Universidad de los Andes
Facultad de Ingeniería
Departamento de Ingeniería de Sistemas y Computación
Tecnología e Infraestructura de Cómputo - ISIS 1311
Laboratorio 7 - Programar Con Concurrencia

Este trabajo debe ser elaborado individualmente

Puede hacer preguntas al asistente docente y a los monitores para resolver dudas. También puede intercambiar ideas con sus compañeros, pero no puede intercambiar con ellos ningún tipo de código.

Tenga en cuenta que los laboratorios están diseñados para apoyar su proceso de aprendizaje e identificar y resolver sus dudas sobre los temas estudiados, antes del parcial.

Objetivos

- Familiarizarse con los conceptos de espera pasiva, espera activa y espera semi-activa en Java.
- Implementar un programa que solucione el problema productor-consumidor de forma concurrente y sincronizada.

Contexto

En la programación concurrente, uno de los desafíos fundamentales es coordinar múltiples hilos que se ejecutan simultáneamente y manejan recursos compartidos. Consideremos un escenario clásico: el problema productor-consumidor. Un hilo produce datos y otro hilo los consume: el consumidor no puede procesar información que aún no existe, por lo que debe esperar a que el productor finalice su tarea.

Para manejar este tipo de escenarios, es fundamental sincronizar la ejecución de los hilos, asegurando que ciertas acciones ocurran en un **orden lógico, controlado y predecible**.

Existen tres estrategias diferentes para manejar esta sincronización:

1) **Espera Pasiva:** El hilo que está esperando **se bloquea por completo** y libera la CPU, entrando en un estado de suspensión sin consumir recursos. Solo **se despierta cuando otro hilo (por ejemplo, el productor)** le envía una señal indicando que la condición ya se cumplió. Esta es la estrategia **más eficiente** en cuanto al uso de recursos, aunque requiere una correcta coordinación para que el hilo que produce los datos notifique al que espera.

2) **Espera Activa:** El hilo que espera **no se bloquea**, sino que **verifica constantemente**, dentro de un bucle, si la condición se ha cumplido (por ejemplo, si ya hay datos disponibles). Aunque es sencilla de implementar, **mantiene a la CPU ocupada**

innecesariamente, incluso cuando no hay nada nuevo que procesar. Esto puede generar un desperdicio de recursos, especialmente si la espera es larga.

3) **Espera Semi-Activa:** El hilo que espera cede el procesador y luego lo vuelve a usar para comprobar si la condición se cumplió, repitiendo este ciclo hasta recibir los datos. Es un **punto intermedio** entre eficiencia y capacidad de respuesta: no bloquea completamente el hilo como en la espera pasiva, pero tampoco consume CPU de forma constante como en la espera activa.

Preparación

Asegúrese que su entorno de trabajo tenga instalado alguna versión de Java. Además, pueden trabajar en el IDE de su preferencia.

Se recomienda escribir/teclear directamente el código ya que favorece la retención de la sintaxis y fortalece la comprensión funcional de los conceptos, lo cual será muy útil **para el parcial**.

Parte 1: Espera Pasiva (Eventos y Señalización)

La espera pasiva es la estrategia más utilizada en Java para lograr una sincronización eficiente entre hilos. Esta se implementa mediante el uso de **eventos**, que son señales que un hilo genera para indicar que algo importante ha ocurrido. Mientras otro que otro hilo puede estar esperando pasivamente ese evento específico para poder continuar con su ejecución.

Este mecanismo de **señalización** permite que los hilos se comuniquen y coordinen de forma controlada, evitando desperdiciar recursos del sistema.

Conceptos fundamentales para entender la sincronización mediante espera pasiva:

- **Objeto Monitor**

  En Java, cada objeto tiene la capacidad de actuar como un monitor de sincronización. Un monitor es esencialmente un mecanismo que controla el acceso a recursos compartidos en un entorno concurrente.

  Cada monitor incluye un **candado (lock)** interno que funciona como una llave de acceso. Este candado protege las secciones críticas del código, que son aquellas partes donde se accede o modifica un recurso compartido entre varios hilos.

