Universidade de Coimbra
Faculdade de Ciências e Tecnologias
Departamento de Engenharia Informática

PROJECTO

Sistemas Operativos – 99/00

Paulo José dos Santos Guilhoto
guilhoto@student.dei.uc.pt - Nº 985011444

Susana Patrícia Costa de S. Rosa
srosa@student.dei.uc.pt - Nº 955010015

Índice

Introdução

O presente trabalho prático visa implementar em linguagem C uma aplicação que nos permite, com recurso a chamadas ao sistema operativo UNIX, classificar e ordenar o número de vezes que um determinado utilizador esteve logado na máquina, desde a última reinicialização desta. Através de um pequeno menu, é facultado ao utilizador o acesso a rankings independentes quer ao nível global dos utilizadores do DEI, quer ao nível mais restrito dos inscritos na lista da cadeira de Sistemas Operativos, que iremos considerar serem alunos desta mesma cadeira.

Para o efeito, era necessário fazer uso de vários processos, sendo dois deles responsáveis pela leitura dos dados provenientes dos ficheiros *all_users.dat* e *so_users.dat*, que contêm informação sobre os utilizadores da máquina e sobre os alunos da cadeira de SO respectivamente. Os ficheiros por nós utilizados foram disponibilizados publicamente na lista da cadeira, não tendo por isso qualquer característica que os diferencie dos utilizados pelos restantes colegas. Os outros processos serão descritos posteriormente. Falta ainda referir que este trabalho prático faz apelo a todos os conhecimentos adquiridos ao longo do semestre, o que implica o uso de memória partilhada, semáforos, mensagens, manipulação de ficheiros, etc.

Finalizando, a chamada ao nosso programa deverá conter quatro argumentos, sendo o primeiro o número P de processos Pworkers (encarregues do tratamento de dados), o segundo argumento será o número N de utilizadores a processar, e os restantes argumentos serão os ficheiros fonte (entenda-se *all_users.dat* e *so_users.dat*). Sendo assim, uma possível chamada será:

```
$ ./project P N all_users.dat so_users.dat
```

Especificação Geral

Basicamente, estamos inicialmente perante um problema do tipo produtor / consumidor. A comunicação entre os produtores, no nosso caso os dois processos Preader que lêem uma linha do ficheiro fonte associado *(all_users.dat ou so_users.dat)* e os consumidores, os Pworkers, é implementada através de uma região de memória partilhada e controlada através de três semáforos:

- WORKER – controla a leitura,
- MUTEX -- controla o acesso às variáveis partilhadas *terminado*, *read_pos, write_pos, por_ler* ,
- EMPTY – impede a escrita de uma nova linha no buffer circular quando a posição de escrita coincide com a posição de leitura, e por isso é inicializado com o tamanho do buffer circular,

um array de 2 semáforos para controlar a concorrência entre os dois processos Preader e consequentemente a escrita em memória, e um array de N semáforos, onde cada um controla um Pworker.

Os três primeiros semáforos referidos foram criados com recurso aos métodos definidos no ficheiro *my_sem.h.* Para os dois arrays seguintes, foi necessário recorrer ao processo de criação tradicional.

Cada vez que é lida uma linha, esta é colocada no buffer circular, inicializado com dez pedaços de memória alocados de forma a poderem conter uma cadeia de caracteres correspondente a um login e a um nome. De seguida, é feito um signal() no semáforo de um Pworker determinado aleatoriamente. O próximo processo Preader a entrar em acção também é determinado de forma aleatória.

Na figura 1, está descrita de uma forma simplificada a memória partilhada utilizada na comunicação entre os Preaders e os Pworkers.

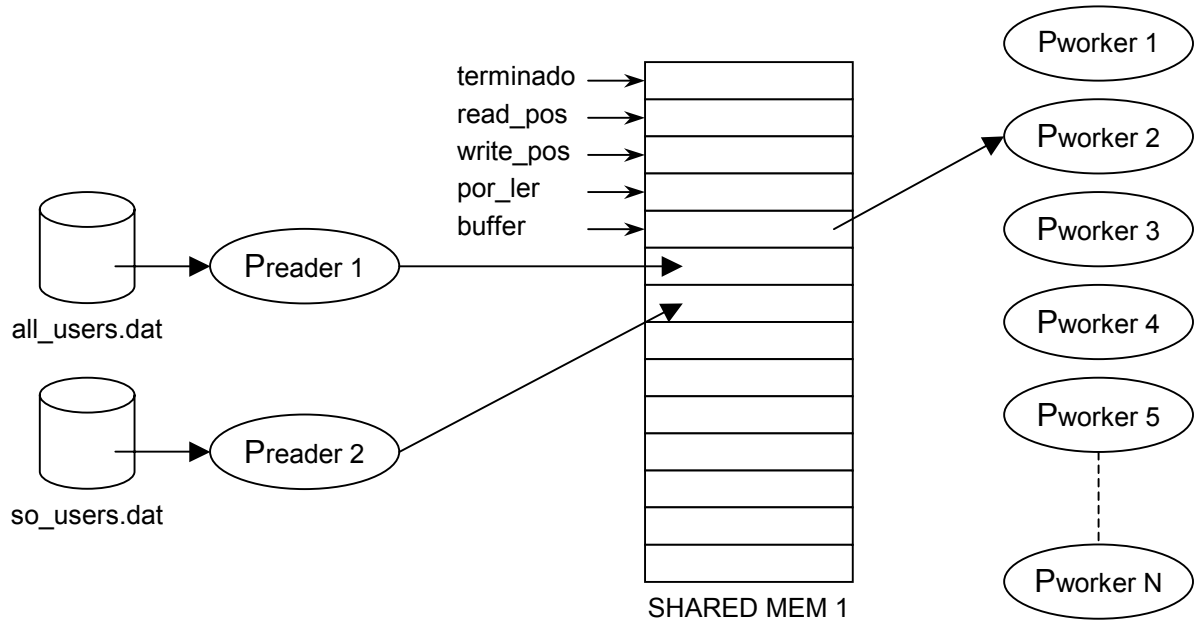

Figura 1 – Esquematização da memória partilhada entre Preaders e Pworkers

Por cada 10 linhas lidas pelos Preaders, é enviada uma mensagem do tipo ping ao Pinput, processo responsável pela comunicação directa com o utilizador, que vai escrever um ponto (.) no ecrã assinalando de forma interactiva como está a decorrer o processamento do programa. A figura 2 representa este envio.

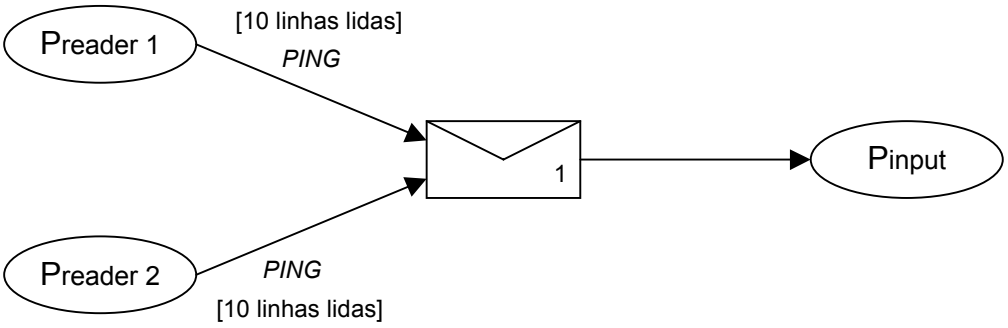

Figura 2 – Representação do envio de mensagens do tipo Ping

Quando um dos Preaders lê N linhas ou chega ao fim do ficheiro, incrementa o valor da variável *terminado*, de modo que os Pworkers fiquem informados de que os produtores já acabaram o seu trabalho. Também é necessário acrescentar que cada vez que é escrita uma linha na memória partilhada, a variável *por_ler* é incrementada e, cada vez que é lida uma linha, é decrementada. De facto, estas duas variáveis são necessárias para garantirmos que é mesmo o último Pworker a entrar em funcionamento que vai enviar um signal ao processo Poracle (quando se verificar a condição *terminado = 2* e *por_ler = 0)*.

