Sistemas Operativos 2019

Gestão de Vendas

José Pedro Silva, José Ricardo Cunha, Válter Carvalho

13 de Maio de 2019



Grupo nº 7
Mestrado Integrado em Engenharia Informática
Universidade do Minho

Conteúdo

1 Introdução

Este relatório irá abordar a realização do projeto realizado no âmbito da Unidade Curricular **Sistemas Operativos**, que consiste na criação de uma rede de gestão de vendas. Este trabalho foi realizado tendo como base todos os conteúdos lecionados nas aulas.

2 Manutenção de artigos

O programa da manutenção de artigos recebe todo o seu input pelo *stdin* e permite executar várias operações, como por exemplo:

- Inserir artigos
- Alterar o nome de artigos
- Alterar o preço de artigos
- Pedir ao servidor para fazer a agregação das vendas

Apresentam-se a seguir os argumentos(*corretos*) necessários, para cada operação, para que o programa funcione de acordo com o que é esperado.

```
./ma
i <nome> <preco>   -> insere novo artigo e mostra o codigo
n <codigo> <nome>  -> altera o nome do artigo
p <codigo> <preco> -> atera o preco do artigo
a                  -> agrega o estado atual das vendas
```

2.1 Funcionamento da manutenção de artigos

A manutenção de artigos, como já foi dito em cima recebe o seu input através do *standard input* e em função daquilo que recebe executa uma operação. De modo a poupar memória o grupo decidiu alocar um *array* com a capacidade de 1024 caracteres uma vez que não existem nomes maiores do que esse tamanho. No ficheiro *artigos*, cada artigo tem alocado 8 bytes para o seu identificador, 8 bytes para a sua referência relativamente ao ficheiro *strings* e por fim 8 bytes para o seu preço. O grupo decidiu guardar a informação em formato texto de modo a poder ver o conteúdo dos ficheiros.

2.1.1 Inserir artigos

Esta operação recebe um nome e um preço e cria no ficheiro *artigos* todos os campos relativos a este artigo, para além disso ainda acrescenta o nome do artigo ao ficheiro *strings* e coloca o stock a zeros no ficheiro *stocks*. No final mostra ao utilizador o código identificador que foi atribuido ao artigo em questão.

2.1.2 Alterar o nome

Esta operação recebe como input o código identificador assim como o seu novo nome. Esta operação acede ao ficheiro *strings* acrescentando o novo nome no fim do mesmo e mudando a sua referência no ficheiro *artigos*. De modo a poder fazer a **compactação de artigos** é criado um ficheiro *deprecated* onde são guardadas as referências para os nomes que estão obsoletos.

2.1.3 Alterar o preço

Esta operação recebe como input o código identificador do artigo assim como o seu novo preço. Nesta operação existe um acesso ao campo do preço do artigo em questão no ficheiro *artigos* e é alterado o preço para o novo preço recebido.

2.1.4 Agregação

Nesta operação não existe qualquer input uma vez que não é o programa da manutenção de artigos que lida diretamente com a agregação. Nesta operação é enviado um sinal **SIGUSR1** ao servidor que por sua vez faz a comunicação com o agregador. De modo a enviar o sinal ao servidor, o programa acede a um ficheiro que contém o **pid** do servidor.

3 Servidor de vendas

O **servidor** é o ponto de coordenação do projeto e, portanto, deve estar **sempre** a correr em *background* antes de qualquer execução de uma das funcionalidades que o Gestor de Vendas apresenta.

É este que é responsável pela atualização dos ficheiros de **vendas** e de **stocks**, assim como o **caching** (ver 6.1) e, por último, as **agregações** (das que têem origem na referência 2).

3.1 Funcionamento do servidor

O servidor é essencialmente um leitor de um *named pipe* (**FIFO**) público em que todos os clientes comunicam o comando a executar pelo servidor assim como o seu **PID** para posterior escrita no canal privado dos clientes (ver 4), que faz por ordem o seguinte:

1. Cria um *handler* para sinais **SIGUSR1** e **SIGUSR2**;
2. Cria uma **cache**;
3. Abre o *named pipe "server"* em modo **RDWR** (ler 3.2);
4. Lê desse *pipe* um buffer com a sequência "PID␣COMANDO";
5. Escreve para o cliente o resultado dos comandos (ler 4);
6. Mantêm-se em loop até ser desligado (leitura seguida de comandos);
7. Liberta a memória alocada pela cache.

Tratamento de sinais definidos pelo utilizador:

- **SIGUSR1**

  Sendo este invocado pela da manutenção de artigos (em 2), o servidor é imediatamente imposto a correr o **agregador** (ver 5), atraveś dum *handler* definido.

