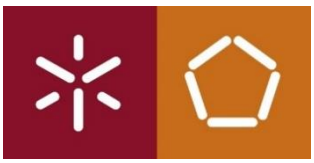
Universidade do Minho



Departamento de Informática

Sistemas Operativos

Gestão de Inventário e Vendas

Grupo Número 70

Diogo Nogueira a78957 - Diogo Araújo a78485 - Hugo Moreira a43148

Relatório Projeto

Índice

1.Introdução

O presente relatório será redigido como suporte ao projeto da Unidade Curricular de Sistemas Operativos que se baseia na criação de um protótipo de sistema de gestão de inventário e vendas que permita um controle de stock eficiente e adaptado aos pedidos efetuados pelos vários clientes.

De modo a facilitar/otimizar todo o algoritmo de resolução de requisitos, o enunciado enumera os programas a desenvolver e quais os ficheiros que devem depois ser usados para armazenar as informações dos artigos em si e igualmente as suas vendas. Este esquema apresenta como ideia base a criação e armazenamento de novos artigos que podem depois estar sujeitos a vendas e alterações de preço e nome. Apesar de todos estes programas atuarem sobre um objetivo comum, encontram-se individualizados em termos de tarefas, que estão distribuídas consoante o nível de prioridade que cada "entidade" deve possuir sobre a totalidade do programa.

Foi perante este conceito que se criaram os vários programas – a manutenção de artigos, que permite uma manutenção inicial dos artigos existentes para venda/compra, estabelecendo-se os seus preços e nomes; o Cliente e Servidor, dois programas que trabalham lado a lado e que juntos se conectam (criando uma ponte entre as necessidades impostas pelos clientes e a possibilidade de isso se efetivamente processar); o agregador, que produz o conjunto de vendas efetuadas para os demais artigos indicando o montante total dessas mesmas vendas.

Todo o processo de gestão de inventário e vendas é então efetuado através do conjunto de ficheiros que contêm a informação sobre o sistema atual – artigos existentes, o seu preço, stock e nome e vendas efetuadas ao longo do tempo. É através deles que se devem retirar as informações antes e após cada operação, atualizando os clientes sobre o resultado dos seus pedidos. Estes ficheiros encontram-se formatados de modo a que os dados possam ser armazenados de forma coerente e que a informação guardada faça sentido, para futura leitura/carregamento da informação (operação que será efetuada por múltiplas vezes durante a execução dos programas). Tendo a essência do programa em si definida, todo o resto do programa consegue ser resolvido e responsivo em termos de requisitos obrigatórios para o Gestor de Vendas.

2.Implementação

2.1. Manutenção de Artigos

Este primeiro programa do sistema de gestão de inventário e vendas serve como impulsionador de todo o restante tratamento de dados. Através dele são feitas as inicializações das informações dos artigos que serão depois usados como referência para compra e venda.

Note-se que este programa está preparado para receber um ficheiro de comandos através do uso do sinal <.

2.1.1. Estrutura de dados

Para guardar a informação relativa a um artigo surgiu a necessidade da criação de uma estrutura de dados, que contém a informação relativa a um artigo.

```
struct Artigo {
    int codigoArtigo;
    double precoArtigo;
};
```

Esta criação da estrutura foi pensada e implementada dada a necessidade da escrita fixa e concreta em termos do tamanho em *bytes* da estrutura. Assim daqui em diante, a escrita para os diversos ficheiros que guardam a informação dos artigos, vendas e stocks têm como base uma estrutura fixa com variáveis fixas também. Assim, foi possível inserir e, mais concretamente, alterar o preço dos artigos sem que ocorresse problemas associados caso a escrita fosse em texto, ou seja, *array* de caracteres. Isso mostrou-nos o problema de que se o preço anterior tivesse um certo tamanho já alocado no ficheiro, ocorreria uma sobreposição ou um "buraco" caso o novo preço tivesse tamanho diferente. Ao optar por esta implementação de estrutura com variáveis fixas abriu-se a possibilidade de utilizar posteriormente funções capazes de apontar para o local correto para existir a alteração dos valores destas variáveis.