Este signal (do tipo *SIGUSR1)* irá 'acordar' o Poracle de modo a que este leve o programa à segunda fase de execução, que iremos falar mais à frente. Só falta aqui acrescentar que no seguimento deste signal, o processo Poracle irá enviar uma mensagem do mesmo tipo que as mensagens ping enviadas anteriormente só que com um conteúdo diferente. Desta forma, o processo Pinput que estava em ciclo à espera de mensagens do tipo 1, irá sair deste, possibilitando assim a continuação do programa. O envio deste signal está representado na figura 3.

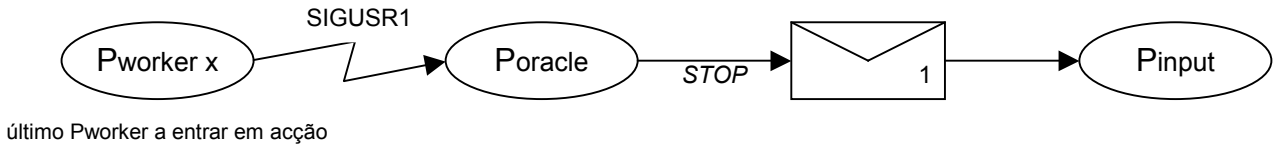

Figura 3 – Envio do signal *SIGUSR1* ao Poracle e sua consequência

Quando um dos processos Pworker recebe algo, a primeira coisa a fazer é identificar de onde vieram os dados. Para isso, apenas é necessário procurar na string lida, o caractere ':' que separa o login do nome do utilizador no ficheiro

all_users.dat. Se este não for encontrado, sabemos que a string provem do ficheiro *so_users.dat*.

De seguida, é necessário escrever a informação recolhida *(<login, nome, num_logins, aluno_so>)* numa tabela que fica em memória partilhada, sendo o campo *num_logins* referente ao número total de vezes que um determinado utilizador esteve logado na máquina desde o último shutdown/startup, e obtido através do comando *last* da shell do Unix. No entanto, surgia aqui um pequeno problema: o comando last dava-nos as datas de todos os logins registados desde a criação do ficheiro de registo, e nós apenas queríamos a informação relativa aos logins efectuados depois do shutdown da máquina. Para isso, foi necessário identificar a data e hora do último reboot (através do comando *last reboot)* para então podermos comparar estes valores com cada registo de login do utilizador. No final, apenas são contabilizados aqueles que foram efectuados após a reinicialização da máquina. Apenas falta acrescentar que, no caso do comando *last reboot* não devolver nenhuma informação, é porque o ficheiro de registo dos logs foi criado após ser feito o último reboot e, nesse caso, todos os logins feitos pelo utilizador são contabilizados.

No entanto, antes de escrever qualquer informação na tabela partilhada, é necessário percorrer sequencialmente cada entrada e comparar o login actual com o login guardado na tabela. Se coincidirem, apenas são preenchidos os campos "em branco". Se o campo *aluno_so* estiver inicialmente a 0, basta passá-lo para 1, indicando que afinal é um aluno da cadeira de Sistemas Operativos. Pelo contrário, se o mesmo campo estiver a 1, é necessário retirar da string lida inicialmente o nome completo do utilizador e colocá-lo na tabela. Se o login não

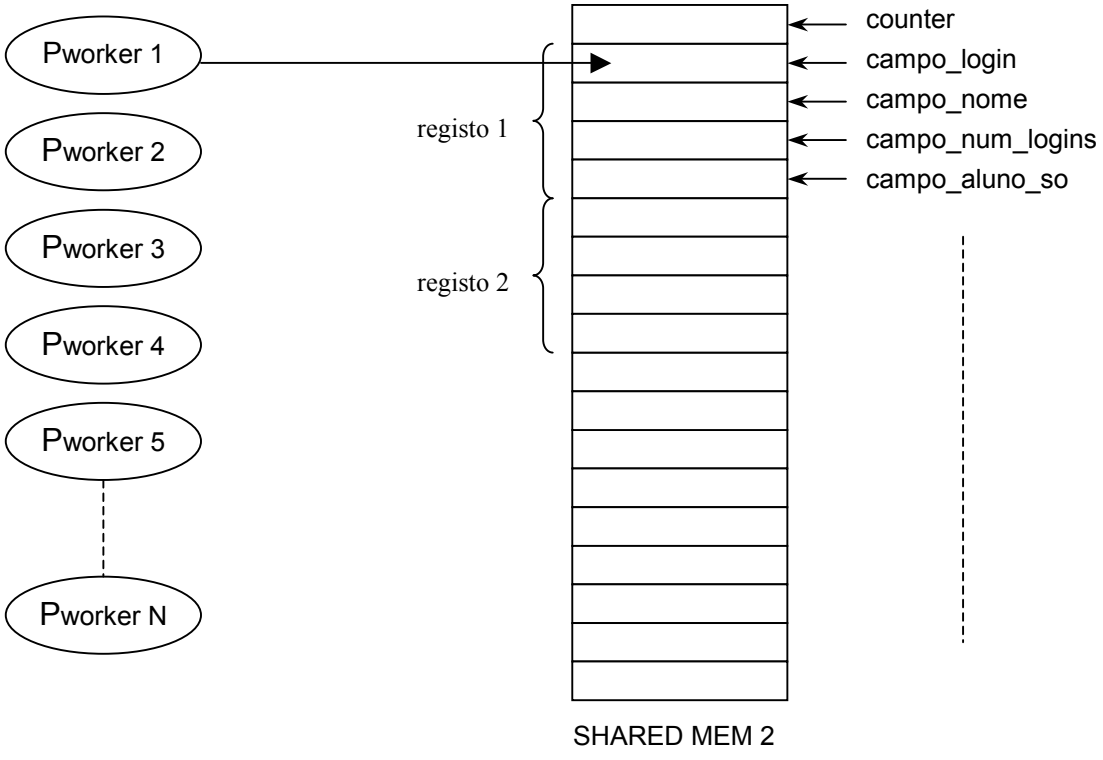

Figura 4 – Esquematização do funcionamento da tabela em memória partilhada

se encontrar ainda na tabela, então é criada uma nova entrada a seguir à última onde serão colocados os dados disponíveis. É fundamental ter aqui em atenção dois pormenores: é necessário haver uma variável partilhada com o número actual de registos na tabela e nunca se deve ultrapassar o número de entradas da tabela em memória partilhada especificado pelo argumento N correspondente ao número de utilizadores a processar.

No entanto, é preciso ler sempre N linhas de ambos os ficheiros *all_users.dat* e *so_users.dat* mesmo que a tabela já tenha N entradas porque é sempre possível encontrar nos ficheiros um utilizador que já tenha um registo nessa mesma tabela e onde apenas são necessários completar alguns campos.

Na figura 4, está representada a estrutura da tabela situada em memória partilhada.

Depois do processo Poracle receber o signal *SIG_USR1,* entramos, na segunda fase do nosso programa (mencionada acima). Em primeiro lugar, é feita uma operação de ordenação da tabela por ordem decrescente do número de logins *(*campo *num_logins).* O algoritmo de ordenação por nós implementado baseou-se no já conhecido Bubble Sort, só que, a dificuldade era acrescida, na medida em que estávamos a lidar com valores obtidos por ponteiros.

Depois de ordenada, é necessário copiar cada entrada da tabela para os ficheiros que têm como função armazenar os rankings *(top_users.dat* e *top_so.dat)*. Cada registo poderá ser copiado para um ou para ambos os ficheiros dependendo se é ou não aluno da cadeira de SO, caso em que o seu registo deverá figurar nos dois rankings.

Em seguida, é enviada a mensagem referida anteriormente (ver figura 3) desencadeando a escrita de:

```
Fim do procedimento!
Tempo total de processamento: x
Número de pings recebidos: y
```

onde x é obtido facilmente através da diferença entre o tempo final de execução e o tempo inicial, registados respectivamente antes destas linhas de output e no início da execução do programa; e y é obtido pela simples contagem do número de mensagens recebidas, do tipo ping, pelo processo Pinput.

