Curso 2019/20

Práctica 2

Programación Concurrente

Guillermo de Azcárate Acosta
Agustín López Gil

Al principio, y para entender más el código que íbamos desarrollando, lo hemos hecho con variables de sincronización con espera activa y simulando los clásicos problemas vistos en las clases de teoría de “Productor-Consumidor” y “Cliente-Servidor” ya que se comunican entre sí mediante peticiones.

La clase **Almazón** es la encargada de crear la jornada de trabajo de los empleados (turno de 8 horas). Además, hemos hecho que, para que funcione el programa, se deba **compilar y ejecutar esta clase** ya que su método *main* y llama a su método *exec*, y este método exec llama al método *exec* de la clase Personal.

La clase principal **Personal** consta de varios métodos y muchas variables.
Dentro de los métodos está el *main* que únicamente sirve para llamar al método que ejecutará los hilos dentro de la clase. Este método de ejecución (*exec*) crea un número de hilos, definido por las diferentes constantes de cada tipo de empleado. Hay un método por cada tipo de empleado más uno para los clientes. Podemos diferenciar cinco tipos de empleado: uno de tipo administrativo, uno cuya función es recoger pedidos, otro que la suya es empaquetarlos, uno que es de limpieza y otro que es el encargado. Cada uno de estos tiene un método asociado.

El cliente es el que realiza los pedidos. Para que vaya cambiando los pedidos que va haciendo, creamos una variable aleatoria que, dependiendo del valor que toma, hace un pedido u otro.

El empleado en el ámbito administrativo se encarga de recibir y atender las peticiones de los pedidos por parte de los clientes. Si se ha realizado correctamente el pedido, procede a avisar al empleado encargado de recoger pedidos que se puede recoger. Además, una vez que el pedido se ha recogido, se ha empaquetado y se ha enviado, es el encargado de enviar un correo electrónico al cliente avisándole de que el pedido se ha hecho de forma correcta.

El empleado encargado de recoger los pedidos se encarga de guardarlos en un almacén de pedidos. Cuando el pedido está listo para recoger, guarda todos los productos que se han pedido, uno a uno procesa los pedidos y guarda los productos en playas. Después, da el visto bueno para proceder a empaquetarlos. Esto último lo hace siempre salvo que el empaquetador detecte algún error en algún pedido, que es cuando procede a revisarlo y toma la mayor prioridad (*MAX_PRIORITY*). Cuando soluciona esos errores, elimina toda la lista de pedidos con fallos.

El empleado que empaqueta los pedidos empaqueta todos los productos que están en las playas y, si no hay errores, los pone directamente en una cinta de salida. Si los hay, los manda a revisar al empleado anteriormente descrito. Por supuesto, avisa al empleado administrativo para que sepa que ese pedido ya se puede enviar.
También, si ha ocurrido algún incidente no previsto en alguna de las playas disponibles, notifica al empleado encargado de la limpieza de que hay que depurar una playa concreta. Generamos ese 5% de probabilidad de que suceda con un número aleatorio y que el valor comprenda solo cinco de cien valores posibles.

El empleado de limpieza únicamente se encarga de limpiar las playas cada vez que se realizan 20 pedidos.

Y por último, el encargado es el hacedor del almacén. Quien lo abre y quien lo cierra. Para ello, creamos un cerrojo que controla el tiempo que está el almacén abierto. Lleva la cuenta de cuántos procesos hay activos. También, una de sus tareas fundamentales es decidir cuándo termina un empleado su jornada laboral. Cuando la terminan, interrumpe todos los procesos activos y los manda a casa.

En cuanto a las **variables** definidas, tanto herramientas de sincronización de alto o bajo nivel o como variables globales enteras, son las siguientes:

N_EMPLEADOS_ADMINISTRATIVO, N_EMPLEADOS_RECOGEPEDIDOS, N_EMPLEADOS_EMPAQUETAPEDIDOS, N_EMPLEADOS_LIMPIEZA, N_EMPLEADOS_ENCARGADO y **N_CLIENTES** indican el número de empleados que trabajan en el almacén y el número de clientes que hacen pedidos.

La variable **CAPACIDAD_PLAYA** indica la cantidad de productos que admite la playa antes de empaquetarlos, **N_PEDIDOS_PARA_LIMPIAR_PLAYAS** indica el número de pedidos que deben haberse realizado para eliminar la playa por completo, y MAX_PRIORITY indica el valor entero de prioridad que tiene un hilo. Esta última solo se utiliza cuando hay algún fallo en un pedido. Por último, **CONVERSION_TIEMPO** es un valor, definido por nosotros, para que en la ecuación que definimos en la parte del código de la rectificación del tiempo, dé tiempo a que se ejecuten los procesos y se vea cómo se desarrolla el código.

En cuanto a variables enteras, las dos están inicializadas a 0 y son **n_empleados** la cual especifica el número de empleados que están trabajando, y **n_pedidosRealizados** que indica el número de pedidos que se han realizado.

Sobre las listas sincronizadas que hemos definido están:

- List<Pedido> **almacen_pedidos** -> Almacena los pedidos que se van realizando
- List<Pedido> **almacen_pedidos_revisados** -> Guarda los pedidos que han sido revisados
- List<Pedido> **almacen_pedidos_recogidos** ->.Almacena los pedidos que se van recogiendo
- List<Pedido> **almacen_pedidos_erroneos** -> Almacena temporalmente los pedidos con fallos
- List<Pedido> **cinta_de_salida** -> Guarda temporalmente los pedidos después de ser revisados, recogidos y haberles puesto la pegatina de envío.

