Sdstore e Sdstored - Armazenamento Eficiente e Seguro de Ficheiros

Relatório do Trabalho Prático de SO

Sistemas Operativos

2º Ano | 2º Semestre

Grupo 98

José Pedro Batista Fonte a91775

João Gonçalo de Faria Melo a95085

Dinis Gonçalves Estrada a97503

Mestrado Integrado em Engenharia Informática

Departamento de Engenharia Informática

Universidade do Minho, Braga, Portugal

Maio 2022

ÍNDICE

Introdução

No âmbito da cadeira de Sistemas Operativos, o grupo 98 desenvolveu um programa em C que pretende implementar um serviço que permitisse aos utilizadores armazenar uma cópia dos seus ficheiros de forma segura e eficiente, poupando espaço de disco.

Para tal o serviço disponibilizará funcionalidades de compressão/descompressão e cifragem/decifragem dos ficheiros a serem armazenados. As 7 transformações disponíveis são:

- **bcompress** / **bdecompress**. Comprime / descomprime dados com o formato bzip2.
- **gcompress** / **gdecompress**. Comprime / descomprime dados com o formato gzip.
- **encrypt** / **decrypt**. Cifra / decifra dados.
- **nop**. Copia dados sem realizar qualquer transformação.

Segundo o enunciado apresentado o trabalho encontra-se divido em funcionalidades básicas e funcionalidades avançadas.

Como funcionalidades básicas o serviço deve permitir a submissão de pedidos para processar e armazenar novos ficheiros bem como recuperar o conteúdo original de ficheiros guardados previamente. É possível ainda consultar as tarefas de processamento de ficheiros a serem efetuadas num dado momento.

Como funcionalidades avançadas o serviço deve permitir obter estatísticas sobre o tamanho do documento de input e de output, implementar a prioridade de pedidos e fechar o servidor graciosamente com o sinal *SIGTERM*.

Para implementar este serviço, foram criados dois programas - *sdstore* e *sdstored,* que fazem o papel de cliente e servidor, respetivamente. O programa *sdstore* permite enviar pedidos ao servidor através dos seus argumentos e mantém o utilizador ocorrente do estado do pedido. O programa *sdstored* vai respondendo aos pedidos dos clientes.

Para correr o serviço, em primeiro lugar, é necessário inicializar o servidor com dois argumentos: o primeiro corresponde ao ficheiro de configuração com o número máximo de transformações, o segundo corresponde ao caminho para as pastas dos executáveis das transformações.

Assim que é inicializado, o servidor está pronto para receber os pedidos dos clientes.

Comunicação entre Servidor (Sdstored) e Cliente (Sdstore)

O envio da informação entre servidor e cliente é feito através de **pipes com nome**.

O problema pode ser descrito como o envio de informação em dois sentidos, um no sentido *cliente→servidor* para enviar pedidos e outro no sentido *servidor→cliente* para enviar a resposta a esse pedido.

Como primeira abordagem, o grupo decidiu criar dois pipes com nome permanentes, chamados *client2server* e *server2client*. O programa cliente abria um descritor de escrita para o pipe *client2server* e um de leitura para o pipe *server2client*. O programa server faria o contrário, um de leitura do *client2server* e outro de escrita para *server2client*.

Esta abordagem parecia a correta até ao momento de executar clientes em simultâneo. O problema que surgiu foi que, como os pedidos corriam concorrentemente, as mensagens de *servidor→cliente* não funcionavam corretamente.

Assim sendo, a segunda abordagem, a apresentada no trabalho, é criar um pipe *cliente→servidor* e vários pipes *servidor→cliente*. Um só pipe *cliente→servidor* é suficiente porque só existe um programa servidor. Os vários pipes *servidor→cliente* resolvem o problema da primeira abordagem pois permitem ao servidor enviar mensagens diferentes para pedidos diferentes ao mesmo tempo.

Mais concretamente, o servidor cria o pipe *client2server* com o descritor de leitura.

O programa cliente cria o pipe *servidor→cliente*, com o descritor de escrita. Para o servidor distinguir os vários pipes criados, o pipe terá o nome do *pid* do processo do cliente, por exemplo, se o *pid* do cliente for "12345" é criado um pipe chamado "12345".

O pipe de cada cliente é temporário pois fica aberto enquanto o pedido está a ser executado e depois é eliminado.

Estruturas de Dados

O servidor utiliza dois tipos de estrutura de dados.

Lista de Transformações – Transf *tracker* e *struct transf*

A ***struct transf*** é criada com o objetivo de guardar o estado de uma transformação e tem como parâmetros:

- **transf** : o nome da transformação;
- **running** : o número de transferências a ser utilizadas;
- **max** : número de transferências máximo.

No servidor é criada uma lista para as 7 transformações possíveis chamando a função *readConfigFile()* que lê o ficheiro de configuração e cria as 7 transformações, guardando o nome da transformação, inicializando o número a correr a zero, e o número máximo com o número no ficheiro de configuração.

