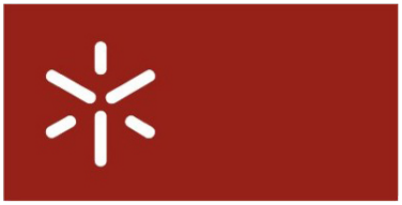

Sistemas Operativos

Mestrado Integrado em Engenharia Informática

Gestão de Vendas

A85227 João Pedro Rodrigues Azevedo
A85729 Paulo Jorge da Silva Araújo
A83719 Pedro Filipe da Costa Machado

Braga
Maio 2019

Conteúdo

Capítulo 1

Introdução

Este trabalho foi realizado no âmbito da unidade curricular de Sistemas Operativos do curso de Mestrado Integrado em Engenharia Informática (MIEI).

Esta UC, assim como este trabalho prático, tinha como principal objetivo a utilização de conhecimentos Teóricos e Práticos relativos à manipulação de ficheiros, gestão de processos, memória e utilização de CPU pelas diferentes entidades que o consomem (Processos).

Deste modo, foram importantes conhecimentos relativos ao escalonamento de processos, sincronização e também concorrência para a realização deste trabalho prático.

1.1 Objectivos

Neste trabalho pretende-se construir um protótipo de um sistema de gestão de inventário e vendas. O sistema deverá ser constituído por vários programas: manutenção de artigos, servidor de vendas, cliente de vendas, e agregador de dados.

A interação de possivéis clientes com os programas são comunicadas ao servidor que deverá controlar e garantir a correta resposta e segurança oferecida pelo mesmo de modo a escalonar o débito de dados e pedidos recebidos garantindo, portanto, uma resposta eficiente aos mesmos.

1.2 Outros aspetos (Organização do projeto)

Este projeto possui uma organização lógica de pastas contendo cada uma um dos programas pedidos, dentro da pasta source "src/".

Na pasta "FILES/"encontram-se os ficheiros ARTIGOS, STRINGS, STOCKS e VENDAS pedidos (e outros, como por exemplo, registo de erros no decorrer dos programas) e na pasta "PipeVendas/"estarão armazenados os pipes de comunicação com o servidor e ficheiros com registos de clientes no sistema (e os seus PIDs) e também alguns registos importantes para o agregador.

Capítulo 2

Descrição do projeto

Neste capítulo serão descritos os procedimentos, estratégias e todos os programas implementados bem como justificações sobre utilização de certas estruturas e ficheiros para o correto funcionamento de todo o sistema.

Além das operações pedidas são realizadas outras para permitir um melhor processo de *debugging* e registo de clientes/pedidos:

- Registo de erros (personalizados) relativos a comandos, números introduzidos, erros de leitura/escrita, etc...
- Registo de todos os clientes com *log in* no sistema
- ...

2.1 Manutenção de artigos

Este programa permite a introdução e modificação de produtos pertencentes ao sistema através de 3 comandos principais (e 1 secundário), sendo eles:

```
$ ma
i <nome> <preco> (insere novo artigo, mostra o codigo)
n <codigo> <novo nome> (altera nome do artigo)
p <codigo> <novo preco> (altera preco do artigo)
a (efetua a agregacao das VENDAS)
...
<EOF>
```

Segue-se a descrição da organização dos ficheiros de dados ARTIGOS [22 bytes/linha], STOCKS [11 bytes/linha] E STRINGS [nº de bytes variáveis/linha]:

```
Codigo | <Referencia> <Preco> [ARTIGOS]
Codigo | <Nome>               [STRINGS]
Codigo | <qt_stock>          [STOCKS]

1|
```

Aqui o Codigo representa a linha no ficheiro [Codigo fixo do artigo], a referência representa a linha onde se encontra no ficheiro strings [Referencia variavel mas inicialmente igual ao codigo] e por fim o preço e quantidade de stock são propriedades do artigo.

É de realçar que os tamanhos fixos de todos os ficheiros (exceto o STRINGS) permitem uma leitura/escrita atómica através de *system calls: read e write.*

Nota: tal não acontece com o ficheiro de nomes o que implica a utilização de uma função que faz a leitura byte a byte de uma string input.

Em relação ao comando "a"é de referir que o pedido de agregação só é realizado se o servidor estiver *online*, ou seja, em execução. Caso contrário o pedido será negado e uma mensagem de erro será impressa no STDERR do "ma".

