Gestão de Vendas

Docentes:

Paulo Sérgio Soares Almeida

António José Pinheiro Coutinho

Discentes:

Bruno Sousa, A84945

Filipe Freitas, A85026

Ricardo Cruz, A86789

Universidade do Minho
Licenciatura em Ciências da Computação
Sistemas Operativos
2018/2019

Índice

1 Estrutura

A fonte do projeto está organizada de forma bastante simples, onde cada executável tem o seu diretório específico, contendo o *main.c* correspondente e a definição de métodos/funções específicos ao funcionamento do programa.

O diretório *"common"* contém as definições de estruturas e de funções utilizadas com mais frequência e que são partilhadas ao longo de todo o projeto.

2 Organização e Metodologia

Tentou-se fazer com que a nomenclatura, ao nível do nome dos ficheiros da fonte, mas também ao nível do nome das funções/métodos e estruturas, fosse coerente e sugestiva e que esta assim se mantivesse ao longo do desenvolvimento.

No mesmo sentido, tentou-se documentar, ao máximo possível, todas as funções juntamente com o seu código, ou no seu *header* file específico. Conseguimos deste modo garantir que quaisquer dúvidas que surjam, poderão normalmente ser resolvidas com consulta à documentação, que pode ser consultada no código ou então gerada, assumindo que o *Doxygen* está já instalado no sistema, com o comando *"make doc"*. O código em si também tem comentários úteis que podem ajudar a entender o porquê de algumas decisões tomadas, ou qual é o propósito de certos blocos de código cujo propósito possa não ser aparente.

Além disso, investiu-se grande parte do tempo de desenvolvimento do na produção de uma *"API"* que possibilitasse uma maior abstração durante a elaboração dos sistemas específicos ao tema da elaboração de um sistema de gestão de vendas/artigos. Isto permitiu melhorar a produtividade de todos os membros do grupo que, depois de terem estudado o funcionamento interno da *"API"* e as suas possíveis aplicações nos diversos sistemas a implementar, puderam concentrar-se com maior facilidade na elaboração dos sistemas que realmente importam num sistema de gestão de artigos/vendas.

Finalmente, desde o início do desenvolvimento do trabalho que a prevenção e deteção de erros é considerada deveras importante. Assim, a implementação de

sistemas capazes de reagir a potenciais erros ou bugs, mantendo os programas num estado estável, foi uma das principais metas.

3 Instruções de Compilação

De seguida apresentam-se os possíveis comandos a executar, com as suas devidas *flags* e argumentos, para se proceder à compilação dos executáveis:

- **make setup** - Prepara a estrutura do projeto;
- **make** $[debug = true\,[/false]]$ - Constrói todos os binários (ma, sv, cv, ag);
- **make** $\langle ma/sv/cv/ag\rangle$ $[debug = true\,[/false]]$ - Constrói o binário especificado;
- **make** $\langle ma/sv/cv/ag\rangle$ **run** - Constrói (se necessário) o binário especificado e depois executa o mesmo;
- **make** $\langle ma/sv/cv/ag\rangle$ **run-gdb** - Constrói (se necessário) o binário especificado e depois executa o mesmo através do *gdb*;
- **make** $\langle ma/sv/cv/ag\rangle$ **run-strace** - Constrói (se necessário) o binário especificado e depois executa o mesmo através do *strace*;
- **make clean** - Limpa os ficheiros resultantes de builds anteriormente efetuadas. É necessário depois fazer *"make setup"* para restaurar a estrutura do projeto.

4 Executáveis Disponíveis

Importante: Todos os executáveis estão disponíveis em $bin/\langle production/debug\rangle/$, dependendo do tipo de *build* executada.

É recomendado executar a *build* com *debug=false* (que é o *default*, se a *flag debug* não for sequer passada), para ter mais performance. Neste caso, os executáveis estão em *"bin/production/"*.

4.1 Manipulador de Artigos (MA)

Invocação na *bash*: "./ma".