Nesta altura, o processo Pinput exibe o seguinte menu:

```
(1) Visualizar o ranking de um utilizador no top global
(2) Visualizar o ranking de um utilizador no top de SO
(3) Visualizar o top global
(4) Visualizar o top de SO
(5) Sair
```

As duas primeiras opções recolhem o login do utilizador a pesquisar no ficheiro em questão e enviam-no ao processo Poracle via mensagem. Um dos campos da mensagem deverá conter a origem da mensagem, isto é, se a escolha recaiu sobre a opção (1) ou sobre a (2). Ao receber a mensagem, o Poracle procura sequencialmente, na tabela situada na memória partilhada, o

login respectivo e vai contando o número de registos percorridos. È óbvio que se estamos à procura de um login que conste do ficheiro *top_so.dat*, todos os registos que tenham o campo *aluno_so=0* são ignorados na pesquisa. Recolhidos os dados do utilizador especificado, estes são devolvidos ao processo Pinput por meio de outra mensagem.

No caso das opções (3) e (4), é efectuada uma simples chamada de rotina que fará a leitura, linha a linha, do ficheiro correspondente.

Na opção (5), é enviada uma mensagem ao Poracle, indicando o final do programa, permitindo a este último sair do ciclo em que estava envolvido e chamar a rotina de limpeza dos recursos IPC e dos ficheiros auxiliares utilizados *(cleanup())*.

A figura 5 ilustra toda esta troca de mensagens entre Pinput e Poracle.

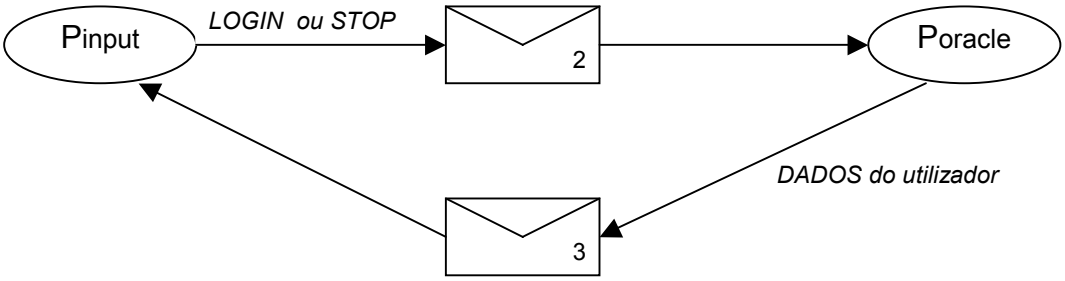

Figura 5 – Troca de mensagens entre Pinput e Poracle

Especificação Interna

`int main()`

Começa-se por extrair os argumentos da chamada do nosso programa e detectar possíveis inconsistências. De seguida, são abertos ou criados todos os ficheiros que irão ser utilizados pelos diversos processos. São então criados todos os recursos de IPC (semáforos, memórias partilhadas e filas de mensagens) necessários ao longo do programa. Só depois é que são criados todos os processos filhos necessários, e são chamadas as funções respectivas descritas mais à frente. Resta esclarecer que foi considerado que o processo pai é o processo Poracle, simplesmente por motivos de comodidade, já que tornava mais simples o envio do signal pelo último Pworker, bastando referir *getppid()* no lugar do PID do processo receptor *(signal(getppid(),SIG_USR1))*.

`int erro_parametros()`

Esta função é chamada sempre que é detectada uma introdução inválida nos argumentos da chamada do programa. Exibe também a sintaxe de uma chamada tipo.

`void processo_Preader(int i)`

Chamada duas vezes (uma por cada Preader), esta função executa um ciclo infinito. Os Preaders, apenas poderão ficar definitivamente parados (através de um *pause()*), caso a linha lida seja igual a *num_users* ou se tenha chegado à última linha do ficheiro.

Cada processo, começa por fazer um wait() no seu respectivo semáforo e de seguida um wait() no semáforo EMPTY, para garantir que não sejam sobrepostas linhas no buffer circular sem que estas tenham sido lidas. Em seguida, é lida uma linha do ficheiro respectivo para a posição correcta (*write_pos*) do buffer. Executada a escrita, é incrementada a variável *write_pos* assim como o número de linhas disponíveis para leitura (*por_ler*).

È então altura de verificar se já foram lidas dez linhas e de enviar uma mensagem tipo ping. Caso se verifique que esta será a última vez que o processo executa, incrementa o valor da variável *terminado.*

Por fim, é feito um signal() no array de semáforos dos Pworkers, numa posição gerada aleatoriamente, e também é feito um signal() num semáforo aleatório dos Preaders para determinar qual o próximo a entrar.

`void processo_Pworker(int i)`

Chamada *num_users* vezes, começa por fazer um wait() no semáforo respectivo *(i)* e no semáforo WORKER, de modo a só ser permitida a entrada de um Pworker de cada vez na 'zona de trabalho'. De seguida, faz uma chamada à função *recebeu(int)* para ser processada toda a informação recolhida, incrementa a posição de leitura *(read_pos)* e decrementa o número de linhas que ainda restam a ler *(por_ler)*.

Basta então verificar se se trata do último Pworker a entrar em funcionamento *(terminado = 2* e *por_ler = 0)* para este poder enviar um signal SIGUSR1 ao Poracle (que é o pai).

Esta função termina com um signal() no semáforo WORKER de modo a deixar a passagem livre a outro processo.

void processo_Pinput();

Escreve '`Estou a processar`' e fica à espera de mensagens do tipo ping (do tipo 1), escrevendo um ponto (.) e incrementando a variável *num_pings* por cada um deles. Caso seja recebida uma mensagem com um conteúdo diferente, sai do ciclo e regista a hora (em segundos) de forma a poder ser calculado o tempo total de processamento.

De seguida, escreve o menu já referido na Especificação Geral e fica à espera de uma opção.

Caso sejam escolhidas uma das duas opções de pesquisa por login (opções (1) e (2)), é enviada uma mensagem do tipo 2 *(msg_input)*, com o login introduzido pelo utilizador, para o processo Poracle. Depois, basta ficar à espera da resposta via mensagem *(msg_searchResults)* e exibir o resultado da pesquisa, apenas se a mensagem trouxer o campo *valid* igual a 1. Caso contrário, esse login não foi encontrado e é exibida uma mensagem de erro.

No caso de serem escolhidas as opções (3) e (4), é chamada a rotina *ver_top(char *)* que exibe o conteúdo do top pretendido.

Se a opção escolhida for a opção (5), é enviada uma mensagem ao Poracle (que tinha ficado à espera de um login por pesquisar) com a indicação de que este pode terminar, e o processo Pinput fica aqui parado.

void processo_Poracle();

Para iniciar, vamos apenas relembrar que este processo entra nesta função somente após ter sido 'acordado' do *pause()*.

Começa então por ordenar a tabela situada em memória partilhada *(ordenaTabela())* e salvaguarda os registos no *top_users.dat* e no *top_so.dat (saveRankings())*. É então enviada uma mensagem do mesmo tipo que os pings (tipo 1) ao processo Pinput de forma a este sair da espera em que estava envolvido.

Depois, fica à espera de uma mensagem *msg_input* (do tipo 2) que seria enviada pelo Pinput caso fosse seleccionada a opção (1) ou (2) do menu. Caso chegue uma mensagem com um login, verifica primeiro a sua origem de forma a saber em que 'população' de utilizadores procurar. Feito isto, envia a mensagem *msg_searchResults* (do tipo 3) com os dados pedidos. Se não foi possível encontrar o login, o campo *valid* da mensagem é preenchido com 0.

