SDStore: Armazenamento Eficiente e Seguro de Ficheiros

Sistemas Operativos

Bárbara Faria	José Pires	Tiago Lima
A85774	A84552	A85126

(29 de maio de 2022)

Conteúdo

1 Introdução

No âmbito da unidade curricular de Sistemas Operativos, foi-nos proposto a implementação de um serviço que permite aos seus utilizadores armazenar uma cópia dos seus ficheiros de forma segura e eficiente, poupando espaço em disco. Este será caraterizado pelas funcionalidades de compressão e cifragem de ficheiros, bem como a submissão de pedidos para os processar e armazenar. Será também possível recuperar o conteúdo original e consultar as tarefas que estão a ser executadas no momento.

2 Cliente

Começamos por desenvolver um cliente (programa *sdstore*) que oferece uma interface com o utilizador via linha de comando. O utilizador poderá agir sobre o servidor através dos argumentos especificados na linha de comando deste cliente.

2.1 Modo de Utilização

Quando executamos o cliente sem qualquer argumento, o programa fornece informação relevante ao modo de utilização do mesmo:

```
$ ./sdstore
[Usage] ./sdstore proc-file samples/file-a outputs/file-a-output [
    transform]...
[Usage] ./sdstore status
```

2.2 Opção *status*

Quando executamos o cliente com a opção *status*, este imprime os pedidos de processamento em execução, bem como o estado de utilização das transformações:

```
$ ./sdstore status
task #i: proc-file input-file output-file [transform]...
transf nop: r/m (running/max)
transf bcompress: r/m (running/max)
transf bdecompress: r/m (running/max)
transf gcompress: r/m (running/max)
transf gdecompress: r/m (running/max)
transf encrypt: r/m (running/max)
transf decrypt: r/m (running/max)
```

Foi criada uma função *sendServerStatus* que recebe como parâmetros os argumentos da função *main*. De maneira a comunicar com o servidor, utilizamos um *pipe* com nome (*fifo*), através do qual enviamos o identificador deste processo (*pid*) e o número de argumentos que a função *main* recebeu. Este último ajudará o servidor a distinguir entre as opções *status* e *proc-file*. O identificador de processo servirá para o servidor enviar informação a este cliente.

2.3 Opção *proc-file*

Quando executamos o cliente com a opção *proc-file*, passamos também como argumentos o caminho do ficheiro a ser processado, o caminho onde o serviço guarda a nova versão do ficheiro e uma sequência de transformações a aplicar.

Em primeiro lugar, verificamos se os argumentos da opção *proc-file* estão sintaticamente corretos com a ajuda da função *parser*. Foi criada também uma

função *sendServerPF* que recebe como parâmetros os argumentos da função *main*.

De maneira a comunicar com o servidor, utilizamos um *pipe* com nome (*fifo*), através do qual enviamos o *pid* e o número de argumentos que a função *main* recebeu. São enviados os ficheiros de input e output assim como as transformações que serão aplicadas. O número de argumentos, para além de ajudar a distinguir entre as opções *proc-file* e *status*, auxiliará no *parsing* das transformações.

Por fim, o cliente informa o utilizador acerca do estado do pedido (*pending*, *processing* e *concluded*) ou do estado do servidor.

2.4 Comunicação Servidor → Cliente

O cliente cria um *pipe* com nome (o nome será o *pid* do cliente) que será utilizado para o servidor enviar a informação relevante ao cliente. A função *receiveFromServer* ocupar-se-á destas tarefas.

3 Servidor

Foi desenvolvido também um servidor (programa *sdstored*), que mantém em memória a informação relevante para suportar as funcionalidades pedidas.

Este programa recebe dois argumentos pela linha de comando: o primeiro corresponde ao caminho para um ficheiro de configuração que é composto por uma sequência de linhas de texto, uma por tipo de transformação, contendo o identificador da mesma e o número máximo de instâncias de uma certa transformação que podem executar concorrentemente num determinado período de tempo. O segundo argumento corresponde ao caminho para a pasta onde os executáveis das transformações estão guardados.

3.1 Estruturas de Dados

As estruturas de dados desenvolvidas para auxiliar na gestão do servidor foram *Processamento* e *Limites*.

3.1.1 Processamento

```
typedef struct{
  char pid[6];
  char comandos[200];
  int completed; // 0 -> pending, 1-> processing, 2-> completed
  int task;
}Processamento;
```

Esta estrutura é usada para armazenar e gerir informação sobre os pedidos feitos por clientes.

3.1.2 Limites

```
typedef struct{
  int nop_atual;
  int bcompress_atual;
  int bdecompress_atual;
  int gcompress_atual;
  int gdecompress_atual;
  int encrypt_atual;
  int decrypt_atual;
}Limites;
```

Esta estrutura é usada para armazenar e gerir informação sobre as instâncias das transformações a correr.

3.2 Inicialização

Ao iniciarmos o servidor, a função *parseArgs* irá, em primeiro lugar, verificar se os argumentos passados ao servidor estão sintaticamente corretos.

Depois, com o auxílio da função *read_Config_File*, extrai os limites máximos de cada transformação (localizados no ficheiro de configuração), cria uma estrutura de dados inicializada com os valores das variáveis a zero, à qual será guardada num ficheiro *log_limites.bin*, que servirá para identificar o número de instâncias de cada transformação a correr num dado momento.

