ETSII
Escuela Técnica Superior de Ingeniería Informática

Universidad Rey Juan Carlos

INGENIERÍA DE COMPUTADORES

PROGRAMACIÓN CONCURRENTE

PRÁCTICA 2

AUTORES:

Oscar Nydza Nicpoñ

Miguel Santiago Herrero

Índice

Descripción general

Se ha realizado un programa concurrente, que consiste en la implementación de un almacén logístico en el cual existen un clase `Clientes` que realiza pedidos, `Personal` los cuales según el tipo de trabajo que tenga hará una cosa, estos trabajos se dividen en `trabajoAdministrativo()`, `trabajoRecogePedidos()`, `trabajoEmpaquetaPedidos()`, `trabajoLimpieza()`, `trabajoEncargado()`, encontramos tambien la clase `Playa` donde se almacenan los pedidos que se han recogido, `Pedido` que guardara las características del pedido que realizó el `Cliente`, además existe la clase `Paquete` el cual se creará cuando el pedido ya esté recogido del almacén y se ponga el sello para ser enviado y por último, la clase `Almazon` en la cual se encuentra el `main()` así como todos los recursos compartidos (listas bloqueantes, constantes, comunicadores para los hilos, etc) por todos los hilos que también se crean en esta clase.

Desarrollo del programa

Clase Personal

Se comenzó el desarrollo de esta práctica por la clase `Personal`, que se encarga de diferenciar, así como de realizar las distintas tareas que hace cada uno de los tipos de empleados de los que dispone el almacén. Esta distinción se hace en la función `tarea()`, que además controla las excepciones. Estas serán explicadas más adelante.

Método `tarea()`.

En este método se divide el trabajo dependiendo del tipo de trabajador que se le haya asignado, pero lo más importante que maneja esta función son las excepciones. Estas a su vez son lanzadas por el Encargado, y, a excepción de los propios Encargados, pone a cambia el estado del Personal de "trabajando" a "no trabajando". Esto en principio altera su comportamiento, pero no ha sido complicado implementarlo de esta manera.
Sospechamos que puede ser debido a que cuando un hilo se duerme, aunque se le interrumpa no para de dormir, es decir, llama al manejador y vuelve a lo que estaba haciendo.
A continuación se explican las distintas tareas dependiendo del tipo de personal :

`trabajoAdministrativo()`: esta función la realiza la tarea asignada al trabajador Administrativo esta consiste en comprobar si hay pedidos realizados por los clientes en las lista `pedidos` si existe alguno notifica al personal del tipo RecogePedidos, para que este realice sus tareas correspondientes, por el contrario si no encuentra ningún pedido en la lista continúa comprobando hasta que el `Cliente` haga un nuevo pedido. Por otro lado, también comprueba la lista de `pedidosEnviados` en la cual se almacenan las características de los paquetes que ya salieron hacia su destino, el administrativo coge y

elimina el primer elemento que encuentre en la lista y enviará en correo al cliente informando del envío del paquete que cogió de la lista.

`trabajoRecogePedidos()`: esta función la realiza la tarea asignada al trabajador RecogePedidos y consiste en recoger y eliminar los pedidos realizados por los clientes en las lista `pedidos`, a continuación llamará a la función `recogerPedido(Pedido inicial)` la cual se encargará de coger los artículos que se encuentran en el pedido y devolver el pedido realizado. En la recogida se pueden cometer errores al recoger el articulo. Por último pondrá el pedido recogido en una playa aleatoria , y se notificará al personal del tipo EmpaquetaPedidos que ya se ha realizado la recolección de los artículos para que éste los envíe. Además esta función comprueba una lista prioritaria de `pedidosErroneos`, la comprueba antes que la de `pedidos`. Si existe algun pedido en esta lista llamará a la función `tratarPedidoErroneo(Pedido mal)` en la que se realiza una comprobación del pedido y se corrigen los posibles errores cometidos en una recolección anterior en la función `recogerPedido(Pedido inicial)` , sin embargo el que detecta si el pedido esta erroneo o no para añadirlo a la lista `pedidosErroneos` es el personal del tipo Empaqueta Pedidos. Si el cliente no hiciera más pedidos el personal de RecogePedidos se bloqueará hasta que el Administrativo le notifique que en la lista `pedidos` hay nuevos.