2.2 Servidor de vendas

Este é o *core* do sistema de vendas, recebe pedidos de clientes através de um pipe comum a todos os clientes e envia respostas através de um pipe indiviual. Tal se processa utilizando o **Process ID** do cliente como sendo o seu identificador **único** que é enviado em conjunto com o seu pedido sendo a respetiva resposta enviada por um pipe por cliente.

Visto que o servidor responde a um conjunto de pedidos dos clientes, tal será importante ser explicitado primeiro, passemos a descrição para o **cliente de vendas**.

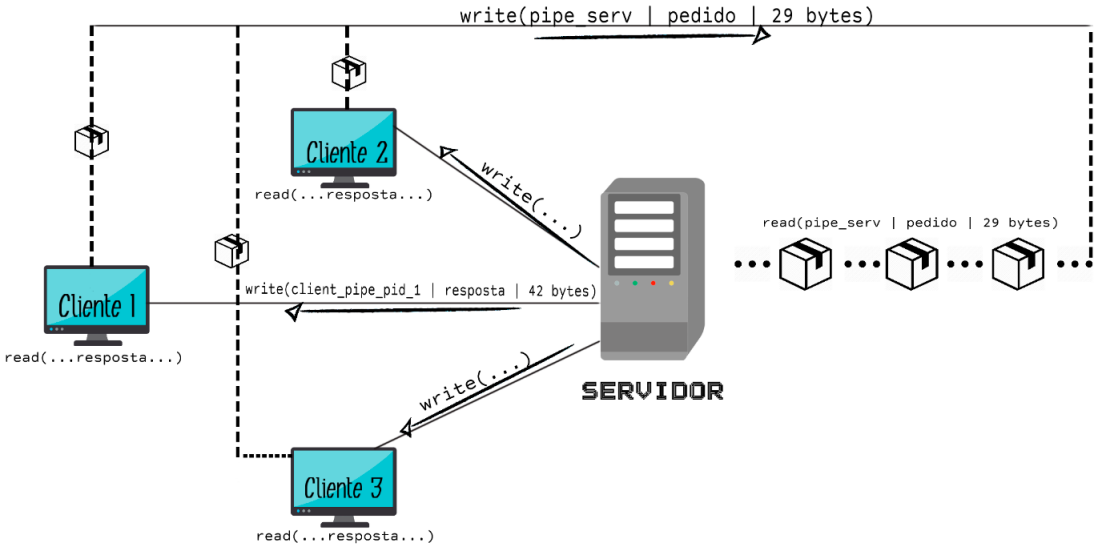

Figura 2.1: Representação do sistema cliente-servidor.

2.2.1 Cliente de Vendas - Comandos permitidos

Um cliente deste sistema de vendas tem a possibilidade de visualizar stock e preço de um artigo e também de alterar a quantidade de stock (aumentando ou diminuindo o mesmo), sendo que uma diminuição de stock implica um registo de uma venda. Segue-se a descrição dos comandos possiveis.

```
$ cv
<codigo> (mostra no stdout stock e preco)
<codigo> <quantidade> (actualiza stock e mostra novo stock)
...
<EOF>
```

De forma geral, um cliente introduz comandos a partir do seu STDIN e recebe as repostas no seu STDOUT (formatadas de forma diferente dependendo do pedido).

2.2.2 Pedidos e Respostas

Os pedidos são enviados para o servidor por um *named pipe* (FIFO - *first in first out*) e são processados, um a um. Eis alguns exemplos de *buffers* contendo pedidos de clientes enviados ao servidor:

```
//Codigo que gera o pedido
sprintf(pedidoBuffer, "%07d %010d %010d", clientPID, codigo, quantidade);

//Exemplo de pedido
//PID CODIGO QUANT
```

Cada pedido tem um tamanho fixo de 29 bytes escritos e lidos atomicamente. Seguem-se exemplos de repostas do servidor:

```
sprintf(bufferEscrita, "0 %07d %010d %010d %010d", clientID,
codigoProduto, quantidadeStock, precoLido);

//Aqui o primeiro byte, o "0",
// representa um dos tipos de resposta
//Chegando ao cliente, mediante essa flag,
// este sabe que resposta apresentar.
```

Cada resposta utiliza 42 bytes. Todas estas constantes, *defines* e *paths* estão presentes no ficheiro "global.h"[1] da diretoria GLOBAL-SOURCE.

