CHAPTER 6

Concurrent execution and networking

6.1 Concurrent execution

Before we start designing solutions for network gaming, we must first be able to run the rules engine in a separate thread. Furthermore, the rules engine should run without any UI component. Ultimately, the goal is to run an engine in a different process on a remote machine.

This section also introduces several common approaches and associated patterns for managing concurrent execution of processing, for example, the parallelization of tree search for planned AIs.

6.1.1 Separate rules engine and user interface

6.1.1.1 Interaction between components

If we want to parallelize processing, we can emphasis the following components:

- The rules engine that changes state using commands;
- The rendering engine that draws the state and produces commands;
- A component which orchestrates the interactions, embodied by the `PlayGameMode` class in the Pacman example game.

These components need common information, such as game state data. Some components modify these shared data, which makes reading this data impossible during the changes. Other components produce messages, such as the rules engine that notifies status changes or the user interface that produces commands. Finally, the whole is subject to real-time constraints, such as the rendering that must be performed at 60 frames per second.

For each type of interaction, a solution must be found to allow access or modification of the data, without errors or blocking. Another usual problem in concurrent execution is *data race*. It occurs when multiple components want to view or edit the same information at the same time. It does not necessarily lead to blocking, but slow down the program. In these situations, components are constantly waiting. The latency generated by these conflicts also comes with scheduling problems: the order in which data is read and written is generally not predictable, which can lead to erratic behavior.

All the work in this section is to identify the problems associated with parallel processing and then find ways to minimize or remove them.

6.1.1.2 Rules engine thread

The first separation step in this chapter is to place the rules engine in a dedicated thread. Hence, it is necessary to determine where is the game state, or more precisely, who "owns" and controls access to the game data. There is no concurrent processing in the rules engine, and it is the only one that changes the game state. As a result, we consider that the rules engine is the sole owner of the game state, and to access it, you have to ask for it. In the opposite case, where there are concurrent accesses to data, as it is the case in conventional databases, it would be necessary to separate and synchronize the parts that modify the database of those that store them.

We define a `RulesThread` class in the `sync` package to manage access to the rules engine and its state. An alternative would have been to enrich the `Rules` class. However, from a conceptual point of view, it gives too much responsibility to this class. Just as methods must be separated into sub-methods when they become too large, it is strongly advised to divide classes that have too many features.

Rules engine thread

The `RulesThread` class handles the execution of the game state updates in the background. The `run()` method of the `Thread` class is implemented, and contains a loop that continuously calls an `update()` method, as long as the `running` attribute is true:

```
public void run()
{
    while(running) {
        update();
    }
}
```

The `running` attribute is used to control the stopping of updates and the life of the thread. A `stopRunning()` method is defined to trigger the end of the thread:

```
public void stopRunning() {
    running = false;
}
```

The `running` attribute is declared `volatile` to ensure that it is modifiable by all threads. It also ensures that there is no risk that there are multiple copies with different values in the caches of the processor cores. It is not necessary to declare the `stopRunning()` method as `synchronized` since the only access to the `running` attribute is in the `run()` method. Concerning these accesses, there are two possible scenarios. In the first scenario, another thread modifies the attribute by calling the `stopRunning()` method during the while loop of the `run()` method: no problem. In the second scenario, the modification is during the while test of the `run()` method: there is then a data run, and therefore a slight block (a few nanoseconds). Since the goal is to stop the game, it only happens once and presents no problem for the user. However, it remains relevant to have thought about this before making a decision, knowing that the synchronization of the `stopRunning()` method would have unnecessarily blocked its callers.

⇒ Note: This usual approach does not allow you to terminate the thread during an update. For a game like Pacman, which updates are very fast, it's not a problem. For other games, it is necessary to place tests to know if the stop is requested regularly. A good approach is to use the mechanism of thread interruption integrated into the Java language. To interrupt a thread, the `interrupt()` method of the `Thread` class is called. Within the update, the `isInterrupted()` method of the `Thread` class can be periodically checked to see if the game has to stop.