- **SIGUSR2**

  Sendo este também invocado por 2, aquando a alteração do preço dum artigo, o servidor através dum *handler* atualiza a **cache** (definida em 6.1) com o preço correto.

3.2 Notas acerca do servidor

Os **FIFOs** exibem um comportamento peculiar, isto é, aquando a sua abertura para leitura, eles bloqueiam até existir um escritor.

Considerando hipoteticamente que o servidor lê o seu **FIFO** público em modo **RDONLY**, quando um cliente fecha a sua conexão ao servidor, este último entra em bloqueio permanente, perdendo a capacidade de comunicação.

Decidimos, então, aplicar a *flag* **RDWR** aquando abertura do descritor, pelo que o servidor passou a ser escritor e leitor de si mesmo, nunca bloqueando e esperando sempre por novos clientes.

Nos handlers acabamos por usar os **SIGUSR1** e **SIGUSR2** porque são aqueles que nos dão mais "criatividade"e que não têem tantas *underlying conditions* como os restantes (i.e SIGINT, SIGCHLD, etc), que implicam um tratamento mais metódico e, francamente, desnecessário.

4 Cliente de vendas

Programa relativamente simples, cria um **FIFO** cujo nome é o seu **PID** e que abre o *named pipe "server"* em modo escrita. O cliente pode mandar um comando (tendo este de ser válido) que eventualmente o servidor irá ler através do *"server"* mencionado anteriormente, utilizando **FIFO** privado para escrever o resultado da execução do mesmo comando.

Garantimos a concorrência porque cada cliente lê do seu próprio canal de comunicação, não há *sapping* de dados de outros clientes, teoricamente, sendo o servidor que gere todo este processo de decisão de escrita das mensagens.

O comando do cliente, como visto em 3, é sempre enviado junto com o seu **PID**, isto é, é criado um buffer que contém estes dois dados e só depois é que é enviado, caso contrário o servidor não saberia que cliente estava a mandar a mensagem. Como nunca obtivemos *pid's* com um tamanho superior a cinco dígitos, o tamanho que decidimos alocar para este, foi também de 5 bytes.

5 Agregador

O **agregador** é um executável que recebe o seu input através do **STDIN**, estando o mesmo no formato de venda. Após a agregação o output é redirecionado para o **STDOUT**. Antes de executar o agregador, o servidor

faz um **dup2** do **STDIN** para um ficheiro cujo nome é uma formatação da data atual, ou seja, a data de agregação (por exemplo 2019-05-11T17:40:24). Dentro deste ficheiro estará a soma dos montantes e a soma das quantidade de vendas de cada artigo que é agregado.

5.1 Funcionamento do agregador

Foi mostrado anteriormente que o servidor espera por um comando de agregação por parte da manutenção de artigos. Neste caso, é redirecionado para o o **STDIN** do agregador o ficheiro de **vendas**. Este tem um *offset* na primeira linha que indica a última venda agregada, pelo que é enviado um conjunto de vendas para o executável do agregador, levando a que o processo se mantenha idêntico.

De forma mais concreta, o agregador irá pegar no conjunto de vendas recebidos e:

1. Cria um ficheiro temporário com as vendas necessárias, recebidas no **STDIN**;
2. Divide em N ficheiros as agregações, com N sendo igual ao número de artigos distintos nesse conjunto de vendas;
3. Cada um desses artigos é visto separadamente sendo atualizados paralelamente (ver 6.2);
4. No final, são todos juntos num só ficheiro com a data de agregação.

A forma como foi tratada a agregação concorrente será vista mais à frente, com mais detalhe.

6 Aspetos valorizados

6.1 Caching de preços

As *system calls* são dispendiosas, quando se tratam de ficheiros potencialmente muito grandes, a situação piora. Deste modo, é importante a implementação de uma cache, em particular, duma cache de preços.

A cache serve para o acesso rápido a preços de artigos "populares", na prática significa que correspondem a artigos vendidos muitas vezes, evitando assim *lseek, open* entre outras funções de sistema.

É de notar que no sentido desta UC, a cache tem de ser muito pequena relativamente à quantidade potencial de produtos, porque a RAM nunca aguenta carregar dados (de uma só vez e mantê-los em memória) com o tamanho que os ficheiros podem ter (Terabytes, Petabytes, etc). A implementação de uma cache muito simples foi feita da seguinte forma:

```
#define CACHE_SIZE 1024

struct data {
    int id;
    int vendas;
    int preco;
};

typedef struct cache {
  struct data* cached[CACHE_SIZE];
  int ocupados;
  int full;
}*Cache;
```

Foi implementada usando um AoS (*array of structs*) com tamanho arbitrário como estrutura principal de mapeamento, em que os campos *ocupados* e *full* são variáveis auxiliares para a sua construção.

