PSP - U4 Exámenes

Descargar estos apuntes

Índice

Ejercicio: Sistema de Reservas de Hoteles

Desarrolla una aplicación **cliente/servidor TCP multihilo sin estados** que simule un sistema de reservas de hoteles en tiempo real, como los usados en webs de viajes. El servidor gestionará habitaciones disponibles de forma compartida entre múltiples clientes.

La aplicación debe permitir a los clientes listar habitaciones disponibles, consultar la disponibilidad de un tipo de habitación concreta, reservar habitaciones y finalizar sesión mostrando un resumen de sus reservas.

Instrucciones Generales

- **Servidor**: Crea un `ServerSocket` en puerto 4444. Usa hilos `Worker` para cada cliente. Gestiona una estructura `ConcurrentHashMap<String, Double>` para precios de habitaciones por tipo (ej: "individual"=50.0, "doble"=80.0). Usa `ConcurrentHashMap<String, Integer>` para stock de habitaciones disponibles por tipo (ej: "individual"=5). Ambas estructuras son compartidas y thread-safe.
- **Cliente**: Envía comandos al servidor y muestra respuestas formateadas.
- **Protocolo**: Cualquier comando en cualquier orden. Mensajes terminan en salto de línea. Usa `PrintWriter` y `BufferedReader`.
- **Datos iniciales**: 3 individuales (50€), 4 dobles (80€), 2 suites (150€).[1][2]

Comandos del Cliente (4 comandos)

Comando	Formato	Descripción
LISTAR	`LISTAR`	Lista tipos disponibles con stock y precio.
STOCK tipo	`STOCK individual`	Muestra stock actual de un tipo (ej: "individual").
RESERVAR tipo cantidad	`RESERVAR doble 2`	Reduce stock si disponible; sino error.
SALIR nombre	`SALIR juan`	Lista reservas del usuario (usa Map adicional si necesitas historial) y cierra conexión.

Respuestas del Servidor

- **LISTAR**: "individual:5:50.0doble:4:80.0suite:2:150.0" (concatenado, sin espacios extra).
- **STOCK tipo**: "OK:3" o "KO:Tipo no existe".
- **RESERVAR tipo cantidad**: "OK:stock_restante" o "KO:No disponible".
- **SALIR nombre**: "Reservas: doble:2 total:160.0" (ejemplo; usa historial por usuario en otra Map<String, Map<String,Integer>> sincronizada).

Salida del Servidor (consola)

Muestra todos los mensajes recibidos/enviados con formato:

```
-- CLIENTE IP:PUERTO comando args
Reservas actualizadas: individual=4, doble=3...
Error: Tipo 'triple' no existe.
Cliente desconectado.
```

Ejemplo completo:

```
Escuchando en puerto 4444
-- CLIENTE /127.0.0.1:54321 LISTAR
-- CLIENTE /127.0.0.1:54321 RESERVAR individual 1
Reservas actualizadas: individual=2, doble=4, suite=2
-- CLIENTE /192.168.1.100:12345 STOCK doble
-- CLIENTE /192.168.1.100:12345 SALIR maria
Cliente desconectado.
```

Salida del Cliente (ejemplo)

```
Introduce comando (LISTAR/STOCK tipo/RESERVAR tipo cant/SALIR nombre): LISTAR
Disponibles: individual:3:50.0 doble:4:80.0 suite:2:150.0
Introduce comando: RESERVAR individual 1
OK, quedan 2 individuales.
Introduce comando: SALIR pepe
Tus reservas: individual:1 total:50.0
```

Notas Técnicas

- Sincroniza accesos a Maps con `synchronized` si no usas ConcurrentHashMap.
- Gestiona excepciones: muestra errores y continúa.
- Prueba con varios clientes simultáneos para verificar concurrencia.[2][3][^1]

Fábrica de bicicletas

Crea una aplicación para simular el funcionamiento de una fábrica de bicicletas.

En la fábrica de bicicletas BalmisBikes tenemos tres tipos de trabajadores que deben coordinarse para cumplir con las siguientes tareas:

- Operarios de cuadros: Producen cuadros de bicicleta. Hay **3 operarios** de cuadros.
- Operarios de ruedas: Producen pares de ruedas. Hay **2 operarios** de ruedas.
- Montadores: Montan bicicletas usando 1 cuadro y 2 ruedas (un par). Hay **4 montadores**.

La producción sigue este proceso:

- Los operarios pueden producir un cuadro o un par de ruedas y dejarlos en su almacén, pero si el almacén está lleno, deben esperar.
  - El almacén de cuadros tiene una capacidad de **4 unidades** y el de pares de ruedas tiene una capacidad de **10 unidades (5 pares)**.
  - Los operarios de cuadros tardan entre **2h y 4h** en producir un cuadro
  - Los operarios de ruedas tardan entre **1h y 3h** en producir un par de ruedas.
- Los montadores solo pueden armar una bicicleta si hay, al menos, 1 cuadro y 1 par de ruedas disponibles. Si no es posible, deben esperar.
  - Los montadores tardan entre **media hora y 2h** en montar una bicicleta.

Se pide realizar una aplicación en Java, siguiendo el modelo productor-consumidor, que simule el funcionamiento de la fábrica, mostrando por consola los mensajes necesarios para entender qué está pasando en cada momento.

La simulación debe ejecutar un mínimo de **20 montajes** exitosos de bicicletas antes de finalizar.

Información del proceso

La aplicación debe mostrar claramente qué está pasando en cada momento.
Los mensajes deben indicar el estado: producción, espera, montaje y retirada de piezas, con el nombre del hilo responsable.