Após toda esta atividade, é criado um ficheiro *log.bin* que irá guardar e atualizar todos os pedidos solicitados.

É também criado um *pipe* com nome (*fifo*) pelo qual o servidor comunicará com os clientes.

3.3 Funcionamento

O servidor irá ler linhas do *pipe*, cada linha correspondendo a um pedido. Estas serão processadas pela função *parseServer*.

3.3.1 Opção *status*

Se for recebida uma opção *status*, a função *parseServer* cria um filho, através da chamada ao sistema *fork*, que executa a função *executeStatus*. Desta forma, o programa poderá continuar a receber pedidos e a processá-los enquanto este é executado.

A função *executeStatus* cria um *pipe* com nome (o nome será o *pid* do cliente que submeteu o pedido) para poder enviar informação ao cliente. A função percorre o ficheiro *log.bin* e identifica os pedidos em execução, enviando a informação destes pelo *pipe*. Por fim, percorre o ficheiro *log_limites.bin* para identificar as instâncias de cada transformação a correr naquele dado momento.

3.3.2 Opção *proc-file*

Caso o servidor receba um pedido com opção *proc-file*, a função *parseServer* cria um filho, através da chamada ao sistema *fork*, que executa a função *writeToLog*. Esta vai guardar a informação do pedido numa estrutura de dados: o *pid* do cliente que realizou o pedido, as transformações a serem aplicadas, os ficheiros de input/output, uma variável que controla o estado do pedido e uma variável que identifica o número da tarefa. Essa estrutura é guardada no ficheiro *log.bin*. De seguida, o processo filho irá utilizar a função *executeServer* para efetuar o pedido.

A função *executeServer* vai receber o *pid* do cliente que realizou o pedido, uma linha com os ficheiros de input/output e as transformações que serão aplicadas. Primeiro, é enviada uma mensagem ao cliente, informando sobre o

estado do pedido (*pending*). Usa-se a função *answerClient* para esse efeito. A comunicação é feita através de um *pipe* com nome (o *pid* do cliente).

É feito um *parsing* da linha que a função *executeServer* recebe. É verificada se as transformações passadas como argumento não excedem os limites impostos pelo ficheiro *config_file.txt*. Para tal, é usada a função *verificaLimites*: se alguma das transformações exceder o limite, é enviada uma mensagem ao cliente através da função*answerClient* a avisá-lo de tal.

Percorremos o ficheiro *log_limites.bin* e procuramos as instâncias das transformações passadas como argumento. Caso o número de instâncias a correr mais o número de instâncias no pedido excedam o limite, não será possível processá-lo. No entanto, este algoritmo foi implementado com o auxílio de um ciclo *while*. Isto significa que estamos a recorrer a *polling*, o que não é desejável. Quando for possível executar o pedido, atualizamos no ficheiro *log.bin* (com a ajuda da função *updateLog*) a variável *completed* da estrutura de dados relativa a este pedido para o valor "1"(em estado de processamento). É enviado ao cliente a mensagem *processing* através da função *answerClient*. Atualizamos o ficheiro *log_limites.bin* (com a ajuda da função *updateLogConfig*) onde vão ser acrescentados o número das instâncias das transformações que vão ser executadas.

Se o número de transformações for igual a 1, apenas temos de redirecionar os descritores do ficheiro de input para o standard input e o do ficheiro de output para o standard output. Recorrendo ao comando *exec*, executamos essa transformação. Caso contrário, para a primeira transformação, redirecionamos o descritor do ficheiro de input para o standard input e para a última redirecionamos o descritor do ficheiro de output para o standard output. O resto da informação é passado através de *pipes* anónimos para as transformações intermédias com os devidos redirecionamentos dos descritores. Mais uma vez, utilizamos o comando *exec* para executar cada uma delas.

Assim que acabar a execução do pedido, atualizamos no ficheiro *log.bin* a variável *completed* da estrutura de dados relativa a este pedido para o valor "2"(terminado). Atualizamos o ficheiro *log_limites.bin* onde vão ser retiradas o número das instâncias das transformações que foram executadas. É enviado ao cliente a mensagem *concluded* através da função *answerClient*.

4 Conclusão

Através deste trabalho prático foi-nos possível pôr em prática todo o conhecimento adquirido ao longo do semestre na Unidade Curricular Sistemas Operativos. Aplicamos com sucesso o acesso a ficheiros através das chamadas ao sistema *read* e *write* (entre outras), a criação de processos filho (textitfork), a execução de programas (*execs*), o redirecionamento de descritores (*dups*) e a utilização de *pipes* (anónimos e com nome).

Fazemos uma avaliação positiva do nosso desempenho neste projeto pois achamos que o objetivo do mesmo foi parcialmente concluído. Em relação aos aspetos menos positivos, recorremos ao *polling* no controlo da execução dos processos, não implementamos as funcionalidades avançadas e a verificação dos limites não está a funcionar corretamente.

