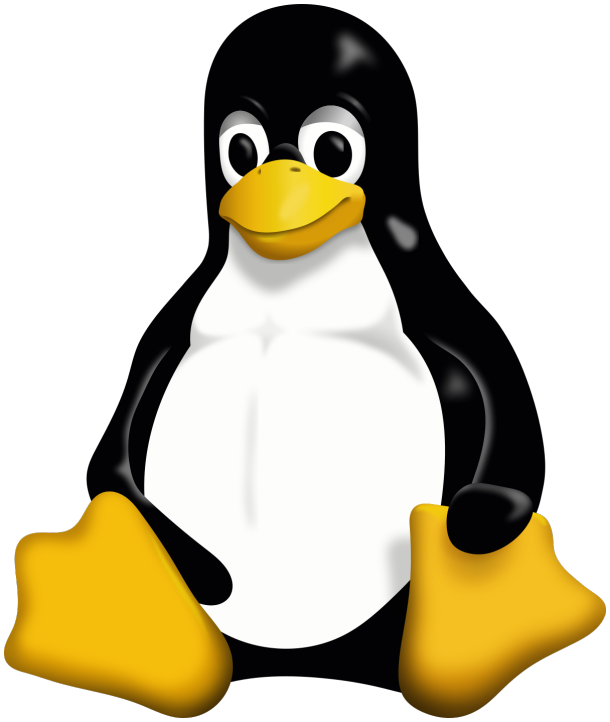

Trabalho Prático - SDStore

Sistemas Operativos

Feito por:

Guilherme Sampaio, a96766
Miguel Neiva, a92945
João Longo, a95536

Licenciatura em Engenharia Informática

28 de maio de 2022

Conteúdo

1 Introdução **1**

2 Objetivos **1**

3 Arquitetura da Aplicação **2**

3.1 O Cliente 2

3.2 O Servidor 3

4 Problemas Encontrados **5**

5 Conclusão **7**

1 Introdução

O trabalho prático de Sistemas Operativos, **SDStore**, implementa um serviço que permite aos seus utilizadores armazenar uma cópia dos seus ficheiros de forma segura e eficiente de modo a poupar espaço em disco. São disponibilizadas quatro tipos de operações primários sobre ficheiros:

- nop : Copia dados sem realizar nenhum tipo de alteração.
- encrypt/decrypt : Cifra ou decifra ficheiros (dados).
- gcompress/gdecompress : Comprime ou descomprime dados com o formato gzip.
- bcompress/bdecompress : Comprime / descomprime dados com o formato bzip.

O programa é composto por duas metades, o servidor, responsável por aceitar e processar pedidos, e o cliente, responsável por enviar pedidos de processamento ao servidor ou pedidos de *status* do servidor.

Neste relatório vão ser discutidos os objetivos do projeto, os problemas encontrados durante a sua resolução e uma descrição da arquitetura da aplicação.

2 Objetivos

Os objetivos/funcionalidades do trabalho podem ser divididos em duas categorias: as funcionalidades básicas e as funcionalidades avançadas. Os objetivos base, alguns já previamente descritos acima, envolvem tanto o servidor como o cliente.

Em primeiro lugar, o servidor deve receber dois argumentos pela linha de comandos, o primeiro indica o ficheiro de configuração e o segundo a diretoria onde os ficheiros binários das operações (nop, bcompress, etc.) estão guardados. O ficheiro de configuração serve para limitar o uso de operações. Como os pedidos vão ser processados concorrentemente vários pedidos vão requerer de várias operações, de modo que haja um limite de operações a correr em simultâneo, foi criado este mecanismo.

```
Por exemplo:
$ ./server config.conf ./tools
```

Em segundo lugar, cada cliente deve solicitar o processamento de dados através de um comando composto pelo caminho do ficheiro a ser processado, o caminho onde o servidor deve guardar a nova versão do ficheiro e uma sequência de operações a aplicar no ficheiro.

```
Por exemplo:
$ ./client (-p priority_value) file_a file_b op1 op2 ... opN
```

Nesta funcionalidade entra o primeiro objetivo avançado - a prioridade dos pedidos. Cada pedido pode, ou não, especificar a sua prioridade (caso não seja especificado o servidor assume a sua prioridade a 0). A prioridade máxima é 5.