Se a mensagem recebida indicar o fim da execução do programa (por opção do utilizador), o Poracle sai imediatamente desta função e vai executar a rotina *cleanup(),* chamada na função *main().*

int recebeu()

Esta função é chamada sempre que é permitido a um processo Pworker ler uma linha da memória partilhada, em primeiro lugar, copia a linha situada na posição de leitura adequada para um buffer auxiliar e faz um signal() de imediato no semáforo EMPTY, de modo a que os Preaders possam escrever outra linha na memória partilhada sem que haja perca de tempo.

Para determinar a origem da linha, basta procurar o caracter *':',* pois sabemos que este só se encontra nos dados provenientes de *all_users.dat.* Depois de separar o login do nome do utilizador (se possível), é chamada a rotina *posiciona(char *)* para direccionar os ponteiros ou para o último registo, ou para um registo que já contenha dados relativos a este login. Neste último caso, vamos verificar o valor de *campo_aluno_so*, porque teremos a garantia de que se este for igual a 0, já temos os dados todos em relação ao utilizador, bastando assim colocar esse campo a 1. Na situação em que seja 1, bastará copiar o nome para *campo_nome* e contar o número de logins através da função *countNum_Logins(login, int)*. Se ainda não existir esse registo, apenas são preenchidos os campos disponíveis.

Na chamada à função *countNum_Logins(char * , int x)*, o valor de x é 0 se houve registo do último reboot, e 1 caso contrário.

int posiciona(char *login)

Percorre a tabela à procura de uma entrada onde *campo_login* é igual a *login*. Dado que os ponteiros vão sendo sempre incrementados com o tamanho do registo, se não houver nenhuma ocorrência do *login*, então estes ficarão a apontar para o último registo, devolvendo 0. Devolve 1, caso contrário.

int readline(int fd, char *buf)

Lê uma linha do ficheiro *fd* e coloca-a no buffer apontado por *buf.*

Devolve 1 se a linha lida for a última do ficheiro, e 0 caso contrário.

int ordenaTabela()

Implementação do algoritmo Bubble Sort, mas desta vez com o objectivo de obter uma ordenação decrescente dos registos de cada utilizador na tabela partilhada pelo *campo_num_logins.* Se necessário, é efectuada a troca dos registos com recurso à função *troca(int,int).*

int troca(int a,int b)

Troca a ordem dos registos *a* e *b* da tabela situada em memória partilhada com recurso a campos auxiliares para armazenar os valores de todos os campos dos registos.

int saveRankings()

Percorre a tabela já ordenada, registo a registo, e grava cada entrada no *top_users.dat* e, se se verificar que o referido utilizador também é aluno da cadeira de SO *(campo_aluno_so = 1),* também é gravada no ficheiro *top_so.dat*.

int reset_pointers()

Coloca os ponteiros *campo_login, campo_nome, campo_num_logins* e *campo_aluno_so* a apontar para o primeiro registo.

int search_All(char *login)

Percorre sequencialmente a tabela, comparando *campo_login* com *login*. Caso sejam coincidentes, são preenchidos os campos da mensagem que irá ser enviada como resposta ao Pinput *(msg_searchResults)* com os dados da tabela respectivos e devolve 1. Caso contrário, devolve 0.

int search_SO(char *login)

Possui a mesma funcionalidade que a função anterior, só que desta vez, apenas são pesquisados os registos que tenham *campo_aluno_so = 1*.

int ver_top(char *filename)

Faz a listagem do ficheiro *filename* com recurso a uma chamada ao comando *more* do sistema. Também foi incluído o algoritmo para a leitura linha a linha do ficheiro.

int getRebootData()

Em primeiro lugar, executa uma chamada ao sistema com o comando "*last reboot | wc –l "* e redirecciona o output para o ficheiro *tmp.01*, com o objectivo de verificar se de facto a última reinicialização da máquina foi efectuada após a criação do ficheiro de registo. Para isso, basta subtrair 2 ao número que consta no ficheiro *tmp.01*, dado que, mesmo que não haja ocorrência de nenhum registo, verifica-se sempre a presença de duas linhas: uma em branco e outra com a data da criação do ficheiro de registo.

Se obtermos o número zero, então todas as entradas no ficheiro de registo são válidas para qualquer utilizador, dado que foram todas registadas após a reinicialização da máquina. Neste caso é devolvido 1.

Caso contrário, é executado o comando *"last reboot | head –1 > tmp.01"*, e são retirados os valores do mês, dia, hora e minutos. É devolvido o valor 0.

int countNum_Logins(char *login, int countAll)

Inicialmente, executa a instrução *"last login | wc –l > tmp.01"*, onde login é passado como parâmetro, e subtrai 2 ao número que consta no ficheiro *tmp.01*. Se o número obtido for zero, devolve imediatamente 0, senão continua.

È verificado o valor do parâmetro *countAll*. Se este for igual a 1, temos a vida simplificada, visto que é apenas necessário devolver o número que obtivemos acima. Caso contrário, as coisas complicam-se um pouco, uma vez que é necessário extrair linha a linha o valor do mês, dia, hora e minutos em que o utilizador esteve logado e compará-los com os valores obtidos pela função *getRebootData().*

Se o mês do login a analisar for igual ao mês do último reboot, basta ir comparando sucessivamente os dias, as horas e os minutos até podermos determinar com clareza e em termos temporais a ordem dos acontecimentos.

No caso de termos um mês diferente do mês do último reboot, vamos ter que considerar, isto se não queremos complicar mais ainda as coisas, que os seis meses a seguir ao mês do reboot (em termos de ordem numérica) vêm de facto depois, em termos cronológicos e os cinco meses antes são anteriores em termos de tempo. Para simplificar, um pequeno exemplo: admitindo que o último reboot foi feito em Janeiro e que o último login foi feito em Fevereiro, então consideramos que este login é posterior à data da última reinicialização (até porque é pouco provável que um determinado utilizador tenha passado 11 meses sem nunca se ter logado e que o ficheiro de registo nunca tenha sido limpo), fazendo com que seja considerado válido. Este método não é de forma alguma completamente correcto, mas verificou-se ser, na maior parte das vezes, bastante eficaz.

Este método repete-se linha a linha até se chegar a um login 'inválido' ou até se esgotarem todos os registos de logins desse utilizador.

`int getMes(char *mes)`

Devolve o valor inteiro correspondente ao mês indicado por extenso pelo ponteiro *mes*.

`int cleanup()`

Tem como objectivo a libertação de todos os recursos IPC utilizados, onde estão incluídos os semáforos, as memórias partilhadas, e as filas de mensagens. Para além disto, também remove um ficheiro auxiliar utilizado para contar o número de logins de um determinado utilizador.

Antes de acabar, o processo que está a executar esta função envia um signal, do tipo *SIGINT* (que até poderia ser outro), a todos os processos de modo a assegurar que serão todos 'mortos' e suicida-se.

