DISTRIBUTED CLIPBOARD

FINAL REPORT

Disciplina: Programação de Sistemas
Docente: João Silva

Miguel Rodrigues Nº76176
Pedro Esteves Nº77060
Grupo 30

11/06/2018

1 Introdução

Neste projeto final de Programação de Sistemas, tem-se como objetivo criar um conjunto de aplicações C que funcionem como um *clipboard* distribuído. Concluindo este projeto, deverá ser possível copiar dados de um computador para outro (ou no mesmo), independentemente to tipo de dados, tendo em conta uma das dez regiões de memória definidas para este programa, sendo possível fazer *paste* desses mesmo dados ou esperar até que uma certa região tenha dados novos disponíveis (função de espera).

É necessário criar o *clipboard* em si, ou seja, aplicação responsável por guardar e partilhar os dados que são copiados ou que devem ser replicados, bem como um conjunto de funções que sirvam de API a aplicações de cópia de dados, independentemente do tipo de dados. Assim, existe também uma aplicação de teste, utilizada para copiar e colar dados lidos do terminal.

No projeto submetido, encontram-se os seguintes ficheiros:

- *clipboard.c* - Código fonte para a aplicação de *clipboard*.
- *clipboard.h* - Ficheiro *header* utilizado tanto pelo *clipboard*, como pelo ficheiro referente às funções da API.
- *library.c* - Ficheiro que contem o códgio para a implementação da API para utilização do *clipboard*.
- *app_teste.c* - Ficheiro de código para a aplicação de teste que recorre à API fornecida para utilizar o *clipboard*.
- *Makefile* - Ficheiro de *make* para as aplicações supramencionadas. Possui também uma função de *clean* (*make clean*).
- PDF de auto-avaliação, conforme requerido.

De modo a agilizar o processo de cópia e colagem, são utilizadas múltiplas *threads* em cada *clipboard*, bem como princípios de exclusão mútua que permitem o correto acesso aos dados guardados. As comunicações entre aplicações é feita utilizando *sockets*.

É, também, feito o tratamento de erros de forma a que as aplicações não encerrem ou tenham comportamentos inadequados em certas situações.

2 Arquitectura do Sistema

O *clipboard* fucniona recorrendo a *multi-threading*, com *pthreads* de forma a que possam ser feitas diversas acções de cópia, de colagem e de espera ao mesmo tempo.

Assim que o programa é iniciado, é feita a verificação para perceber se é suposto conectar-se a um *clipboard* já existente,utilizando os argumentos "-c EndereçoIP Porto, ou se é suposto estar sozinho. Tendo um *clipboard* "pai", é criada uma *thread* para apenas para comunicação com esse *clipboard* (*thread_recebe_parent()*). Não tendo pai, esse passo é ignorado e são criadas as *threads* comuns a ambos os casos: uma *thread* para aceitar novas ligações de *clipboards* (*thread_code_filhos()*) e uma para aceitar ligações de apps (*thread_aceita_apps()*).

Dentro da *thread* de aceitação de novas apps, é criada uma nova *thread* para cada aplicação diferente que se ligue ao *clipboard* (*thread_code_apps()*), tal como no caso da *thread* de aceitação de *clipboards*, é utilizada uma nova *thread* para cada nova ligação (*thread_code_clipboard_filhos()*), tirando assim o máximo uso dos princípios de exclusão múltipla aplicados e permitindo que tudo aconteça quase em simultâneo.

Aquando da atualização dos dados nas regiões de memória, o *clipboard* que recebe novos dados envia-os para o seu pai, que fará o mesmo até que um *clipboard* não tenha pai. A partir desse, os novos dados são enviados para todos os filhos e cada filho envia para os seus respetivos filhos e assim sucessivamente, garantindo assim que a informação se encontra sincronizada e atualizada em todos os *clipboards*.

Cada *clipboard* gera um valor pseudo-aleatório entre 8000 e 8100 para o porto a utilizar e imprime-o no terminal, de forma a que outros se possam ligar a ele.

A API para qualquer aplicação utilizar o *clipboard* consiste em ter uma função que permite a conexão ao mesmo, retornando o respetivo *file descriptor* e uma função para fecho da conexão, bem como uma função para cada uma das possíveis ações (cópia, colagem e espera), cujos argumentos são os mesmos para as três: *file descriptor* do *clipboard*, região a utilizar, tamanho dos dados a receber ou enviar, e os dados a transmitir ou um *buffer* para receção de dados.

2.1 Protocolo de comunicação

A comunicação entre aplicações e o *clipboard* local é feita através de *sockets* de *stream* do tipo UNIX, situadas na diretoria de onde o programa é corrido, tal como especificado no enunciado, com o nome CLIPBOARD_SOCKET. Para comunicar entre *clipboards*, são utilizadas *sockets* de *stream* do tipo AF_INET. Para comunicar, para além das funções de *bind*, *accept* e *listen*, são utilizadas as funções de *send* e de *recv*.