  La regla fundamental es: Solo un hilo puede poseer el candado a la vez. Esto significa que, cuando un hilo adquiere el candado de un objeto, ningún otro hilo podrá ejecutar código sincronizado (bloques o métodos encerrados con synchronized) sobre ese mismo objeto hasta que el candado sea liberado. Esta restricción garantiza que no se produzcan conflictos ni inconsistencias al acceder a datos compartidos (evitando condiciones de carrera).

- **Método wait()**

  El método wait() es la herramienta que Java proporciona para implementar la espera pasiva. Este método hace que un hilo se detenga temporalmente cuando no puede continuar su ejecución (Ej. Cuando espera a que se cumpla cierta condición).

  Cuando un hilo ejecuta wait() sobre un objeto monitor, suceden tres cosas importantes:

  1) El hilo **libera el candado** que poseía sobre ese objeto monitor. Esto es crucial porque permite que otros hilos adquieran el candado y trabajen con el recurso compartido, potencialmente generando la condición esperada por el hilo que liberó el candado.

  2) El hilo se añade a la **cola de espera** del objeto monitor y pasa a un estado de espera pasiva, quedando completamente bloqueado y **sin consumir tiempo de CPU**. Esta es la característica clave de la espera pasiva: el hilo no está verificando activamente la condición, sino que está dormido esperando ser despertado.

  3) El hilo permanecerá en ese estado dormido hasta que otro hilo invoque notify() sobre el **mismo objeto monitor**, momento en el cual intentará readquirir el candado para continuar.

  Es importante destacar que wait() solo puede ser invocado desde dentro de un bloque o método sincronizado (synchronized). El hilo debe poseer el candado del objeto antes de poder liberarlo, si no arrojará una excepción.

- **Método notify()**

  El método notify() se utiliza para señalizar a los hilos que están en espera pasiva que algo ha cambiado y que pueden intentar continuar su ejecución. Este método es el complemento necesario de wait() para implementar el patrón de espera pasiva completo.

  Cuando un hilo ejecuta notify() sobre un objeto monitor:

  1) **Despierta a uno** de los hilos que está en la cola de espera de ese objeto monitor (si hay varios esperando, se elige uno de manera arbitraria).

  2) El hilo despertado sale del estado de espera pasiva y pasa a un **estado "listo"**. Sin embargo, **no continúa inmediatamente**: primero debe competir con otros hilos para readquirir el candado del objeto monitor.

  3) Una vez que el hilo despertado logra adquirir el candado (cuando el hilo que llamó a notify() lo libera), puede continuar su ejecución desde el punto inmediatamente después de donde invocó wait().

  Es importante notar que el hilo que llama a notify() no libera inmediatamente el candado, continúa ejecutándose hasta que sale del bloque sincronizado o hasta que él mismo invoque wait().

Para entender mejor los conceptos, construiremos un programa sencillo que ilustra cómo funcionan los métodos wait() y notify() en Java.

En este programa, tendremos dos hilos que colaboran entre sí a través de un contador compartido. El primer hilo permanece bloqueado en espera pasiva hasta que detecta un cambio en el contador, mientras que el segundo hilo se encarga de disminuir periódicamente ese contador y notificar al primer hilo cada vez que se produce una modificación.

El programa utiliza un objeto compartido llamado **MonitorContador**, que contiene un contador inicializado con un valor determinado. Sobre este monitor actúan los dos hilos:

- El hilo **Oyente** permanece en espera pasiva, es decir, bloqueado hasta que el contador cambia. Cada vez que recibe una notificación, se despierta, imprime el nuevo valor del contador y vuelve a esperar.
- El hilo **Notificador**, por su parte, decrementa el contador cada cierto tiempo y notifica al Oyente cada vez que realiza un cambio.

El programa finaliza automáticamente cuando el contador llega a cero, momento en el que ambos hilos terminan su ejecución.

Cree una carpeta llamada **Parte1** que contenga los siguientes archivos:

- Oyente.java
- Notificador.java
- MonitorContador.java
- Main.java

A continuación, escriba en cada archivo el código tal cual aparece en las siguientes imágenes:

```java
public class MonitorContador {
    private int contador;

    public MonitorContador(int inicial) {
        this.contador = inicial;
    }

    public synchronized int getContador() {
        return contador;
    }

    public synchronized void esperarCambio(Thread thread) {
        // TO DO
    }

    public synchronized void decrementarYNotificar(Thread thread) {
        // TO DO
    }
}
```