Anexos

A – Código do programa

```
/*************************************************************/
/* FICHEIRO: projecto.c                                      */
/* AUTORES:  Susana Rosa & Paulo Guilhoto                    */
/* VERSAO:   1.06                                            */
/* DATA:     30/01/2000                                      */
/*************************************************************/


#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <sys/shm.h>
#include <sys/msg.h>
#include <sys/time.h>
#include <time.h>
#include <signal.h>
#include "my_sem.h"


#define LOGIN_SIZE 12
#define NAME_SIZE 30
#define NUM_LOGINS_SIZE sizeof(int)
#define ALUNO_SO_SIZE sizeof(int)
#define SIZE_OF_BUFFER 10

#define MAX_WORKERS 20

#define SEMKEY IPC_PRIVATE
#define SHMKEY1 IPC_PRIVATE
#define SHMKEY2 IPC_PRIVATE

#define SIZE_OF_MSG1 LOGIN_SIZE+sizeof(int)
#define SIZE_OF_MSG2 NAME_SIZE+3*sizeof(int)


int WORKER,MUTEX,EMPTY;
int semid1,semid2;
int shmid1,shmid2;
int msgqid;

struct sembuf worker_wait[MAX_WORKERS];
struct sembuf worker_signal[MAX_WORKERS];
struct sembuf reader_wait[2];
struct sembuf reader_signal[2];


struct my_msg1{      // msg para os pings e para enviar logins do Pinput para o
Poracle
  long mtype;
  char data[LOGIN_SIZE];
  int  origin;
} msg_ping,msg_input,msg_receive1;


struct my_msg2{      // msg para resposta do Poracle ao Pinput
  long mtype;
  char nome[NAME_SIZE];
  int  num_logins;
  int  ranking;
  int  valid;
} msg_searchResults,msg_receive2;
```

```
int num_workers, num_users;
int registo_size = LOGIN_SIZE+NAME_SIZE+NUM_LOGINS_SIZE+ALUNO_SO_SIZE;
int linha_size = LOGIN_SIZE+NAME_SIZE;
int fd1,fd2;
FILE *file3,*file4;

char lastMonth[4];
int lastDay;
int lastHour;
int lastMin;
int rebootVazio;

int *terminado;
int *read_pos;
int *write_pos;
int *por_ler;
char *first;
char *buffer;
int *counter;
char *campo_login;
char *campo_nome;
int *campo_num_logins;
int *campo_aluno_so;

time_t *t;
long sec_ini,total_sec;


int erro_parametros();

void processo_Preader(int);
void processo_Pworker(int);
void processo_Pinput();
void processo_Poracle();

void sig_handler();
int recebeu();
int posiciona();
int readline(int,char *);
int ordenaTabela();
int troca(int,int);
int saveRankings();
int reset_pointers();
int search_All(char *);
int search_SO(char *);
int ver_top(char *);
int getRebootData();
int countNum_Logins(char *,int);
int getMes(char *);
void cleanup();

extern char *shmat();



/*----------------------------------- MAIN -----------------------------------
*/
main(int argc,char **argv) {

ushort init1[MAX_WORKERS];
ushort init2[2];
union semun aux;

int i,j;
int proc_id;
```

```
/*************************************************
**     TRATAMENTO DOS PARAMETROS DE ENTRADA         **
*************************************************/

if (argc!=5)
  erro_parametros();
if (sscanf (argv[1], "%d", &num_workers) == 0)
  erro_parametros();
if (sscanf (argv[2], "%d", &num_users) == 0)
  erro_parametros();



/*************************************************
**           ABERTURA DOS FICHEIROS                **
*************************************************/

// Abre ficheiro all_users.dat
if ((fd1=open(argv[3],O_RDONLY)) == -1) {
  printf("O ficheiro %s nao existe!\n",argv[3]);
  exit(-1);
}


// Abre ficheiro so_users.dat
if ((fd2=open(argv[4],O_RDONLY)) == -1) {
  printf("O ficheiro %s nao existe!\n",argv[4]);
  exit(-1);
}



/*************************************************
**           CRIACAO DOS FICHEIROS                 **
*************************************************/

// Cria ficheiro top_users.dat
if ((file3=fopen("top_users.dat","w+")) == 0) {
  printf("Nao foi possivel criar o ficheiro top_users.dat\n");
  exit(-1);
}


// Cria ficheiro top_so.dat
if ((file4=fopen("top_so.dat","w+")) == 0) {
  printf("Nao foi possivel criar o ficheiro top_so.dat\n");
  exit(-1);
}


/*************************************************
**         REDIRECCIONAMENTO DOS SINAIS            **
*************************************************/

/* Redireccionamento dos sinais */
for (i=1;i<20;i++)
  if (i!=17)                // o signal 17 nao e' redireccionado por causa das
chamadas
    signal(i,cleanup);      // ao sistema


signal(SIGUSR1,sig_handler);


/*************************************************
**         CRIACAO DA MEMORIA PARTILHADA           **
*************************************************/
```

```
/* Criacao memoria partilhada para comunicacao entre readers e workers */
shmid1 = shmget(SHMKEY1,4*sizeof(int)+linha_size*SIZE_OF_BUFFER,0777|IPC_CREAT);
if (shmid1 == -1) {
  printf("Nao foi possivel criar a regiao de memoria partilhada\n");
  exit(-1);
}

terminado = (int *) shmat(shmid1,0,0);
read_pos = (int *)(terminado + sizeof(int));
write_pos = (int *)(read_pos + sizeof(int));
por_ler = (int *)(write_pos + sizeof(int));
buffer = (char *)(por_ler + sizeof(int));



/* Cria tabela partilhada */
shmid2 = shmget(SHMKEY2,sizeof(int)+registo_size*num_users*5,0777|IPC_CREAT);
if (shmid2 == -1) {
  printf("Nao foi possivel criar a regiao de memoria partilhada\n");
  exit(-1);
}

counter = (int *) shmat(shmid2,0,0);                // indica o # de registos
presentes na shm
first = (char *)(counter + sizeof(int));            // referencia
campo_login = (char *)(first);
campo_nome = (char *)(campo_login + LOGIN_SIZE);
campo_num_logins = (int *)(campo_nome + NAME_SIZE);
campo_aluno_so = (int *)(campo_num_logins + NUM_LOGINS_SIZE);


/* Inicializacao */
*counter = 0;
*terminado = 0;
*read_pos = 0;
*write_pos = 0;
*por_ler = 0;



/*****************************************************
**             CRIACAO DOS SEMAFOROS               **
*****************************************************/

// Semaforos '+ simples'. Sao criados gracas as rotinas simplificadas do
ficheiro my_sem.c
WORKER = sem_create(1);
MUTEX = sem_create(1);
EMPTY = sem_create(SIZE_OF_BUFFER);


// --- Array de semaforos para os Pworkers ----
semid1 = semget(SEMKEY,num_workers,0777|IPC_CREAT);
if (semid1 == -1) {
  printf("Nao foi possivel criar os semaforos\n");
  exit(-1);
}

for(i=0;i<num_workers;i++)
  init1[i] = 0;
aux.array = &init1[0];

semctl(semid1,num_workers,SETALL,aux);

for(i=0;i<num_workers;i++) {
  worker_wait[i].sem_op = -1;
  worker_wait[i].sem_flg = SEM_UNDO;
  worker_wait[i].sem_num = i;
```

```
}

for(i=0;i<num_workers;i++) {
  worker_signal[i].sem_op = 1;
  worker_signal[i].sem_flg = SEM_UNDO;
  worker_signal[i].sem_num = i;
}


// --- Array de semaforos para os Preaders ---
semid2 = semget(SEMKEY,2,0777|IPC_CREAT);
if (semid2 == -1) {
  printf("Nao foi possivel criar os semaforos\n");
  exit(-1);
}

init2[0] = 1;                                          // Preader do all_users
comeca
init2[1] = 0;
aux.array = &init2[0];

semctl(semid2,2,SETALL,aux);

for(i=0;i<2;i++) {
  reader_wait[i].sem_op = -1;
  reader_wait[i].sem_flg = SEM_UNDO;
  reader_wait[i].sem_num = i;
}

for(i=0;i<2;i++) {
  reader_signal[i].sem_op = 1;
  reader_signal[i].sem_flg = SEM_UNDO;
  reader_signal[i].sem_num = i;
}



/*****************************************************
**          CRIACAO DAS FILAS DE MENSAGENS           **
*****************************************************/
// Mensagens do tipo ping
msgqid = msgget(IPC_PRIVATE,0777|IPC_CREAT);
if (msgqid == -1) {
  printf("Nao foi possivel criar as filas de mensagens\n");
  exit(-1);
}

msg_ping.mtype = 1;
msg_input.mtype = 2;
msg_searchResults.mtype = 3;