Lista de Pedidos – Request lista e *struct request*

A ***struct request*** é criada com o objetivo de guardar toda a informação de um pedido e tem como parâmetros:

- **pid**: que guarda o *pid* do cliente que fez o pedido;
- **running**: uma *flag* que indica se o pedido está a ser executado;
- **priority**: que guarda a prioridade do pedido;
- **number_arg** : que guarda o número de argumentos na lista de *strings*;
- **max**: o número máximo de argumentos na lista de *strings*;
- **args** : guarda os argumentos numa lista de *strings*;

A lista de strings **args** funciona de forma dinâmica de modo a ser mais eficiente e concisa em termos de memória.

No servidor a função *init_requestList()* aloca espaço para a lista de pedidos com o seu tamanho passado como argumento. A lista de pedidos também funciona de forma dinâmica de modo a não estar limitada a um certo número de pedidos.

Para adicionar um pedido utiliza a função *addRequest()*, para limpar um pedido utiliza a função *limpaRequest()*.

A função *addRequest()* primeiramente verifica se existe algum espaço vazio na lista, depois aloca espaço para 8 *strings* e faz *parsing* da linha para preencher com os argumentos. O campo do *pid* e da *priority* são criados e o *running* fica a zero. A lista dos *args* guarda todos os argumentos.

De seguida, verifica se o pedido é valido de acordo com a configuração. Se não for limpa o pedido e retorna 0, se for retorna a posição do pedido.

A função *limpaRequest()* limpa o pedido, mudando tudo para zero e dando *free()* a todos os *args*.

Cliente – *Sdstore*

O programa cliente começa por criar o pipe do cliente, verificar os argumentos e abrir um descritor de escrita para o pipe *server2client*. De seguida, trata do envio do pedido para o servidor.

O envio do pedido é feito pelo envio de uma *string* separada por ";" com todos os argumentos necessários. A *string* é criada com a concatenação de todos os argumentos recebidos. Em primeiro lugar é enviado o *pid* do cliente (para o servidor saber que pipe abrir) e depois são concatenados por ordem os argumentos recebidos.

O cliente trata de verificar se os argumentos são válidos, exceto os ficheiros de input e output. Como a prioridade é opcional, se o cliente não inserir a prioridade esta é passada como "0" para não ter qualquer efeito na ordenação da lista. Um exemplo do referido é:

Para o comando:

./sdstore proc-file 1 file1.txt file2 nop bcompress encrypt

É enviada a mensagem:

12345; proc-file;1;file1.txt;file2;nop;bcompress;encrypt

Depois de enviada a mensagem, fecha o descritor de escrita para o servidor.

De seguida abre-se o descrito de leitura do pipe *servidor→cliente* para esperar receber a mensagem do servidor.

No entanto, existe um edge-case que dá problemas. Quando no servidor existe um pedido pendente, o descritor de escrita é encerrado logo o pipe seria fechado, e assim o programa cliente encerrava sem a resposta do servidor. Deste modo, a solução arranjada foi abrir no cliente um

descritor de escrita para o pipe *servidor→cliente*, denominado *helper*, que só é encerrado o pipe se o servidor passar a mensagem *"concluded"* ou se o pedido for *./sdstore status*.

Para encerrar o programa, elimina o pipe *servidor→cliente*, fecha o descritor de leitura e liberta todo o espaço alocado em memória.

Servidor – *Sdstored*

O programa *sdstored* começa por verificar o número de argumentos, abrir descritor de leitura para o ficheiro de configuração e para o pipe *client2server*, assim como associar o sinal *SIGTERM* à função *exitServer()*.

De seguida, o servidor cria as duas listas de estruturas mencionadas em cima. O ***tracker*** é criado passando-lhe o ficheiro de configuração e para a **lista** é alocado espaço de dois pedidos na memória.

Assim que as listas são criadas, o servidor espera pela mensagem do cliente. O processamento de mensagens primeiro faz um parsing dos dois primeiros argumentos. De acordo com estes dois argumentos o servidor só pode executar 4 tipos de operações:

- **exit**: sair do servidor
- **termina-pedido**: terminar o pedido passado
- **proc-file**: processa um pedido com transformações
- **status**: envia o estado do sistema

O descritor de escrita para o pipe *servidor→cliente* só é aberto depois de verificar que a mensagem recebida não é *exit*, nem *termina-pedido*. Isto é necessário porque quando o intuito é executar *exit* ou *termina-pedido* o 1º argumento não corresponde a nenhum pipe *servidor→cliente*. No caso do exit, é "0000" pois é desnecessário, no caso do termina-pedido, é o *pid* do pedido.

Operação *exit*

Para terminar o servidor o exit liga a variável global *exit=1* e corre a função *isRequestListEmpty()* que só deixa fechar o servidor quando a lista estiver vazia.

Operação *termina-pedido*

Operação responsável por decrementar o *tracker* de transformações e limpar o pedido da lista. Assim que o pedido é terminado tenta encontrar o próximo na lista para ser executado.