`trabajoEmpaquetaPedidos()`: aquí el Empaqueta Pedidos realiza sus tareas. Elige una playa aleatoria en la que trabajar y comprueba si existe algun pedido en ella. Si hay coge el primero de la lista, lo guarda en una variable y lo elimina de dicha playa, en la manipulación del pedido se puede ensuciar una playa. Si esto ocurre llamará al personal de limpieza para que la limpie. Por último comprueba si el pedido tiene todos los productos que el cliente pidió al inicio llamando a la función `comprobarPedido(Pedido p)`, en ella comprueba si el RecogePedidos no cometió ningún error. Si efectivamente si cometió un error se añade el pedido a la lista de `pedidosErroneos` y no se envía hasta que se arregle el error, pero si el envío está correcto se le pone el sello, se envía y se añade el pedido a la lista de `pedidosEnviados` para que el Administrativo notifique al cliente que su pedido se envió.
Por último si no tiene más pedidos que tratar se bloqueará hasta que Recoge Pedidos les vuelva a despertar poniendo pedidos en alguna playa.

`trabajoLimpieza():` esta función la realizará el Personal de Limpieza, exclusivamente se ejecutará cuando un Empaqueta Pedidos detecte una playa sucia o cuando se lleve un número determinado de pedidos (en nuestro caso cada 10 pedidos). una vez que el proceso es llamado este llamará a una función `limpiarPlaya()` la cual se encarga de limpiar o todas las playas porque han pasado esos 10 pedidos o una playa en concreto estaba sucia. Una vez terminada la limpieza, notifica a todos los EmpaquetaPedidos que están esperando. Volverá a bloquearse hasta que reciba otra llamada para limpiar.

`trabajoEncargado():` en esta función se ejecuta el código del Encargado cuyo cometido principal es hacer cada cierto tiempo (en nuestro caso 10 veces por cada turno de trabajo) un informe incluyendo el estado de las playas, si están en uso o no y el número de elementos que contienen, el número total los paquetes enviados, los pedidos realizados por los clientes que aún no se han atendido y los pedidos erróneos aún no corregidos. Para esto se llama a la función `comprobarAlmazon()`. Además informa del estado de todo el personal, si está esperando o está en ejecución. Esto se consigue haciendo un conjunto de todos los hilos en ejecución y contrastando con la lista de trabajadores que tenemos, consultamos sus atributos, como el id y el estado del propio hilo.

Clase Cliente

Esta clase simula el comportamiento de un cliente típico. Su ciclo de vida consiste en hacer compras cada X tiempo aleatorio (entre 0 y `MAX_HORAS_ENTRE_PEDIDOS`). También implementa una simulación rudimentaria del tiempo de descanso. Este tiempo correspondería al tiempo en el que un cliente duerme y es calculado aleatoriamente (1 entre 3 posibilidades de dormir). El cliente se dormiría 8 horas, que corresponde a ⅓ del día.
Sobra decir que mientras duerme no hace pedidos.

Clases Pedido y Paquete

La clase Pedido guarda el id del pedido, si está pagado o no, la lista de productos que se quiere comprar y la nota original, para poder comparar en caso de error a la hora de recoger los pedidos.
La clase Paquete es una clase simple que guarda la lista de productos y un booleano que simboliza si tiene el sello puesto o no.

Clase Playa

Se trata de otra clase simple que guarda una lista de los pedidos y si está sucia o no. Además, proporciona métodos para insertar, eliminar y devolver pedidos (add, poll, peek respectivamente).

Clase Almazon

Esta clase se encarga principalmente de inicializar las estructuras de datos y de poner a los hilos a ejecutar, asignar turnos, etc. Aquí se guardan toda la información sobre el almacén que pueden necesitar los miembros del personal. Contiene todas las listas, los locks que usamos como canales de comunicación, las constantes que usa el programa para el cálculo de distintos parámetros y algunos enteros que llevan la cuenta de los pedidos que entran y salen.