Por pensamento análogo, a escrita dos nomes dos artigos no ficheiro apropriado não requisitou a necessidade duma estrutura para o mesmo. Dado que o ficheiro dos nomes poderia conter nomes de artigos obsoletos/antigos, assumiu-se que a última escrita do nome daquele código de artigo era o seu nome mais recente, logo a última entrada no ficheiro *strings.txt*.

2.1.2. Algoritmo de Resolução

O *source code* deste primeiro programa teve de ser logo pensado para ser responsivo em termos de leitura via *stdin* (como referido desde início), tanto manualmente como através da leitura de um ficheiro de texto que contivesse um conjunto de comandos pré-preparados para leitura. Só com esta base inicial implementada é que se consegue garantir que depois todo o teste de resultados seja o mais rápido e otimizado possível. Foi a partir desta implementação inicial que se conseguiu então desenvolver o restante código da manutenção de artigos, concretizando de uma melhor forma aquilo que foi proposto no enunciado.

Leitura caracter a caracter
(Caracter '\n' é o que identifica o fim de um comando)

↓

Filtragem/interpretação do comando

```
void interpretaComandoMA(char *buffer)
```

Função que filtra de que comando se trata. Aqui faz-se uso da função ***strtok*** da biblioteca do C que delimita uma *string* com base em um delimitador. O delimitador passa pelo caracter espaço (no caso dos comandos base, como **i nomeArtigo precoArtigo**) e pelo caracter mudança de linha (no caso do agregador a pedido, comando **a**).

↓ **se "i"**

```
void insereArtigo(char *nomeArtigo, double precoArtigo)
```

↓ **se "n"**

```
void alteraNomeArtigo(int codigoArtigo, char *nomeArtigo)
```

se "p" ↓

```
void alteraPrecoArtigo(int codigoArtigo, double precoArtigo)
```

Perante a filtragem do comando, chamam-se as devidas funções que depois tratam de efetuar todo o restante do processo. Em termos de ficheiros, esta primeira fase apenas necessita de listar os artigos existentes (linha a linha) juntamente com o seu preço num ficheiro de nome *artigos.txt*. O nome de cada artigo é mencionado num ficheiro à parte de nome *strings.txt* que conterá a posição que identifica o produto em si. Assim, por cada inserção que se faça para um artigo, acrescenta-se uma nova linha no ficheiro *strings.txt* contendo esse mesmo código identificador e o seu nome. (Dada essa ideia, sabe-se que o nome mais recente é sempre o mais próximo do END OF FILE).

- A função `void insereArtigo(char *nomeArtigo, double precoArtigo)` trata da escrita em ambos os ficheiros. Dados os parâmetros obtidos na filtragem de informação de cada comando, cria-se uma `struct Artigo` que depois de "povoada" é escrita no ficheiro dos artigos em modo O_APPEND. Já para o ficheiro das *strings*, apenas se insere uma *string* normal contendo o código e o nome (não existindo a tal necessidade em manter um tamanho fixo por cada entrada/adição).

Em termos de algoritmo, esta função verifica se o ficheiro *artigos.txt* já existe, e caso já exista, trata de obter o código do último artigo inserido, de modo a conseguir calcular o novo código. Este posicionamento em termos de ficheiro consegue-se pela função `lseek(ficheiroArtigos, -sizeof(struct Artigo), SEEK_END)` e de um modo muito mais simples dado que estamos a lidar com linhas de tamanho fixo, facilitando-se assim na sobreposição de escrita.

- A função `void alteraNomeArtigo(int codigoArtigo, char *nomeArtigo)` escreve apenas no ficheiro *strings.txt* o código + nome. Sempre no final do ficheiro (seguido do caracter de mudança de linha).