O protocolo de comunicação consiste em enviar, em primeiro lugar, uma *string* de 30 bytes com a informação relevante ao que se quer fazer (região escolhida, função a realizar e tamanho dos dados a receber ou enviar) e apenas depois de lida essa informação é realizada a ação pedida: copiar algo, colar algo ou esperar até que a região indicada sofra mudanças.

No entanto, há uma exceção a este protocolo: quando um *clipboard* novo se conecta a um já existente. Nesta situação, após a conexão do novo programa ter sido aceite pelo *clipboard* pai, este último envia uma mensagem de 200 bytes constituída por dez valores. Estes dez valores representam o tamanho dos dados armazenados em cada região, respetivamente. De seguida, são enviados os dados de todas as regiões cujo tamanho difere de 0, sendo que o novo *clipboard* recebe e guarda os dados das respetivas zonas. O valor 0 para o tamanho indica que ainda não existe nada guardado nessa região.

2.2 Tratamento de Pedidos

2.2.1 Copy

A ação de *copy* consiste em enviar dados para o *clipboard* ao qual a aplicação está conectado, utilizando a função da API *clipboard_copy()*, especificando qual a região a ser usada e qual o tamanho dos dados a enviar. Esta informação é enviada numa primeira *string*, sendo que os dados são enviados de seguida, com o *clipboard* à espera de receber algo com o tamanho indicado.

Quando os dados são recebidos no *clipboard*, este envia-os para o seu pai. O *clipboard* pai enviará para o seu pai e assim sucessivamente. Quando não houver pai, o *clipboard* em causa (o que não tem pai) percorre todos os filhos que tem, enviando os novos dados pela função *envia_para_filhos()*, que por sua vez enviam para os seus filhos e assim sucessivamente, até que todos os *clipboards* conectados tenham os dados atualizados, seguindo o protocolo de comunicação descrito.

2.2.2 Paste

Na ação de *paste*, a aplicação de teste recorre à API para utilizar a função *clipboard_paste()* para obter dados da região especificada. O utilizador tem de introduzir a região pretendida e o tamanho a receber, que são enviados na primeira *string*, conforme descrito anteriormente. De seguida, a função da API espera pelo envio do *clipboard*, pela *thread* designada para esta app, que enviará os dados da região pretendida até ao tamanho fornecido (ou mais pequeno se os dados armazenados forem menores que o tamanho pedido). Não havendo nada na região pretendida, é enviada uma mensagem em vazio que a API reconhece como sendo uma região sem nada guardado.

2.2.3 Wait

Utilizando a função *clipboard_wait()* da API, é realizada a função de espera. Tal como na função de colagem, é necessário informar a região pela qual se quer esperar e o tamanho de bytes a receber e posteriormente esperar pelos dados vindos do *clipboard*. No *clipboard*, a *thread* responsável por esta aplicação vai ficar num estado adormecido (em vez de espera ativa, para poupar recursos) até que que receba um sinal a informar que a região escolhida foi alterada e aí irá enviar esses novos dados para a aplicação que estava à espera dos mesmos. O *clipboard* tanto poderá receber dados para essa região vindos de outras aplicações ou de outros *clipboards* aos quais esteja ligado, seja o pai ou filhos.

3 Sincronização

3.1 Regiões Críticas

As regiões críticas neste trabalho prendem-se com os acessos às dez regiões de memória utilizadas. Assim, cada acesso às mesmas deve ser protegido utilizando princípios de exclusão múltipla. Estes acessos são feitos sempre que um *clipboard* recebe dados novos de uma app que esteja ligada ao mesmo, cada vez que um *clipboard* novo se conecta ou cada vez que outro *clipboard* atualiza os dados, tendo de replicar os novos conteúdos por todos os *clipboards* conectados.

3.2 Exclusão Múltipla

De modo a implementar exclusão múltipla, o acesso às regiões críticas é feito utilizando *pthread_mutex_t*, sendo que é utilizando um *mutex* diferente para cada uma das dez regiões, de modo a limitar o código restrito a cada uma das regiões e não bloquear o funcionamento a acessos a regiões diferentes.

São também utilizadas 10 variáveis de *pthread_cond_t* de modo a que a função de *wait* ocorra de forma eficiente, sem ter de utilizar espera ativa, o que consumiria mais recursos. Cada condição é associada ao *mutex* respetivo a essa mesma região e é utilizada a função *pthread_cond_signal()* para dar o sinal de desbloqueio relativo à região em causa quando os dados dessa região são atualizados.