6.1.1 Problemas da implementação

A forma como esta cache foi implementada, através de declaração estática, significa que para a venda de apenas 1 artigo, terá exatamente o mesmo tratamento que um conjunto de vendas com 1 milhão de artigos potencialmente diferentes, ou seja, serão todos de alguma forma inseridos na cache.

Há um massivo problema nestes casos, num aloca-se memória a mais que não é usada, noutro aloca-se memória a menos que é constantemente atualizada. Ora, pegando no segundo caso, significa que estaremos a perder produtos na cache, é impossível colocar todos eles em apenas $CACHE\_SIZE$ posições distintas, tem de haver trade-off algures, como por exemplo remover o artigo menos vendido e colocar o novo nesse local.

Para além disto, como os artigos não são inseridos usando um mapeamento como *HashTables*, não existe qualquer tipo de ordem de inserção nem ordenação dentro da cache, não é possível saber que artigo ocupa qual posição sem percorrer toda a estrutura de dados.

Os tempos de acesso passam, então, a depender do tamanho do array, dado por N, que coincide também com o número de produtos inseridos, le-

vando a que a complexidade para o acesso a um código de produto seja Θ(N), limitada no máximo por $N = CACHE\_SIZE$.

6.1.2 Possíveis melhorias da implementação

Um dos problemas vistos anteriormente, mostra o facto de que a cache é estaticamente declarada. E se não o fosse?

Alocando dinamicamente a cache poderia ser uma melhor alternativa. Há sempre o incoveniente da memória RAM não ser tão grande quanto ficheiros (potencialmente), mas o inconveniente de alocar espaço a mais, assim como espaço a menos, é resolvido. É sempre alocado espaço de acordo com as necessidades.

Ótimo, com a cache otimizada para suportar tamanhos arbitrários, passamos ao segundo grande problema, os tempos de acesso.

O melhor caso possível de acesso numa estrutura de uma estrutura *array-like* é $\Theta(1)$, e de alguma forma cada artigo saber a posição que ocupava dentro da estrutura era certamente exequível, seja por apontadores, por *hashing* ou qualquer estratégia de memória partilhada escolhida arbitrariamente.

Sendo assim, aplicando uma estratégia deste tipo evitava procuras de complexidade Θ(N), como visto anteriormente, causadas pela procura iterativa pelas $CACHE\_SIZE$ posições, melhorando imenso o tempo de execução (exponecialmente mais rápidos, talvez).

6.2 Agregação concorrente

O agregador, após a criação do ficheiro temporário em cima mencionado, vai verificar quantos artigos de ID diferente estão na *stream* de vendas fornecida. Seguidamente, são criados N *fork's* que gerem cada um o seu artigo, criando um ficheiro próprio com o código do artigo, atualizado individualmente.

De seguida, é varrida toda a coleção desses ficheiros e incluídos num ficheiro cujo nome é a data atual, também mencionado acima, com as vendas agregados corretamente e, mais importante, concorrentemente.

É um processo concorrente, porque só há apenas um descritor de ficheiro para cada um dos processos filho gerados, não há qualquer tipo de *race condition* e são completamente independentes entre si.

É garantido, também, que não são deixados sub-ficheiros por agregar porque o processo pai faz uma iteração para os N processos criados, com o

comando de *wait(NULL)*, ou seja, só arranca com o *joining* após terem sido criados todos os ficheiros necessários.

6.3 Compactação do ficheiro strings

Como já vimos antes, após ser feita a alteração do nome de um artigo, este é adicionado no fim do ficheiro *strings*.

Esta implementação gera um problema que é o espaço ocupado pelos nome que estão obsoletos. De modo a reduzir o espaço ocupado do ficheiro *strings* existe a compactação de strings.

A compactação de strings é, resumidamente, um programa que limpa todas as strings que estão obsoletas e altera a referência no ficheiro *artigos*.

Como funciona, então, a compactação? De modo a saber que nomes estão obsoletos o grupo criou um ficheiro *deprecated* que contem as informações relativas ao artigo cujo o nome foi alterado e criou ainda o ficheiro *bytes* que contém o número total de bytes obsoletos, de modo a permitir verificar quando foram atingidos os 20%.

Quando os 20% são atingidos dá-se então a compactação de strings que consiste em ler todo o conteúdo so ficheiro *deprecated* e percorrer os artigos corrigindo a referência para a string identificadora. No final da execução ambos os ficheiros *deprecated* e *bytes* são eliminados recorrendo à função *unlink*.

7 Conclusão

Em suma, após a conclusão deste trabalho o grupo considera que fez um bom trabalho, uma vez que todos os pontos do enunciado foram cobertos, assim como todos os conteúdos lecionados nas aulas práticas, sendo estes, **fork's**, **exec's**, **dup's**, **pipes** e por fim **sinais**.