- A função `void alteraPrecoArtigo(int codigoArtigo, double precoArtigo)` recorre também ao posicionamento do apontador em termos de leitura ficheiro. Assim que encontra o artigo cujo preço pretende ver alterado (através do *match* de código de artigo), posiciona o apontador para o início dessa estrutura antiga, inserindo aí a nova estrutura com os dados mais recentes.

Adicionalmente, a manutenção de artigos permite que o programa em si comunique com o servidor de vendas, solicitando-lhe uma agregação a pedido que aplicaria as vendas apenas

até ao momento em que a agregação é efetivamente solicitada. Esta adição fez proveito da fase de interpretação de comandos, mandando pelo *pipe* (usado pelo Cliente-Servidor) uma estrutura própria criada para o efeito e que permite ao servidor identificar cada pedido recebido **(percebe-se mais à frente de que estrutura se está a falar em termos de escrita e leitura para o *pipe*)**.

2.2. Cliente e Servidor de Vendas

O cliente de vendas relaciona-se diretamente com o servidor de vendas, uma vez que depende desse para ver em prática todas as suas solicitações. É através de um *pipe* com nome, criado pelo programa do servidor, que o cliente consegue enviar os seus pedidos, ao mesmo tempo que o servidor os interpreta e executa consoante vão entrando pelo "canal" em si.

Em termos de algoritmo, o programa servidor de vendas assemelha-se bastante à manutenção de artigos, no sentido em que tem de existir uma pré-filtragem de comandos e os seus respetivos parâmetros e só depois a seleção da função perante o que o comando pretende ver processado (inclusive, esta filtragem faz uso da mesma função ***strtok***).

Note-se que o programa do cliente está também adaptado para receber um ficheiro de comandos com o uso do sinal <. Relativamente ao output em si, o grupo decidiu ir colocando num ficheiro *log.txt* sempre que o stock era atualizado ou pedido, facilitando a verificação final dos resultados obtidos e poupando *printf* desnecessários na parte do terminal do servidor.

2.2.1. Estrutura de dados

A estrutura `Artigo` usada logo na manutenção de artigos é proveniente do *file header* correspondente ao Servidor de Vendas, que adicionalmente possui mais três estruturas essenciais para o funcionamento do sistema.

```
struct Artigo {
    int codigoArtigo;
    double precoArtigo;
};
```

```
struct Venda {
    int codigoArtigo;
    int quantArtigo;
    double montanteTotal;
};
```

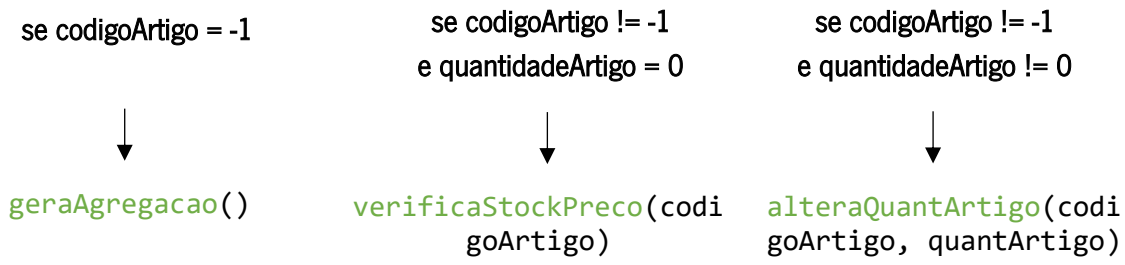
```
struct ArtigoStock {
    int codigoArtigo;
    int quantArtigo;
};
```

Dado que era suposto existir um ficheiro *stock.txt* contendo a quantidade atual dos vários artigos assinalados e tendo em conta que esta quantidade ia sofrer alterações sempre que houvesse uma compra/venda, o pensamento de escrever no ficheiro em modo estrutura teve também de estar presente, daí a criação da `struct ArtigoStock` que consegue facilmente resolver este requisito.