A Código do Cliente

A.1 sdstore.c

```c
#include "auxClient.h"

int parser(int argc, char * argv[]){

    //./sdstored pro old new bla bla bla
    if(strcmp(argv[1],"proc-file")!=0){
        printf("Op   o  inv lida: %s\n[Usage] ./sdstore proc-file samples/file-a outputs/file-a-output [transform]...\n",argv[1]);
        return -1;
    }

    for (int i = 4; i < argc; i++){
        if (strcmp(argv[i],"nop")!=0 && strcmp(argv[i],"bcompress")!=0 && strcmp(argv[i],"bdecompress")!=0
                    && strcmp(argv[i],"gcompress")!=0 && strcmp(argv[i],"gdecompress")!=0
                    && strcmp(argv[i],"encrypt")!=0 && strcmp(argv[i],"decrypt")!=0){

            printf("Comando invalido: %s\n[Usage] nop bcompress bdecompress gcompress gdecompress encrypt decrypt\n",argv[i]);
            return -1;
        }
    }
    return 0;
}


int main(int argc, char *argv[]){
    int status;

    if (argc<2){
        printf("[Usage] ./sdstore proc-file samples/file-a outputs/file-a-output [transform]...\n");
        printf("[Usage] ./sdstore status\n");
        return -1;
    }
    else if (argc == 2){
        if(strcmp("status",argv[1])!=0){
            printf("[Usage] ./sdstore status\n");
            return -1;
        }
    }

    else{
        if (parser(argc, argv)<0){
            return -1;
        }
    }

    if (argc==2){
        sendToServerStatus(argc,argv);
```

```
47          receiveFromServer();
48
49      }
50      else{
51          sendToServerPF(argc,argv);
52          receiveFromServer();
53      }
54
55      return 0;
56  }
```

A.2 auxClient.c

```c
#include "auxClient.h"

char read_buffer[BUFFER_SIZE];
int read_buffer_pos = 0;
int read_buffer_end = 0;

/*--------------------------- READ LINE ---------------------------
    */

int readc(int fd, char *c)
{
    if (read_buffer_pos == read_buffer_end)
    {
        read_buffer_end = read(fd, read_buffer, BUFFER_SIZE);
        switch (read_buffer_end)
        {
        case -1:
            perror("read_line");
            break;
        case 0:
            return 0;
            break;
        default:
            read_buffer_pos = 0;
        }
    }
    *c = read_buffer[read_buffer_pos++];
    return 1;
}

ssize_t readln(int fd, char *line, size_t size)
{
    int res = 0;
    int i = 0;
    while (i < size && (res = readc(fd, line + i) > 0))
    {
        i++;
        if ((line)[i - 1] == '\n') {
            line[i-1] = '\0';
            return i;
        }
    }
    return i;
}


/*--------------------------- SEND TO SERVER PROC-FILE
    ---------------------------*/

void sendToServerPF(int argc, char *argv[]){
    char * path = "/tmp/fifo";

    int fd = open(path,O_WRONLY), j;

    char buffer[4096]="";
```

```
54
55      sprintf(buffer,"%d %d",getpid(),argc-2);
56
57      for(int i = 2; i < argc ; i++){
58          sprintf(buffer,"%s %s",buffer,argv[i]);
59
60      }
61
62      j = sprintf(buffer,"%s \n",buffer);
63      write(fd,buffer,j);
64      //write(1,buffer,j);
65
66      close(fd);
67  }
68
69
70  /*---------------------------- SEND TO SERVER STATUS
        ----------------------------*/
71
72  void sendToServerStatus(int argc, char *argv[]){
73      char * path = "/tmp/fifo";
74
75      int fd = open(path,O_WRONLY), j;
76
77      char buffer[4096]="";
78
79      j = sprintf(buffer,"%d %d %s \n",getpid(),argc-1,argv[1]);
80      write(fd,buffer,j);
81
82      //write(1,buffer,j);
83
84      close(fd);
85  }
86
87
88  /*---------------------------- RECEIVE FROM SERVER
        ----------------------------*/
89
90  void receiveFromServer(){
91      int fifo_client_r,fifo_client_w,bytes_read;
92      char path[12];
93      char buffer[4096];
94
95      for (int i = 0;i<11;i++){
96          path[i] = '\0';
97      }
98
99      sprintf(path,"/tmp/%d",getpid());
100     mkfifo(path, 0666);
101
102     if ((fifo_client_r = open(path, O_RDONLY)) == -1) {
103
104         perror("rd open");
105     }
106
107     if((fifo_client_w = open(path, O_WRONLY)) == -1){
108         perror("wr open");
```

```
    }

    while((bytes_read = readln(fifo_client_r, buffer, 4096)) > 0) {
        char * str = (char*) malloc(bytes_read*sizeof(char));
        strcpy(str,buffer);

        if(strcmp(str,"stop")==0){
            close(fifo_client_w);
        }
        else{
            write(1,buffer,bytes_read);
            write(1,"\n",1);
        }
    }

    close(fifo_client_r);
}
```

A.3 auxClient.h

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <string.h>
#include <sys/wait.h>
#include <stdlib.h>

#ifndef MY_HEADER_H
#define MY_HEADER_H

#define BUFFER_SIZE 4096

int readc(int fd, char *c);
ssize_t readln(int fd, char *line, size_t size);

void sendToServerPF(int argc, char *argv[]);

void sendToServerStatus(int argc, char *argv[]);

void receiveFromServer();

#endif
```

B Código do Servidor

B.1 sdstored.c

```
#include "auxServer.h"

