TÉCNICO LISBOA

Clipboard Distribuído

Projeto Final

Daniel de Schiffart 81479
João Gonçalves 81040

Instituto Superior Técnico
Mestrado em Engenharia Aeroespacial
Programação de Sistemas

Junho de 2018

Conteúdo

1 Introdução

Para o projeto final da unidade curricular de Programação de Sistemas do segundo semestre do ano curricular de 2017/2018 foi desenvolvido e implementado um *clipboard* distribuído em C. O objetivo do projeto proposto consistia na implementação do ante-referido *clipboard* através de uma biblioteca, uma API e um processo local que controlassem o funcionamento do mesmo.

O funcionamento do *clipboard* distribuído é análogo ao funcionamento da função de copiar-colar implementada em muitos diferentes programas e sistemas. Uma API permite a outras aplicações comunicar com um processo local (o *clipboard* local) e guardar e retirar dados temporariamente, que serão sincronizados com outras instâncias do mesmo *clipboard* local que estão em funcionamento em outros sistemas, desde que estejam conectados entre si. O conjunto das instâncias deste *clipboard* terá então os mesmos dados guardados em todos os sistemas, facilitando assim a comunicação de dados entre estes. Cada conjunto de *clipboards* interligados tem espaço para armazenamento de dez variáveis diferentes, identificados de 1 a 10.

As várias instâncias de *clipboards* locais mantêm uma hierarquia entre si. Quando um clipboard é iniciado, ou conecta-se a outro ou servirá para outras instâncias se conectarem. Neste segundo caso, esta primeira instância estará no topo da hierarquia, e quaisquer outras instâncias que se conectem estarão um nível abaixo na hierarquia.

Para a comunicação entre *clipboards* são usados *sockets* de domínio Internet, enquanto que para a comunicação com clientes são usados sockets de domínio UNIX.

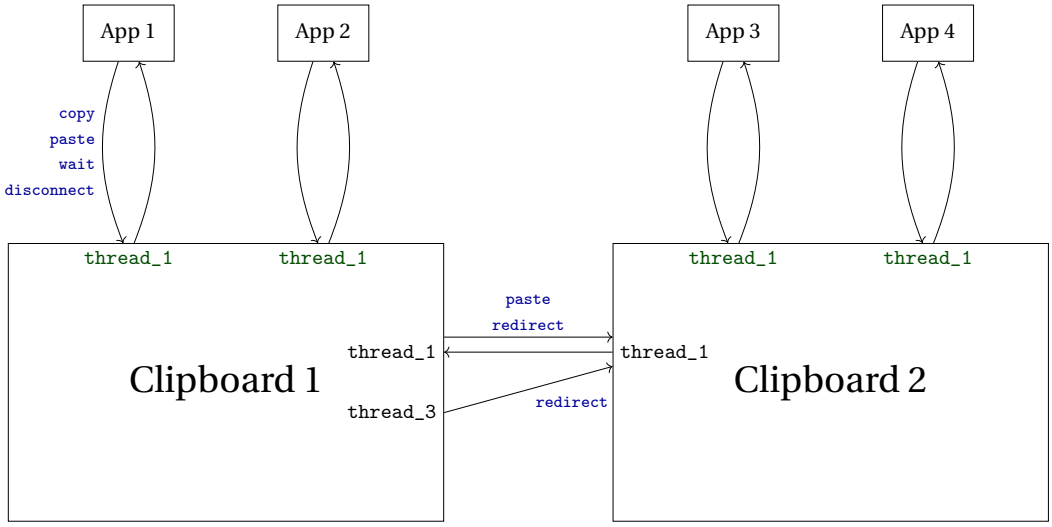

Figura 1: Esboço da arquitectura de implementação e comunicação.

2 Arquitetura

O programa desenvolvido encontra-se dividido num conjunto de cinco ficheiros de código fonte, mais dois ficheiros adicionais desenvolvidos para duas aplicações de teste. São estes ficheiros

- `clipboard.c` – O núcleo do *clipboard*. Contém o código para a função principal `int main` e a *thread* principal do programa, assim como a inclusão das bibliotecas utilizadas.