Refletindo-se já sobre última fase deste projeto, teve de se ter em atenção que o papel do agregador seria interpretar todas as vendas registadas, agrupando-as de acordo com o artigo a que pertenciam, a quantidade total vendida e o montante obtido com tudo isso. Caso a escrita no ficheiro a *vendas.txt* fosse feita como no ficheiro *strings.txt*, em que apenas se inseria linha a linha um novo registo, depois seria muito mais difícil e custoso ler e consecutivamente agrupar todos estes dados. Assim, com recurso a uma outra estrutura, consegue-se mais espontaneamente retirar os códigos e quantidades dos vários artigos, facilitando depois o processamento de agregação de informação.

Finalmente, e falando agora em termos de leitura/escrita no *pipe*, sabia-se que esse processo se tornaria mais intuitivo e linear se os dados que estivessem a circular pelo canal fossem todos do mesmo tamanho – solução obtida mais uma vez através de uma estrutura criada a pensar nos vários comandos que o Servidor de Vendas teria de interpretar.

```
typedef struct comando {
    int codigoArtigo;
    int quantidadeArtigo;
} Comando;
```



Aqui percebe-se que a `struct comando` foi construída de modo a poder ser universal tanto para os comandos da manutenção de artigos como para os comandos do cliente de

vendas. Não só a estrutura possibilita uma troca de dados uniforme via *pipe*, como simplifica a interpretação/filtragem dos parâmetros referente a cada comando, tendo em conta que se consegue logo recolher tanto o código do artigo como a sua quantidade (no caso de se querer atualizar o stock tanto em termos de venda como compra).

2.2.2. Algoritmo de Resolução

Os algoritmos desenvolvidos para esta interação Cliente-Servidor foram idênticos aos da manutenção de artigos, já que as operações de compra e venda nada mais são que uma atualização do ficheiro *stock.txt* (que está por si só formatado também em termos de estruturas) e consequentemente do ficheiro *vendas.txt*.

- A função `void geraAgregacao()` faz uso de bibliotecas que permitem adquirir o *LocalDateTime* atual para que depois se possa produzir o ficheiro de texto correspondente a cada agregação.

Visto que um dos requisitos deste sistema passa por desenvolver um programa que faça a agregação das vendas, uma agregação a pedido consiste basicamente em:

1. Obter o *LocalDateTime* que servirá depois como nome do ficheiro para onde a agregação será impressa.

```
time(&tempoAtual);
informacao = localtime(&tempoAtual);

strftime(datePlusTime,sizeof(datePlusTime),"%FT%T",
informacao);

nome = strdup(datePlusTime);
strcat(nome, ".txt");
```

2. Criar um processo filho a partir do processo pai atual (processo programa servidor) dado que se invocássemos a *system call* **execlp** nesse processo principal, o programa servidor morreria, podendo causar problemas caso os clientes de vendas estivessem ainda a tentar escrever através do seu *pipe*.

```
pid_t pid = fork();
```

3. Ao identificar-se o processo filho, faz-se o uso do redireccionamento de descritores de ficheiros, permitindo assim que tudo o que originalmente estaria a ser mostrado no *stdout*, é redirecionado para o ficheiro de texto (com o *LocalDateTime*) que realmente se pretende ver criado. Após se fazer isso e se executar o programa do agregador, fecha-se o processo filho, permitindo que o servidor continue a executar normalmente o seu trabalho.

```
if(pid==0){
    int ficheiroOutput = open(nome, O_CREAT|O_RDWR|O_APPEND,
0777);
    dup2(ficheiroOutput, 1);
    execlp("./ag", "./ag", "vendas.txt", NULL);
    _exit(0);
}
```

- A função `void verificaStockPreco(int codigoArtigo)` trata o ficheiro de igual forma à maioria das funções já abordadas. Neste caso, precisa de percorrer tanto o ficheiro *artigos.txt* para retirar o preço associado ao artigo daquele código e depois percorrer o ficheiro *stock.txt* conseguindo o nível de unidades existentes para esse mesmo artigo.