De seguida temos que o utilizador deve ser informado (através do STDOUT) do estado do seu pedido, isto é, se este já foi processado, se foi metido na lista de espera, se já concluiu. Aqui também entra uma das funcionalidades avançadas, nomeadamente que ao concluir o processamento deve ser indicado o número de *bytes* lidos e escritos.

```
Por exemplo:
[at 11:21:34:366227] Job id: 13054
[*] Pending...
[*] Job queued...
[*] Completed (bytes-input: 85, bytes-output: 134)
```

Em quarto lugar, o servidor deve suportar o processamento concorrente de pedidos, ou seja, vários pedidos podem executar simultaneamente.

E finalmente, o cliente deve conseguir fazer um pedido do estado do servidor, isto é, saber quantos e quais os pedidos aguardam processamento ou que estão em execução, assim como o estado de utilização das operações.

```
Por exemplo:
[at 11:33:40:669802] Job id: 13640
[SERVER STATUS] tmp/stc_13640
Queued Up Jobs:
-- no jobs queued up --
In Execution Jobs:
-- no jobs currently executing --
Resources (using/max):
nop:         0/10
gcompress:   0/10
gdecompress: 0/10
bcompress:   0/10
bdecompress: 0/10
encrypt:     0/10
decrypt:     0/10
```

Existe uma funcionalidade adicional, término gracioso, porém, esta não foi implementada.

3 Arquitetura da Aplicação

Como já referido, o programa SDStore é composto pelo cliente e pelo servidor. O cliente comunica com o servidor e envia-lhe pedidos e o servidor fica à espera de novos pedidos, lê-los e processa-os.

3.1 O Cliente

O cliente é sem dúvida a metade mais simples do programa. Tem como objetivo enviar pedidos de processamento/*status* ao servidor e espera que este lhe responda.

Um aspeto importante que tem de ser analizado é o facto de, para comunicar entre cliente e servidor pode ser usado apenas um FIFO, porém para comunicar entre servidor

e cliente tem de ser usado um FIFO para cada cliente. Mas como é que garantimos a unicidade do FIFO?

De modo a que os FIFO's sejam únicos devemos usar o identificador do processo do cliente atual, assim, para cada cliente vai ser criado um FIFO próprio a esse cliente. A criação do FIFO entre o servidor e cliente é feita no cliente e quando é enviado um pedido ao servidor, o nome do FIFO do cliente atual é enviado também.

Deste modo, garantimos que um cliente tem um FIFO só dele e que o servidor sabe sempre por onde enviar mensagem para comunicar com o cliente.

Após a abertura do FIFO do cliente-servidor (cts) e da criação do FIFO do servidor-cliente (stc), podemos começar a formatar a nossa mensagem. A mensagem a enviar ao servidor vai ser uma *string* que contém o STC e todos os argumentos concatenados e separados por espaços. De seguida, a mensagem é enviada ao servidor e entramos num estado de espera.

O estado de espera, nada é mais, que um *read()*, ficamos a ouvir o que o servidor nos tem a dizer, e caso o que tenhamos lido seja uma mensagem de *status* ou de indicação de finalização podemos terminar a execução do programa.

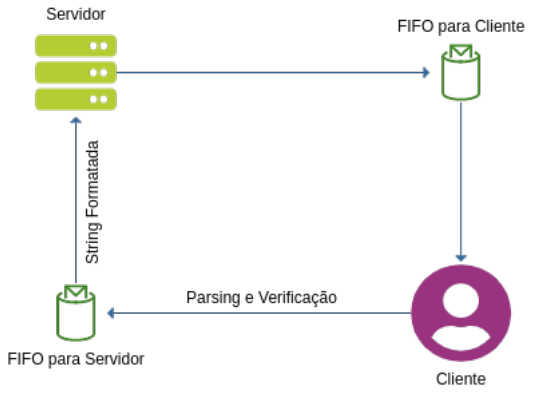

Figura 1: Arquitetura do Cliente

3.2 O Servidor

Contrariamente ao cliente, esta parte do programa é bastante mais complexa e é responsável por receber pedidos, decidir o tipo do pedido e executar o pedido. Estes pedidos (jobs) podem ser categorizados em três tipos: HELP, PROC-FILE e STATUS.