Figura 1. Clase MonitorContador.java.

```java
public class Oyente extends Thread {
    private MonitorContador monitor;

    public Oyente(String nombre, MonitorContador monitor) {
        super(nombre);
        this.monitor = monitor;
    }

    @Override
    public void run() {
        System.out.println("[" + this.getName() + "]: Iniciado. Valor inicial = " + monitor.getContador());
        while (monitor.getContador() > 0) {
            monitor.esperarCambio(this);
        }
        System.out.println("[" + this.getName() + "]: Finalizado");
    }
}
```

Figura 2. Clase Oyente.java.

```java
public class Notificador extends Thread {
    private MonitorContador monitor;
    private long intervalo;

    public Notificador(String nombre, MonitorContador monitor, long intervalo) {
        super(nombre);
        this.monitor = monitor;
        this.intervalo = intervalo;
    }

    @Override
    public void run() {
        while (monitor.getContador() > 0) {
            try {
                Thread.sleep(intervalo);
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
            monitor.decrementarYNotificar(this);
        }
        System.out.println("[" + this.getName() + "]: Finalizado");
    }
}
```

Figura 3. Clase Notificador.java.

```java
public class Main {
    public static void main(String[] args) {
        System.out.println("=== EJEMPLO DE ESPERA PASIVA CON MONITOR ===\n");

        MonitorContador monitor = new MonitorContador(5);

        Thread oyente = new Oyente("Oyente", monitor);
        Thread notificador = new Notificador("Notificador", monitor, 2000);

        oyente.start();
        notificador.start();

        try {
            notificador.join();
            oyente.join();
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }

        System.out.println("\n=== PROGRAMA TERMINADO ===");
    }
}
```

Figura 4. Clase Main.java.

Como se puede observar en la **Figura 1**, la clase MonitorContador tiene dos métodos marcados con **TO DO**. Debe implementar estos métodos para completar la lógica de la espera pasiva, y además debe incluir mensajes en consola que permitan verificar la correcta sincronización entre los hilos.

```
=== EJEMPLO DE ESPERA PASIVA CON MONITOR ===

[Oyente]: Iniciado. Valor inicial = 5
[Oyente]: Dormido en espera pasiva
[Notificador]: Contador decrementado a 4
[Oyente]: Despierto. Cambio detectado: 5 -> 4
[Oyente]: Dormido en espera pasiva
[Notificador]: Contador decrementado a 3
[Oyente]: Despierto. Cambio detectado: 4 -> 3
[Oyente]: Dormido en espera pasiva
[Notificador]: Contador decrementado a 2
[Oyente]: Despierto. Cambio detectado: 3 -> 2
[Oyente]: Dormido en espera pasiva
[Notificador]: Contador decrementado a 1
[Oyente]: Despierto. Cambio detectado: 2 -> 1
[Oyente]: Dormido en espera pasiva
[Notificador]: Contador decrementado a 0
[Oyente]: Despierto. Cambio detectado: 1 -> 0
[Oyente]: Finalizado
[Notificador]: Finalizado

=== PROGRAMA TERMINADO ===
```

Figura 5. Ejemplo de la salida del programa (parte 1).

Una vez implementados, **ejecute/verifique el programa al menos tres veces** y responda las siguientes preguntas (En forma de comentarios en el archivo Main.java):

- ¿Qué hace exactamente el hilo Oyente cuando llama al método esperarCambio()? ¿En qué estado queda?
- ¿Qué ocurre con el hilo **Oyente** si elimina la llamada a notify() dentro del método decrementarYNotificar()? Realice el cambio y ejecute el programa para observar su comportamiento.
- ¿Qué ocurriría si en vez de notify() se usara notifyAll() en decrementarYNotificar()? ¿Habría alguna diferencia en este caso? ¿Qué pasaría en el caso hipotético de que hubiera más de un oyente?
- ¿Por qué los métodos de la clase MonitorContador están declarados como synchronized?
- ¿Qué podría ocurrir si se elimina synchronized de esperarCambio() y se ejecuta el programa?

Al responder, no se limite a “lo que hace el código”, sino que explique **qué sucede con los hilos, en qué estado se encuentran y por qué ocurre** dicho comportamiento.

Parte 2: Espera Activa

La espera activa es una estrategia básica de sincronización en la que un hilo no se suspende mientras espera que ocurra un evento, sino que verifica constantemente si la condición que necesita para continuar se ha cumplido.