2.3 Agregador

O agregador utiliza os registos de vendas presentes no ficheiro VENDAS e coleciona vendas agregando a quantidade vendida e o montante (preco x quantidade comprada) numa mesma entrada de um novo ficheiro gerado (cujo nome é dado pela data:hora:min no momento de agregação).

A implementação passou pela criação de uma HashTable[2], cuja *key* é o codigo do produto, após *parsing* de uma venda e o respetivo value é composto pela quantidade comprada e montante de venda.

As entradas que têm *hashing* para a mesma posição agregam-se num lista de registos de venda sendo posteriormente convertidas apenas numa venda (agregada). A imagem seguinte descreve a estrutura implementada:

[1] Source que contém uma coleção de funções auxiliares ao projeto e constantes importantes.
[2] São usadas estruturas otimizadas, GLib.

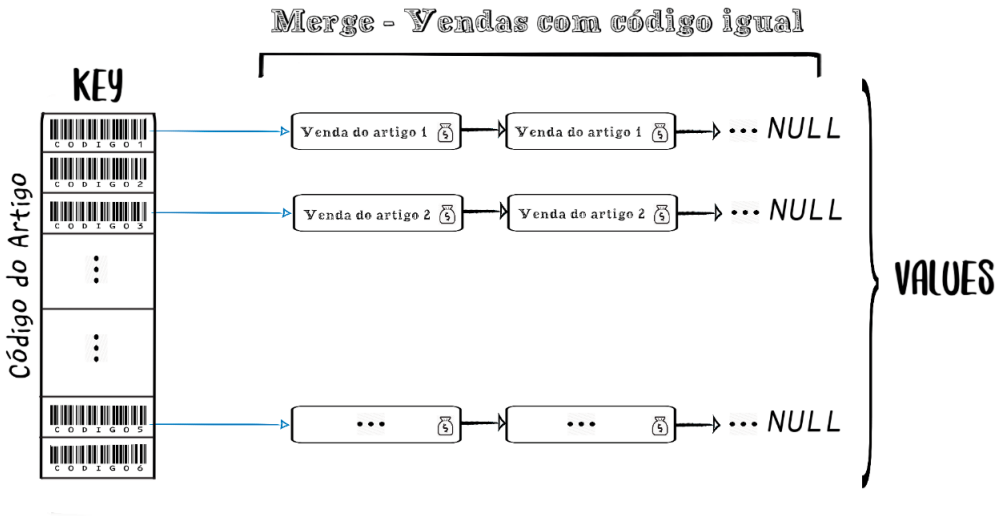

Figura 2.2: Estrutura utilizada no agregador.

A importância da criação desta HashTable passa pela procura de otimização do processo de agregação devido à sua procura constante *O(1)*.

2.3.1 Agregação concorrente

Para este tipo de agregação tem se em consideração a criação de processos que efetuam a agregação de partes do ficheiro VENDAS, divindo o trabalho de forma equivalente. Ora isto traz a noção de paralelismo ao projeto, apesar de que a execução não é necessariamente paralela (Devido à gestão de processos do Sistema Operativo) mas providencia um melhoramento de performance ao trabalho. Mais à frente será descrita a implementação adotada.

Quanto à implementação a solução é simples, trata-se de arranjar uma forma de dividir trabalho e criar processos q.b. para agregar concorrentemente. O grupo decidiu estabelecer um limite inferior de linhas para criação de processos com *fork()* e, pela experiência, concluimos que 5000 seria um numero razoável devido aos testes que o programa poderá estar sujeito.

Um exemplo muito custoso (testado inicialmente pelo grupo) seria colocar um limite de 1000...Num caso de termos 150 mil registos venda o programa criaria 150 processos em concorrência para efetuarem a agregação, o que, rapidamente, se tornaria impossivél.

No capítulo **"Valorizações"** apresenta-se uma representação visual da agregação.

Capítulo 3

Valorizações

As valorizações descritas no enunciado foram tidas em consideração visto que foram uma oportunidade de revelar os conhecimentos adquiridos e explorar diferentes conceitos aparentemente "teóricos"mas que puderam ser aplicados na prática.