Mal o servidor receba um *job* o processo *"Dispacher"* faz um *parsing* incial da mensagem para determinar o seu tipo. Caso seja um pedido HELP então, através do STC Fifo (fornecido na mensagem), é enviado uma mensagem de ajuda com todas as funcionalidades do programa. Caso seja um pedido STATUS então, por um *pipe* anónimo enviamos um pedido STAT ao nosso segundo processo *"Queue Manager"*.

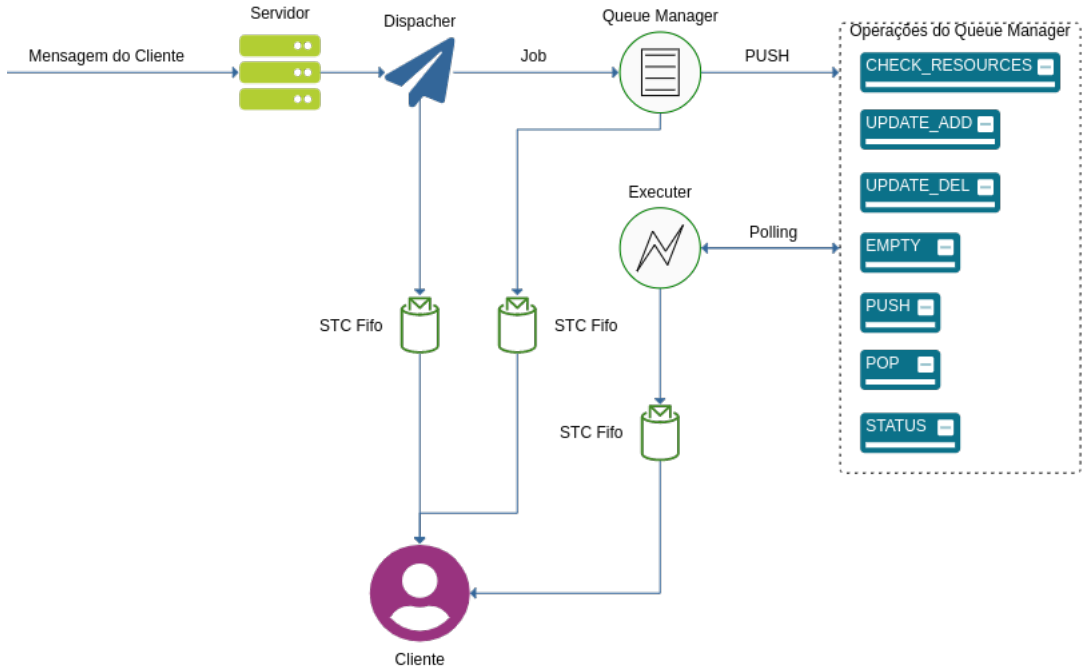

Figura 2: Arquitetura do Servidor

O *"Queue Manager"* é o processo mais importante deste trabalho, tem 7 operações distintas que podem ser "chamadas"através do envio de um inteiro negativo por um *pipe*. A operação CHECK RESOURCES notifica se há recursos disponíveis para executar um *job*, a operação UPDATE ADD e UPDATE DEL atualizam a fila de prioridade (removendo ou adicionando um pedido), a operação POP retira um elemento da fila de prioridade, a operação EMPTY notifica se há a fila está vazia ou não e a operação PUSH insere um pedido na fila.

Finalmente, caso o pedido seja do tipo PROC-FILE, então é enviado ao *"Queue Manager"* a mensagem para este fazer o seu *parsing* e adicionar à *priority queue*.