- `clipboard.h` e `clipboard-dev.h` – Incluem a declaração das funções e todos os macros utilizados. O primeiro ficheiro inclui as funções que compõem a API para implementação em outras aplicações, e o segundo ficheiro contém as declarações das funções restantes.

- `library.c` – Contém o código das funções pertencentes à API, declaradas em `clipboard.h`.

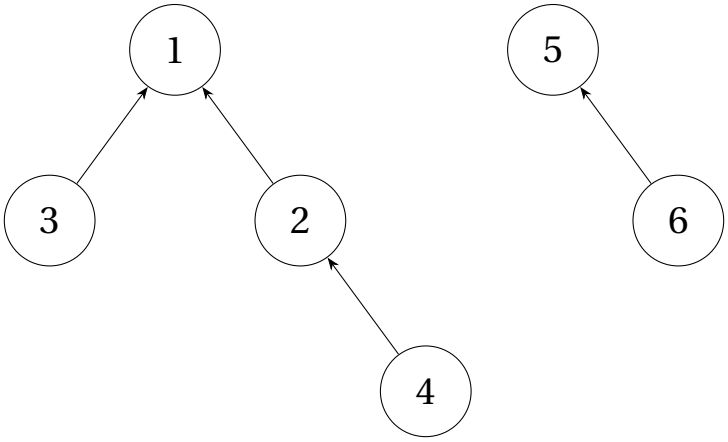

Figura 2: Hierarquia de instâncias do *clipboard*.

- `clipboard_func.c` – Código auxiliar ao ficheiro `clipboard.c`. Inclui as funções correspondentes às *threads* e ao tratamento de sinais, assim como outras funções utilizadas.
- `app_teste.c` e `app_teste2.c` – Código correspondente às duas aplicações de teste referidas.

2.1 Estruturas de dados

Os dados são armazenados numa estrutura do tipo `struct clip_data`, declarada global. De `clipboard-dev.h`:

```
43 struct clip_data{
44
45         void* data[10];
46         size_t size[10];
47         short waiting[10];
48         short writing[10];
49 };
```

Os últimos dois campos são usados para gerir os pedidos de `clipboard_wait()`.

Os *peers*, têm o seu *file descriptor* guardado num vetor dinâmico implementado com `struct clip_peers`.

```
28 typedef struct clip_peers{
29
30         int count;
31         int * sock;
32         int master;
33 }C_peers;
```

Nesta estrutura, guarda-se ainda o `fd` do *master*, ou seja, o *clipboard* conectado com a hierarquia mais alta, que é usado na sincronização global. Seguindo o exemplo da figura 2, `master` do clipboard 4 vai guardar o `fd` de 2, e o `master` do clipboard 1 vai guardar −1.

2.2 *Threads*

Cada instância de um *clipboard* usa três tipos de *threads* diferentes, mais o `main` em si, para executar todos os pedidos de forma eficiente e rápida.

2.2.1 `main`

A função `main` efetua o arranque, que consiste na inicialização do *clipboard*, sincronização deste (caso em modo conectado), lançamento das outras *threads* relevantes, e de seguida entra num ciclo infinito para

aceitar ligações com aplicações.

2.2.2 `thread_1`

Serviço para aplicações ou outros *clipboards*. Esta *thread* é lançada para cada nova conexão que é efetuada, e faz *exit* quando esta é interrompida.

As funcionalidades são:

- Desconectar.
- *Copy* – escrever numa posição local, e reencaminhar para todos os *peers* ligados a esta instância.
- *Paste* – devolver uma posição.
- *Wait* – esperar até uma posição ser alterada e depois devolve-la.
- *Redirect* – idêntico ao *copy*, mas específico para usar com outros *clipboards*.

2.2.3 `thread_2`

Aceitação de ligações com outros *clipboards*. Uma e uma só *thread* deste tipo está a correr, em ciclo infinito, para aceitar ligações com *peers*, lançando depois uma `thread_1`. Também é aqui criada a socket de domínio `AF_INET`, e imprimido o porto através o qual outro *clipboard* se pode conectar.