/*---------------------------------- EXECU  O DE OP  ES
    ----------------------------------*/

//fun ao que envia ao cliente informa  o sobre as tasks a correr
     naquele momento
int executeStatus(char pid[]){
  char path[11];
  for (int i = 0;i<11;i++){
    path[i] = '\0';
  }

    sprintf(path,"/tmp/%s",pid);
    mkfifo(path, 0666);

    int fifo_client = open(path,O_WRONLY);

    int f_log1, f_log2;

  if ((f_log2 = open("log.bin", O_RDONLY))==-1){
    perror("[execute status] Erro ao abrir o ficheiro log");
    return -1;
  }

  int j;
  char buffer[4096] = "";

  Processamento aux;

  while((read(f_log2, &aux, sizeof(aux))) > 0){

    if(aux.completed==1){

      j = snprintf(buffer, sizeof(buffer), "task #%d: proc-file %s\
    n", aux.task, aux.comandos);

      write(fifo_client,buffer,j);
    }
    }

    close(f_log2);

    if ((f_log1 = open("log_limites.bin", O_RDONLY))==-1){
    perror("[execute status] Erro ao abrir o ficheiro log");
    return -1;

  }

  Limites aux1;

  if(read(f_log1, &aux1, sizeof(aux1))<0){
    perror("exec status ler limites");
```

```
51      return -1;
52    }
53
54    j = snprintf(buffer, sizeof(buffer), "transf nop: %d\\%d (running
        \\max)\n", aux1.nop_atual,nop_max);
55    write(fifo_client,buffer,j);
56
57    j = snprintf(buffer, sizeof(buffer), "transf bcompress: %d\\%d (
        running\\max)\n", aux1.bcompress_atual,bcompress_max);
58    write(fifo_client,buffer,j);
59
60    j = snprintf(buffer, sizeof(buffer), "transf bdecompress: %d\\%d
        (running\\max)\n", aux1.bdecompress_atual,bdecompress_max);
61    write(fifo_client,buffer,j);
62
63    j = snprintf(buffer, sizeof(buffer), "transf gcompress: %d\\%d (
        running\\max)\n", aux1.gcompress_atual,gcompress_max);
64    write(fifo_client,buffer,j);
65
66    j = snprintf(buffer, sizeof(buffer), "transf gdecompress: %d\\%d
        (running\\max)\n", aux1.gdecompress_atual,gdecompress_max);
67    write(fifo_client,buffer,j);
68
69    j = snprintf(buffer, sizeof(buffer), "transf encrypt: %d\\%d (
        running\\max)\n", aux1.encrypt_atual,encrypt_max);
70    write(fifo_client,buffer,j);
71
72    j = snprintf(buffer, sizeof(buffer), "transf decrypt: %d\\%d (
        running\\max)\n", aux1.decrypt_atual,decrypt_max);
73    write(fifo_client,buffer,j);
74
75    close(f_log1);
76
77      write(fifo_client,"stop\n",5);
78      close(fifo_client);
79
80      return 0;
81  }
82
83
84  //fun ao que executa os comandos enviados pelo cliente
85  void executeServer(char pid[], char line[], int numeroArg) {
86
87    answerClient(pid, "pending\n");
88
89    char delim[] = " ";
90
91    char* cmds[50];
92
93    int j = 0, status;
94
95    char* token = strtok(line, delim);
96    cmds[j] = token;
97    j++;
98
99    while (token != NULL) {
100     //printf(" %s\n",token);
```

```
    token = strtok(NULL, delim);
    cmds[j] = token;
    j++;
  }

  cmds[j] = NULL;

  if (verificaLimites(cmds, numeroArg, pid) == 0) {

    int forigin, fdestination, status, i;

    int fstdout = dup(1);

    while (verificaDisp(cmds, numeroArg) != 1) {
      sleep(2);
    }

    updateLog(pid, 1);

    updateLogConfig(cmds, numeroArg, 0);

    answerClient(pid, "processing\n");

    if (numeroArg == 1) {

      switch (fork()) {
        case -1:
          perror("cria   o de fork");
          _exit(-1);

        case 0:

          forigin = open(cmds[0], O_RDONLY);
          if (forigin < 0) {
            perror("[execute proc-file] open file f_origin");
            _exit(-1);
          }

          fdestination = open(cmds[1], O_CREAT | O_WRONLY | O_TRUNC
    , 0666);
          if (fdestination < 0) {
            perror("[execute proc-file] open file f_destination");
            _exit(-1);
          }

          dup2(forigin, 0);
          dup2(fdestination, 1);
          close(forigin);
          close(fdestination);

          strcat(path, cmds[2]);
          //write(fstdout,path,sizeof(path));
          execl(path, cmds[2], NULL);
          perror("exec numeroarg == 1");
          _exit(0);

        default:
```

```
157           wait(&status);
158       }
159
160       updateLog(pid, 2);
161       updateLogConfig(cmds, numeroArg, 1);
162       answerClient(pid, "concluded\n");
163       answerClient(pid, "stop\n");
164     }
165     else {
166
167       //sleep(5);
168       int NC = numeroArg;
169
170       int p[NC - 1][2];
171
172       i = 0;
173
174       while (i < NC) {
175
176         if (i == 0) {
177
178           pipe(p[0]);
179
180           switch (fork()) {
181             case -1:
182               perror("cria  o de fork");
183               exit(-1);
184
185             case 0:
186               forigin = open(cmds[0], O_RDONLY);
187               if (forigin < 0) {
188                 perror("[execute proc-file] open file f_origin");
189                 _exit(-1);
190               }
191
192               close(p[0][0]);
193
194               dup2(p[0][1], 1);
195               dup2(forigin, 0);
196               close(forigin);
197               close(p[0][1]);
198
199               strcat(path, cmds[2 + i]);
200               //write(fstdout,path,sizeof(path));
201               execl(path, cmds[2 + i], NULL);
202               perror("exec numeroarg != 1 and i==0");
203               close(1);
204               _exit(0);
205
206             default:
207               close(p[0][1]);
208           }
209         }
210
211         else if (i == NC - 1) {
212
213           switch (fork()) {
```