A diferencia de la espera pasiva, donde el hilo "duerme" y libera el candado hasta ser notificado, en la espera activa el hilo permanece ejecutándose, revisando una variable compartida (o un estado) una y otra vez.

Esto permite que el hilo reaccione inmediatamente cuando la condición cambia, pero también tiene un costo: puede desperdiciar tiempo de CPU si la condición tarda en cumplirse.

El mecanismo central de la espera activa es un bucle de verificación constante. Un hilo ejecuta repetidamente una condición, por ejemplo:

```
while (!condicionCumplida) {
    // No hace nada útil, solo espera
}
```

Figura 6. Estructura de la espera activa.

En este fragmento, el hilo no se bloque ni libera recursos. Está activo todo el tiempo, consumiendo CPU mientras espera que otro hilo modifique la variable **condicionCumplida**. Este tipo de bucles son sencillos de implementar, pero no escalan bien cuando hay muchos hilos o cuando los tiempos de espera son largos, ya que cada hilo que espera está ocupando el procesador sin hacer trabajo útil.

Basándose en el mismo programa de la parte 1, implemente ahora, en lugar de la espera pasiva, **la espera activa,** para que el Oyente esté constantemente revisando los cambios del contador.

```
=== EJEMPLO DE ESPERA ACTIVA CON MONITOR ===

[Oyente]: Iniciado. Valor inicial = 5
[Notificador]: Contador decrementado a 4
[Oyente]: Cambio detectado: 5 -> 4
[Notificador]: Contador decrementado a 3
[Oyente]: Cambio detectado: 4 -> 3
[Notificador]: Contador decrementado a 2
[Oyente]: Cambio detectado: 3 -> 2
[Notificador]: Contador decrementado a 1
[Oyente]: Cambio detectado: 2 -> 1
[Notificador]: Contador decrementado a 0
[Oyente]: Cambio detectado: 1 -> 0
[Oyente]: Finalizado
[Notificador]: Finalizado

=== PROGRAMA TERMINADO ===
```

Figura 7. Ejemplo de la salida del programa (parte 2).

Cree una carpeta llamada **Parte2** que contenga los siguientes archivos con la implementación de la espera activa:

- Oyente.java
- Notificador.java
- MonitorContador.java
- Main.java

Parte 3: Espera Semi-Activa

La espera semi-activa es una estrategia intermedia entre la espera activa y la espera pasiva. Su objetivo es reducir el desperdicio de CPU que ocurre en la espera activa, sin necesidad de bloquear completamente el hilo como en la espera pasiva.

En esta técnica, el hilo que espera sigue verificando constantemente si la condición que necesita se ha cumplido, pero cede el procesador en cada iteración para que otros hilos puedan avanzar.

Para lograr esto, en Java se utiliza el método **yield()** de la clase Thread. Este método es una sugerencia al planificador de hilos para que el hilo actual ceda voluntariamente la CPU. Esto no significa que el hilo se bloquee ni entre en espera pasiva; simplemente indica que está dispuesto a dejar que otros hilos en estado "listo" se ejecuten.

```
while (!condicionCumplida) {
    Thread.yield();  // El hilo cede el procesador
}
```

Figura 8. Estructura de la espera semi-activa.

En este bucle, el hilo continúa esperando activamente la condición, pero entre cada verificación le da una oportunidad al sistema operativo de ejecutar otros hilos, evitando así que monopolice la CPU de manera continua.

Basándose en el mismo programa de la parte 2, implemente ahora, en lugar de la espera activa, **la espera semi-activa**, de modo que el Oyente esté constantemente revisando los cambios del contador, pero sin consumir la CPU de forma continua.

Para ello, cree una carpeta llamada **Parte3** que contenga los siguientes archivos con la implementación del espera semi-activa:

- Oyente.java
- Notificador.java
- MonitorContador.java
- Main.java

Parte 4: Ejercicio práctico

Implemente un programa concurrente que solucione el problema productor-consumidor. Este se basa en la existencia de un conjunto de productores que generan objetos y un conjunto de consumidores que los procesan. El proceso de producción y consumo se realiza a través de un búfer, el cual almacena temporalmente dichos objetos.