Sincronización entre hilos (exclusiones mutuas)

Para acceder a diferentes puntos críticos de la memoria de manera que las funciones de escritura y lectura se hagan de forma óptima se han implementado una serie de Lock (ReentrantLock para ser más concretos) o cerrojos que nos permiten restringir los accesos a las variables compartidas sin que haya conflicto entre hilos. En concreto, se usa para el acceso al array de playas, a la lista de pedidos, la lista de pedidos erróneos y la lista de pedidos enviados. A continuación se explica más detalladamente el porqué fue necesario:

`mutexPlayas:`

Este mutex o cerrojo se ha implementado debido a que en momento en que el Empaqueta Pedidos quiere coger y eliminar un pedido de la playa este puede ser interrumpido a mitad de la acción y otro personal de su mismo tipo puede entrar en esta sección crítica y hacer una comprobación de manera equivocada y "quitar al primer Empaqueta Pedidos el pedido que iba a coger. Ejemplo, solo hay un pedido en la playa y dos procesos (de tipo Empaqueta Pedidos) comprueban que aún existe al menos un pedido en la playa ambos intentan cogerlo y eliminarlo de la playa y uno de los dos (el último) no encuentra nada y salta una excepción.Para ello se crea el `mutexPlayas` para que esto no ocurra y solo un proceso la vez pueda comprobar si hay o no pedidos en la playa y coger uno de la misma.

`mutexPedidos:`

Este otro cerrojo se ha creado para evitar que dos o más recogepedidos intenten coger un único pedido o un número menor a los procesos que quieren coger un elemento de la lista, y que un proceso o varios al haber sido interrumpidos antes de hacerse con el valor de dicha lista salte la excepción una vez se vuelva a ejecutar porque la lista ya está vacía. De nuevo se crea este mutex para que uno a uno los procesos comprueben y cogan cada elemento de la lista sin interrupciones de otros hilos.

`mutexPedidosEnviados:`

Este mutex está implementado para evitar lo mismo que evita `mutexPedidos` pero en este caso en la lista de `pedidosEnviados`. Asi obliga a cada proceso del tipo Recoge Pedidos comprobar y coger de dicha lista uno a uno los valores sin las interrupciones de diferentes procesos.

`mutexPedidosErroneos:`

Este mutex fue creado para hacer la misma función que `mutexPedidosEnviados` pero en este caso para la lectura, comprobación y edición de la lista `pedidosErroneos`.

`mutexNotificacionLimpieza:`
Su función es la de obligar a que solo un Empaqueta pedidos a la vez realice la notificación al personal de limpieza y además que nadie,debido a una interrupción, modifique la variable limpiar ya que la necesita el personal de limpieza para saber si ha de `limpiar` (compartida entre todos los hilos) o no.

`mutexLimpia:`
Por último este mutex se implementó para que mientras se está limpiando una o todas las playas nadie interrumpa a la limpieza ya que si esta fuera interrumpida otros procesos pueden cambiar el valor de la variable compartida `playaALimpiar` que contiene el número de la playa que tiene que limpiar, corriendo el riesgo de que una playa se quede sucia y bloquee a todos los hilos.

Estructuras de Datos

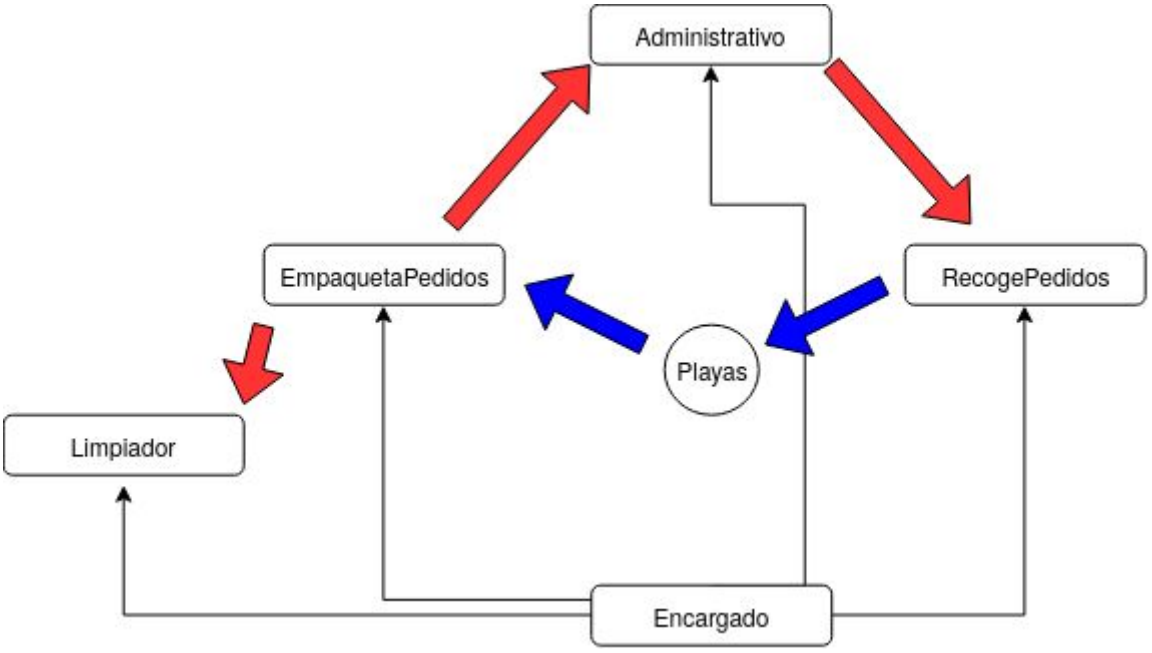
En el enunciado se especificaban algunas estructuras de datos que se debían usar. Estas han sido implementadas de la siguiente manera:

- Lista de pedidos: esta lista ha sido implementada mediante una `LinkedBlockingQueue<Pedido>`, principalmente porque al ser una lista enlazada nos permite que sea "infinita" (no se especificaba ningún límite de pedidos). Además, nos proporciona operaciones atómicas que ayudan a la hora de hacer el programa concurrente.
- Playas: se trata de un array clásico que guarda objetos `Playa`. Estos, a su vez, tienen un `ArrayBlockingQueue` que guarda los pedidos ya tratados por los RecogePedidos. Se usa un array clásico porque se debe de poder acceder a varias playas a la vez de manera concurrente sin bloqueos.
- Cinta: de nuevo, se usa una `LinkedBlockingQueue<Pedido>` (por los mismos motivos que se usaba en la lista de pedidos) que guarda todos los pedidos que han sido enviados.
- Lista de pedidos erróneos: se usa otra `LinkedBlockingQueue<Pedido>` por las mismas razones que anteriormente, solo que aqui entran los pedidos que los RecogePedidos recogen mal.

Adicionalmente se ha usado una estructura de datos de apoyo. Esta es `pedidosEnviados`. Una lista que guarda los pedidos que han sido enviados para que el Administrativo pueda comunicar al cliente que ya se ha enviado su pedido.

Comunicación entre hilos

En este apartado se explicarán los distintos mecanismos de comunicación que se han usado. Para ayudar a la visualización del esquema que hemos seguido, referir a la siguiente figura:



Aquí se puede ver que vamos a tener 3 canales de comunicación entre hilos principalmente.
Estos son:

- Entre Administrativo y RecogePedidos: se trata de una llamada bloqueante, es decir, una de las partes se queda bloqueada hasta que la otra le llame. Esto lo conseguimos con el uso de monitores. Cada `Personal` recibe al menos un canal de comunicación (`Object canalComunicacion` y `Object canalComunicacion2`) que debe compartir con el hilo con el que desea comunicarse. Como descubrimos mientras intentábamos establecer la comunicación entre estos hilos, se debe usar la misma instancia del objeto. De esta manera, Un Administrativo puede notificar a un solo RecogePedidos mediante la función `notify()` sobre su `canalComunicacion`. Esto despertara solamente a uno de los RecogePedidos, que continuará con su trabajo hasta que se bloquee esperando a que le llamen de nuevo.
- La comunicación entre EmpaquetaPedidos y RecogePedidos es indirecta, es decir, no hablan directamente entre ellos. Uno coloca pedidos en una Playa y el otro cuando ve que hay algo, lo recoge.
- Entre EmpaquetaPedidos y Administrativo se establece una comunicación similar a la anterior. Uno coloca en la lista y el otro ve si hay algún elemento y actúa en consecuencia (notificando al cliente).

- EmpaquetaPedidos también se comunica con los Limpiadores de manera similar a cómo los Administrativos se comunican con los RecogePedidos, sin embargo, existen diferencias, principalmente la existencia de 2 canales de comunicación. Uno de ellos se utiliza para notificar a los Limpiadores y otro para notificar a los EmpaquetaPedidos. Esto se ha hecho así debido a que si se usaba solamente un canal de comunicación, al haber más de un Limpiador, se despertaban entre sí y entraban en un bucle que, aunque no infinito, resultaba un problema.
- Por último, los Encargados se comunican con todos los demás trabajadores mediante interrupciones. Estas cambiarían el estado del trabajador de "trabajando" a "descansando" para simular los ciclos de turnos que existirían en el almacén.