```
214             case -1:
215               perror("cria  o de fork");
216               exit(-1);
217
218             case 0:
219               fdestination = open(cmds[1], O_CREAT | O_WRONLY |
    O_TRUNC, 0666);
220               if (fdestination < 0) {
221                 perror("[execute proc-file] open file f_destination
    ");
222                 _exit(-1);
223               }
224
225               dup2(p[i - 1][0], 0);
226               dup2(fdestination, 1);
227
228               close(p[i - 1][0]);
229               close(fdestination);
230
231               strcat(path, cmds[2 + i]);
232               execl(path, cmds[2 + i], NULL);
233
234               perror("exec numeroarg != 1 and i==NC-1");
235               _exit(0);
236             default:
237               close(p[i - 1][0]);
238           }
239         }
240
241         else {
242           pipe(p[i]);
243
244           switch (fork()) {
245             case -1:
246               perror("cria  o de fork");
247               _exit(-1);
248
249             case 0:
250               close(p[i][0]);
251
252               dup2(p[i - 1][0], 0);
253               dup2(p[i][1], 1);
254               close(p[i - 1][0]);
255               close(p[i][1]);
256
257               strcat(path, cmds[2 + i]);
258               execl(path, cmds[2 + i], NULL);
259               perror("exec numeroarg != 1 and i!=0 and i!=NC-1");
260               _exit(0);
261
262             default:
263               close(p[i - 1][0]);
264               close(p[i][1]);
265
266           }
267         }
268         i++;
```

```
269       }
270
271       for (int k = 0; k < NC; k++) {
272         //int status;
273         wait(&status);
274       }
275
276       updateLog(pid, 2);
277       updateLogConfig(cmds, numeroArg, 1);
278       answerClient(pid, "concluded\n");
279       answerClient(pid, "stop\n");
280     }
281   }
282   else {
283     updateLog(pid, 2);
284   }
285 }
286
287 /*---------------------------------- PARSING
      ----------------------------------*/
288
289 // fun   o que faz parse aos argumentos enviados pelo cliente e
      executa-os
290 void parseServer(char line[],int n){
291
292   int length=0;
293   char * n_argc;
294   char * pid;
295
296   pid = strtok(line, " ");
297   length+=strlen(pid);
298   //printf("%s\n",pid);
299
300   n_argc = strtok(NULL," ");
301   length+=strlen(n_argc)+2;
302
303   int m = atoi(n_argc);
304   //printf("%d\n",m);
305   /*
306
307   printf("%d\n",length);
308   printf("%s\n",line+length);
309   */
310   if(m==1){ //op ao status
311
312     int pid_filho = fork();
313
314     switch(pid_filho){
315       case -1:
316         perror("erro cria   o de fork (status)");
317         _exit(-1);
318
319       case 0:
320         executeStatus(pid);
321         _exit(0);
322
323       default:
```

```
324         printf("Filho %d a executar o comando status\n", pid_filho)
    ;
325     }
326   }
327
328   else{ //op ao proc-file
329
330       int pid_filho = fork();
331
332     switch(pid_filho){
333         case -1:
334           perror("cria   o de fork");
335           _exit(-1);
336
337       case 0:
338           writeToLog(pid,line+length,task);
339         executeServer(pid,line+length,m-2);
340         _exit(0);
341
342       default:
343         printf("Filho %d a executar o comando proc-file %s\n",
    pid_filho, line+length);
344         task ++;
345       }
346     }
347 }
348
349
350 //fun ao que faz parse aos argumentos recebidos pelo servidor
351 int parseArgs(int argc, char* argv[]){
352
353   int log;
354   if (argc == 1 || argc > 3){
355     write(1,"[Usage] ./sdtored <path_config_file> <
    path_tranformations>\n",59);
356     return -1;
357   }
358
359   if(read_Config_File(argv[1])<0){
360     return -1;
361   }
362
363   if((log = open("log.bin",O_WRONLY | O_CREAT | O_TRUNC, 0660))
    ==-1){
364     perror("Impossivel criar log file");
365     return -1;
366   }
367
368   strcpy(path,argv[2]);
369
370   int len = strlen(argv[2]);
371
372   if (argv[2][len-1] != '/'){
373     strcat(path,"/");
374   }
375
376   return 0;
```

```
377 }
378
379
380 /*------------------------------------ MAIN
        -------------------------------------*/
381
382 int main(int argc, char* argv[]) {
383
384   if(parseArgs(argc,argv)<0){
385     return -1;
386   }
387
388   int n, fd, fd2, task;
389   char * path_fifo = "/tmp/fifo";
390     mkfifo(path_fifo, 0666);
391
392   char line[4096];
393
394   fd = open(path_fifo, O_RDONLY);
395   if (fd == -1) {
396     perror("opening fifo");
397   };
398
399   fd2 = open(path_fifo, O_WRONLY);
400   if (fd2 == -1) {
401     perror("opening fifo");
402   };
403
404   task = 0;
405
406   while((n = readln(fd, line, 4096)) > 0) {
407     //write(1,line,n);
408     //write(1,"\n",1);
409     parseServer(line,n);
410   }
411
412   close(fd);
413
414   return 0;
415 }
```

B.2 auxServer.c