La salida por consola debe permitir distinguir fácilmente qué trabajador produce qué y cuándo (colores opcionales, nombres obligatorios).
El nombre de los trabajadores se puede establecer como "OperarioCuadros-1", "OperarioRuedas-2", "Montador-3", etc.

Simulación en escala

Se pide simular en escala el proceso de cada trabajador.
Cuando se indica que un trabajador tarda entre X e Y horas en realizar su tarea, puede tardar un tiempo aleatorio (todo el rango posible de tiempo) entre esos valores, no solo horas enteras.
Cada hora real = 100 ms en la simulación.

Gestión de excepciones en el código

Controla y maneja excepciones en tu código (no las escales), mostrando la pila cuando sea necesario.

Parametrización del código

Ten en cuenta que el número de operarios, capacidades de almacén, tiempos de producción y montaje, etc., deben estar parametrizados en constantes al inicio del código para facilitar su modificación.
Es recomendable probar diferentes configuraciones para comprobar el correcto funcionamiento de la aplicación en distintos escenarios.

Opciones de prueba del sistema

Aquí tienes varios conjuntos de valores para los parámetros principales del ejercicio de la fábrica de bicicletas. Estos sets permiten simular y analizar diferentes escenarios de concurrencia, carga y sincronización para poner a prueba la robustez y flexibilidad del código

Escenario	Operarios Cuadros	Operarios Ruedas	Montadores	Capacidad Cuadros	Capacidad Ruedas	Bicicletas a montar
Básico (equilibrado)	2	3	4	5	10	20
Alta presión en montaje	2	3	10	5	10	30
Buffer reducido	2	3	4	2	4	15
Muchos operarios	5	5	4	7	12	25
Cuellos de botella montaje	2	3	8	3	6	20
Montaje más lento	2	3	2	5	10	20
Máxima simultaneidad	4	6	6	10	15	40
Montadores esperando	1	2	6	3	6	20
Producción escasa	1	1	2	2	4	10
Gran escala	8	10	12	25	40	100

Estos conjuntos permiten comprobar:

- Esperas largas de montadores por piezas (más montadores que operarios o buffers pequeños).
- Saturación de almacenes (más operarios que montadores).
- Escenarios equilibrados donde todos trabajan a ritmo similar.
- Parámetros extremos para analizar cuellos de botella y posibles condiciones de carrera o deadlocks.

Rúbrica de corrección

Cada criterio se evalúa individualmente con una nota de 0 a 5, según el nivel alcanzado.

Se recomienda ponderar más los apartados de “Sincronización y concurrencia”, “Implementación de hilos” y “Gestión de buffers”, dada la naturaleza de la actividad.

Para nota máxima, es imprescindible que no haya condiciones de carrera, deadlocks o bloqueos injustificados, y que la actividad sea reproducible modificando los parámetros principales.

Criterio	Ponderación	Excelente (5)	Bien (3)	Mal (1)	Insuficiente (0)
Diseño de Clases y Modularidad	0,1	Todas las clases cumplen su función, bien diseñadas y con responsabilidades claras. Código modular, reutilizable y fácil de mantener. Excepciones bien capturadas, se usa printStackTrace y mensajes claros para depuración.	Estructura adecuada, aunque algo mejorable en separación de responsabilidades. Se gestionan la mayoría de excepciones, pero podrían mejorar los mensajes o detalles.	Se mezclan responsabilidades o hay redundancia, pero la solución funciona. Se atrapan excepciones pero se ignoran o falta detalle en el tratamiento.	Diseño confuso, poco modular o incorrecto. No se gestionan excepciones, el código puede interrumpirse inesperadamente.
Implementación de Hilos	0,2	Todos los hilos cumplen la lógica, gestionados correctamente y lanzados desde el main.	Los hilos se crean y controlan, pero hay leves errores o incoherencias menores.	Al menos un hilo tiene errores significativos o faltan instrucciones clave.	Errores graves, los hilos están mal implementados, faltan o solo hay uno.
Sincronización y Concurrencia	0,3	Perfecto uso de synchronized, wait, notifyAll y/o bloqueos para evitar condiciones de carrera y asegurar acceso seguro a los buffers.	Sincronización generalmente correcta, pero con pequeñas ineficiencias.	Hay algún error o falta de cobertura en condiciones límite, pero sin bloquear la ejecución.	Falta sincronización, hay condiciones de carrera o deadlocks.

Criterio	Ponderación	Excelente (5)	Bien (3)	Mal (1)	Insuficiente (0)
Manejo de Buffers (Objetos compartidos)	0,1	Buffers seguros y parametrizables, gestionan correctamente los desbloqueos y bloqueos.	Los buffers funcionan, pero son rígidos o con algún detalle mejorable.	Falta parametrización o la gestión de bloqueo/desbloqueo no es óptima pero válida.	Gestión incorrecta o ausencia de buffers concurrentes.
Salida y Mensajes por Consola	0,2	Mensajes claros, con hilos identificados (nombres/colores) y representan bien la situación.	Mensajes comprensibles y sincronizados, pequeña confusión puntual.	Los mensajes existen, pero no siempre claros, faltan detalles de identificación.	Mensajes insuficientes, ambiguos o inexistentes.
Parametrización y Flexibilidad	0,1	Todos los parámetros son configurables (nº hilos, capacidades), fácil de probar escenarios.	La mayoría de parámetros son configurables, falta alguno menos relevante.	Apenas parametrización, el código depende de constantes internas no modificables.	Ejercicio rígido, para modificar hay que tocar demasiado el código fuente.