- A função `void alteraQuantArtigo(int codigoArtigo, int quantArtigo)` desempenha um papel mais dual dado que ao mesmo tempo que altera a quantidade de um determinado artigo, tem de perceber se a quantidade em si é um valor negativo para que depois possa ser inserida uma nova venda no ficheiro *vendas.txt.* Isso exige a existência de uma função `void insereNovaVenda(int codigoArtigo, int quantArtigo, int novaQuantArtigo)` que seja invocada assim que se perceba que uma venda está a ser feita - **Através de uma simples comparação com o número 0. Se a quantidade é maior que esse 0 então é positiva e estamos a entrar com *stock*. Se a quantidade é menor que esse 0 então é negativa e estamos a retirar *stock*.**

Fora estas funções principais, a ligação Cliente-Servidor funciona de forma muito simples. Do lado do Cliente, por cada comando/pedido que é solicitado, existe uma interpretação prévia que filtra a informação desse comando e o transforma na `struct comando` acima abordada. Essa struct é enviada pelo *pipe* e recebida pelo Servidor. Quando o Servidor recebe, interpreta esse comando mais facilmente (uma vez que está já em modo estrutura) e aplica então uma destas três funções consoante o pedido.

Resta referir que todos os algoritmos desenvolvidos para as funções presentes no Cliente de Vendas como no Servidor, fazem uso também das funções usadas para a manutenção como o `lseek` e os vários *open* e *write* para os ficheiros em si.

2.3. Agregador de Vendas

Responsável por ler e iterar o ficheiro *vendas.txt* (formatado em modo de estruturas), agrupando as vendas de cada artigo de modo a somar as quantidades vendidas e o montante total de todas essas vendas.

Note-se que o programa do agregador está adaptado por colocar o resultado do seu output num ficheiro de texto através do sinal >, ou simplesmente guardar num ficheiro com o *LocalDateTime* no caso do agregador a pedido.

2.3.1. Estrutura de Dados

Para a implementação do agregador de vendas, pensou-se na criação de uma lista ligada auxiliar em que contivesse os vários (códigos de) artigos e assim conseguir contabilizar e agregar a quantidade e montante total vendido até então. Dessa maneira, não se criou com *hashcode* dado que no projeto cada artigo já tinha um código único a identificar o mesmo.

```
typedef struct vendas {
        int codigo;
        int quantidadeTotal;
        double montanteTotal;
        struct vendas *proximaVenda;
} RegistoVenda, *Vendas[SIZE];
```

Como está supracitado, a criação desta estrutura facilita a junção de toda a informação que será lida no ficheiro *vendas.txt*, para depois haver o *output* apropriado da agregação de todas as vendas dos diversos artigos até então.

2.3.2. Algoritmo de Resolução

Para a algoritmia pensada para o agregador houve a utilização de toda a matéria inicial dada como preparação para a Unidade Curricular, em que se manipula uma lista ligada, apontadores e contadores de forma a conseguir obter a agregação necessária e pedida.

Quando o agregador corre pela primeira vez, existe uma inicialização inicial da estrutura falada anteriormente, sendo que para o tamanho predefinido (10000 artigos com vendas) acontece uma inicialização dum nodo com todos os seus elementos vazios e a sua posterior colocação na lista ligada, para que desta forma exista a alocação correta antes da leitura do ficheiro *vendas.txt* e a sua interpretação.

```
int ficheiroVendas = open("vendas.txt", O_RDONLY, 0666);

    while(read(ficheiroVendas, &vendaLida, sizeof(struct Venda)))
interpretaVenda(vendaLida);
```