- Se encontrar um pedido faz um goto para a parte de execução de pedidos, mudando primeiro o descritor de escrita do pipe *servidor→cliente* para o do pedido encontrado.
- Se não encontrar nenhum pedido encerra tudo normalmente o servidor continua à espera de novos pedidos.

Operação *proc-file*

Para processar o pedido, primeiro ele é adicionado dinamicamente à lista. Depois verifica se ele pode ser executado consoante as transformações a ser utilizadas no momento.

- Se não for possível, fica na lista de pedidos como pedido pendente e será executado assim que se terminar um pedido.
- Se for possível correr o pedido, liga a *flag* do pedido para indicar que está correr e aumenta o *tracker* com as transformações do pedido.

A parte de executar as transformações é feita por um processo filho de modo a ser possível correr pedidos concorrentemente.

Como o processo-filho copia toda a memória, apenas se executa as transformações pois não necessita de mudar nada nas listas de pedidos e de transformações. Para executar um pedido utiliza a função *executar_pedido()*.

Primeiro o *executar_pedido()* envia a mensagem *"processing"* para o cliente, depois abrem-se um descritor de leitura para o input-file e um de escrita para o output-file.

- Caso seja apenas uma transformação, cria-se um processo-filho e muda-se o STDIN para o input-file e o STDOUT para o output-file. E o pai espera que termine.
- Caso sejam mais do que um, cria-se vários processos-filho, ligam-se entre eles por pipe anônimos e cada um executa uma transformação. O primeiro processo muda o STDIN para o descritor do input-file e muda o STDOUT para a escrita do pipe para o primeiro processo intermédio. Os processos intermédios mudam o STDIN para a leitura do pipe que liga o processo anterior e o STDOUT para escrita no pipe que liga o processo seguinte. O último processo muda o STDIN para o pipe que o liga ao processo anterior e o STDOUT para o descritor do output-file. No final o pai espera que todos os filhos encerrem.

No final da execução envia a mensagem *"concluded (bytes-input: X, bytes-output: Y)"*.

Depois de executado o pedido tem de ser encerrado.

Para encerrar um pedido é necessário limpá-lo da lista e decrementar o *tracker* das transformações, como isto não pode ser feito no processo-filho, o processo envia a mensagem para o server com a formatação *"[pedido.pid];termina-pedido"*. Como a operação *termina-pedido* ocorre no processo-pai é possível alterar a memória principal.

No final fecha tudo normalmente e o servidor continua à espera de novos pedidos.

Operação *status*

Para o servidor enviar o estado do mesmo, em concorrência com os outros pedidos este também executa o *status* dentro de um processo-filho. O *status* percorre a lista de transformações e de pedidos e escreve para o cliente cada um. De notar que ao enviar os pedidos, este verifica se não são pedidos limpos.

No final fecha tudo normalmente e o servidor continua à espera de novos pedidos.

Funcionalidades Extra

Tamanho do ficheiro *input* e *output*

O tamanho dos ficheiros é calculado usando a função *lseek(ficheiro,0,SEEK_END)* que é utilizada no final da função *executar_pedido()*.

Prioridade

A prioridade de cada pedido é atingida com a combinação de dois fatores: organizar a lista com o algoritmo *Merge Sort* e executar os pedidos de acordo com o seu *pid*.

A primeira parte foi atingida a 100%, estando o algoritmo bem construído com a função *mergesortLista()* que seria utilizada sempre que se adiciona um novo pedido à lista.

A segunda parte não foi atingida, pois o grupo só se apercebeu da mesma tarde demais.

O trabalho foi todo construído com as funções a receber a posição do pedido na lista em vez do *pid*. No entanto, quando a prioridade é implementada as posições mudam sempre que é adicionado um novo pedido. Isto implica que todas as funções tenham de receber o *pid* e só depois encontrar a posição e executar a função.

SIGTERM

A implementação do sinal *SIGTERM* é atingida usando associando do sinal à função *exitServer()* que envia uma mensagem para o servidor "0000;exit" que ao ser processada liga a *flag* global *exit* e espera que a lista de pedidos fique vazia com a função *isRequestListEmpty()*.

Conclusão

O grupo pretendia atingir todos as funcionalidades propostas, mas tal não se concretizou.

Nas funcionalidades básicas o grupo considera que todos os objetivos foram cumpridos, mas nas funcionalidades avançadas devido à falta de tempo o grupo não conseguiu implementar tudo.

O trabalho em geral pareceu indicado tanto a nível de dificuldade como de carga de trabalho. E as aulas práticas pareceram o veículo ideal de aprendizagem e suporte para fazer o trabalho.

Por último, o grupo também acha indicado uma distribuição de notas diferente.

- José Pedro Batista Fonte *a91775* (+ 4 valores)
- João Gonçalo de Faria Melo *a95085* (- 2 valores)
- Dinis Gonçalves Estrada *a97503* (- 2 valores)