2.2.4 `thread_3`

Sincronização global. Igualmente à `thread_2`, uma instância desta *thread* corre por cada *clipboard*. Caso o *clipboard* seja o topo de uma árvore, a cada 30 segundos ocorre uma sincronização, de forma que todos os dados são replicados para os *peers* a ele ligados.

2.3 API

As funções referidas pertencentes à API são cinco e estão declaradas no ficheiro `clipboard.h` da forma

```
3  int clipboard_connect(char * clipboard_dir);
4  int clipboard_copy(int clipboard_id, int region, void *buf, size_t count);
5  int clipboard_paste(int clipboard_id, int region, void *buf, size_t count);
6  void clipboard_close(int clipboard_id);
7  int clipboard_wait(int clipboard_id, int region, void* buf, size_t count);
```

2.4 Protocolo de Comunicação

A comunicação entre instâncias separadas do *clipboard* é realizada com recurso a *sockets* UNIX do tipo `SOCK_STREAM`.

O uso dos sockets é feito usando a estrutura `struct message` declarada em `clipboard-dev.h` como:

```
19 typedef struct message{
20
21         short entry;
22         char msg[MESSAGE_SIZE];
23         size_t size;
24         short flag;
25
26 }Message;
```

O macro `MESSAGE_SIZE` tem o valor de 256 (bytes), que deverá assegurar a maioria dos pedidos ao sistema com apenas uma mensagem, para o uso típico deste. Caso isto se verifique, toda a informação necessária é transmitida em apenas uma mensagem: `entry` contêm a posição do *clipboard* em questão, `size` o tamanho da mensagem, e `flag` é um código que distingue as operações a fazer. Caso a mensagem tenha mais de `MESSAGE_SIZE` bytes, o campo `size` passa a conter o tamanho dos dados que faltam receber, estes são divididos em partes, e enviados consecutivamente, como exemplificado neste segmento de código, proveniente da função `clipboard_copy`:

```
61          while (count > MESSAGE_SIZE){
62
63                  m.size=count;
64
65                  memcpy(m.msg, buf, MESSAGE_SIZE);
66
67                  err = send(clipboard_id, &m, sizeof(Message),0);

72                  sent+=MESSAGE_SIZE;
73                  buf = (char*)buf + MESSAGE_SIZE;
74                  count = count - MESSAGE_SIZE;
75          }
76
77          if (count > 0){
78                  m.size = count;
79                  memcpy(m.msg, buf, count);
80                  err = send(clipboard_id, &m, sizeof(Message),0);

85                  sent+=count;
86          }
```

Do lado do receptor, a estrutura é idêntica.

2.5 Fluxo de Tratamento de Pedidos

A todas estas rotinas, acrescentam-se operações de alocação de memória e tratamento de erros.

2.5.1 *Copy*

A função de *copy* do *clipboard* recebe uma mensagem e sincroniza-a com todas as instâncias do *clipboard* a que está conectada.

Primeiro, os dados são totalmente copiados para um *buffer* local, seguindo o protocolo de comunicação já descrito.

De seguida, os dados são inseridos na estrutura de memória do *clipboard*, e verifica-se se a entrada está marcada como em espera, sendo efetuado um `pthread_cond_broadcast()` caso afirmativo.

Finalmente, retorna-se uma mensagem a reportar erros, e os dados são reencaminhados para todos os *peers*.

2.5.2 *Paste*

A função de *paste* devolve a totalidade dos dados de uma entrada.

Começa por copiar os dados da entrada para um *buffer* local, com o caso excecional de a entrada estar vazia.

De seguida, são enviados de acordo com o protocolo de comunicação.

2.5.3 *Wait*

A função de *wait* espera até uma entrada ser alterada para de seguida devolver os novos conteúdos.

A espera é sinalizada através de um campo da estrutura de dados do *clipboard*, e entra-se em espera passiva, através de uma variável condicional.

Quando sinalizada de outra instrução de *copy*, passa a executar como um *paste*.

3 Sincronização

3.1 Identificação das Regiões Críticas

As regiões críticas a proteger são aquelas que procuram ler/escrever nas estruturas de dados descritas na secção 2.1.

