Projeto Sistemas Operativos 2018/19

Gestão de Vendas

Universidade do Minho

Mestrado Integrado em Engenharia Informática

Grupo PL66
Rui Santos - a67656
Luís Vila - a84439
Luís Ramos - a83930

1. Introdução

Este relatório aborda a resolução do projeto prático da unidade curricular Sistemas Operativos, do 2º ano do curso de Mestrado Integrado em Engenharia Informática. O projeto consiste, resumidamente, em construir um protótipo de um sistema de gestão de inventário e vendas.

O objetivo era a criação de quatro programas diferentes, manutençao de artigos, servidor de vendas, clientes de vendas e agregador de dados, e depois estabelecer a comunicação entre eles de modo que cada um funcionasse de modo autônomo. Para fazer realizar essa comunicação de forma correta, recorremos à matéria lecionada nas aulas práticas e teóricas.

Ao longo deste relatório iremos referir algumas das decisões tomadas, explicando cada uma delas ao pormenor. Também iremos referir algumas alternativas e soluções encontradas para os problemas que iam surgindo consoante íamos avançado no projeto, de modo a demonstrar todo o nosso conhecimento.

2. Modulos

2.1 Manutenção de artigos

O nosso ma (manutencao de artigos) é um programa simples que lê comandos do stdin até encontrar um EOF, que faz com que este termine. Cada comando recebido é separado nos seus 3 argumentos (ou apenas num, no caso do comando 'a' para o agregador), e consoante o primeiro argumento do comando é escolhida a função interna a utilizar. O 'i' cria um artigo,o 'p' e o 'n' mudam o preço e nome respectivamente.

O resultado da operação é então escrito no ficheiro *ARTIGOS* e, no caso de 'i' e 'n', o ficheiro *STRINGS* também é alterado. Acabamos por ter que fazer uma adição neste módulo para inicializar os stocks no ficheiro *STOCKS* sempre que um produto novo é inicializado.

Escolhemos usar ficheiros de texto para todos os nossos dados. O formato escolhido para o ficheiro *ARTIGOS* foi o seguinte

XXXXXXXXXX YYYYYYYYYY ZZZZZZZZZZ
XXXXXXXXXX YYYYYYYYYY ZZZZZZZZZZ

Em que XX representa o código do artigo, YY o offset no ficheiro *STRINGS* onde está guardado o nome e ZZ o preço do artigo. Cada linha tem 36 bytes incluindo os espaços e os parágrafos.

O formato do ficheiro *STRINGS*

XXXXXXXXXX
XXXXXX
XXXXXXXXXXXXXXXXXX

É simplesmente um ficheiro de texto com os nomes separados por parágrafos, mas neste caso de tamanho variável.

Quanto ao ficheiro STOCKS utilizamos o seguinte formato

XXXXXXXXXX YYYYYYYYYY
XXXXXXXXXX YYYYYYYYYY

Em que XX representa o código do artigo, YY o stock. Cada linha tem 24 bytes incluindo os espaços e parágrafos.

2.2. Servidor de vendas

O nosso servidor de vendas também funciona com um loop. Este fica à espera de comandos e processa-os sequencialmente, mas neste caso, lê de um *named pipe* que todos os clientes tem acesso para escrever.

Quando é lido um comando, se este só tiver um argumento é chamada a função de mostrar informação sobre stock e produto, a qual é depois retornada ao cliente. Caso tenha dois é chamada a função de alterar stock. Se o segundo argumento for negativo, e houver stock suficiente então é registada uma venda e devolvida a informação sobre o novo stock, caso contrário o servidor responderá “Stock inválido”.

No que toca ao funcionamento em si, optamos por não fazer um “mini-servidor”, ou seja um fork por cliente, pois isso iria causar muito problemas para lidar com as alterações concorrentes de stocks.

Pensamos ainda em implementar este método para apenas quando fosse uma operação de consulta, visto que aqui não criava nenhum problema de concorrência, mas como não tivemos muito tempo restante, o ganho de performance não pareceu vantajoso em relação à hipótese de introdução de erros.

Incluímos ainda um sleep no ciclo principal do servidor, pois não há razão para ele tentar ler milhões de vezes por segundo num pipe vazio. Talvez pudéssemos ter feito isto de forma melhor utilizando sinais.

Quanto à comunicação cliente-servidor, optamos por ter um fifo com nome conhecido, em que todos os clientes podem escrever ao mesmo tempo (desde que escrevam todo o comando de uma vez usando o write, sendo assim a escrita garantidamente atómica), sendo que cada um escreve uma string composta pelo seu identificador único (neste caso o pid).

Este pid é usado de seguida para abrir um fifo pessoal do cliente, no qual o servidor escreverá a resposta.

O ficheiro de VENDAS tem o seguinte formato

XXXXXXXXXX YYYYYYYYYY ZZZZZZZZZZ
XXXXXXXXXX YYYYYYYYYY ZZZZZZZZZZ

Em que XX representa o código, YY a quantidade, e ZZ o montante. Cada linha tem 36 bytes incluindo espaços e parágrafos.