Update the game state

The `update()` method of the `RulesThread` class updates the game state at the defined frequency. This one starts by watching if the time spent since the beginning of the last update is sufficient:

```
private long lastUpdate = 0;
public void update()
{
    long now = System.nanoTime();
```

```
    State state = rules.getState();
    if ((now-lastUpdate)<state.getEpochDuration()) {
        return;
    }
    lastUpdate = now;
```

It follows the update of the state. This one is synchronized, to prohibit any access during the update:

```
    synchronized(this) {
        rules.update();
    }
```

The end of the method pauses the unused time so as not to make the processor work unnecessarily:

```
    long elapsed = System.nanoTime() - lastUpdate;
    long milliSleep = (state.getEpochDuration()-elapsed)/1000000;
    if (milliSleep > 0) {
        try {
            Thread.sleep(milliSleep);
        } catch (InterruptedException ex) {
        }
    }
}
```

Simple reading of the rules engine

If another thread needs to read data in the rules engine, a simple getter is too dangerous. Even synchronized, it would offer no guarantee on non-concurrent access to data. A simple but inefficient solution is to create a synchronized method with a lambda function:

```
public synchronized void processRules(
    Consumer<Rules> consumer) {
    consumer.accept(rules);
}
```

To use this method, we provide a lambda function with a `Rules` type argument. For example, to save the game state in the `PlayGameMode` class:

```
public void saveState(String fileName) {
    rulesThread.processRules((rules) -> {
        State state = rules.getState();
        state.save(fileName);
        savedState = state.clone();
```

```
    });
}
```

The `processRules()` method of the `RulesThread` class executes all lambda code with the assurance that the rules engine is locked all the way through. So, if the update is in progress, the other thread is waiting. If the other thread is running this method, the update must wait before starting. This simple solution is relevant in cases where these locks are not troublesome, such as saving or loading the game state.

6.1.1.3 Transfer commands (Double Buffer)

The user interface produces commands when the user interacts. For example, in the `handleInputs()` method of the `PlayGameMode` class, the keyboard is consulted to see if the arrows are pressed. If it is the case, a corresponding command is produced.

A basic approach to transfer these commands to the rules engine thread is to create a synchronized method in the `RulesThread` class to add commands, or directly use the `processRules()` method presented above. This approach causes many locks. Indeed, the keyboard is consulted 60 times per second, and each time, if a command is added with this approach during an update, the main thread is then blocked. The display is then frozen, and the entire game seems to pause the time of the update. In the case where the update took the maximum allowed time, the duration of the pause is as long. For the Pacman example game, where the update can take up to 80 milliseconds, it's still playable, but the display is unstable. For games with much larger updates, it completely wipes out the user experience.

A usual solution in the field of parallel execution is to use a *double buffer*. It's not an "official" design pattern, but it's still very popular. The first example in this book is the case of the double buffer presented for display in **Chapter 3**. When the graphics card transfers the data from one buffer to the screen, the other can be modified without any blocking. This principle was also used for the rendering engine, with display data placed in layers (e.g., `Layer` child classes). The processor freely modifies this data and then copied it into the graphics card. The difference with the previous case is that there is a copy of one buffer to another and not a swap of the two buffers. Unlike the copy, swapping is extremely fast and therefore leads to the shortest possible blocking time.

For the transfer of commands, the solution is to create a command array `commands` in the `PlayGameMode` class. Then, in the `handleInputs()` method of the `PlayGameMode` class, when an arrow is pressed, a new command is added to that array:

```
synchronized(commands) {
    if (keyboard.isKeyPressed(KeyEvent.VK_RIGHT)) {
        commands.put(currentChar,new DirectionCommand(
```

```
                currentChar,Direction.EAST));
        }
        if (keyboard.isKeyPressed(KeyEvent.VK_LEFT)) {
            commands.put(currentChar,new DirectionCommand(
                    currentChar,Direction.WEST));
        }
        if (keyboard.isKeyPressed(KeyEvent.VK_DOWN)) {
            commands.put(currentChar,new DirectionCommand(
                    currentChar,Direction.SOUTH));
        }
        if (keyboard.isKeyPressed(KeyEvent.VK_UP)) {
            commands.put(currentChar,new DirectionCommand(
                    currentChar,Direction.NORTH));
        }
}
```