Durante a sua execução, o *ma* fica à espera do *input*/comandos do utilizador, guardando-o num *array*, $char^{**}argvMA$, onde cada elemento é uma *string* que corresponde a um argumento introduzido pelo utilizador.

- Se em $argvMA[0]$ estiver "i", então procede-se à inserção de um novo artigo cujo nome e preço estão em $argvMA[1]$ e $argvMA[2]$ (convertido para *double*), respetivamente;
- Se em $argvMA[0]$ se se encontrar um "n", então procede-se a alteração do nome do artigo com código equivalente a $argvMA[1]$, passando este artigo a ter o novo nome em $argvMA[2]$, que é registado no ficheiro $STRINGS$;
- Caso $argvMA[0]$ seja "p", passa-se à efetivação do preço do novo preço, em $argvMA[2]$ do antigo cujo código é $argvMA[1]$.
- Se em $argvMA[0]$ estiver "a", então é enviada uma ordem ao servidor para este executar o agregador.

4.2 Servidor de Vendas (SV)

Invocação na *bash*: "./sv ".

O Servidor de Vendas é responsável pelo controlo de pedidos de venda e pela alteração efetiva do stock. Para este efeito, atua apenas depois da receção de um pedido do Cliente de Vendas (cv).

É também responsável pelo controlo da invocação do Agregador (ag) quando o utilizador insere o comando "a" no ma.

O servidor funciona apenas num único *thread* e num único processo. Recebe instruções dos clientes de venda, sendo que a cada operação possível corresponde um código de instrução, e despacha para o código correto o respetivo tratamento da instrução recebida.

Esta arquitetura *"single-threaded"* tenta garantir a inexistência de *race conditions* que poderiam originar com vários clientes a atingir o servidor ao mesmo

tempo, mas, ao mesmo tempo, também é responsável por diminuir a sua performance em relação a um modelo com vários processos ou *threads*. Por esta razão, a *FIFO* principal recebe instruções e envia respostas em binário, e não em texto ASCII, pois estas são mais rápidas e eficientes de interpretar e enviar.

Devido a estas e outras otimizações, conseguimos fazer com que o servidor, sem *cache*, consiga processar 1 milhão de pedidos de venda do mesmo artigo em menos de 30 segundos, provenientes de vários clientes (100 nos nossos testes, sendo que cada cliente executa 10000 pedidos de venda do mesmo artigo), e sem causar *race conditions*.

Estas instruções são recebidas de uma *FIFO* principal (atualmente com o nome *"fifo_sv"*, controlável numa macro em *src/common/sv_protocol.h*).

Mais detalhadamente, o *loop* do servidor funciona assim: (src/sv/main.c):

1. Da *FIFO* principal, o servidor lê um único *char*, corresponde à instrução que se pretende executar.

2. Da *FIFO* principal, o servidor lê de seguida um *pid_t*, correspondente ao *PID* do processo de cliente/ma que quer executar a instrução. Este *PID* vai servir como nome da *FIFO* que permite responder ao cliente.

3. Agora, o servidor processa a instrução: compara com as instruções reconhecidas, e despacha o controlo para o código apropriado capaz de reconhecer a instrução que o cliente pretende ver executada (ficheiro src/sv/main.c, funções de nome *exec_**).

4. Por último, o servidor verifica se tudo foi executado com sucesso, fechando a *FIFO* de resposta, e libertando outros recursos previamente utilizados.

4.3 Cliente de Vendas (CV)

Invocação na *bash*: "./cv" .

O Cliente de Vendas (cv) é interface interativa pela qual o utilizador consegue obter informação de um dado artigo e alterar o stock disponível desse mesmo artigo.

De forma semelhante ao ma, recebe um *array* cujos argumentos correspondem aos argumentos do comando introduzido pelo utilizador, sendo que o primeiro argumento deverá sempre ser o código correspondente ao artigo que se pretende inspecionar.

