RA2. Desarrolla aplicaciones compuestas por varios hilos de ejecución analizando y aplicando librerías específicas del lenguaje de programación. a) Se han identificado situaciones en las que resulte útil la utilización de varios hilos en un programa.

b) Se han reconocido los mecanismos para crear, iniciar y finalizar hilos.

c) Se han programado aplicaciones que implementen varios hilos.

d) Se han identificado los posibles estados de ejecución de un hilo y programado aplicaciones que los gestionen.

e) Se han utilizado mecanismos para compartir información entre varios hilos de un mismo proceso.

f) Se han desarrollado programas formados por varios hilos sincronizados mediante técnicas específicas.

g) Se ha establecido y controlado la prioridad de cada uno de los hilos de ejecución.

h) Se han depurado y documentado los programas desarrollados.

i) Se ha realizado control de versiones en el código generado.

j) Se ha analizado el contexto de ejecución de los hilos.

k) Se han analizado librerías específicas del lenguaje de programación que permiten la programación multihilo.

l) Se han reconocido los problemas derivados de la compartición de información entre los hilos de un mismo proceso.

m) Se ha utilizado una metodología para la planificación, asignación y desarrollo de las funcionalidades generadas.

1. Crea un repositorio en github, llámalo práctica2.2-psp, en este repositorio debes subir los siguientes ejercicios y enviar el enlace del repositorio.

   https://github.com/ivan-sb1/Practica2.2-psp

2. En el siguiente enlace puedes encontrar un programa que usa el concepto syncronized de la programación multihilo. Analiza el programa, ejecútalo y explica lo que hace.
   https://drive.google.com/file/d/1pIWv63lyiliSqQhes8Z1Y_jM6RWBuX64/view?usp=sharing

   1. Clase Turnos

   Esta clase gestiona la sincronización entre hilos, asegurando que se ejecuten en un orden específico.

- **Atributo**:
  - turno: Entero que indica el turno actual (0 para A, 1 para B, 2 para C).
- **Métodos**:
  - esperarTurno(int miTurno):
    - Bloquea el hilo hasta que sea su turno.
    - Usa wait() para esperar de manera eficiente.
  - siguiente():
    - Avanza al siguiente turno de forma circular (0→1→2→0...).
    - Notifica a todos los hilos en espera con notifyAll().

   2. Clase HiloImprimePorTurno

   Extiende Thread para crear hilos que imprimen números en un orden específico.

- **Atributos**:
  - turno: Referencia al objeto compartido Turnos.
  - miTurno: Identificador del turno (0, 1 o 2).
  - inicio: Número inicial para la secuencia.
  - paso: Incremento entre números (siempre 3).
- **Método run()**:
  - Bucle que imprime números del 1 al 10, pero cada hilo maneja una secuencia específica:
    - Hilo A: 1, 4, 7, 10
    - Hilo B: 2, 5, 8
    - Hilo C: 3, 6, 9
  - Usa esperarTurno() para sincronización.
  - Imprime el número y pasa al siguiente turno.
- **Método main**:
  - Crea una instancia compartida de Turnos.

- Inicializa tres hilos (A, B, C) con sus respectivos turnos y secuencias.
- Inicia los hilos y espera a que terminen con join().

3. En los apuntes, en la página 38 tienes el ejemplo de un programa productor-consumidor. Analízalo, desarróllalo y ejecútalo, una vez realizados estos pasos explica su funcionamiento.
   https://aulasciclos2526.castillalamancha.es/pluginfile.php/806684/mod_resource/content/1/PSPUT2%20-%2002%20Programacio%CC%81n%20Sincronizada%20Multihilo.pdf

   1. Clase ProductorConsumidor

   Es la clase principal que inicia los hilos productor y consumidor.

- Crea una instancia de Buffer compartida.
- Inicia dos hilos:
  - **Hilo Productor**: Produce 20 elementos, con una pausa de 500ms entre cada producción.
  - **Hilo Consumidor**: Consume 20 elementos, con una pausa de 1000ms entre cada consumo.

   2. Clase Buffer

   Gestiona el almacenamiento compartido entre productor y consumidor.

   Atributos:

- capacity: Capacidad máxima del búfer (10 elementos).
- count: Contador de elementos actuales en el búfer.

   Métodos sincronizados:

1. **producir()**:
   - Si el búfer está lleno, el hilo productor espera.
   - Añade un elemento al búfer.
   - Notifica a los hilos en espera.
   - Muestra el estado actual del búfer.
2. **consumir()**:
   - Si el búfer está vacío, el hilo consumidor espera.
   - Elimina un elemento del búfer.
   - Notifica a los hilos en espera.
   - Muestra el estado actual del búfer.

   Sincronización:

- Los métodos están marcados como synchronized para garantizar la exclusión mutua.
- Se usa wait() para que los hilos esperen cuando no pueden realizar su operación.

- Se usa notifyAll() para despertar a los hilos en espera cuando hay cambios en el estado del búfer.

4. Simula un sistema de control de acceso a una sala de conferencias en un edificio de oficinas.
   En este caso, habrá varios empleados que desean usar una sala de conferencias compartida.
   Los empleados deben esperar si la sala está ocupada o si hay demasiados empleados dentro de la sala, ya que la sala tiene un límite de ocupación (por ejemplo, 5 personas como máximo)

   Escenario:
   Sala de conferencias compartida: Tiene una capacidad máxima de 5 personas. Si ya hay 5 personas en la sala, el siguiente empleado debe esperar fuera.
   Empleados: Son hilos que intentan entrar en la sala. Si la sala está llena, deben esperar hasta que alguien salga. Además, solo puede haber un número máximo de empleados dentro al mismo tiempo.
   Salida: Cuando un empleado sale de la sala, debe notificar a los demás empleados esperando fuera para que puedan entrar si hay espacio.

   Reglas:
   La sala tiene una capacidad máxima de 5 personas.
   Hay 30 empleados. Si llegan y ven que la sala está llena deben esperar fuera, tienen 3 intentos, deja 12 segundos entre intentos.
   Un empleado solo puede entrar cuando hay espacio disponible (cuando otro haya salido).
   Los empleados están 10 segundos en la sala y tardan 1,5 en salir.
   Cuando un empleado sale de la sala, debe notificar a los demás empleados para que puedan entrar.

5. Simulación Cuenta Bancaria
   Crea una simulación de una cuenta bancaria compartida por 5 clientes. Cada cliente debe poder ingresar dinero, sacar dinero y ver su saldo. Sin embargo, el banco tiene una limitación, sólo 2 clientes pueden acceder a la cuenta simultáneamente.

   Reglas:

Define las clases necesarias con sus métodos.
Debes implementar la clase Thread para el manejo de los hilos.
Cada hilo representará un cliente.
Asegúrate de evitar condiciones de carrera.
El cliente aleatoriamente realizará una operación de ingresar o de sacar dinero.
El programa debe terminar cuando todos los clientes hayan realizado 3 operaciones del tipo ingresar o sacar dinero.
Cada operación debe tardar un tiempo aleatorio entre 1 y 3 segundos.
En las operaciones de ingresar y sacar dinero, la cantidad debe ser aleatoria entre 10 y 1000 euros.
Al finalizar cada operación, el cliente debe consultar el saldo de la cuenta.
Comprueba que, en las operaciones de retirada de dinero, haya suficiente saldo. En caso contrario, muestra un mensaje.
Imprime por consola todos los movimientos para poder controlar que se está haciendo de manera correcta.