Se planteó también una comunicación mediante Exchanger en vez de usar monitores, pero la idea fue descartada debido a que un hilo se bloquea hasta que el otro le "notifique" y solo se puede hacer entre 2 hilos.

Ejecución y simulación

Para ejecutar este programa correctamente se deben retocar una serie de parámetros que encontramos en las distintas clases.
Empezando por la clase `Almazon`, que contiene la mayoría de estos parámetros necesarios en el programa. Estos son:

- `nSegundosSon24HorasReales`. Es poco intuitivo, pero se le pasan ms a esta función. Por ello, solo se debe modificar el valor anterior a la multiplicación (XX en XX * YY). Este valor ajusta cuantos segundos va a durar la simulación.
- `NUM_TURNOS`: Ajusta el número de turnos.
- `NUM_PLAYAS`: Ajusta el número de playas.

Los siguientes parámetros ajustan el número de clientes y el número de trabajadores de cada tipo **por turno**.

En la clase `Cliente` podemos encontrar el número de productos que pueden ver del almacén, el tiempo máximo en horas que puede pasar entre pedidos si no se duerme 8 horas y la posibilidad de que el pedido no haya sido pagado.
Este último parámetro no está en uso debido a que causaba problemas en la ejecución.

La clase `Personal` contiene 2 constantes configurables, siendo la posibilidad de error a la hora de recoger el pedido y la posibilidad de que el pedido se rompa en la playa y la ensucie.

La clase `Playa` contiene como dato configurable el tamaño máximo de la misma.

Decisiones para la simulación

Para la correcta visualización de la simulación, se decidió añadir colores a los distintos tipos de trabajadores. Estos son:

- Morado para el Administrativo.
- Verde para el RecogePedidos.
- Azul para el EmpaquetaPedidos.
- Amarillo para los Limpiadores.
- Rojo para los Encargados.

Se tomó esta decisión debido a que resultaba muy complicado seguir lo que estaba pasando cuando el programa funcionaba más velozmente. De esta manera ha sido posible encontrar errores de bloqueo y solucionarlos más fácilmente.

Se ha decidido también la eliminación de los cambios de turno ya que no hemos sido capaces de hacerlos. Nuestro planteamiento usando interrupciones no ha sido el correcto. Otras ideas que hemos tenido pero que no hemos tenido tiempo de desarrollar han podido ser el uso de monitores de nuevo o sincronización en barrera.

Hemos decidido pausar el programa 3 segundos antes de comenzar para poder ver quienes van a trabajar.

También, debido a que inicialmente se pretendía hacer 2 turnos, se crean el doble de hilos trabajadores de los que se especifican en los parámetros. Unos duermen y otros trabajan. Sin embargo, ya que no se hacen cambios de turno, solo trabajan los que se especifican en los parámetros.