4 Gestão de Recursos

Como referido anteriormente, este programa dá uso a várias *threads* e *sockets* que precisam de ser encerradas e fechadas. Sempre que uma aplicação é fechada (enviando 0 bytes pelo *socket* correspondente ou não sendo possível enviar nada) a *thread* dessa mesma aplicação é fechada, tal como com os *clipoards* conectados, tanto pai como filhos. Às três *threads* que podem ser criadas no *main* (aceitar applicações, aceitar filhos e ligar a um pai) é feito *join*, sendo que o programa irá encerrar apenas após estas *threads* terem sido encerradas.

O encerramento de um *clipboard* é feito utilizando o sinal SIGINT (Ctrl + C). Em vez de terminar imediatamente a aplicação, é utilizado um *signal handler* para que uma nova função seja chamada com este sinal. Nesta função, *sigint_flagger()*, todas as *threads* por encerrar são canceladas, bem como toda a memória alocada para os dados das dez regiões (um vetor de dez posições de estruturas do tipo Mensagem, conforme descrito no Listing 1) é libertada. Numa lista duplamente ligada são guardadas as informações dos filhos conectados para que se possa encerrar a sua ligação e fechar a *socket* correspondente, tal como um identificador do pai (no caso de existir). Para que as *threads* possam ser encerradas a partir de outra *thread*, há que permitir que seja feito *cancel* à *thread* (utilizando *pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, NULL)*) e que seja feito de forma assíncrona (*pthread_setcanceltype(PTHREAD_CANCEL_ASYNCHRONOUS,NULL)*).

Também na API é fechada a conexão, através da função *clipboard_close()*.

Tanto na API, como no *clipboard*, é necessário utilizar variáveis auxiliares, por exemplo para enviar a primeira *string* do protocolo de comunicação, sendo sempre utilizada memória dinâmica e respetiva libertação da mesma, não havendo erros de *valgrind* referentes a fugas de memória (ou problemas de ligações de *sockets*).

De formar a reutilizar algum código, dado que as replicações podem acontecer em situações diferentes, para replicar dados pelos filhos, é utilizada a função *envia_para_filhos()*. O restante código encontra-se organizado pela função da *thread* respetiva e apresenta muito poucos comentários, apenas alguns que servem de guia para o que é feito em algumas situações, mas que poderiam estar bastante mais completos.

Listing 1: Declarações e variáveis globais do *clipboard*.

```
//Estrutura para as dez regioes de dados.
typedef struct Mensagem{
  void* mensagem;
  size_t size;
  int cont;
}Mensagem;

struct Mensagem mensagens[10];

pthread_mutex_t mutex[10]= {PTHREAD_MUTEX_INITIALIZER};/*1 mutex para cada regiao*/
pthread_cond_t cond[10]= {PTHREAD_COND_INITIALIZER};/*1 cond para cada regiao*/

//estutura para guardar clipboards conectados
typedef struct ClipB{
  int fd;
  struct ClipB *next;
  struct ClipB *prev;
  pthread_t thread;
}ClipB;

struct ClipB *parent;

ClipB*filhos;

pthread_t aceita_filhos;
pthread_t aceita_apps;
pthread_t parent_thread;
int parent_fd=-1;

pthread_t thread_ids[100]={-1};/*threads apps*/
```

5 Tratamento de Erros

No que toca ao tratamento de erros, os erros a ter em conta são os relacionados com as ligações e os sinais utilizados.

No que toca a erros na comunicação estes são impressos utilizando a função de *perror* e a acção seguinte depende do local do programa onde se encontra o erro. Quando uma aplicação regista um erro de ligação, por exemplo, retorna-se o valor indicado na API descrita no enunciado, enquanto que quando um *clipboard* encontra um erro num envio ou receção de mensagem de uma aplicação, esta é eliminada das aplicações às quais está ligada, acontecendo o mesmo quando o erro provém da ligação com um filho ou com um pai.

De forma a que os programas não encerrem erroneamente quando esperam receber dados e recebem um sinal que não é suposto (por o outro interveniente da comunicação ter utilizado CTRL + C, por exemplo), é ignorado o sinal de SIGPIPE, fechando a ligação ou retornando o valor indicado na API, no caso do ficheiro *library.c*.

6 Conclusão

O projeto foi realizado com sucesso, implementando todos os procedimentos necessários e assegurando que não há erros de execução ou de cópia/leitura de dados. Foi testando em vários computadores com distribuições de Linux diferentes, apresentando um funcionamento de acordo com o esperado em todos.

Com a realização do projeto, obteve-se alguma experiência em trabalhar com várias *threads* ao mesmo tempo, o que é interessante dado que assim é possível executar vários processos em simultâneo, bem como trabalhar com *sockets*, que nos permite realizar comunicações entre processos.