No caso de `struct clip_data`, os acessos ocorrem sempre que se faz um *copy/redirect*, *paste*, *wait*, ou operação equivalente. A exclusão é efetuada com `mutexes`, de maneira que as leituras também bloqueiam os acessos. Isto permite que tanto a escrita como a leitura sejam essencialmente atómicas, não havendo mistura de dados. Como são usados *buffers* locais para operações com os dados, a zona crítica pode ser restrita apenas a uma operação de alocamento de memória e de cópia.

No caso de `struct clip_peers`, os acessos ocorrem apenas quando há o reencaminhamento de dados, quer originado por um *copy/redirect* ou por uma sincronização periódica, ou quando se insere ou remove um valor no vetor dinâmico.

3.2 Implementação de Exclusão Mútua

De `clipboard.c`:

```
20 //Secure data structures
21 pthread_mutex_t m_clip = PTHREAD_MUTEX_INITIALIZER;
22 pthread_mutex_t m_peers = PTHREAD_MUTEX_INITIALIZER;
```

3.2.1 `clip_data`

A exclusão mútua desta estrutura é feita usando o `mutex m_clip`. De `clipboard_func.c`:

```
104 //Critical region
105 pthread_mutex_lock(&m_clip);
106
107 //Check if we are rewriting the same data
108 if(clipboard.size[m.entry]==size)
109         if(!memcmp(clipboard.data[m.entry], buf, size))
110                 err=1;
111 if (err != 1){
112         clipboard.data[m.entry] =
113                         realloc(clipboard.data[m.entry], size);
114         if (clipboard.data[m.entry] == NULL){
115                 perror("realloc");
116                 exit(-1);
117         }
118         memcpy(clipboard.data[m.entry], buf, size);
119         clipboard.size[m.entry]=size;
120         print_entry(m.entry);
121
122         //Signal waiting threads
123         if (clipboard.waiting[m.entry] > 0){
124                 clipboard.writing[m.entry]=1;
```

```
                pthread_cond_broadcast(&cv_wait);
        }
}
pthread_mutex_unlock(&m_clip);
//End critical region
```

Este excerto é retirado de uma operação de *copy*, onde os novos dados já estão guardados em `buf`. A zona crítica consiste numa alocação de memória, uma cópia, e uma sinalização a *threads* em espera passiva. No caso de uma leitura, é ainda mais simples:

```
//Critical region
pthread_mutex_lock(&m_clip);
size=clipboard.size[m.entry];
if (size == 0){
        pthread_mutex_unlock(&m_clip);
```

```
        continue;
}else{
        buf=realloc(buf, size);
        if(buf==NULL){
                perror("realloc");
                exit(-1);
        }
        memcpy(buf, clipboard.data[m.entry], size);
        pthread_mutex_unlock(&m_clip);
}
//end critical region
```

3.2.2 `clip_peers`

Neste caso, a exclusão é mais apertada, visto querermos garantir que o protocolo de comunicação com várias mensagens não seja violado. Desta maneira, este vetor dinâmico é bloqueado durante toda a transmissão de dados entre *peers*, para além das operações de escrita e leitura.

```
// Replicate to other clipboards - critical region
pthread_mutex_lock(&m_peers);
if(peers->count > 0){
        m.flag = REDIRECT;
        buf2=buf;

        for(i=0; i<peers->count; i++){

                if (client_fd != peers->sock[i]){
                        printf("Sending to peer with fd %d\n",
                                peers->sock[i]);

                        buf=buf2;
                        m.size=size;
```

Protocolo de comunicação

```
                }
        }
        buf=buf2;
}
//End critical region
pthread_mutex_unlock(&m_peers);
```

A exclusão é implementada também quando pretendemos remover um *peer* do vector, como exemplificado aqui.

```
43 err = recv(client_fd, &m, m_size, 0);
44 if(err <= 0){
45         printf("Disconnected\n");
46         close(client_fd);
47         if(mode==PEER_SERVICE){
48                 pthread_mutex_lock(&m_peers);
49                 remove_fd(peers, client_fd);
50                 pthread_mutex_unlock(&m_peers);
51         }
52         pthread_exit(NULL);
53 }
```

Também de maneira idêntica quando se acrescenta um elemento, na `thread_2`.