Utilizando a estrutura Venda falada anteriormente na secção do Cliente e Servidor, existe a leitura do ficheiro *vendas.txt* em que a cada venda lida, existe a sua interpretação sendo que se baseia em verificar se o artigo da venda lida do ficheiro já se encontra na lista ligada da agregação. Ao existir a comparação e pesquisa pelo código de artigo e pelo seu nodo equivalente, existem duas opções. No caso de ser a primeira vez em que este código de artigo está a ser agregado cria-se um nodo para o mesmo com os valores da quantidade e montante lidos do ficheiro. Por fim, adiciona-se esse nodo à lista ligada e continua-se a analisar o resto do ficheiro das vendas de artigos. No caso de já existir contabilização sobre o artigo vendido, apenas se soma às quantidade e montante existentes, os novos valores, agregando assim corretamente a informação das vendas desse artigo.

Na consequência da leitura de todo o ficheiro *vendas.txt* ocorre uma última iteração por toda a lista ligada de forma a escrever, para o *stdout*, a contabilização de cada artigo que teve pelo menos uma venda efetuada até então e irá mostrar todas as quantidades adquiridas e também o montante monetário das mesmas.

3. Considerações Finais e Discussão de Resultados

Após a finalização do projeto pedido e análise do mesmo, demonstrou-se que é possível assimilar e demonstrar as diversas vertentes que a Unidade Curricular leciona e criar um programa capaz de suster uma gestão eficaz de inventário e vendas. Através dos mais diversos métodos, tais como a utilização de processos-filhos, à necessidade da criação de *pipes* com nome até à utilização de estruturas e a manipulação de ficheiros ao nível mais baixo possível não só da linguagem C, mas também sobre todo o entendimento teórico de Sistemas Operativos, conseguiu-se produzir uma solução final que atendesse ao máximo de requisitos pedidos.

Apesar de toda o conceito do funcionamento do sistema estar desde início explícito, existem algumas considerações finais que retratam as dificuldades superiores deste projeto e de como o grupo as procurou solucionar. Em termos de funcionalidades base, o sistema funciona, considerando que é possível listar os vários artigos, os seus nomes e preços, para que depois possam existir compras e vendas solicitadas pelos vários clientes que comuniquem com o servidor. Assim, consegue-se atualizar *stock* e registar vendas de modo a que exista depois uma agregação fiável e de acordo os artigos vendidos e quantidade/montante total de cada um desses. Este cenário de implementar um sistema que depositasse toda a informação em ficheiros, obrigou a ter em mente as permissões de escrita e leitura para as várias entidades, tendo em conta os programas que precisariam de aceder aos vários ficheiros para o processamento da informação. Dada a necessidade de acesso por parte de todos os programas, deram-se permissões de leitura e escrita para o *owner*, *group* e outros (0666), garantindo assim que a informação poderia ser lida e alterada consoante a necessidade atual do sistema.

Estas aprendizagens obtidas pela frequência das aulas práticas tiveram de ser fortificadas pela teoria em si. O grupo sabia que a imposição de concorrência dos múltiplos clientes de vendas precisaria ser resolvida com recurso a métodos de proteção das zonas críticas do código, assegurando que não existisse mais do que um processo a trabalhar sobre a mesma operação. Dessa forma, e dado que se implementou a ideia de se criar um processo filho sempre que o servidor recebida um comando vindo do cliente, teria de ser implementado um mecanismo de sincronização como semáforos para que existisse então esse controlo em termos de *stock*. Ainda que existindo o conhecimento deste requisito, o grupo não obteve sucesso a tentar implementar o mesmo, ficando assim o programa sujeito a algumas quebras de *stock* quando se tentam aplicar vários clientes.

Ainda assim, o grupo acredita ter feito um bom trabalho dado que aplicou a maioria dos conhecimentos oferecidos pela Unidade Curricular em si. O balanço é positivo, acreditando sempre numa melhoria em termos de resolução e obtenção de resultados.