Enquanto tudo isto acontece, existe mais um processo a correr, o *Executer*. Este é o único processo utilizar as operações do *queue manager* com exceção do PUSH. O *Executer* é responsável por retirar pedidos da fila de espera e executá-los. Isto acontece através de *polling*, isto é, o *Executer* envia constantemente pedidos EMPTY ao *Queue manager*, caso o *manager* responda que não está vazia, então é enviado um pedido POP para receber o pedido. Ao receber o pedido, o próximo passo é verificar se pode executar ou não (devido ao limite de operações concorrentes), para isso envia um pedido CHECK RESOURCES e, mais uma vez, caso haja recursos enviamos um pedido UPDATE ADD, executamos o pedido e enviamos um pedido UPDATE DEL. Desta maneira garantimos sempre que os recursos são atualizados e que só executamos o pedido quando temos a certeza que está tudo livre.

Este processo fica mais claro com a Figura 3, anexada abaixo.

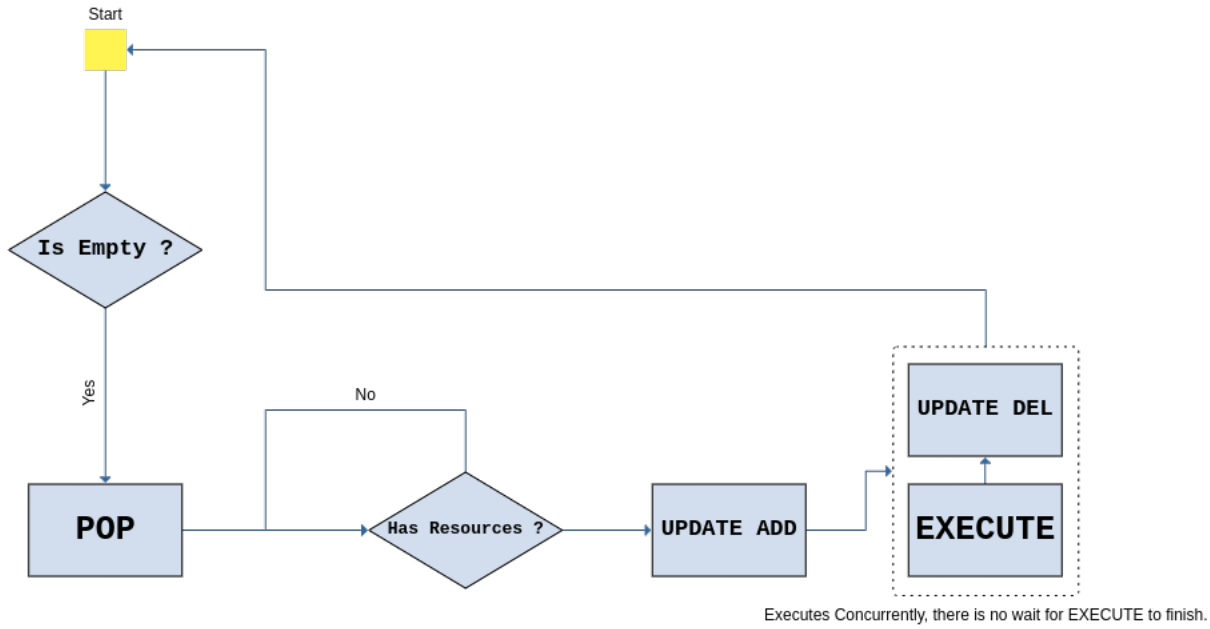

Figura 3: Processo de Execução

4 Problemas Encontrados

O primeiro problema encontrado, já referido acima, foi o facto de ter de existir um canal de comunicação entre o servidor e o cliente único, cada cliente tem de ter um FIFO próprio por onde o servidor envia dados.

Grande parte do trabalho já estava feito quando descobri a necessidade deste aspeto (claro que se tivesse havido um bom planeamento não teria tido este mesmo problema) e grande parte do código teve de ser alterado, tanto do cliente como do servidor.