/*****************************************************
**                      RELOGIO                     **
*****************************************************/
//Salvaguarda dos valores actuais do relogio
sec_ini = (long)(time(t));

srand(time(t));



/*****************************************************
**                    PRINCIPAL                      **
*****************************************************/
```

```
rebootVazio = getRebootData();                // vai buscar a informacao do
ultimo reboot


// ---- pai de todos os processos e' o Poracle ----


for(i=0;i<num_workers;i++)                    // criacao dos processos Pworker
  if (fork()==0) {
    processo_Pworker(i);
  }


for(j=0;j<=1;j++)                             // criacao dos processos Preader
  if (fork()==0){
    processo_Preader(j);                      // j=0 -> allusers     j=1 ->
so_users
  }


proc_id = fork();                             // criacao do processo Pinput
if (proc_id==0){                              // Pinput
  processo_Pinput();
}
else {                                        // Poracle
  pause();
  processo_Poracle();
  cleanup();                                  // chama rotina de limpeza
}

} // do main




/*-------------------------------- ERRO_PARAMETROS ----------------------------
---*/
erro_parametros() {
  printf("Formato incorrecto de parametros\n");
  printf("Usage: project N P all_users.dat so_users.dat\n");
  printf(" -> N : numero de utilizadores da maquina\n");
  printf(" -> P : numero de processos trabalhadores\n");
  exit(0);
}




/*-------------------------------- PROCESSO_PREADER ---------------------------
---*/
void processo_Preader(int j) {
  int linhas_lidas=0;
  int eof=0;
  int nrand;

  while(1) {
    semop(semid2,&reader_wait[j],1);
    sem_wait(EMPTY);

    if (j==0)
      eof = readline(fd1,(char *)(buffer + linha_size*(*write_pos)));
    else
      eof = readline(fd2,(char *)(buffer + linha_size*(*write_pos)));

    sem_wait(MUTEX);
    *write_pos = (*write_pos + 1) % SIZE_OF_BUFFER;
    *por_ler  = *por_ler +1;
```

```
    sem_signal(MUTEX);

    linhas_lidas++;
    if ((linhas_lidas % 10) == 0){
      strcpy(msg_ping.data,"ping");
      msgsnd(msgqid,&msg_ping,SIZE_OF_MSG1,0);
    }

    if ((linhas_lidas==num_users) || (eof==1)) {
      if (j==0)
        close(fd1);
      else
        close(fd2);

      sem_wait(MUTEX);
      *terminado = *terminado + 1;
      sem_signal(MUTEX);
    }

    nrand = (int)(rand() % num_workers);      // faz-se um signal num Pworker
determinado
    semop(semid1,&worker_signal[nrand],1);    // aleatoriamente

    if ((linhas_lidas==num_users) || (eof==1)) {
      semop(semid2,&reader_signal[1-j],1);    // ´e feito um signal no outro
Preader
      pause();                                // fica parado
    }

    if (*terminado == 0){                     // 2 Preaders a trabalhar
      nrand = (int)(rand() % 2);
      semop(semid2,&reader_signal[nrand],1);
    }
    else                                      // so' ja existe um Preader
      semop(semid2,&reader_signal[j],1);      // faz o signal nele proprio
  }
}




/*------------------------------- PROCESSO_WORKER -----------------------------
--*/
void processo_Pworker(int i) {
  while(1){
    semop(semid1,&worker_wait[i],1);

    sem_wait(WORKER);
    recebeu(i);
    *read_pos = (*read_pos + 1) % SIZE_OF_BUFFER;    // passa p/ posicao de
leitura seg.

    sem_wait(MUTEX);
    *por_ler  = *por_ler - 1;                 // temos menos uma linha para ler

    if ((*terminado == 2) && (*por_ler <= 0)){      // e' o ultimo Pworker a
trabalhar
      kill(getppid(),SIGUSR1);
      pause();
    }
    sem_signal(MUTEX);
    sem_signal(WORKER);
  }
}
```

```
/*------------------------------- PROCESSO_PINPUT ------------------------------
---*/
void processo_Pinput() {
  int continua=0;
  int num_pings=0;
  char login[LOGIN_SIZE];
  int opcao=0;


  printf("Estou a processar ");
  fflush(stdout);
  do {
    msgrcv(msgqid,&msg_receive1,SIZE_OF_MSG1,1,0);   // fica bloqueado a espera
de 1a msg
    if (strcmp(msg_receive1.data,"ping")==0) {
      printf(".");
      fflush(stdout);
      num_pings++;
    }
    else continua=1;                          // recebeu uma mensagem para sair
do ciclo
  } while (continua == 0);


  total_sec = (long)(time(t)) - sec_ini;      // calculo do tempo total em
segundos


  printf("\n\nFim do procedimento!\n");
  printf("Tempo total de processamento: %ld seg \n",total_sec);
  printf("Numero de pings recebidos: %d\n",num_pings);
  fflush(stdout);


  while (opcao!=5) {                          // mostra Menu
    printf("\n(1) Visualizar o ranking de um utilizador no top global\n");
    printf("(2) Visualizar o ranking de um utilizador no top de SO\n");
    printf("(3) Visualizar o top global\n");
    printf("(4) Visualizar o top de SO\n");
    printf("(5) Sair\n");

    printf("\nIntroduza sua opcao: ");
    scanf("%d",&opcao);

    switch (opcao){
      case 1: printf("\nIntroduza um login valido: ");
              scanf("%s",login);
              strcpy(msg_input.data,login);
              msg_input.origin = 0;
              msgsnd(msgqid,&msg_input,SIZE_OF_MSG1,0);
              msgrcv(msgqid,&msg_receive2,SIZE_OF_MSG2,3,0);
              if (msg_receive2.valid==0)
                printf("Este login nao consta no top global\n");
              else {
                printf("Nome: %s\n",msg_receive2.nome);
                printf("Numero de logins: %d\n",msg_receive2.num_logins);
                printf("Ranking no top global: %d\n",msg_receive2.ranking);
              }
              fflush(stdout);
              break;

      case 2: printf("\nIntroduza um login valido: ");
              scanf("%s",login);
              strcpy(msg_input.data,login);
              msg_input.origin = 1;
              msgsnd(msgqid,&msg_input,SIZE_OF_MSG1,0);
              msgrcv(msgqid,&msg_receive2,SIZE_OF_MSG2,3,0);
```

```
              if (msg_receive2.valid==0)
                printf("Este login nao consta no top de SO\n");
              else {
                printf("Nome: %s\n",msg_receive2.nome);
                printf("Numero de logins: %d\n",msg_receive2.num_logins);
                printf("Ranking no top de SO: %d\n",msg_receive2.ranking);
              }
              break;

      case 3: ver_top("top_users.dat");
              break;

      case 4: ver_top("top_so.dat");
              break;

      case 5: printf("\nExiting\n");
              break;

      default: printf("\nOpcao incorrecta - tente novamente\n");
    }
  }

  strcpy(msg_input.data,"exit");
  msgsnd(msgqid,&msg_input,SIZE_OF_MSG1,0);

  pause();
}



/*------------------------------ PROCESSO_PORACLE -----------------------------
---*/
void processo_Poracle() {
  int valido=0;
  int continua=0;

  ordenaTabela();
  saveRankings();

  strcpy(msg_ping.data,"stop");
  msgsnd(msgqid,&msg_ping,SIZE_OF_MSG1,0);          // envia msg para o Pinput
sair do ciclo

  do {
    msgrcv(msgqid,&msg_receive1,SIZE_OF_MSG1,2,0);  // fica a espera de 1a msg
por parte do
    if (strcmp(msg_receive1.data,"exit")!=0) {      //      Pinput
      if (msg_receive1.origin == 0)                 // ve se o login provem do
opcao 1 ou 2
        valido = search_All(msg_receive1.data);
      else
        valido = search_SO(msg_receive1.data);
      if (valido==0)                                //login nao encontrado no
respectivo top