3.1 *Caching* de preços

A cache de preços foi implementada e apresenta uma estrutra simples. Trata-se de um array, de 10 posições, cujas células[1] contêm estruturas que armazenam os artigos.

Funções para operar sobre a cache, fazer *lookup*, *add*, *init*, etc...estão presentes no respetivo módulo **"cache.c"**.

Segue-se uma pequena descrição, visual e estrutural, da cache:

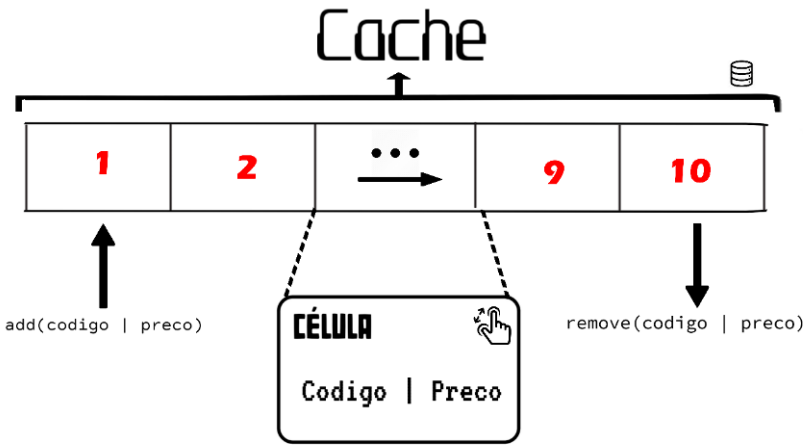

Figura 3.1: Cache de preços.

```
struct celula {

        int codigo;
        int preco;

};
```

[1] posições do array

```
struct cache {

        CELULA *cache;

        int next;
        int max;
        int nr_elementos;
};
```

O algoritmo de gestão da cache, ou seja, de remoção de elementos é feito removendo o último elemento, ou seja, o primeiro de todos inserido visto que devido à aleatoridade dos pedidos de vendas não é muito relevante ter algoritmos especiais e/ou *hashs* complicados, sendo que, um só será removido caso sejam introduzidos 10 novos códigos para a cache, mas, se um produto estiver a ser continuamente requisitado este não será removido.

No nosso entender, não faz sentido ter *counters* com o número de vezes que um código foi requisitado pois caso este deixe, por aleatoridade, de ser usado ficará na cache sem ser usado, ocupando um espaço importante **numa cache tão reduzida**.

Isto reduz assim o número de vezes que o servidor recorre aos ficheiros de dados, diminuindo operações custosas ao nivél do Sistema Operativo de *buffering*, através de *system calls* com operações de *read* e *write*.

3.2 Agregação concorrente

Segue-se a representação referida anteriormente:

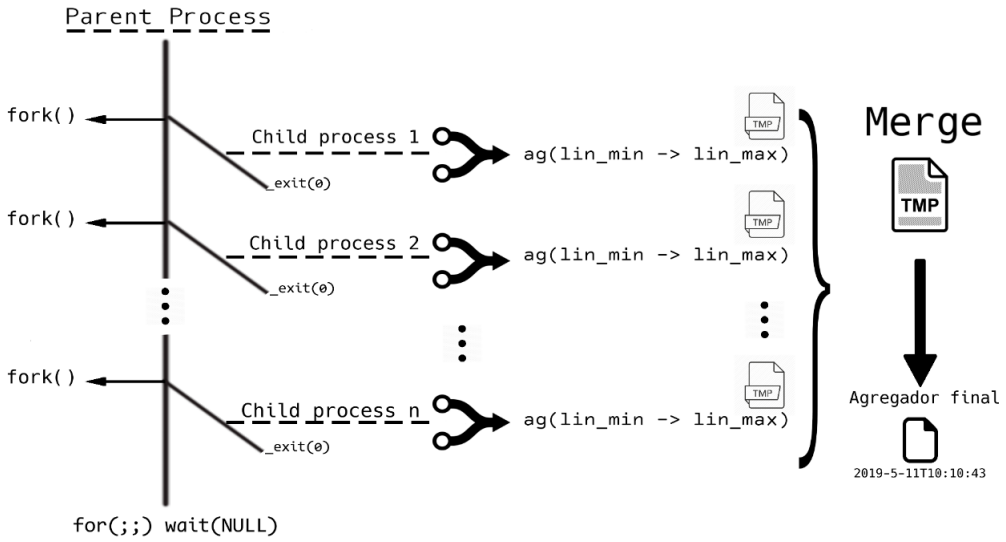

Figura 3.2: Processo de agregação concorrente.