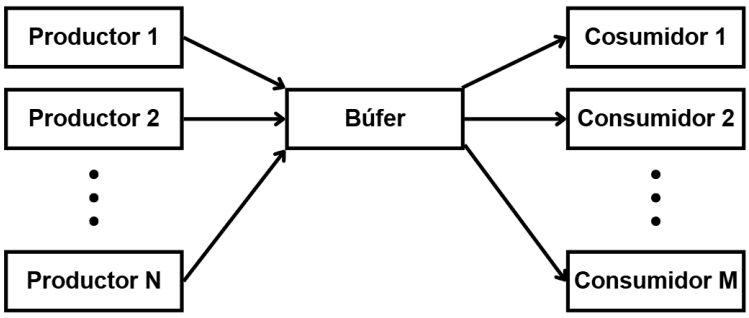

Figura 9. Diagrama productor-consumidor con búfer.

Cree una carpeta llamada **Parte4** que contenga la implementación del programa concurrente. En general, su programa debe seguir la estructura del siguiente diagrama y cumplir con los requisitos mencionados posteriormente:

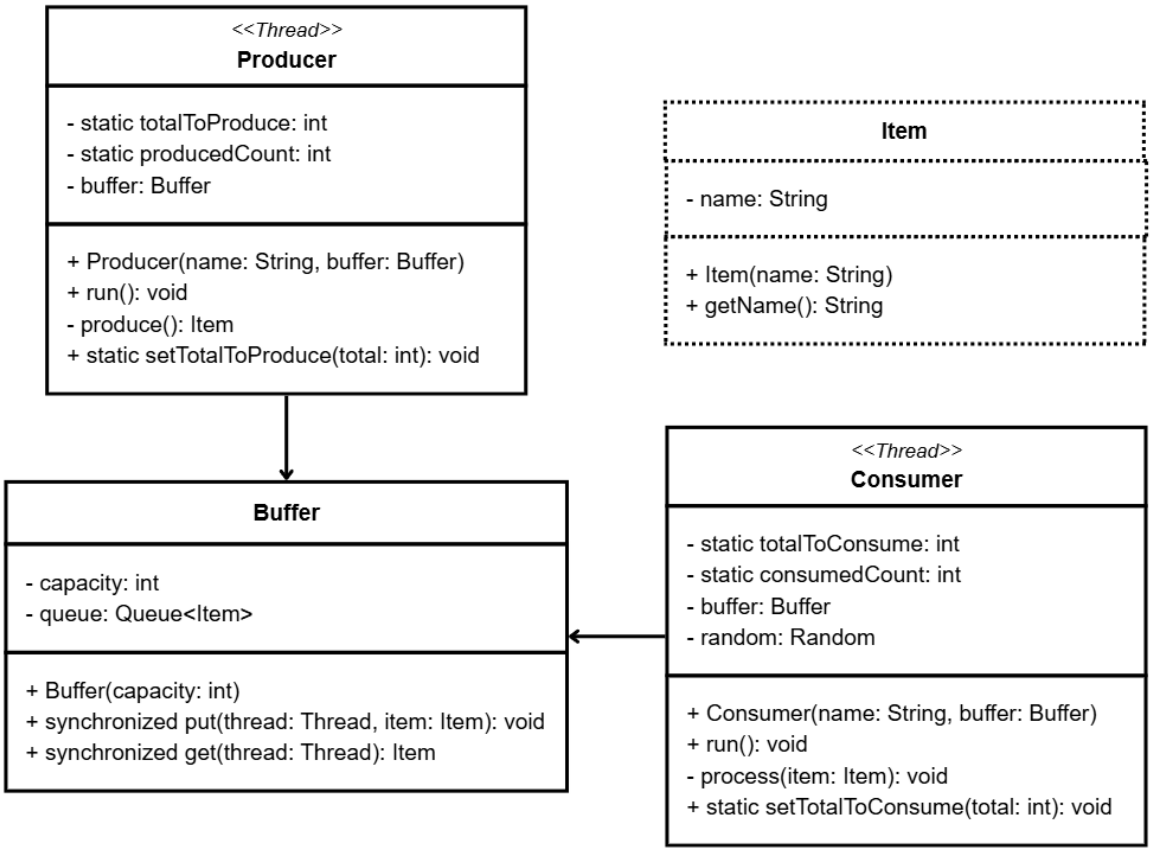

Figura 10. Diagrama de clases guía para el desarrollo del programa.