A maneira mais fácil de resolver este problema foi enviar ao servidor por que FIFO é que este teria de comunicar comigo (cliente).

```
pid_t client_id = getpid();
char cts_fifo[64];
sprintf(cts_fifo, "tmp/stc_%d", client_id);
```

O segundo problema encontrado toca na arquitetura do projeto, um erro bastante óbvio, mas que devido à pouca testagem inicial se manteve durante demasiado tempo. Antes de existirem as operações UPDATE ADD, UPDATE DEL e CHECK RESOURCES no *queue manager*, os dados relativos à utilização de recursos eram guardados no *executer*. O modo de execução seguia a mesma ordem, o mesmo *flow* da Figura 3, porém se eu quisesse manter a concorrência entre pedidos não haveria maneira de eu saber que um processamento acabou de executar.

```
/* ... update resources usage ... */

pid_t exec_fork = fork();
if (exec_fork < 0)
{
```

```
    print_error("Could not fork process @ executing job.\n");
    _exit(FORK_ERROR);
}

if (exec_fork == 0)
{
    print_log("Executing a job.\n", log_file, false);
    execute(current_job);

    char *exec_string = xmalloc(sizeof(char) * 128);
    sprintf(exec_string, "Executed job (%s).\n", current_job.fifo);
    print_info(exec_string);
    free(exec_string);

    char *completed_message = xmalloc(sizeof(char) * 128);
    generate_completed_message(completed_message, current_job.from,
   current_job.to);

    send_status_to_client(current_job.fifo, completed_message);

    _exit(EXIT_SUCCESS);
}

/* ... update resources usage ... */
```

O problema com isto é que sem maneira de como saber quando o processamento termina, não podemos desalocar os recursos, isto é, os recursos iriam ser adicionados, é criado o processo de execução, mas os recursos, mesmo antes do *job* terminar, seriam deslocados, pois, a execução do processo pai continua.

Assim o comando *status* e o mecanismo de verificação de recursos nunca iria funcionar já que os valores dos recursos ia ser sempre 0.

Com os valores de recursos no *queue manager* é possível desalocar recursos através do processo de execução, ao comunicar através de um *pipe* como o *manager* (implementação atual).

```
/* ... atualizacao de recursos ... */

pid_t exec_fork = fork();
if (exec_fork < 0)
{
    print_error("Could not fork process @ executing job.\n");
    _exit(FORK_ERROR);
}

if (exec_fork == 0)
{
    close(input_com[0]);
    close(del_pipe[0]);

    print_log("Executing a job.\n", log_file, false);
    execute(current_job);

```

```
18      char *exec_string = xmalloc(sizeof(char) * 128);
19      sprintf(exec_string, "Executed job (%s).\n", current_job.fifo);
20      print_info(exec_string);
21      free(exec_string);
22
23      char *completed_message = xmalloc(sizeof(char) * 128);
24      generate_completed_message(completed_message, current_job.from,
        current_job.to);
25
26      send_status_to_client(current_job.fifo, completed_message);
27
28      /* 1: Enviar pedido UPDATE_DEL ao queue_manager. */
29      /* 2: Enviar o pedido a ser desalocado. */
30
31      close(input_com[0]);
32      close(del_pipe[0]);
33
34      /* Terminar execucao */
35      _exit(EXIT_SUCCESS);
36  }
```

5 Conclusão

O projeto apresenta, praticamente, todas as funcionalidades pedidas (com exceção do término gracioso) e mais duas adicionais (mensagem de ajuda e ficheiro de log). Desta forma acredito que o trabalho está bem conseguido, porém poderia ter feito algumas coisas de maneira diferente e mais eficiente (especialmente refatoração de código).

Foi um projeto muito interessante, pois mostrou muitos aspetos da linguagem C e do sistema operativo dos quais não fazia ideia e ajudou a consolidar tudo o que foi ensinado nas aulas práticas. Sem dúvida uns dos trabalhos mais divertidos do ano.