```
#include "auxServer.h"

char read_buffer[BUFFER_SIZE];
int read_buffer_pos = 0;
int read_buffer_end = 0;

/*--------------------------- ANSWER CLIENT
    ---------------------------*/

//fun   o que envia ao cliente informa  o sobre o estado de um
    processamento de um ficheiro
void answerClient(char pid[], char mens[]){
    char path[11];

    for (int i = 0;i<11;i++){
        path[i] = '\0';
    }

    sprintf(path,"/tmp/%s",pid);
    mkfifo(path, 0666);

    int fifo_client = open(path,O_WRONLY);

    write(fifo_client,mens,strlen(mens));
    close(fifo_client);
}


/*--------------------------- READ LINE ---------------------------
    */

int readc(int fd, char *c)
{
    if (read_buffer_pos == read_buffer_end)
    {
        read_buffer_end = read(fd, read_buffer, BUFFER_SIZE);
        switch (read_buffer_end)
        {
        case -1:
            perror("read_line");
            break;
        case 0:
            return 0;
            break;
        default:
            read_buffer_pos = 0;
        }
    }
    *c = read_buffer[read_buffer_pos++];
    return 1;
}

ssize_t readln(int fd, char *line, size_t size)
{
    int res = 0;
```

```
53      int i = 0;
54      while (i < size && (res = readc(fd, line + i) > 0))
55      {
56          i++;
57          if ((line)[i - 1] == '\n') {
58              line[i-1] = '\0';
59              return i;
60          }
61      }
62      return i;
63  }
64
65
66  /*--------------------------- READ CONFIG FILE
        ----------------------------*/
67
68  //fun ao que l   o ficheiro de configura  es
69  int read_Config_File(char * path_config){
70      int config_file, log;
71
72      if((config_file = open(path_config,O_RDONLY))==-1){
73          perror("Impossivel abrir config file");
74          return -1;
75      }
76
77      char buffer[200];
78      int i = 0, bytes;
79
80      if((bytes = read(config_file,buffer,200))<0){
81          perror("leitura do config file");
82          return -1;
83      }
84
85      close(config_file);
86
87      char * limites[7];
88      char delim[] = "\n";
89
90      int j = 0;
91
92      char * token = strtok(buffer, delim);
93      limites[j] = token;
94      j++;
95
96      while( token != NULL ) {
97          token = strtok(NULL, delim);
98          limites[j] = token;
99          j++;
100     }
101
102     Limites aux;
103
104     for (j = 0; j<7; j++){
105         if(strstr(limites[j],"nop")!=NULL){
106             int n = strlen(limites[j]);
107             nop_max = limites[j][n-1] - '0';
108             aux.nop_atual = 0;
```

```
109         }
110         else if(strstr(limites[j],"bcompress")!=NULL){
111             int n = strlen(limites[j]);
112             bcompress_max = limites[j][n-1] - '0';
113             aux.bcompress_atual = 0;
114         }
115         else if(strstr(limites[j],"bdecompress")!=NULL){
116             int n = strlen(limites[j]);
117             bdecompress_max = limites[j][n-1] - '0';
118             aux.bdecompress_atual = 0;
119         }
120         else if(strstr(limites[j],"gcompress")!=NULL){
121             int n = strlen(limites[j]);
122             gcompress_max = limites[j][n-1] - '0';
123             aux.gcompress_atual = 0;
124         }
125         else if(strstr(limites[j],"gdecompress")!=NULL){
126             int n = strlen(limites[j]);
127             gdecompress_max = limites[j][n-1] - '0';
128             aux.gdecompress_atual = 0;
129         }
130         else if(strstr(limites[j],"encrypt")!=NULL){
131             int n = strlen(limites[j]);
132             encrypt_max = limites[j][n-1] - '0';
133             aux.encrypt_atual = 0;
134         }
135         else if(strstr(limites[j],"decrypt")!=NULL){
136             int n = strlen(limites[j]);
137             decrypt_max = limites[j][n-1] - '0';
138             aux.decrypt_atual = 0;
139         }
140     }
141 
142     int f_log;
143 
144     if((f_log = open("log_limites.bin",O_WRONLY | O_CREAT | O_TRUNC
    , 0660))==-1){
145         perror("Impossivel criar log file");
146         return -1;
147     }
148 
149     if(write(f_log,&aux,sizeof(aux))<0){
150         perror("Write para o log");
151         return -1;
152     }
153 
154     close(f_log);
155 
156     return 0;
157 }
158 
159 
160 /*--------------------------- UPDATE LOG
    ----------------------------*/
161 
162 //fun ao que atualiza certo pedido feito pelo cliente
163 int updateLog(char pid[], int flag){
```

