MANUAL TÉCNICO

201503777- Francisco Humberto Lezana Ramos
201504385 - Fernando Josué Flores Valdez

Sistemas Operativos 2

Problema del Almacén

Recursos Compartidos

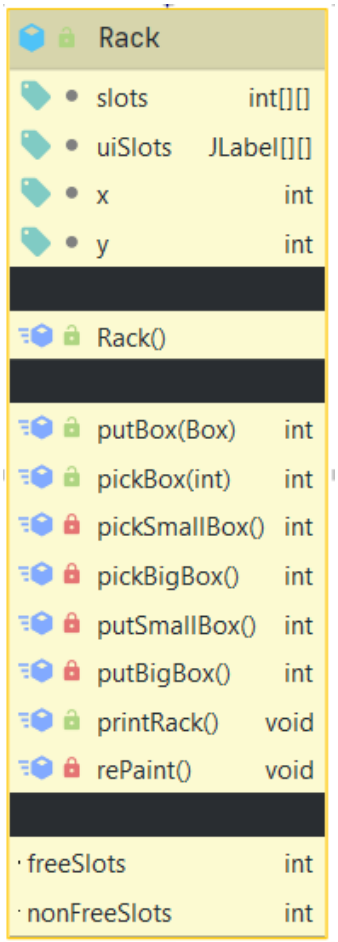
Debido a que existen 2 tipos de cajas, las cuales pueden ser agregadas o removidas del estante, se opta por utilizar una sola matriz de datos, la cual representa los espacios disponibles del estante.

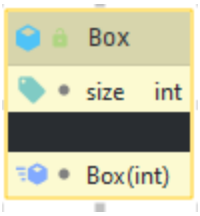
Este recurso es accesado y compartido por 4 hilos.

1. Se agrega una caja pequeña
2. Se agrega una caja grande
3. Se toma una caja pequeña
4. Se toma una caja grande

Estos 4 hilos acceden independientemente al mismo recurso ya que son acciones que pueden darse en un mismo tiempo y que por lo tanto es necesario que se ejecuten con autonomía.

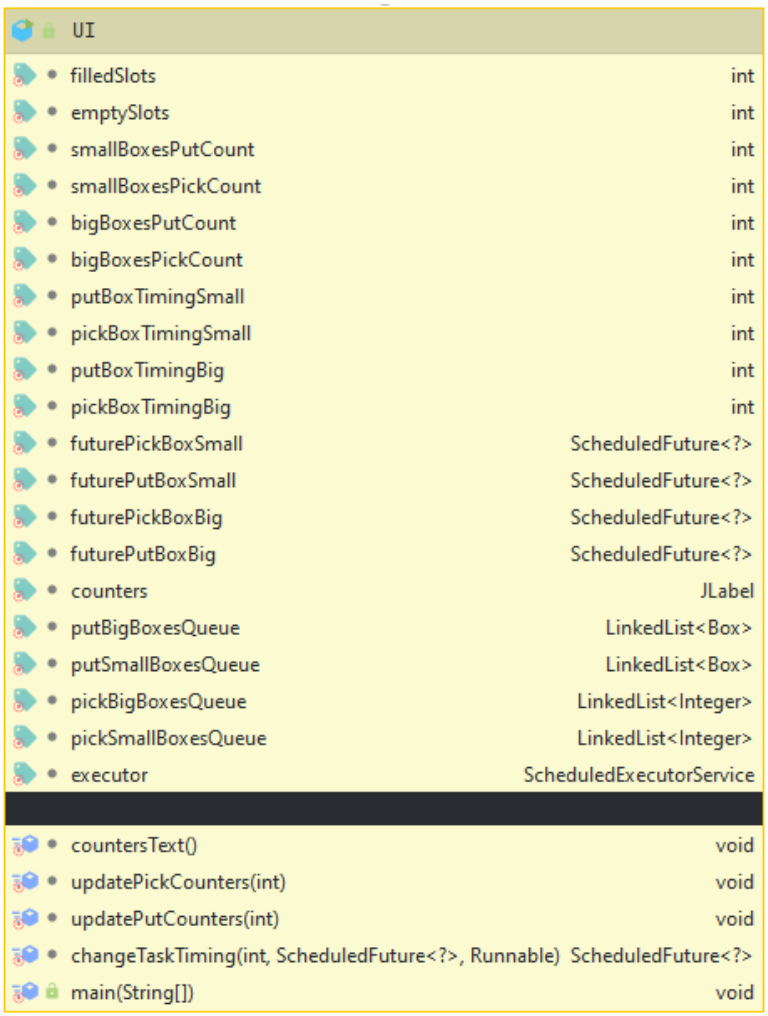
Diagrama UML de clase principal





Concurrencia

Para la concurrencia se hace uso de la clase principal Rack, esta tiene ya sus estructuras y métodos implementados, los cuales afectan a la estructura, por ello lo único que se necesita es ejecutar estos métodos y funciones en un tiempo t determinado, esto lo haremos a través de la clase UI, la cual contiene los métodos y funciones necesarias para ejecutar concurrentemente los eventos que modifican las estructuras de una instancia de tipo Rack.