2.3. Cliente de vendas

O nosso cliente de vendas tem um funcionamento bastante parecido ao ma (manutenção de artigos). Este lê do stdin até encontrar um parágrafo, no qual executa o comando, ou EOF para terminar, mas em vez de processar o comando, concatena o seu pid no início do mesmo, e envia-o para o pipe público do servidor, esperando de seguida por uma resposta no seu pipe pessoal.

Essa resposta é depois impressa no stdout do cliente.

2.4. Agregador

A função do nosso agregador consiste em sintetizar as vendas efetuadas.

Tendo o “ma” a correr, e utilizando o comando que chama o agregador, podemos passar pelo stdin um input de várias “Vendas”, até o EOF.

Para impedir que o ‘ma’ parasse de correr, optamos por utilizar um fork, de modo a correr o agregador. Esse fork redireciona o seu stdin para o ficheiro escolhido no ‘ma’, e o seu stdout para um novo ficheiro, cujo nome será a data e hora em que o agregador foi chamado.

O agregador vai então ler vendas do stdin e vai juntá-las de forma a agregar toda a informação contida nelas, ou seja, se o agregador ler uma “Venda” do produto X com N1 quantidade vendida, e uma “Venda” do mesmo produto X com N2 quantidade vendida, vai juntar essa informação, ficando a “Venda” registada como uma “Venda” do produto X com N1 + N2 quantidade vendida. Faz o mesmo com o montante total.

Depois de se dar a agregação, o agregador escreve no stdout resultado final.

Ainda a referir que o plano inicial era que, de modo a tornar a agregação mais rápida, fosse usado o último ficheiro de agregação e só fosse lido do vendas a partir de um certo offset. Embora conseguirmos guardar o dito offset e apenas ler essas vendas, tivemos dificuldades em que o agregador (que é genérico) não interrompesse no fim do primeiro EOF, tendo até tentando tratar o input dos dois ficheiros no ma (embora este passo extra vá contra o propósito da otimização), mas sem tempo para dar debug acabamos por reverter à solução original de iterar todo o ficheiro vendas

3. Aspectos valorizados

3.1. Agregação concorrente

No enunciado do trabalho prático é mencionado que seria valorizada a implementação de um agregador que executasse concorrentemente, algo que tentamos fazer recorrendo à utilização de forks para conseguir correr o agregador concorrentemente.

Porém, devido a um lapso na nossa interpretação do enunciado, tivemos que mudar a nossa abordagem ao desenvolvimento do nosso agregador, pelo que acabamos por não implementar uma forma de conseguir correr o agregador concorrentemente.

Sucintamente, a nossa ideia para o agregador era dividir o input de vendas em 4 partes, e agregá-las individualmente recorrendo à utilização da criação de 4 forks que executassem concorrentemente. Após esta operação, queríamos que o processo pai agregasse finalmente o resultado das agregações concorrentes, chegando assim ao resultado desejado.

3.2. Caching de preços

Neste tópico não trabalhamos tanto quanto o desejado. Chegamos a ter uma cache simples com uma política simples de tamanho fixo que guardava todos o artigos e a sua informação respectiva, mas não chegamos a implementar qualquer tipo de sistema de gestão para verificar quais eram os produtos menos usados, de modo a limpar esses para não sobrecarregar a cache, ou algum tipo mecanismo para ter uma cache de tamanho variável.

3.3 Compactação do ficheiro STRINGS

Uma vez mais, acabamos a não implementar nada deste aspecto principalmente por falta de tempo, mas na teoria seria um algoritmo simples. A nossa ideia era guardar no início do ficheiro STRINGS, uma linha com comprimento fixo que teria 2 números, um a indicar a quantidade de nomes escritos e outro a indicar quantos nomes estariam a ser usados.

Quando um novo artigo fosse introduzido eram os dois incrementados. Quando apenas fosse mudado o nome, então só o primeiro é que seria incrementado. Sempre que era mudado um nome seria verificado se passava dos 20% de lixo, e caso isso se verificasse, então era iniciada uma limpeza.

O processo para limpar que tínhamos em mente, embora possivelmente pouco eficiente, era percorrer o *ARTIGOS* e ir guardando todos os nomes encontrados num novo STRINGS (ou numa estrutura, para depois substituir tudo). Outra hipótese era guardar num ficheiro NomesNãoUsados, que continha os offsets de todos os nome por usar.

4. Conclusão

Consideramos que conseguimos cumprir todos os objectivos “base” do trabalho razoavelmente bem, a aplicação permite que vários utilizadores “martelem” milhões de comandos ao mesmo tempo sem perder nenhum comando, mantendo consistência de dados com uma performance razoável.

No entanto, podia ser melhorado em vários aspectos, tais como, fazer “mini-servidor” como descrito em cima em alguns casos. Podíamos ter optado por usar ficheiros binários, que fariam melhor utilização do espaço visto que só guardamos inteiros, e que provavelmente tornariam as leituras e escritas mais simples, visto não ser preciso ter tanto cuidado com Strings em C e os problemas associados às mesmas.

Não conseguimos implementar até ao fim nenhum dos aspectos valorizados, principalmente por falta de tempo passado a dar debug no servidor. Embora tenhamos tido uma versão concorrente do agregador quase a funcionar, acabamos a ter que a recomeçar por causa de um lapso de interpretação do enunciado, e acabamos a não conseguir re-implementar.