```
164     int f_log;
165 
166     if ((f_log = open("log.bin", O_RDWR))==-1){
167         perror("[update log] Erro ao abrir o ficheiro log");
168         return -1;
169     }
170 
171     Processamento aux;
172     int w;
173 
174     while((w = read(f_log,&aux,sizeof(aux)))>0){
175 
176         if (strcmp(aux.pid,pid)==0){
177 
178             aux.completed = flag;
179 
180             off_t o = lseek(f_log,-sizeof(aux),SEEK_CUR);
181 
182             write(f_log,&aux,sizeof(aux));
183 
184             break;
185         }
186     }
187 
188     if(w<0){
189         perror("Write");
190         close(f_log);
191         return -1;
192     }
193 
194     close(f_log);
195 
196     return 0;
197 }
198 
199 
200 /*--------------------------- UPDATE LOG CONFIG
    ----------------------------*/
201 
202 int updateLogConfig(char * cmds[], int numeroArg, int flag){
203     int log;
204 
205     if((log = open("log_limites.bin", O_RDONLY))==-1){
206         perror("[update config file] Erro ao abir o ficheiro log");
207         return -1;
208     }
209 
210     Limites aux;
211 
212     if(read(log,&aux,sizeof(aux))==-1){
213         perror("[update config file] Erro a ler do ficheiro");
214         return -1;
215     }
216 
217     close(log);
218 
219     if(flag==0){
```

```
220
221         for (int i = 0; i < numeroArg; i++){
222             if(strcmp(cmds[i+2],"nop")==0){
223                 aux.nop_atual += 1;
224             }
225
226             else if(strcmp(cmds[i+2],"bcompress")==0){
227                 aux.bcompress_atual = aux.bcompress_atual+1;
228             }
229
230             else if(strcmp(cmds[i+2],"bdecompress")==0){
231                 aux.bdecompress_atual += 1;
232             }
233
234             else if(strcmp(cmds[i+2],"gcompress")==0){
235                 aux.gcompress_atual += 1;
236             }
237
238             else if(strcmp(cmds[i+2],"gdecompress")==0){
239                 aux.gdecompress_atual += 1;
240             }
241
242             else if(strcmp(cmds[i+2],"encrypt")==0){
243                 aux.encrypt_atual += 1;
244             }
245
246             else if(strcmp(cmds[i+2],"decrypt")==0){
247                 aux.decrypt_atual += 1;
248             }
249         }
250     }
251
252     else{
253
254         for (int i = 0; i < numeroArg; i++){
255             if(strcmp(cmds[i+2],"nop")==0){
256                 aux.nop_atual -= 1;
257             }
258
259             else if(strcmp(cmds[i+2],"bcompress")==0){
260                 aux.bcompress_atual -= 1;
261             }
262
263             else if(strcmp(cmds[i+2],"bdecompress")==0){
264                 aux.bdecompress_atual -= 1;
265             }
266
267             else if(strcmp(cmds[i+2],"gcompress")==0){
268                 aux.gcompress_atual -= 1;
269             }
270
271             else if(strcmp(cmds[i+2],"gdecompress")==0){
272                 aux.gdecompress_atual -= 1;
273             }
274
275             else if(strcmp(cmds[i+2],"encrypt")==0){
276                 aux.encrypt_atual -= 1;
```

```
277             }
278 
279             else if(strcmp(cmds[i+2],"decrypt")==0){
280                 aux.decrypt_atual -= 1;
281             }
282         }
283     }
284 
285     if((log = open("log_limites.bin", O_WRONLY ))==-1){
286         perror("[update config file] Erro ao abir o ficheiro log");
287         return -1;
288     }
289 
290     if(write(log,&aux,sizeof(aux))<0){
291         perror("[log update] Escrita do ficheiro!");
292         return -1;
293     }
294 
295     return 0;
296 }
297 
298 
299 /*--------------------------- WRITE LOG ----------------------------
    */
300 
301 //fun ao que escreve num ficheiro log os pedidos feitos ao
    servidor
302 int writeToLog(char pid[], char cmds[], int task){
303     int f_log;
304 
305     if ((f_log = open("log.bin", O_WRONLY | O_APPEND, 0644))==-1){
306         //printf("Msg: %s, Nr: %d\n", sterror(errno),errno);
307         perror("Erro ao abrir o ficheiro log");
308         return -1;
309 
310     }
311 
312     Processamento aux;
313 
314     strcpy(aux.comandos,cmds);
315     strcpy(aux.pid,pid);
316     aux.completed = 0;
317     aux.task = task;
318 
319     if(write(f_log,&aux,sizeof(aux))<0){
320         perror("Write para o log");
321         close(f_log);
322         return -1;
323     }
324 
325     close(f_log);
326 
327     return 0;
328 }
329 
330 
331 /*--------------------------- VERIFICA LIMITES
```