Antes de ejecutar

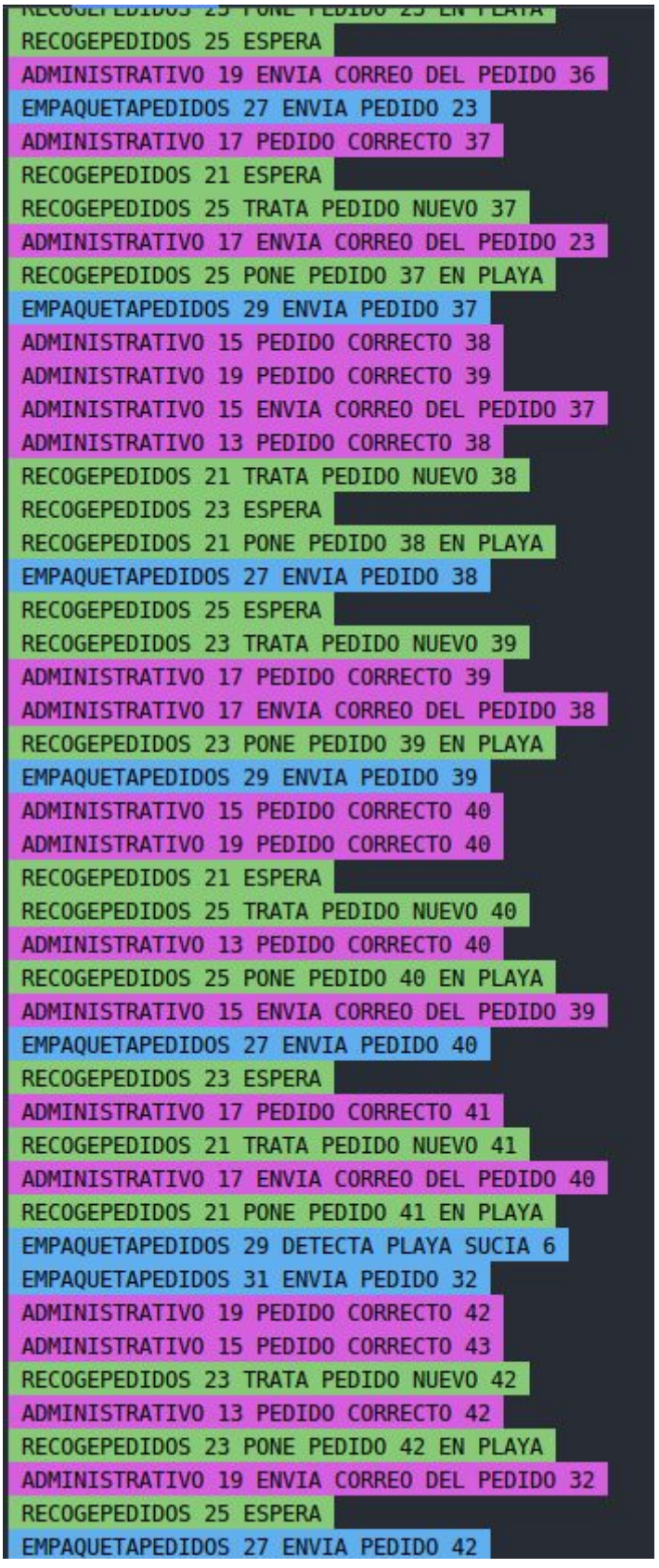
El programa ha sido probado en entornos UNIX (tanto Linux como MacOS). No hemos probado el programa en Windows. Es posible que de problemas a la hora de dormir hilos debido a lo que se puede ver aquí.

“The granularity of sleeps is generally bound by the thread scheduler's interrupt period. In Linux, this interrupt period is generally 1ms in recent kernels. In Windows, the scheduler's interrupt period is normally around 10 or 15 milliseconds”

Es decir, si los tiempos que se duermen los procesos es menor que 10 o 15 ms puede dar problemas. Como esos tiempos son calculados dependiendo del tiempo que se especifica como parámetro, queda fuera de nuestro alcance.

Ejecución típica

Teniendo todo esto en cuenta, la ejecución típica del programa es la siguiente:



Pruebas realizadas

Para asegurar la estabilidad del programa, se han realizado pruebas de estrés hasta los 30.000 pedidos procesados aproximadamente con resultados satisfactorios. Los posibles errores durante la ejecución (que no se envíen tantos pedidos como se desearía) son debidos a la elección de los parámetros.

Problemas encontrados

Durante el desarrollo nos hemos topado con varios errores que queremos dejar reflejados. Fueron los siguientes:

Problemas con las instancias de los canales de comunicación.

Este problema era debido a que no estábamos creando las instancias de los canales de comunicación de manera correcta. Estábamos creando por ejemplo un Exchanger por cada instancia de la clase Personal y tratábamos de que se comunicaran entre sí. Preguntamos en StackOverflow y la solución fue instanciar los canales de comunicación fuera de la clase Personal y pasarlos como argumentos a los empleados que queríamos que se comunicaran entre sí.

Problemas con la conversión de tiempo.

Este problema surgió principalmente debido a que intentamos hacerlo más complejo de lo que debíamos. La solución fue simplificar las conversiones y hacer el número de turnos estático (aunque luego no conseguimos implementarlos correctamente).

Problemas en la ejecución.

Una vez solventados los problemas anteriores, nos dimos cuenta que al ejecutar se producían bloqueos en todos los hilos. Esto fue debido a la sobrecarga de pedidos que llegaban, que hacía que las playas se ensuciaran y nadie notificase a los limpiadores. Un ajuste de los parámetros y el tiempo de simulación fue la solución.

Posibles mejoras.

Principalmente aquí queremos destacar que queda mucho espacio para la mejora. Un par de ellas podrían ser:

- El Encargado imprime en un archivo, no en la terminal. Esto podría simplificar la lectura de los resultados notablemente.
- Implementar los turnos.