Examen1: Desarrollar una aplicación en lenguaje C (o Java, según tu asignatura) que cumpla los siguientes requisitos:

1. Un proceso principal ("padre") lanza *N* procesos hijos de manera paralela (por ejemplo N = 4 o 5).
2. Cada proceso hijo realiza una tarea intensiva (por ejemplo: calcular la suma de los números entre dos límites, procesar un fichero de datos, generar números aleatorios y filtrar según condición).
3. Los procesos hijos deben **comunicarse** con el proceso padre mediante un mecanismo de IPC: puede ser tuberías (pipes), colas de mensajes, memoria compartida, etc.
4. Cada hijo envía su resultado parcial al padre. El padre, una vez que todos los hijos han terminado, recopila los resultados, los integra y produce una salida final (por ejemplo: "La suma total es …", "Se han procesado X registros", …).
5. Asegurar que los procesos se ejecutan en paralelo (no secuencialmente) y que la aplicación aprovecha concurrencia/multiprocesamiento.
6. Opcional: añadir sincronización (ej.: los hijos deben esperar una señal del padre para comenzar; el padre espera a que todos terminen; etc.).
7. Documentar la solución incluyendo diagrama de procesos, mecanismo de comunicación elegido, y análisis de paralelismo.

```
public class Hijo {

   public static void main(String[] args) {

      int inicio = Integer.parseInt(args[0]);

      int fin = Integer.parseInt(args[1]);

      int suma = 0;

      for (int i = inicio; i <= fin; i++) {

         suma += i;

      }

      System.out.println(suma); // enviar resultado al padre

   }

}
```

```
import java.io.*;


public class Padre {
   public static void main(String[] args) throws IOException, InterruptedException {
      int N = 4;
      int total = 100;
      int rango = total / N;
      int sumaTotal = 0;


      Process[] procesos = new Process[N];
      BufferedReader[] lectores = new BufferedReader[N];


      for (int i = 0; i < N; i++) {
         int inicio = i * rango + 1;
         int fin = (i == N - 1) ? total : (i + 1) * rango;
         procesos[i] = new ProcessBuilder("java", "Hijo", String.valueOf(inicio),
String.valueOf(fin))
                        .start();
         lectores[i] = new BufferedReader(new
InputStreamReader(procesos[i].getInputStream()));
      }


      for (int i = 0; i < N; i++) {
         String line = lectores[i].readLine();
         sumaTotal += Integer.parseInt(line);
```

```
            procesos[i].waitFor(); // esperar a que termine
        }

        System.out.println("La suma total es: " + sumaTotal);
    }
}
```

Examen2:

Crear un proyecto Java o Python denominado: PSP-DAM-ACTEVA01 . Contexto: Un restaurante desea simular el proceso de gestión de pedidos en línea . El flujo de trabajo es el siguiente: 1. El Cliente genera pedidos (ejemplo: "Pedido-1", "Pedido-2"). 2. El Cocinero toma los pedidos y los prepara. 3. El Repartidor recoge los pedidos listos y los entrega. 4. Un Gestor coordina toda la simulación. Cada actor (Cliente, Cocinero, Repartidor) debe ser un proceso independiente . Clases/Procesos a implementar : 1. Cliente : genera pedidos y los coloca en la cola de pedidos. 2. Cocinero : toma pedidos de la cola, los procesa (prepara) y los pasa a la cola de pedidos listos. 3. Repartidor : toma los pedidos listos de la cola y los entrega. 4. GestorRestaurante : coordina la ejecución y finalización de todos los procesos.Condiciones del ejercicio : - Deben implementarse mínimo 3-4 clases (una por cada rol). - La comunicación entre procesos debe realizarse exclusivamente mediante colas (multiprocessing.Queue en Python o BlockingQueue en Java). - El Cliente debe generar varios pedidos. - El Cocinero debe simular el tiempo de preparación. - El Repartidor debe simular el tiempo de entrega. - Al finalizar, el Cliente enviará una señal de fin para cerrar todos los procesos correctamente.

```
import java.util.ArrayList;

import java.util.List;


public class PSPDAMACTEVA01 {


    public static void main(String[] args) {

        Object lock = new Object();
```

```
        List<String> pedidos = new ArrayList<>();

        Thread cliente = new Thread(new Cliente(pedidos, lock));
        Thread cocinero = new Thread(new Cocinero(pedidos, lock));
        Thread repartidor = new Thread(new Repartidor(pedidos, lock));

        cliente.start();
        cocinero.start();
        repartidor.start();

        try {
            cliente.join();
            cocinero.join();
            repartidor.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        System.out.println(" ✅ Restaurante cerrado");
    }
}

// ---------------------- Cliente ----------------------
class Cliente implements Runnable {
    private final List<String> pedidos;
    private final Object lock;

    public Cliente(List<String> pedidos, Object lock) {
```

```
        this.pedidos = pedidos;
        this.lock = lock;
    }

    @Override
    public void run() {
        try {
            for (int i = 1; i <= 5; i++) {
                synchronized (lock) {
                    String pedido = "Pedido-" + i;
                    pedidos.add(pedido);
                    System.out.println("[Cliente] Genera " + pedido);
                    lock.notifyAll();
                }
                Thread.sleep(500);
            }

            // Señal de fin
            synchronized (lock) {
                pedidos.add("FIN");
                lock.notifyAll();
            }

        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

```
// ---------------------- Cocinero ----------------------
class Cocinero implements Runnable {
    private final List<String> pedidos;
    private final Object lock;

    public Cocinero(List<String> pedidos, Object lock) {
        this.pedidos = pedidos;
        this.lock = lock;
    }

    @Override
    public void run() {
        try {
            while (true) {
                String pedido;
                synchronized (lock) {
                    while (pedidos.isEmpty()) {
                        lock.wait();
                    }
                    pedido = pedidos.remove(0);
                }

                if (pedido.equals("FIN")) {
                    synchronized (lock) {
                        pedidos.add("FIN");
                        lock.notifyAll();
                    }
```

```
            break;
        }

        System.out.println("[Cocinero] Preparando " + pedido + "...");
        Thread.sleep(1000);
        synchronized (lock) {
          pedidos.add(pedido + " listo");
          System.out.println("[Cocinero] " + pedido + " listo");
          lock.notifyAll();
        }
      }
    } catch (InterruptedException e) {
      e.printStackTrace();
    }
  }
}

// ---------------------- Repartidor ----------------------
class Repartidor implements Runnable {
  private final List<String> pedidos;
  private final Object lock;

  public Repartidor(List<String> pedidos, Object lock) {
    this.pedidos = pedidos;
    this.lock = lock;
  }

  @Override
```

```
public void run() {
   try {
      while (true) {
         String pedido;
         synchronized (lock) {
            while (pedidos.isEmpty()) {
               lock.wait();
            }
            pedido = pedidos.remove(0);
         }

         if (pedido.equals("FIN")) {
            break;
         }

         if (pedido.contains("listo")) {
            System.out.println("[Repartidor] Entregando " + pedido.replace(" listo", ""));
            Thread.sleep(1000);
         } else {
            // Si no está listo aún, lo reinsertamos
            synchronized (lock) {
               pedidos.add(pedido);
               lock.wait(200);
               lock.notifyAll();
            }
         }
      }
   } catch (InterruptedException e) {
```

```
            e.printStackTrace();
        }
    }
}
```