Requisitos:

1) Hay N productores, M consumidores y un Búfer con capacidad finita.

2) Los productores conocen el número total de ítems a producir y los consumidores conocen el número total de ítems a consumir. Tenga en cuenta que el total de ítems a producir es igual al total de ítems a consumir.

3) La comunicación entre los productores y el búfer debe implementarse con **espera pasiva**. Cuando el búfer está lleno, el productor que quiera almacenar un ítem debe dormirse sobre el monitor del búfer hasta que haya espacio disponible.

4) La comunicación entre el búfer y los consumidores debe implementarse con **espera semi-activa**. Los consumidores deben revisar constantemente si hay un ítem disponible para consumir en el búfer, liberando el procesador cuando lo encuentren vacío.

5) Cuando un consumidor toma un ítem del búfer, debe despertar a los productores que estén dormidos sobre el monitor del búfer. Debe pensar si es correcto utilizar notify() o notifyAll().

6) Una vez el consumidor tenga un ítem debe simular su procesamiento durando un tiempo aleatorio entre 1 y 3 segundos. Al terminar, sigue revisando en el búfer si hay ítems para procesar.

7) El programa debe generar prints a lo largo de la ejecución para visualizar que se cumpla correctamente la sincronización.

8) Cree un archivo Main.java que contenga un método main que inicialice y ejecute toda la configuración del sistema. El hilo principal debe esperar la finalización de todos los hilos antes de imprimir un mensaje indicando que la ejecución del programa ha terminado correctamente.

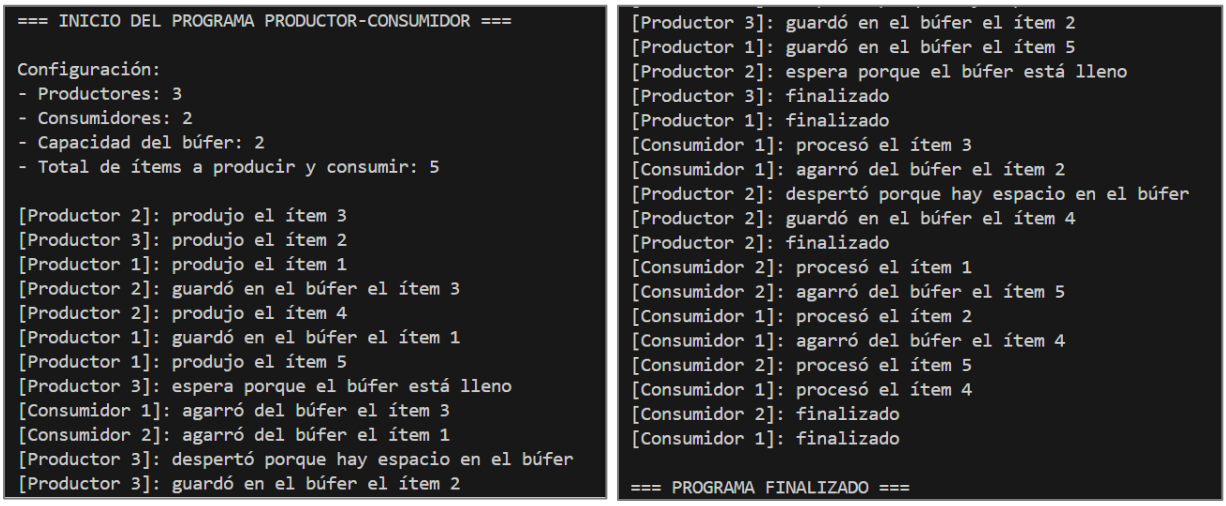

Figura 11. Ejemplo de la salida del programa (parte 4).

Condiciones de entrega

- **Debe entregar en un .zip las carpetas solicitadas en cada parte del laboratorio junto con los comentarios que respondan a las preguntas propuestas.**
- Política de entrega tarde. Para las entregas tarde, se aplicará la siguiente política: por cada 30 minutos, o fracción, de retraso con respecto a la hora de entrega establecida en Bloque Neón habrá una penalización de 0,5/5.
- La calificación de este laboratorio está condicionado a la asistencia, en caso de no asistir a clase o no firmar la hoja de asistencia **su nota será 0**.