Las listas de **productosDisponibles**, **productosEnRebajas**, **productosNovedosos** y **productosAgotados** almacenan, cada una, cuatro productos distintos con el objetivo de que el cliente haga diferentes pedidos.

La lista de hilos llamada **lista_hilos** se encarga de guardar los hilos que se crean.

La lista **playa** almacena los pedidos que se recogen antes de empaquetarlos.

En cuanto a los cerrojos, tenemos los siguientes:

- **pedido_revisado** -> Comprueba si se ha revisado el pedido.
- **pedido_recogido** -> Comprueba si se ha recogido el pedido en cuestión.
- **pedido_empaquetado** -> Comprueba si el pedido ha sido empaquetado.
- **limpiar_playas** -> Da el visto bueno para que se efectúe la limpieza de la playa.
- **hacer_pedido** -> Da pie a crear un pedido y posteriormente meterlo al almacén de pedidos. Lo definimos para evitar inconsistencias.
- **enviar_mensaje** -> Actúa de cerrojo para que el empleado administrativo mande el mensaje al cliente.

Y finalmente, hemos declarado varios semáforos cuya función es:

- **sem_pedidos_realizados** -> Controla que se ha realizado el pedido.
- **sem_listo_para_recoger** -> Controla que el pedido esté listo para ser recogido.
- **sem_listo_para_empaquetar** -> Controla que el listo esté preparado para empaquetarse.
- **sem_listo_para_enviar** -> Controla si el pedido está listo para enviarse.
- **sem_limpiar_playas** -> Actúa como comunicador al empleado de limpieza para que proceda a limpiar las playas una vez se ha llegado al número límite de pedidos.
- **sem_empezar_jornada** -> Actúa como semáforo al empezar el horario de la jornada laboral.
- **sem_pedido_erroneo** -> Controla las acciones a realizar si hay algún fallo en algún pedido.
- **sem_limpiar_playaConcreta** -> Controla que se limpie una playa en concreto a pedido del empaquetador de pedidos.

Después, las clases auxiliares **Producto** (que únicamente contiene un String que define el nombre del producto y la clase o categoría a la que pertenece) y **Pedido** (que como variables tiene un identificador del pedido, la dirección del cliente y una lista con los productos que contendrá el pedido).
Hemos decidido no crear la clase auxiliar Playa ya que lo contemplamos como lista de objetos de tipo Producto y para la funcionalidad que se nos pide implementar, consideramos que así es suficiente.

Después, de una semana trabajando con la práctica, desarrollando casi todos los métodos de las diferentes clases, vemos como en el método de lo que realiza el cliente, al realizar un pedido (es decir, añadir un pedido a la lista de pedidos) nos da el error "ArrayIndexOutOfBoundsException" que aparece cuando queremos acceder a una posición inexistente (mayor que la longitud del array) o que el índice es negativo. Para solucionarlo, lo pusimos en un while, ya que queríamos que el cliente estuviese constantemente realizando pedidos y encuadramos eso en un *try catch* para que en la segunda parte pusiese un mensaje de advertencia.
Días más tarde nos apareció un error de "NullPointerException" debido a que había veces que la primera posición y otras veces varias de las primeras posiciones del array de pedidos tenían como valor null. Tras depurarlo varias veces, vimos que quizás el problema podía estar en que no realizaba bien la actualización de valores y la sincronización del array entre los procesos activos. Viendo eso, probamos a pasar las **listas** que teníamos definidas como siempre, a que fuesen **sincronizadas**. Pasamos de la primera declaración a la segunda:
"List <Pedido> almacen_pedidos = new ArrayList<>()"
"List <Pedido> almacen_pedidos = Collections.synchronizedList(new ArrayList<>())"

Hemos incluido el **factor de rectificación de tiempo**, el cual hemos llamado CONVERSION_TIEMPO y que, cuando un empleado ha alcanzado su hora de fin de jornada, ese hilo se va a dormir con la siguiente sentencia:
(24 / CONVERSION_TIEMPO) * 1000);
habiendo definido el valor del factor a 2 para que dé tiempo repetidas veces a que se manden varios pedidos y se vea todo el procedimiento.

El número de empleados, el número de pedidos que debe haber para que se limpien las playas y alguna que otra variable más lo hemos **definido como constantes** para configurar y modificarlos más fácilmente.

En el enunciado se nos dice que si el empaquetador **detecta algún fallo** en algún pedido, inmediatamente ese **hilo** que lo ejecuta se torna de **máxima prioridad** frente a cualquier otro. Para que no resulte demasiado pesado, y bajo nuestra opinión, hemos decidido que solo haya fallos en dos pedidos y lo hemos hecho cuando el identificador del hilo que lo ejecuta valga 30 (por defecto) para que no sea muy repetitivo pero sea fácil de identificar y verificar que así se hace.

Para que acabe la ejecución del programa, simplemente hemos puesto un "**break**" dentro de cada "catch" de cada método. Al principio pensábamos que al interrumpirse un hilo, ese hilo finalizaba su ejecución. Pero al ver que no era así, supimos que el hilo, al interrumpirse, volvía al inicio del bucle. Por ello escribimos esa sentencia para solucionarlo.