Para poder hacer uso de hilos que corren al mismo tiempo y que cambian su tiempo de ejecución en tiempo real se recurre a las utilidades de la librería Executors.

```
static ScheduledExecutorService executor = Executors.newScheduledThreadPool(4);
```

Se le indica a una nueva instancia de este tipo, que va a tener a cargo la ejecucion de 4 hilos al mismo tiempo.

```
futurePutBoxSmall = executor.scheduleAtFixedRate(putBoxSmall, 0, putBoxTimingSmall, TimeUnit.SECONDS);
```

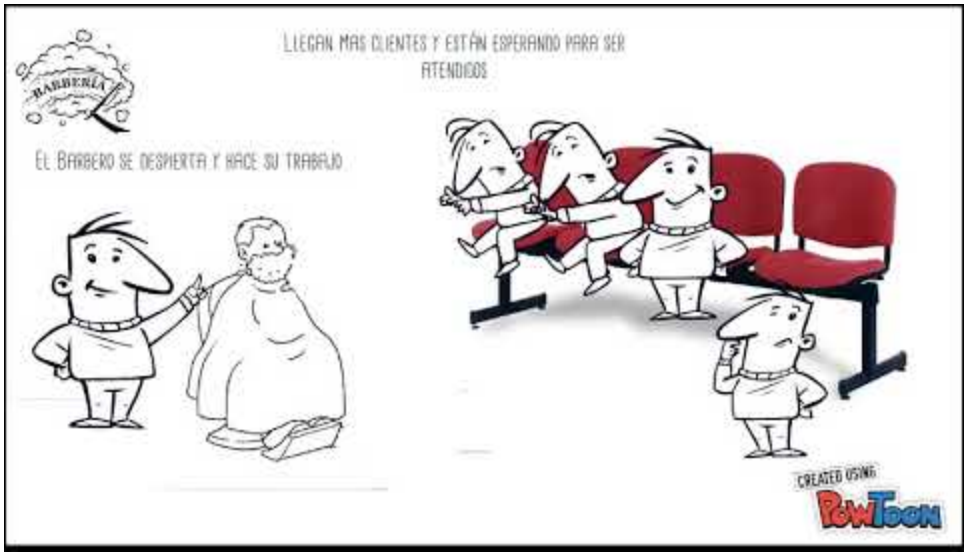
Posterior a ello realizamos un guardado temporal de las referencias de cada una de estas tasks, las cuales están asociadas a un proceso diferente del ciclo y que contarán con tiempos variables cada una.

```
futurePutBoxSmall = changeTaskTiming(putBoxTimingSmall, futurePutBoxSmall, putBoxSmall);
```

Para poder alterar el tiempo de cada una de esas task ya inicializadas hacemos uso de métodos especializados, los cuales van a devolvernos nuevamente la referencia posterior a haber realizado el cambio de tiempo en la ejecución de una tarea.

Problema del Barbero

El problema consiste en una barbería en la que trabaja un barbero que tiene un único sillón de barbero y varias sillas para esperar. Cuando no hay clientes, el barbero se sienta en una silla y se duerme. Cuando llega un nuevo cliente, éste o bien despierta al barbero o si el barbero está afeitando a otro cliente se sienta en una silla (o se va si todas las sillas están ocupadas por clientes esperando). El problema consiste en realizar la actividad del barbero sin que ocurran condiciones de carrera. La solución implica el uso de semáforos y objetos de exclusión mutua para proteger la sección crítica.



Para desarrollar la solución del barbero dormilón se utilizara un objeto que simulara ser nuestra barbería, un hilo principal que es el hilo del barbero y por último hilos de clientes que se generarán en tiempos aleatorios.

Barbero:

El barbero será un hilo corriendo todo el tiempo en el cual se estará consultado el estado del barbero si se encuentra dormido o no, de estar despierto atenderá al cliente y notificará que ya termino.

La existencia del barbero está determinada por un arreglo de tamaño , con el que se manejan estados según la posición del barbero que está durmiendo/atendiendo.