```
    ---------------------------*/
332
333 int verificaLimites(char * cmds[], int numeroArg, char pid[]){
334     int nop = 0, bcom = 0, bdecom = 0, gcom = 0, gdecom = 0,
    encrypt = 0, decrypt = 0;
335
336     for (int i = 0; i<numeroArg; i++){
337         if (strcmp("nop",cmds[2+i])==0){
338             nop += 1;
339             if(nop>nop_max){
340                 answerClient(pid,"O n mero de utiliza  es do
    comando nop excede o limite suportado pelo servidor\n");
341                 answerClient(pid,"stop\n");
342                 return -1;
343             }
344         }
345         else if (strcmp("bcompress",cmds[2+i])==0){
346             bcom += 1;
347             if(bcom>bcompress_max){
348                 answerClient(pid,"O n mero de utiliza  es do
    comando bcompress excede o limite suportado pelo servidor\n");
349                 answerClient(pid,"stop\n");
350                 return -1;
351             }
352         }
353         else if (strcmp("bdecompress",cmds[2+i])==0){
354             bdecom += 1;
355             if(bdecom>bdecompress_max){
356                 answerClient(pid,"O n mero de utiliza  es do
    comando bdecompress excede o limite suportado pelo servidor\n")
    ;
357                 answerClient(pid,"stop\n");
358                 return -1;
359             }
360         }
361         else if (strcmp("gcompress",cmds[2+i])==0){
362             gcom += 1;
363             if(gcom>gcompress_max){
364                 answerClient(pid,"O n mero de utiliza  es do
    comando gcompress excede o limite suportado pelo servidor\n");
365                 answerClient(pid,"stop\n");
366                 return -1;
367             }
368         }
369         else if (strcmp("gdecompress",cmds[2+i])==0){
370             gdecom += 1;
371             if(gdecom>gdecompress_max){
372                 answerClient(pid,"O n mero de utiliza  es do
    comando gdecompress excede o limite suportado pelo servidor\n")
    ;
373                 answerClient(pid,"stop\n");
374                 return -1;
375             }
376         }
377         else if (strcmp("encrypt",cmds[2+i])==0){
378             encrypt += 1;
379             if(encrypt>encrypt_max){
```

```
380                 answerClient(pid,"O n mero de utiliza  es do
    comando encrypt excede o limite suportado pelo servidor\n");
381                 answerClient(pid,"stop\n");
382                 return -1;
383             }
384         }
385 
386         else{
387             decrypt += 1;
388             if(decrypt>decrypt_max){
389                 answerClient(pid,"O n mero de utiliza  es do
    comando decrypt excede o limite suportado pelo servidor\n");
390                 answerClient(pid,"stop\n");
391                 return -1;
392             }
393         }
394     }
395     return 0;
396 }
397 
398 /*--------------------------- VERIFICA DISPONIBILIDADE
    ---------------------------*/
399 
400 int verificaDisp(char * cmds[], int numeroArg){
401     int f_log;
402 
403     if ((f_log = open("log_limites.bin", O_RDONLY))==-1){
404         perror("[verifica disp] Erro ao abrir o ficheiro
    log_limites");
405     }
406 
407     Limites aux;
408 
409     if(read(f_log,&aux,sizeof(aux))==-1){
410         perror("[verificaDisp] Erro a ler do ficheiro log_limites")
    ;
411     }
412 
413     close(f_log);
414 
415     int nop = aux.nop_atual, bcom = aux.bcompress_atual, bdecom =
    aux.bdecompress_atual;
416     int gcom = aux.gcompress_atual, gdecom = aux.gdecompress_atual;
417     int encrypt = aux.encrypt_atual, decrypt = aux.decrypt_atual;
418 
419     for (int i = 0; i<numeroArg; i++){
420         if (strcmp("nop",cmds[2+i])==0){
421             nop += 1;
422             if(nop>nop_max){
423                 return 0;
424             }
425         }
426         else if (strcmp("bcompress",cmds[2+i])==0){
427             bcom += 1;
428             if(bcom>bcompress_max){
429                 return 0;
430             }
```

```
431         }
432         else if (strcmp("bdecompress",cmds[2+i])==0){
433             bdecom += 1;
434             if(bdecom>bdecompress_max){
435                 return 0;
436             }
437         }
438         else if (strcmp("gcompress",cmds[2+i])==0){
439             gcom += 1;
440             if(gcom>gcompress_max){
441                 return 0;
442             }
443         }
444         else if (strcmp("gdecompress",cmds[2+i])==0){
445             gdecom += 1;
446             if(gdecom>gdecompress_max){
447                 return 0;
448             }
449         }
450         else if (strcmp("encrypt",cmds[2+i])==0){
451             encrypt += 1;
452             if(encrypt>encrypt_max){
453                 return 0;
454             }
455         }
456 
457         else{
458             decrypt += 1;
459             if(decrypt>decrypt_max){
460                 return 0;
461             }
462         }
463     }
464 
465     return 1;
466 }
```

B.3 auxServer.h

```c
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <string.h>
#include <sys/wait.h>
#include <stdlib.h>

#ifndef MY_HEADER_H
#define MY_HEADER_H

#define BUFFER_SIZE 4096

int task, nop_max, bcompress_max, bdecompress_max, gcompress_max,
    gdecompress_max, encrypt_max, decrypt_max;
char path[100];

typedef struct{
  char pid[10];
  char comandos[4096];
  int completed; // 0 -> pending, 1-> processing, 2-> completed
  int task;
}Processamento;

typedef struct{
  int nop_atual;
  int bcompress_atual;
  int bdecompress_atual;
  int gcompress_atual;
  int gdecompress_atual;
  int encrypt_atual;
  int decrypt_atual;
}Limites;

void answerClient(char pid[], char mens[]);

int readc(int fd, char *c);
ssize_t readln(int fd, char *line, size_t size);

int read_Config_File(char * path_config);

int updateLog(char pid[], int flag);

int updateLogConfig(char * cmds[], int numeroArg, int flag);

int writeToLog(char pid[], char cmds[], int task);

int verificaLimites(char * cmds[], int numeroArg, char pid[]);

int verificaDisp(char * cmds[], int numeroArg);

#endif
```