        msg_searchResults.valid = 0;
      msgsnd(msgqid,&msg_searchResults,SIZE_OF_MSG2,0);    //envia msg com os
dados
    }                                                       //respectivos
    else
      continua=1;
  } while(continua==0);
}
```

```
/*---------------------------------- READLINE ----------------------------------
---*/
int readline(int fd,char *buf) {
  char str[LOGIN_SIZE+NAME_SIZE+1]="";
  int indice=0;
  char bytelido='\0';
  int num_byteslidos = 0;

  while (bytelido!='\n'){                          // le uma linha caractere a
caractere
    read(fd,&bytelido,1);
    str[indice]=bytelido;
    indice++;
  }
  str[indice-1] = '\0';
  strcpy(buf,str);


  num_byteslidos = read(fd,&bytelido,1);
  if (num_byteslidos == 0)                         // fim
    return(1);

  lseek(fd,-1,SEEK_CUR);                           // recua uma posicao

  return(0);
}



/*---------------------------------- RECEBEU -----------------------------------
---*/
recebeu(int num_worker) {
  int preader;
  char login[LOGIN_SIZE];
  char nome[NAME_SIZE];
  char buf[LOGIN_SIZE+NAME_SIZE];
  int num_logins = 0;
  int i,j;
  int tmp;

  strcpy(buf,buffer + linha_size*(*read_pos));  // copia a string da mem p/ um
buffer aux.
  sem_signal(EMPTY);                               // e faz de imediato um signal
no semaforo
                                                   // EMPTY p/ poder ser colocada
outra linha
                                                   // e nao haver mais percas de
tempo

  if (index(buf,(int)(':'))!=NULL) {               // envio do all_users.dat
    i=0;                                           // obtencao do login a partir do
buffer
    while(buf[i]!=(char)(':')) {
      login[i] = buf[i];
      i++;
    }
    login[i]='\0';
    for (j=i+1;j<=strlen(buf);j++)                 // obtencao do nome
      nome[j-(i+1)] = buf[j];

    preader = 2;
  }
  else {
    strcpy(login,buf);
    preader = 1;
  }
```

```
  if (posiciona(login)==1) {                    // registo ja existente
    if (*campo_aluno_so == 0){                  // temos tudo menos o
campo_aluno_so
      *campo_aluno_so = 1;
    }
    else {
      strcpy(campo_nome,nome);
      if (rebootVazio!=1)
        *campo_num_logins = countNum_Logins(login,0);
      else
        *campo_num_logins = countNum_Logins(login,1);
    }
  }
  else {                                        // registo ainda nao existe
    if ((*counter + 1) <=num_users) {           // ve se e' possivel criar um
novo
      *counter = *counter + 1;                  // incrementa o contador de
registos
      strcpy(campo_login,login);

      if(preader==1){                           // provem de so_users.dat
        *campo_aluno_so = 1;                    //-> basta registar o valor do
campo aluno_so
      }
      else {                                    // provem de all_users.dat
        strcpy(campo_nome,nome);
        if (rebootVazio!=1)                     // verifica se ha registo do
ultimo reboot
          *campo_num_logins = countNum_Logins(login,0);
        else
          *campo_num_logins = countNum_Logins(login,1);

        *campo_aluno_so = 0;
      }
    }
  }
}



/*---------------------------------- POSICIONA ------------------------------
---*/
int posiciona(char *login) {  /* devolve 0 se NAO existe na matriz */
  int i=0;

  reset_pointers();

  while (i<*counter) {                       // percorre as entradas em busca
de um login igual
    if (strcmp(login,campo_login)==0) {         // encontra o login pedido
      campo_nome = (char *)(campo_nome + i*registo_size);
      campo_num_logins = (int *)(campo_num_logins + i*registo_size);
      campo_aluno_so = (int *)(campo_aluno_so + i*registo_size);
      return(1);
    }
    campo_login = (char *)(campo_login + registo_size);          // aponta para
o seguinte
    i++;
  }
                                                              // pesquisa
infrutifera
  campo_nome = (char *)(campo_nome + i*registo_size);      //->aponta p/ o
ultimo
  campo_num_logins = (int *)(campo_num_logins + i*registo_size);
  campo_aluno_so = (int *)(campo_aluno_so + i*registo_size);

  return(0);
}
```

```
/*--------------------------------- ORDENATABELA ------------------------------
---*/
ordenaTabela() {

  // implementacao do algoritmo BubbleSort, de modo a proporcionar uma ordenacao
  // decrescente

  int i,j;
  int trocou = 1;

  reset_pointers();

  for(i=0;(i<num_users-1) && (trocou==1);i++) {
    trocou = 0;
    for(j=num_users-1;j>i;j--) {
      if(*(campo_num_logins + j*registo_size) > *(campo_num_logins + (j-
1)*registo_size)) {
        troca(j,j-1);
       trocou = 1;
      }
    }
  }
}



/*------------------------------------ TROCA ---------------------------------
--*/
troca(int i, int j) {
  char tmp_login[LOGIN_SIZE];
  char tmp_nome[NAME_SIZE];
  int tmp_num_logins;
  int tmp_aluno_so;

  // aux = a
  // a = b
  // b = aux

  strcpy(tmp_login,(char *)(campo_login + i*registo_size));
  strcpy(tmp_nome,(char *)(campo_nome + i*registo_size));
  tmp_num_logins = *((int *)(campo_num_logins +i*registo_size));
  tmp_aluno_so = *((int *)(campo_aluno_so +i*registo_size));


  strcpy((char*)(campo_login + i*registo_size),(char *)(campo_login +
j*registo_size));
  strcpy((char*)(campo_nome + i*registo_size),(char *)(campo_nome +
j*registo_size));
  *((int *)(campo_num_logins +i*registo_size)) = *((int *)(campo_num_logins
+j*registo_size));
  *((int *)(campo_aluno_so +i*registo_size)) = *((int *)(campo_aluno_so
+j*registo_size));

  strcpy((char *)(campo_login + j*registo_size),tmp_login);
  strcpy((char *)(campo_nome + j*registo_size),tmp_nome);
  *((int *)(campo_num_logins +j*registo_size)) = tmp_num_logins;
  *((int *)(campo_aluno_so +j*registo_size)) = tmp_aluno_so;
}



/*--------------------------------- SAVERANKINGS ------------------------------
---*/
saveRankings() {
  int i=0;
```

```
  char log[3];

  reset_pointers();

  for (i=0;i<*counter;i++){
    fprintf(file3,"%s  ",campo_login + i*registo_size);
    fprintf(file3,"%s  ",campo_nome + i*registo_size);
    fprintf(file3,"%d  ",*(campo_num_logins + i*registo_size));
    fprintf(file3,"%d\n",*(campo_aluno_so + i*registo_size));

    if (*(campo_aluno_so + i*registo_size) == 1) {               // e' um aluno
de SO
      fprintf(file4,"%s  ",campo_login + i*registo_size);
      fprintf(file4,"%s  ",campo_nome + i*registo_size);
      fprintf(file4,"%d  ",*(campo_num_logins + i*registo_size));
      fprintf(file4,"%d\n",*(campo_aluno_so + i*registo_size));
    }
  }

  fclose(file3);
  fclose(file4);
}



/*---------------------------------- RESET_POINTERS ----------------------------
---*/
reset_pointers() {
  campo_login = (char *)(first);
  campo_nome = (char *)(first + LOGIN_SIZE);
  campo_num_logins = (int *)(first + LOGIN_SIZE + NAME_SIZE);
  campo_aluno_so = (int *)(first + LOGIN_SIZE + NAME_SIZE + NUM_LOGINS_SIZE);
}



/*----------------------------------- SEARCHALL ---------------------------------
---*/
search_All(char *login){
  int i=0;
  reset_pointers();

  while(i<*counter){                                  // pesquisa entre todos os
utilizadores
    if (strcmp((char *)(campo_login + i*registo_size),login)==0){
      strcpy(msg_searchResults.nome,(char *)(campo_nome + i*registo_size));
      msg_searchResults.num_logins = *((int *)(campo_num_logins +
i*registo_size));
      msg_searchResults.ranking = i+1;
      msg_searchResults.valid = 1;