Para verificar el estado del barbero, se consulta por medio de una bandera, y se actualiza la imagen de la interfaz gráfica, de no estar ocupada la silla del barbero este espera hasta que lo despierte un cliente, hasta ese momento esa bandera se actualizará y procederá a atender al cliente.

```
private void nuevoProceso() {
    int contador = -1;
    boolean bandera = false;
    for (int x = 0; x < arrayBarberos.length; x++) {
        if (!arrayBarberos[x]) {
            arrayBarberos[x] = true;
            contador = x + 1;
            bandera = true;
            break;
        }
    }
    if (bandera) {
        despertar(contador);
    } else {
        //Si todos los barberos están ocupados, examina las sillas
        for (int x = 0; x < arrayEspera.length; x++) {
            if (!arrayEspera[x]) {
                arrayEspera[x] = true;
                contador = x + 1;
                bandera = true;
                break;
            }
        }
        if (bandera) {
            // Asignar silla
            ocuparSilla(contador);
        }
    }
}
```

Los métodos utilizados para la lógica del barbero son los siguientes:

```
public int despertar(int barbero) {
    switch (barbero) {
        case 1: {
            jLabelBarberoA.setVisible(true);
            atenderCliente atender = new atenderCliente();
            atender.contador = barbero;
            atender.start();
            break;
        }
    }
    return 1;
}

public int dormir(int barbero) {
    switch (barbero) {
        case 1:
            jLabelBarberoA.setVisible(false);
            jLabelestadol.setText("Durmiendo");
            break;
    }
    return 1;
}
```

Atención al cliente:

El barbero atiende al cliente obteniendo el tiempo que se ingresa desde la interfaz gráfica , luego procede a activar el proceso de cliente satisfecho en donde se incrementan contadores y se busca a un nuevo cliente

```
public class atenderCliente extends Thread {

    private int contador = -1;

    public void run() {
        String velocidad = jTextFieldl.getText();

        try {
            Thread.sleep(Integer.parseInt(velocidad) * 100);
        } catch (InterruptedException ex) {
            Logger.getLogger(interfaz.class.getName()).log(Level.SEVERE, null, ex);
        }
        try {
            clienteSatisfecho();
        } catch (InterruptedException ex) {
            Logger.getLogger(interfaz.class.getName()).log(Level.SEVERE, null, ex);
        }
    }
```

para notificar que ya termino del corte se consume este método sincronizado y se notifica a todos los subprocesos que ya se actualizo la bandera.

```
private void clienteSatisfecho() throws InterruptedException {
    arrayBarberos[contador - 1] = false;

    if (contador == 1) {
        actualizarcobro(1);

    }
    verificarEspera();
}
```

CLIENTE:

El cliente correrá un hilo en el que intentará acceder a la barbería, si no hay espacio el cliente simplemente es ignorado por el sistema y no se agrega a la cola de clientes existentes . Si existe un lugar disponible el cliente nuevo se encola

```
public void run() {
    while (true) {
        // Inicio semáforo
        for (int x = 0; x < arrayBarberos.length; x++) {
            if (!arrayBarberos[x]) {
                try {
                    Thread.sleep(2000);
                } catch (InterruptedException ex) {
                    Logger.getLogger(interfaz.class.getName()).log(Level.SEVERE, null, ex);
                }
                dormir(x + 1);
            } else {
                activar(x + 1);
            }
        }

        for (int x = 0; x < arrayEspera.length; x++) {
            if (!arrayEspera[x]) {
                desocuparSillaEspera(x + 1);
                if (!modelo.isEmpty()) {
                    ocuparSilla(indexSillaVacia() + 1);
                    modelo.remove(0);
                    arrayEspera[x] = true;
                }

            }
        }
    }
}
```

SILLA:

Las sillas están representadas por un arreglo el cual maneja estados según la posición que se ocupa, visualmente sincronizada con labels para mostrar el funcionamiento.

```
public boolean[] arrayEspera = new boolean[20];      // Array para verificar las sillas disponibles
```

```
private void ocuparSilla(int silla) {
    switch (silla) {
        case 1:
            jLabelEspera1.setVisible(true);
            break;
        case 2:
            jLabelEspera2.setVisible(true);
            break;
        case 3:
            jLabelEspera3.setVisible(true);
            break;
        case 4:
            jLabelEspera4.setVisible(true);
            break;
        case 5:
            jLabelEspera5.setVisible(true);
            break;
```

Problema del Space Invaders

Uso de hilos propuesto

Para el problema se propone utilizar 3 hilos para la elaboración del juego.

El 1er hilo JugarThread que es el principal que es el encargado de detectar las pulsaciones del teclado para poder mover al jugador, también es el encargado de mostrar los componentes en el tablero, este hilo se ejecuta cada 2 milisegundos.

El 2do hilo CrearInvasorThread es el encargado de crear a los enemigos este genera una lista de enemigos y los despliega en el tablero este se ejecuta cada 25 segundos.

El 3er hilo MoverInvasorThread es el hilo encargado de mover a los posibles enemigos en el tablero, también realizaría las validaciones si llego al final del tablero o si el jugador lo elimino, este hilo se ejecuta cada 45 milisegundos.

Situaciones en las cuáles es posible que se den: deadlocks, livelocks

- Se utiliza ReentrantReadWriteLock para bloquear al momento de insertar o de leer la lista de enemigos y la lista de disparos. Se utiliza el writeLock().lock() cuando se va a insertar en la lista . Se utiliza readLock().lock() cuando se lee la lista. Y al finalizar esas acciones writeLock().unlock(); o readLock().unlock() para mejorar el uso de concurrencia en las acciones realizadas.