- Se apenas existir um argumento, ou seja, se apenas se introduzir o código do artigo, então procede-se à escrita no *stdout* do stock desse artigo e o seu preço.
- Se forem detetados dois argumentos, onde o segundo se trata de um número real que especifica o *"delta"* a aplicar ao stock do artigo, efetua-se a alteração do stock disponível do artigo aplicando-se então a quantia do *"delta"*, que pode ser positiva, ou negativa.

Nota: tanto a escrita no *stdout* do stock e preço de um artigo como a aplicação do *"delta"* no stock de um artigo são feitos pelo servidor, sendo o cliente apenas responsável por enviar ao servidor, através da *FIFO* principal acima mencionada, a instrução a executar, assim como os respetivos argumentos dessa instrução.

O cliente, juntamente com a instrução, envia pela *FIFO* principal o seu *PID*. Este seu *PID* é depois utilizado pelo servidor para abrir uma *FIFO* de nome $fifo\_\langle pid \rangle$, onde irá colocar a resposta que pretende dar ao cliente.

4.4 Agregador (AG)

A execução deste programa não tem origem direta na interação do utilizador com a interface do ma. Pelo contrário, o ma, ao ler o comando "a" introduzido por parte do utilizador, faz um *request* ao Servidor de Vendas que, por sua vez, dá início à agregação.

A sua função é produzir, de forma organizada, para o *stdout*, toda informação básica de cada artigo que tenha pelo menos uma venda efetuada, como o seu código, quantidade total disponível e montante total gerado. Este output deve estar no formato do ficheiro de vendas.

Antes de ser executado, o Servidor é responsável por configurar todo o ambiente de execução do agregador: (ficheiro src/sv/main.c, função *exec_ag*)

1. É o Servidor que configura o *File Descriptor de STDIN* do Agregador, de modo a que este leia a input de uma *pipe* anónima criada pelo Servidor, e não da *bash* de onde normalmente estaria a ler;

2. É o Servidor que configura o *File Descriptor de STDOUT* do Agregador, de modo a que este escreva a sua output para o ficheiro agregado.

3. É o Servidor que coloca na *pipe* anónima que serve de *STDIN* ao Agregador os dados que este tem para agregar.

   Após o Agregador concluir, é o Servidor que atualiza a data da última agregação no ficheiro de VENDAS.

O algoritmo utilizado na agregação é bastante simples:

1. Primeiro, é consultado o ficheiro de ARTIGOS para sabermos quantos artigos existem à venda;

2. De seguida, é criada uma *array* com tamanho suficiente para conter os stocks de todos os artigos que existem à venda;

3. Agora, para cada linha lida do *stdin*, é interpretada como uma entrada de stock, e acrescentada à *array*, sendo que o índice na *array* corresponde ao código do artigo. Se o artigo já existir, então as quantidades e os montantes são somados; se o artigo ainda não existir na *array*, este é acrescentado.

4. Por último, é percorrida a *array* e impressos no *stdout* os dados agregados.

5 Otimizações

Ao longo de todo o projeto, uma das principais otimizações que conseguimos realizar foi manter todas as estruturas de dados em formatos binários. Isto

otimiza a sua leitura e escrita para os respetivos ficheiros, assim como o seu processamento mesmo enquanto na memória.

É importante de realçar que a *FIFO* principal de onde o servidor recebe instruções sofreu uma otimização adicional mas bastante importante: todo o código que envia instruções ao servidor (que se encontra em *src/common/sv_backend.c/h*), envia essas instruções e seus parâmetros com uma única chamada ao sistema *write()*. Deste modo, com os *atomic write's*, como são conhecidos, garantimos que não há instruções de vários clientes que ficam entrelaçadas umas com as outras, causando inconsistências que incapacitariam a recuperação por parte do servidor. Estes *atomic write's* também são de muito poucos bytes, de modo a garantirmos que estão abaixo de qualquer limite que o Sistema Operativo possa impor.

Esta utilização de *atomic write's* já não se verifica em outros *write()'s*, devido à sua complexidade, e porque não existem os problemas de *race conditions*, que existem com o código de interpretação de instruções do servidor.