      return(1);
    }
    i++;
  }
  return(0);
}



/*----------------------------------- SEARCHSO ----------------------------------
---*/
search_SO(char *login){
  int i=0,j=0;
  reset_pointers();

  while(i<*counter){                                  // pesquisa entre os alunos
de SO
    if ((*(int *)(campo_aluno_so + i*registo_size)) == 1) {
```

```
      if (strcmp((char *)(campo_login + i*registo_size),login)==0){
        strcpy(msg_searchResults.nome,(char *)(campo_nome + i*registo_size));
        msg_searchResults.num_logins = *((int *)(campo_num_logins +
i*registo_size));
        msg_searchResults.ranking = j+1;
        msg_searchResults.valid = 1;
        return(1);
      }
      j++;
    }
    i++;
  }

  return(0);
}



/*---------------------------------- VER_TOP ---------------------------------
---*/
ver_top(char filename[15]) {
  int ret;
  char line[linha_size];
  FILE *filex;
  char instrucao[50];

  printf("******************** %s ********************\n",filename);

  strcpy(instrucao,"more ");
  strcat(instrucao,filename);
  ret= system(instrucao);

  // utilizamos para ler o ficheiro o comando 'more' do sistema por ser mais
pratico
  // no entanto, a seguir esta' um algoritmo pronto a usar que nao recorre a
esta chamada

  /*
  if ((filex=fopen(filename,"r")) == 0) {
    printf("Nao foi possivel abrir o ficheiro %s\n",filename);
    exit(-1);
  }

  while (fgets(line,linha_size,filex)!=NULL){
    printf("%s",line);
  }

  fclose(filex);
  */

  printf("******************************************************\n");
}



/*--------------------------------- SIG_HANDLER ------------------------------
---*/
void sig_handler() {}



/*------------------------------- GETREBOOTDATA --------------------------------
*/
int getRebootData() {
  int filex;
  int ret;
  char buf[10];
  int i=0,output_lines;
```

```
  ret = system("last reboot | wc -l > tmp.01");
  filex = open("tmp.01",O_RDONLY);
  do {
    read(filex,&buf[i],1);
    i++;
  } while (i<7);
  sscanf(buf,"%d",&output_lines);
  if (output_lines-2 == 0){                  // nao ha qq registo
    close(filex);
    return (1);
  }

  ret = system("last reboot | head -1 > tmp.01");
  filex = open("tmp.01",O_RDONLY);
  lseek(filex,43,SEEK_SET);

  // ----- le mes -----
  i=0;                                       // vai guardar os valores de
referencia para o
  do {                                       // mes, dia, hora e minutos
    read(filex,&buf[i],1);
    i++;
  } while(i<3);
  buf[i]='\0';
  strcpy(lastMonth,buf);
  lseek(filex,1,SEEK_CUR);

  // ----- le dia -----
  i=0;
  do {
    read(filex,&buf[i],1);
    i++;
  } while(i<2);
  buf[i]='\0';
  sscanf(buf,"%d",&lastDay);
  lseek(filex,1,SEEK_CUR);

  // ----- le hora ----
  i=0;
  do {
    read(filex,&buf[i],1);
    i++;
  } while(i<2);
  buf[i]='\0';
  sscanf(buf,"%d",&lastHour);
  lseek(filex,1,SEEK_CUR);

  // --- le minutos ---
  i=0;
  do {
    read(filex,&buf[i],1);
    i++;
  } while(i<2);
  buf[i]='\0';
  sscanf(buf,"%d",&lastMin);


  close(filex);
  return (0);
}



/*------------------------------ COUNTNUM_LOGINS ------------------------------
---*/
int countNum_Logins(char *login, int countAll) {
  int filex;
```

```
  int ret;
  char buf[10];
  char instrucao[100];
  char mes[4];
  int dia,hora,min;
  int conta=0,i,equal;
  int output_lines;

  strcpy(instrucao,"last ");
  strcat(instrucao,login);
  strcat(instrucao," | wc -l > tmp.01");
  ret = system(instrucao);
  filex = open("tmp.01",O_RDONLY);
  i=0;
  do {
    read(filex,&buf[i],1);
    i++;
  } while (i<7);
  sscanf(buf,"%d",&output_lines);
  if (output_lines-2 == 0){                    // nao ha qq registo
    close(filex);
    return (0);
  }

  if (countAll==1){                            // significa que nao ha registo do
last reboot
    close(filex);                              // -> logo, este foi feito antes
da criacao do
    return(output_lines-2);                    // ficheiro de registo
  }

  close(filex);

  strcpy(instrucao,"last ");
  strcat(instrucao,login);
  strcat(instrucao," > tmp.01");

  ret = system(instrucao);
  filex = open("tmp.01",O_RDONLY);


  do {
    lseek(filex,43,SEEK_CUR);

    // ----- le mes -----
    i=0;
    do {
      read(filex,&buf[i],1);
      i++;
    } while(i<3);
    buf[i]='\0';
    strcpy(mes,buf);
    lseek(filex,1,SEEK_CUR);

    // ----- le dia -----
    i=0;
    do {
      read(filex,&buf[i],1);
      i++;
    } while(i<2);
    buf[i]='\0';
    sscanf(buf,"%d",&dia);
    lseek(filex,1,SEEK_CUR);

    // ----- le hora ----
    i=0;
    do {
      read(filex,&buf[i],1);
```

```
      i++;
    } while(i<2);
    buf[i]='\0';
    sscanf(buf,"%d",&hora);
    lseek(filex,1,SEEK_CUR);

    // --- le minutos ---
    i=0;
    do {
      read(filex,&buf[i],1);
      i++;
    } while(i<2);
    buf[i]='\0';
    sscanf(buf,"%d",&min);


    if (strcmp(mes,lastMonth)==0){            // e' necessario comparar os dias
      if (dia<lastDay){
        close(filex);
        return (conta);
      }
      if (dia==lastDay){                      // e' necessario comparar as horas
        if (hora<lastHour){
          close(filex);
          return (conta);
        }
        if (hora==lastHour)                   // e' necessario comparar os
minutos
          if (min<=lastMin){
            close(filex);
            return(conta);
          }
      }
    }
    else{                                     // procura mes -> ate 6 para a
frente -> valido
      i=1;                                    // -> caso contrario -> invalido
      equal=0;
      do{
        if (getMes(mes)==((getMes(lastMonth)+i)%12))
          equal=1;
        i++;
      }while ((i<=6) && (equal==0));

      if (equal==0){
        close(filex);
        return(conta);
      }
    }

    conta++;
    lseek(filex,3,SEEK_CUR);
    read(filex,&buf[1],1);

    lseek(filex,18,SEEK_CUR);                 // desloca para o mes do login
seguinte

  } while(conta < output_lines-2);
}


/*------------------------------------ GETMES ---------------------------------
---*/
int getMes(char *mes){
  if (strcmp(mes,"Jan")==0)
    return (1);
  if (strcmp(mes,"Feb")==0)
    return (2);
```

```
  if (strcmp(mes,"Mar")==0)
    return (3);
  if (strcmp(mes,"Apr")==0)
    return (4);
  if (strcmp(mes,"May")==0)
    return (5);
  if (strcmp(mes,"Jun")==0)
    return (6);
  if (strcmp(mes,"Jul")==0)
    return (7);
  if (strcmp(mes,"Aug")==0)
    return (8);
  if (strcmp(mes,"Sep")==0)
    return (9);
  if (strcmp(mes,"Oct")==0)
    return (10);
  if (strcmp(mes,"Nov")==0)
    return (11);
  else
    return(12);

  return(0);
}



/*------------------------------------ CLEANUP ---------------------------------
---*/
void cleanup() {
  shmctl(shmid1,IPC_RMID,0);                  // remove a 1a zona memoria
partilhada
  shmctl(shmid2,IPC_RMID,0);                  // remove a 2a zona memoria
partilhada

  sem_rm(WORKER);                             // remove semaforos individuais
  sem_rm(MUTEX);
  sem_rm(EMPTY);
  semctl(semid1,num_workers,IPC_RMID,0);      // remove array de semaforos dos
Pworkers
  semctl(semid2,2,IPC_RMID,0);                // remove array dos Preaders

  msgctl(msgqid,IPC_RMID,0);                  // remove fila de mensagens

  unlink("tmp.01");                           //remove ficheiro auxiliar

  kill(0,2);                                  // manda um signal a todos os
processos para terminarem

  exit(0);
}
```