Ciências ULisboa
Faculdade de Ciências da Universidade de Lisboa

RELATÓRIO DO QUARTO EXERCÍCIO DE PROGRAMAÇÃO PARALELA E CONCORRENTE

Relatório por
Guilherme Lopes – 52761

- ## Descrição geral da Arquitetura

  A arquitetura da resolução foi construída tendo como base o ***Actor Model*** fornecido pelo docente da cadeira na página da mesma, que utiliza de unidades únicas e independentes com estado próprio chamadas de *Actors*, onde é possível que executem concorrentemente e comuniquem entre si através de mensagens. A arquitetura foi divida em camadas de *Actors* com as suas respetivas responsabilidades. Os tipos de *Actors* escolhidos foi um ***Customer Actor*** que tem a responsabilidade padrão de decidir que trabalho será feito por outros *Actors*, como lidar e processar a resposta dos mesmos e é quem decide quando encerrar tudo, foi também utilizado um ***Manager Actor*** que recebe as mensagens do *Customer*, gere e decide para onde enviar dependendo do tipo de mensagem (só existe um, no caso) enquanto tenta manter o equilíbrio da carga de trabalho entre os *Employees* e é quem é responsável por lidar com *crashes* inevitáveis em algum *Actor*, por fim foram utilizados cinquenta ***Employees*** que têm a responsabilidade de ler os ficheiros em paralelo e decidir o que fazer perante o que foi lido. As criações dos *Employees* necessários foram feitas no *Manager*, que também decide quantos vão ser inicialmente criados e quem vai receber a próxima mensagem, enquanto mantém a carga de trabalho igual entre todos os *Employees*.
  O número de *Employees* foi escolhido, em parte, ao acaso, porém é um número que pareceu equilibrado em relação ao número total de ficheiros, para fazer bom uso do *Actor* e do seu paralelismo sem sobrecarregar *mailboxes*. Possivelmente teria que ser ajustado durante a execução do programa para testes mais intensivos e maior escalabilidade.

- ## Justificação da escolha da Arquitetura

  O *Actor Model* foi escolhido ao invés de *Goroutines* ou *Erlang* devido à familiaridade com a linguagem Java e pelo *Actor Model* fornecido ser bastante simples de aprender e utilizar. *Akka* não foi escolhido por considerar uma tarefa complicada aprender uma *framework* mais complexa e rica (preparada para ser mais escalável) do que o necessário para uma tarefa consideravelmente simples. A Arquitetura do programa foi desenhada para manter a simplicidade e com um número reduzido de tipos/camadas de *Actors*, mantendo o código coeso e sucinto face à natureza recursiva dos ficheiros enquanto atende aos requisitos de execução sequencial e paralela que se encontram no enunciado.

- ## Explicação de como foi garantida a execução sequencial das operações *create*, *move* e/ou *delete*

  A execução sequencial de tais operações foi garantida através das respostas dos *Employees* serem enviadas de volta para o *Customer*, que, por ser único, guarda todas as respostas na sua *mailbox* e só pode executar a resposta às mensagens dos *Employees* sequencialmente. Mais detalhadamente, após cada *Employee* decidir, em paralelo, se será necessário alterar o nome do ficheiro respetivo, cada *Employee* irá enviar uma mensagem de resposta, com informação do que deve ser feito ao nome do

ficheiro, onde o *Customer* irá executar um por vez dependendo qual mensagem será retirada da sua *mailbox*.

- ## Como foi paralelizado o programa

  Para alcançar leituras paralelas, apenas foi necessário que o *Manager* transmita cada uma das mensagens criadas para a *mailbox* dos vários *Employees* existentes para ler os ficheiros. Com isto, cada ficheiro foi distribuído por cada um dos *Employees*, onde é lido concorrentemente entre os vários *Employees* o que cada um contém nas suas *mailbox*. Isto ocorre pela natureza do *Actor Model* dar automaticamente o uso devido às *Threads*.

- ## Operações de sincronização requeridas pelo programa

  A única operação, ou no caso, mecanismo de sincronização utilizado, foi a estrutura de dados ***ConcurrentLinkedQueue*** para o tipo do objeto da *mailbox* dos *Actors*. Caso ocorresse *deadlocks* no programa, uma forma de tentar prevenir isso seria a utilizar a função *yield*.
  Além da questão das operações, para sincronizar o manuseio de ficheiros sem ocorrer problemas, é tudo processado sequencialmente no *Customer* mantendo a natureza recursiva dos ficheiros, garantindo que cada ficheiro seja modificado apenas se seu parente (ficheiro de nível acima) já tenha sido modificado.

- ## Como se lidou com a estrutura recursiva dos ficheiros

  Como foi referido anteriormente, foram criados os *Employees* que leem ficheiros paralelamente. Inicialmente o *Customer* envia mensagens com cada um dos ficheiros iniciais dentro da *tree* ao *Manager* que distribui por cada *Employee*.
  Cada *Employee* lê os ficheiros na sua *mailbox* e envia resposta de volta ao *Customer* que renomeia o ficheiro, caso seja necessário, e em caso afirmativo envia novamente uma nova mensagem com o ficheiro renomeado ao *Manager* que distribui por um *Employee*, caso o ficheiro não seja renomeado no *Customer*, este envia mensagens com cada um dos ficheiros dentro do respetivo ficheiro anterior ao *Manager* que distribui por cada um dos cinquenta *Employees*.
  Este processo é então repetido até os ficheiros da profundidade X não terem mais nenhum conteúdo em si, ou seja, mais nenhuns ficheiros "aninhados" e todos os ficheiros tiverem sido lidos.

- ## Outras informações

  O código programa a ser avaliado é o que se encontra na *package "assignment4_v3"*. Em relação aos dois programas nas outras duas packages ("*assignment4"* e "*assignment4_v2"*), o primeiro não funciona e o segundo funciona apenas para o caso específico do *Shell Script*, ou seja, não é propriamente recursivo.
  As *substrings* dos números a serem substituídos e dos números que irão substituir são definidas na ***main class***. Este programa foi escrito assumindo que quem o utiliza tenta substituir o nome dos ficheiros por uma *substring* que não existe anteriormente nesse mesmo diretório, ou seja, o programa não lida automaticamente com *input* de um número "duplicado" no nome dos outros ficheiros e irá funcionar incorretamente caso não exista esse cuidado.
  O programa foi construído e testado no ***Windows***, onde se notou que a reconstrução dos ficheiros, caso fosse necessário renomeá-lo, não era necessário pois o seu conteúdo permaneceu intacto, por isso, a única coisa que é realmente feita é a renomeação diretamente no ficheiro com a função *renameTo()* da biblioteca *java.io.File* utilizada para manusear ficheiros. Por essa razão é possível que existam problema ao testar o programa em outros sistemas operativos e também devido ao

funcionamento do acesso a diretórios, nomeadamente do caracter '\' e '/' para acesso ao caminho dos ficheiros, que é diferente no *Windows* e em *Linux* por exemplo, no caso foi utilizado '\'.