3.3 Sincronização global

A sincronização global é aplicada para garantir que, na maioria do tempo, uma árvore de *clipboards* contém a mesma informação. Imaginemos que no exemplo da figura 2, executamos um pedido de *copy* nos *clipboards* 3 e 4, ao mesmo tempo. O sistema propaga as mensagens em ambas as direções na cadeia, e no final, uma metade pode ter dados diferentes da outra.

A solução implementada é a sincronização completa a cada 30 segundos, iniciada neste caso pelo *clipboard* 1. Para tal, a `thread_3` é acordada por um `SIGALARM`, verifica se é o topo da árvore e envia os dados. O código relevante é todo o de `thread_3_handler()`.

4 Gestão de Recurso e Tratamento de Erros

4.1 Memória dinâmica

Os arrays dinâmicos usados são operados com a função `realloc()`, de maneira que a memória usada é sempre a mínima possível.

Em caso de terminação do programa com `CTRL-C`, as posições do clipboard são igualmente libertadas. Contudo, os *buffers* locais a algumas *threads* não são, temos *memory leak*.

4.2 Outros recursos

Os *file descriptors* são libertados com `close()` sempre que é detetado que a ligação foi interrompida do outro lado, ou há um pedido de desconexão.

Para as mesmas condições, as instâncias de `thread_1` são terminadas com `pthread_exit()`, libertando os respetivos recursos. As restantes *threads* são suposto correrem durante todo o uso do sistema, logo são terminadas quando o programa é terminado.

4.3 Tratamento de erros

Em muitos casos, usam-se funções ou *system calls* cujo sucesso depende de aplicações não necessariamente confiáveis quanto à sua robustez, ou de processos a correr noutra máquina. Desta forma, procura-se que os erros esperados não afetem o sistema de forma significativa.

4.3.1 *Sockets*

A comunicação entre processos é feita com *sockets*, usando as funções `send()` e `recv()`. O valor de retorno destas funções indica se houve um erro, e o procedimento adotado é, quando isto acontece, interromper

a ligação com o cliente/peer, evidentemente não completando o seu pedido, e sair em ordem invocando `close()` e `pthread-exit()`. A API foi usa o mesmo conceito, mas não fecha a ligação, passando essa decisão à aplicação através do valor de retorno.

Para o caso do `send()`, é necessário desativar o sinal `SIGPIPE`, o que é aceitável visto protegermos todas as invocações desta função. Como exemplo, de `clipboard_func.c`:

```
167 err = send(peers->sock[i], &m,m_size,0);
168 if (err == -1){
169         perror("send");
170         remove_fd(peers, peers->sock[i]);
171         close(peers->sock[i]);
172         i--;
173         continue;
174 }
```

Outro exemplo é o caso do `accept()` ser terminado devido ao `SIGALARM`. Quando este é o caso, simplesmente voltamos a invocar esta função.

```
381 client_fd = accept(sock_fd, (struct sockaddr *) & client_addr, &size_addr);
382 if(client_fd == -1) {
383         if (errno == EINTR)
384                 continue;
385
386         perror("accept");
387         exit(-1);
388 }
```

No caso do `bind()`, faz-se `unlink()` ao endereço e repete-se a chamada. `clipboard.c`:

```
180 err = bind(sock_fd, (struct sockaddr *)&local_addr, sizeof(local_addr));
181 if(err == -1) {
182         unlink(SOCK_ADDRESS);
183         printf("Unlinked previous address.\n");
184         err = bind(sock_fd, (struct sockaddr *)&local_addr, sizeof(local_addr));
185         if(err == -1) {
186                 perror("bind");
187                 exit(-1);
188         }
189 }
```

Outros erros ocorridos na preparação dos *sockets* são reportados e o programa termina.

4.3.2 Outros erros

O programa testa os valores de retorno das chamadas de `realloc()`, terminando em caso de erro.