Erlang assignment

Alexander Tang

2014-2015

1 Task 1: tile behaviour

I will devote this section to a brief description of how the game works for *main:playnoblaster()*. The main calculations happen in the **tileoperation/7** method. Parameters given include functions to determine the tile propagation, future tiles and stop conditions to both of these functions. This way one method is generalized to take care of any directions, if given the correct function parameter. **futuretilelist/4** returns all possible future tiles in a list, depending on the given tile and direction. **processtile/4** checks which location the given tile will move to and updates the values accordingly.

2 Task 2: tile failure

Tiles are created in *manager.erl* (with **spawn_link/3**). I spawn an additional process that runs **managetiles/0** that accomplishes this. This process its sole duty will become to trap exit signals from tiles and replacing these tiles with their value when they died. The reason for creating an external process is that it's available to catch exit signals all the time. If I were to use the original manager process, I'd have to insert traps almost everywhere. This becomes a nightmare to maintain if this were to become a larger program.

When a tile dies, it sends a message to its linked process. To make it send its tile value before it dies, I modified *exit(killed)* to *exit({killed, Id, CurrentValue})*. This way I can respawn a linked tile instantly. An alternative to modifying the *exit(killed)* would be to send a message with its value to tilemanager before it dies. But I'd then have to add a variable in **tilemanagerloop/0** that holds the variable and then receive the expected exit signal. This means I'd have to receive two messages to handle one dying tile. Having the variable linger around in the loop after it's no longer needed (when the new tile has been spawned) leaves a bad taste. It is also prone to concurrency problems if two tiles happen to send their values at the same time before one has the chance to send the exit signal. This is unlikely since the blaster kills one tile at a time, but it's a possibility.

Lastly, monitors could have been used as an alternative to links. Since *manager.erl* and *tile.erl* share a unidirectional relation (tile doesn't need to know about the state of manager), monitors are certainly a good choice. However I don't think there is any particular benefit to having a monitor over a link. You could setup multiple nested monitors between tile and manager, but that's not desirable in our case. So in this case I think it's a matter of preference but both will work fine.

To summarize:
An external process 'tilemanager' takes care of creating and recreating tiles. When tiles die, they send their value to tilemanager who recreates the tile with its value restored. This happens independent of the actual gameplay.

3 Task 3: tile concurrency

The timer is simply replaced by **readyup/1**. When tiles are done propagating commands, the last tile will send a ready signal to manager. When manager receives 4 ready signals, it'll get out of the receive block of **readyup/1** and continue the rest of 'sendData'. After rigorous testing I've noticed there is a slight chance of a tile getting killed just as it's needed for calculation. In that small window between a tile dying and it getting recreated by tilemanager, it could be needed. Therefore, I included a try catch in **processtile/4** so it would wait 10 milliseconds and retry if an error were to occur during the message send.

An issue that caused me many hours of grief is that the tiles created by tilemanager would sometimes not register fast enough for the collector/broadcaster. I solved this by having the initial 16 tiles send back a confirmready message to manager (through **confirmready/1**) indicating succesful registration before allowing manager to progress. Why recompiling *tile.erl* before running *main:play()* made it work before I implemented this will remain a mystery...