3.3 Compactação do ficheiro STRINGS

A compactação deste ficheiro surge, de forma clara, devido à possibilidade de alteração de nomes de artigos na *manutenção de artigos*. Com isto criam-se entradas **obsuletas** que podem ser perfeitamente compactadas.

Para resolver este problema é importante esclarecer dois pontos: calcular a percentagem de **lixo** no STRINGS e **compactar**.

Para calcular a percentagem de lixo apenas dividimos o número de linhas do ficheiro ARTIGOS pelo número de linhas do ficheiro STRINGS, subtraímos por 1 e obtemos a percentagem requerida.

Ora o processo de compactação também é simples, começamos por renomear os ficheiros de ARTIGOS, STRINGS e STOCK para nomes temporários. De seguida cria-se ficheiros novos com os nomes originais. Percorremos o ARTIGOS e inserimos novamente a associação código ¡-¿ nome ¡-¿ preco utilizando as funções já definidas no programa **ma** que geram novas referências para os códigos.

Reconhece-se que se trata de uma solução não completamente dentro do mundo de Sistemas Operativos, mas consideramos que tiramos partido de performance e gestão de memória, visto que, após compactação todo o lixo é removido e não são criadas estruturas desnecessárias, como *hashtables* ou *linked lists* para armazenar tudo em memória.

Capítulo 4

Notas relevantes

4.1 Compilação do projeto

Este projeto possui uma *Makefile* presente no src/ que, conjuntamente com outras *Makefile* secundárias, efetuam a compilação de todo o projeto, criação de diretorias onde serão armazenados ficheiros *object* e *linking* com estruturas **GLib** o que implica a pré-instalação destas bibliotecas.

4.2 Bibliotecas implementadas

Como já foi referido anteriormente, todas as estruturas auxiliares, **cache**, **artigos**, constantes e *paths* importantes encontram-se armazenadas na diretoria *GLOBAL-SOURCE*, assim como a maior parte de funções auxiliares presentes nos programas implementados.

4.3 *Tracing* de erros e debugging

Este projeto revelou-se algo complicado a nivél de deteção de erros e *debugging* e, de modo a facilitar isso mesmo, para além do *report* dado pelo **stderr** o grupo efetuou um registo de todos os erros relativos a comandos inválidos, números *out of range*, etc.. guardados no ficheiro "/FILES/ERRORLOG.log".

Segue-se uns exemplos de erros detetados:

```
$cat ERRORLOG.log

[./cv] client inserted an invalid command.
[./ma] invalid command.
[./cv] client requested code that does not exist.
[./cv] client requested stock update to negative.
...

<EOF>
```

4.4 Sinais - aplicações neste projeto

Existem algumas aplicações de sinais durante o código, nomeadamente, pelo servidor.

Acontece aquando a terminação do processo do servidor quando este recebe um SIGINT e, ao invés de morrer de imediato, "arruma a casa", isto é, elimina todos os ficheiros desnecessários/temporários, registos de clientes *"loggados"* e termina o processo de todos os clientes que possam estar ligados ao servidor, enviando um sinal para os mesmos (um SIGINT também).

Capítulo 5

Considerações finais

Este projeto descreve uma extrema importância no olhar que todos os programadores devem ter acerca do mundo *Out-of-core* que aparentemente é mais relevante do que saber escrever umas linhas de código.

A noção de como é feita a gestão de memória e de procesos foi crucial na perceção e atenção no código que pode ser escrito e na noção que apesar de tudo, nada é garantido, por exemplo, a partir de um simples *printf(...)*.

Podia ter sido melhor considerados fatores de segurança devido a permissão de ficheiros para utilização/alteração/remoção e podia ter sido economizada memória em certas operações.

Existem *writes* no *stdin* presentes no servidor/cliente desnecessários, usados para o utilizador do sistema ter a noção da existência de uma cache, de pedidos ao servidor para atualizar precos, stocks, entre outros.

Apesar de tudo, o grupo considera que o *core* do trabalho foi implementado com sucesso e responde de uma forma correta a todas as questões propostas.