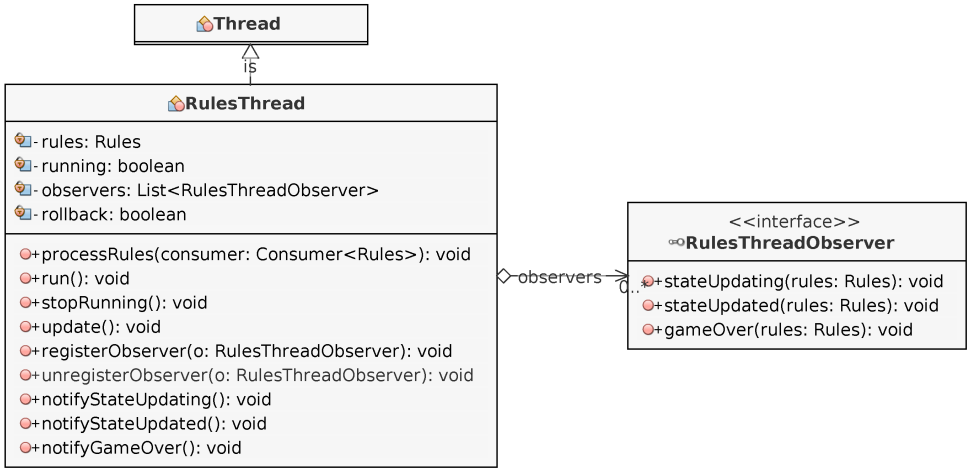
We synchronize the whole on the `commands` array: thus, the array can be safely modified without causing an error if another thread tries to access it.

Subsequently, the rules engine thread adds these commands to the game engine at the best time just before the update. Then, it empties the commands array, which no longer has any interest in referencing the added commands. The blocking time then corresponds to the maximum time between the creation of commands (like in the `synchronized` block above) and that of the transfer of the commands, e.g. at most a few hundred nanoseconds. In terms of data structures, there are two arrays: the one in the rules engine and the one in the `PlayGameMode` class. So there is a copy of the references from one array to another, but not commands: it is a hybrid case of double-buffer, between copy and permutation.

6.1.1.4 Efficient reading of the rules engine (Observer Pattern)

For many components, it is necessary to have access to the rules engine and the game state. As noted above, it is not feasible to leave free access to this data. It is also not efficient to very frequently request an access window, as we did with the `processRules()` method of the `RulesThread` class. In general, it seems complicated to allow components which require access to the data to request it at any time. The simplest way is to let the data owner choose these moments.

To get it, we use the Observer Pattern again. The owner of the rules engine, `RulesThread`, is made observable:



We consider three notification categories:

- `stateUpdating()` when the update is about to be started;
- `stateUpdated()` when the update has just finished;
- `gameOver()` when the game and the thread are finished.

The `notifyXXX()` methods call the corresponding methods with the rules engine as an argument.

Operations before the update

The `PlayGameMode` class, as the main orchestrator, is the best candidate to observe the rules engine. For the `stateUpdating()` method, called before a new state update, the main aim is to add commands. The first commands to be added are the ones caused by the player (e.g., the ones stored in the `commands` array):

```
public void stateUpdating(Rules rules) {
    synchronized(commands) {
        commands.entrySet().forEach((command) -> {
            rules.addCommand(command.getKey(),
                command.getValue());
        });
        commands.clear();
    }
```

The operation is synchronized with the `commands` array, to ensure that it is not changed during the operation.

Then, we add the commands of the AIs, which also makes it possible to run them in the thread of the rules engine (the `stateUpdating()` method of the observers are called by the `notifyStateUpdating()` method, itself called by the rules engine when it is about to update the state):

```
    for(int index=0;index<5;index++) {
        if (index == currentChar)
            continue;
        Command command= ais[index].createCommand();
        if (command != null) {
            rules.addCommand(index, command);
        }
    }
}
```

Operations after the update

The `stateUpdated()` method, called just after an update, determines whether the game is over. If this is the case, the `stopRunning()` method of the `RulesThread` class is called to terminate the rules engine thread:

```
public void stateUpdated(Rules rules)
{
    State state = rules.getState();
    if (state.getGumCount() == 0) {
       rulesThread.stopRunning();
    }
    Characters chars = state.getChars();
    Pacman pacman = chars.getPacman();
    if (pacman != null && pacman.getStatus() == PacmanStatus.DEAD
     && pacman.getStatusTime() == 0) {
        rulesThread.stopRunning();
    }
}
```

End-of-game operations, such as changing the game mode, can not take place until a rules engine thread is running. It is the reason why the above method merely asks to stop it.

Operation after the end of the rules engine thread:

The `gameOver()` method, called just before the end of the engine thread, performs the game mode change operations. The different cases of victories and defeats are the same as before: