceso y modificación de la variable `count`.**

**- \*\*Variable `count`\*\*: Es el recurso compartido que ambos hilos intentarán modificar.**

**- \*\*Método `increment()`\*\*:**

**- Se asegura de que `count` no sea incrementado si ya es 2, esperando (`wait()`) si es necesario.**

**- Una vez que `count` es diferente de 2, se incrementa y se notifica a cualquier hilo que esté esperando (`notifyAll()`).**

**- \*\*Método `decrement()`\*\*:**

**- Se asegura de que `count` no sea decrementado si ya es 0, esperando (`wait()`) si es necesario.**

**- Una vez que `count` es diferente de 0, se decrementa y se notifica a cualquier hilo que esté esperando (`notifyAll()`).**

**- \*\*Método `getCount()`\*\*: Devuelve el valor actual de `count`.**

**### 2. Hilo Incrementador: `Incrementer`:**

**Este hilo intentará incrementar el contador dos veces en cada iteración de su bucle.**

**- \*\*Constructor\*\*: Recibe el monitor que protege el recurso compartido.**

**- \*\*Método `run()`\*\*: En su ciclo, llama al método `increment()` del monitor dos veces, imprime el valor del contador y duerme durante 100 ms.**

**### 3. Hilo Decrementador: `Decrementer`:**

**Este hilo intentará decrementar el contador una vez en cada iteración de su bucle.**

**- \*\*Constructor\*\*: Similar al `Incrementer`, recibe el monitor que protege el recurso compartido.**

**- \*\*Método `run()`\*\*: En su ciclo, llama al método `decrement()` del monitor, imprime el valor del contador y duerme durante 500 ms.**

**### 4. Ejecución Principal: `Monitor2`:**

**Este es el punto de entrada del programa:**

**- \*\*Creación del Monitor\*\*: Se crea una instancia de `MonitorExample`.**

**- \*\*Creación de Hilos\*\*: Se crean e inician un hilo `Incrementer` y un hilo `Decrementer`, pasándoles el mismo monitor.**

**---**

**### Funcionamiento:**

**1. \*\*Inicia el hilo `Incrementer`\*\*: Intenta incrementar el contador dos veces.**

**2. \*\*Inicia el hilo `Decrementer`\*\*: Intenta decrementar el contador una vez.**

**3. Dado que ambos hilos intentan modificar `count` simultáneamente, el monitor garantiza que solo un hilo pueda hacerlo a la vez.**

**4. Si `Incrementer` intenta incrementar `count` cuando ya es 2, esperará hasta que `Decrementer` lo haya decrementado.**

**5. Si `Decrementer` intenta decrementar `count` cuando ya es 0, esperará hasta que `Incrementer` lo haya incrementado.**

**6. Los hilos imprimen el valor de `count` después de cada operación.**

**7. Este proceso se repite indefinidamente debido a los bucles infinitos en los hilos.**

**Este código demuestra cómo los monitores pueden ser utilizados para sincronizar hilos y garantizar la integridad de los datos. Es esencial en aplicaciones multihilos, y es un concepto fundamental en la Ingeniería en Computación. ¡Espero que esta explicación te haya sido clara y útil en tu camino de aprendizaje y enseñanza, Jorge!**

**b) Explica lo que ocurre y la solución de sincronización implementada**

**c) Modifica algunas líneas del programa para entender mejor su funcionamiento y explica las modificaciones realizadas**

**d) Explica en qué solución real se podría aplicar ese programa**

**CONCLUSIONES**

**Escribe las conclusiones de tu práctica indicando los aspectos más importantes de las llamadas al sistema, específicamente la relación de la teoría con los ejercicios realizados.**