Stream Processing

Sistemas Operativos
Mestrado Integrado em Engenharia Informática

André Teixeira - a71930
Mário Ferreira - a70441
Tiago Sá - a71835

Universidade do Minho

Contents

Capítulo 1

Introdução

Este trabalho prático, realizado sob o âmbito de instrumento de avaliação da Unidade Curricular de Sistemas Operativos, pretende a criação de um programa de processamento de eventos em fluxo, ou *stream processing*, de modo a forçar a utilização por parte dos alunos de todos os principais conceitos e ideologias lecionados na UC. Este projeto incorpora portanto e permite a exploração de todas as principais noções de comunicação dentro de um sistema informático, principalmente utilizando texto e valores, o seu tratamento e comandos de execução de tarefas.

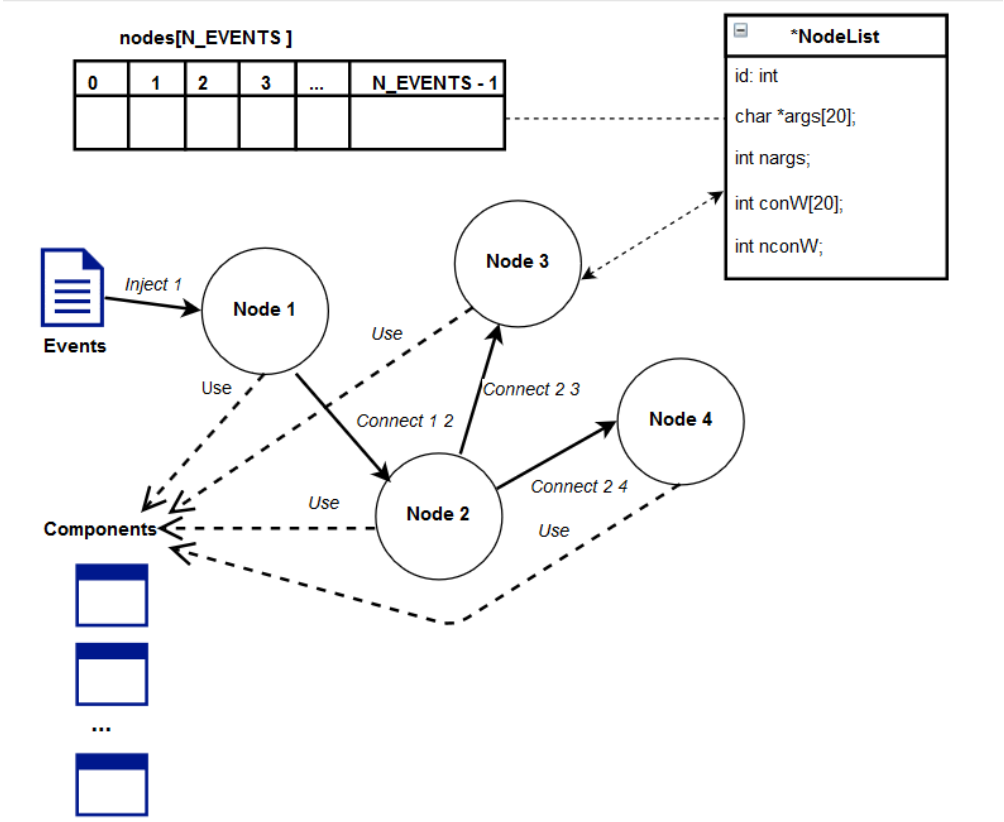
Existem portanto duas facetas principais do projeto. O desenvolvimento e implementação de alguns componentes elementares do sistema, que conseguissem suportar tarefas básicas necessárias para o tratamento de texto e numerais, e a subsequente implementação de um sistema de controlo que permita encadear comandos destas mesmas funcionalidades de forma simples, de acordo com a proposta dos docentes. A criação da rede de processamento final, baseada nos componentes criados e em *pipelines*, é portanto o objetivo a atingir, e assegurar que esta possui quer as funcionalidades básicas esperadas, quer algumas funcionalidades avançadas adicionais foi o foco do grupo. Deste modo pretendemos cumprir os requisitos enunciados pelos docentes, explorar os conceitos e noções associados a este ramo do estudo de informática, e desenvolver as nossas capacidades na manipulação de sistemas operativos de modo a facilitar desafios e projetos futuros.

Neste documento estão presentes todas as principais decisões e estratégias tomadas no desenvolvimento do projeto, assim como a explicitação do processo utilizado na implementação de cada componente individual do programa e testes realizados para assegurar a sua qualidade, servindo este relatório como acompanhamento ao programa desenvolvido e complemento do trabalho realizado.

Capítulo 2

Implementação

2.1 Arquitetura



2.2 Componentes

A primeira das tarefas propostas pelos docentes foi o desenvolvimento e construção dos componentes deste sistema que realizam um conjunto de tarefas simples, porém essenciais.

Os componentes recebem eventos com um tamanho máximo ($PIPE\_BUF$) e produzem eventos alterados, por exemplo, o component **const** <valor>apenas acrescenta o valor associado ao const no fim do evento. Como tal, cada componente estará à "espera" de receber eventos no seu input

até que seja pressionado *ctrl+d* e produzirá eventos no seu output.

O sistema deverá recorrer a estes componentes para encadear os eventos, de forma a produzir resultados que possam ser úteis, utilizando para isso vários componentes e eventualmente redirecionar no fim para um ficheiro de texto.

Os componentes construídos foram os pedidos, **const, filter, window, spawn** e ainda **grep**, **cat** e **tee**. Mais poderiam ter sido construídos e teriam sido úteis, no entanto estes chegavam para iniciar o controlador e começar a implementação do sistema propriamente dito.

2.3 Controlador

O controlador será o programa principal. Este sistema irá permitir definir uma rede de processamento de streams, em que cada nó executa o componente que a ele está definido e executa transformações nos eventos.

O problema foi dividido em sub-problemas de forma a perceber o que se pretende e para desenvolver o programa de forma mais eficaz, assim como divisão de tarefas.

2.3.1 Estrutura de dados

Em primeiro lugar, nós pretendemos ter uma rede de processamento, logo, o primeiro passo será então criar os nós. Como tal, foi construída uma estrutura para suportar o armazenamento de todos os nós e as suas características, como os comandos que estão destinados a realizar. Para este trabalho, usamos um array de nós, sendo que o número de nós máximo definido foi 200.

```
NodeList nodes[N_EVENTS];
```

A estrutura pode ser observada no escerto indicado abaixo:

```
typedef struct node {
  int id;          /* id/number of the node */
  char *args[20]; /* the arguments of the command */
  int nargs;       /* number of arguments in args */
  int conW[20];    /* id's of nodes this node is connect (to write) */
  int nconW;       /* number of nodes this node is connected (to write) */
} *NodeList;
```

As funções necessárias para construir e manter o sistema no que diz respeito aos nós, são as seguintes. São relativamente básicas e permitem iniciar uma nova estrutura de dados, verificar se um **id** existe na estrutura e retorna a sua posição no array, criar um novo nó para inserir na estrutura, remover um nó e ainda alterar o componente de um nó.

```
void initNodeList(struct node **node);
int existNode(struct node **node, int id);
int newNodeList(struct node **node, int id, char **args, int n);
void removeNode(struct node **node, int id);
void changeComponent(struct node **node, int id, int narg, char **arg);
```

2.3.2 Nodes

Para construir um nó, o utilizador deverá inserir *node <id ><cmd ><args...>*. O programa reconhece este comando e irá proceder à chamada da função node que definimos.

```
if (!(strcmp(cmd, "node")))
  node(narg, arg);
```

Esta função irá inserir na estrutura o nodo desejado com as suas características. Para além disto, irá lançar um novo processo (através da *system call* **fork**, que irá criar um pipe com nome, cujo nome será o id do nó a criar. Este processo ficará até o término do programa a correr, à espera de ler eventos do seu pipe com nome. O processo pai decorrerá normalmente e voltará para a main, à espera de receber novos comandos. Em seguida podemos ver um excerto do código que permite criar um novo processo, neste criar o pipe com nome e ficar à espera de escritas para esse pipe.

```
if(!fork()){
  mkfifo(idS, 0666);
  f = open(idS, O_RDONLY);
    if (f==-1)
    perror("Erro na abertura do pipe\n");
  while ((r=(readln(f, buf, 128)))) {
```

Desta forma, sempre que é criado um nó, ele cria um novo processo e um pipe com o nome do seu id, portanto, único. Se for chamado um comando para criar um novo nó com o mesmo id, o programa irá ignorar este comando.

2.3.3 Connect

Para realizar uma conexão entre nós, o utilizador deverá inserir *connect <id ><ids >*. O programa reconhece este comando e irá proceder à chamada da função connect que definimos.

```
else if (!(strcmp(cmd, "connect"))) {
  connect(narg, arg);
  int f = open(arg[0], O_WRONLY);
  if (f==-1)
    perror("Erro a abrir o pipe\n");
    strcat(buf2, "\n");
    write(f, buf2, r+1);
}
```

Este comando permite definir ligações entre nós, mais concretamente, ligar a saída de um nó à entrada de um ou vários outros nós. Por exemplo *connect 1 2 3*, deve ligar a saída do nó 1 à entrada do nó 2 e 3.

Para tal, na main, ao reconhecer este comando, chama a função **connect** que fará a inserção correta na estrutura, ou seja, em nó[X]-¿conW vai constar os nós para o qual este nó deve escrever, X representa a posição do array em que o nó que queremos está no array. Se já constar lá, não faz nada, se não constar, coloca à frente dos que já existirem.

A maior dificuldade aqui era fazer esta inserção na estrutura do processo que foi lançado relativo ao nó que desejamos. Uma vez que foi criado anteriormente um novo processo, este manteve uma cópia do estado atual de todas as variáveis, e portanto, uma vez que não há memória partilhada, foi necessário mandar um comando para o pipe desse nó. Este nó, que está à espera de escritas no pipe, vai ler este comando e interpretá-lo como tal, em vez de um evento, e vai proceder no seu processo á chamada da função connect. Desta forma, mantemos a consistência do programa, sendo que ambos os processos (principal e do nó id) têm na sua estrutura, relativos a si, a mesma informação.

2.3.4 Disconnect

Para realizar uma conexão entre nós, o utilizador deverá inserir *disconnect <id1 ><id2 >*. O programa reconhece este comando e irá proceder à chamada da função disconnect que definimos.

```
else if (!(strcmp(cmd, "disconnect"))) {
  disconnect(arg[0], arg[1]);
  int f = open(arg[0], O_WRONLY);
    if (f==-1)
      perror("Erro a abrir o pipe\n");
    strcat(buf2, "\n");
    write(f, buf2, r+1);
}
```

Este comando permite definir retirar ligações entre nós, que anterirmente foram conectados. Por exemplo *disconnect 1 2*, deve "desligar" a saída do nó 1 para a entrada do nó 2, sendo que ficaria apenas a enviar para o stdout.

Para tal, na main, ao reconhecer este comando, chama a função **disconnect** que fará a remoção correta na estrutura, ou seja, em nó[X]-¿conW vai constar os nós para o qual este nó deve escrever, X representa a posição do array em que o nó que queremos está no array. Se constar lá, será retirado, caso contrário nada fará.

Tal como no connect, a maior dificuldade aqui era fazer esta remoção na estrutura do processo que foi lançado relativo ao nó que desejamos. A estratégia foi a mesma.

2.3.5 Inject

Para realizar uma injeção na rede (inserindo eventos num nó), o utilizador deverá inserir *inject <id1 >* ou *inject <id1 >cat <.txt >*. O programa reconhece este comando e irá proceder à chamada da função disconnect que definimos. Pode portanto, ler do input eventos, ou ler de um ficheiro de texto onde já existem eventos.

```
else if (!(strcmp(cmd, "inject")))
  inject(narg, arg);
```

Depois de ler o evento (seja do input ou ficheiro de texto), é chamada a função **inject**. Esta função, abre o pipe com nome do id onde queremos injetar e escreve lá o evento. Uma vez que isto acontece, o nó desse id, que está à espera de eventos, irá ler e interpretar como evento. De seguida, realiza as transformações necessárias, utilizando para isso a chamada da *system call* **execvp**, uma vez que os comandos do nó estão guardados na estrutura em node[X]-¿args, em que X é a posição onde o nó está guardado na estrutura. Para tal, foi necessário recorrer ao uso de pipes anónimos.

Assim que estas transformações são realizadas, é chamada a função **writeNode** que vai escrever, dependendo das conexões deste nó. Se tiver alguma conexão, vai escrever para esse nó, ou nós, caso contrário, escreverá para o output. Se este nó tiver conexões e precisar de escrever para outros nós, o writeNode vai abrir o pipe com nome dos nós para onde deseja escrever, envia os eventos para os nós, e estes fazem a mesma coisa que este nó acabou de fazer, recebe os eventos, faz transformações e volta a enviar, seja para o stdout ou outros nós.

Capítulo 3

Funcionalidades básicas

3.1 Permanência de uma rede

Esta funcionalidade é garantida, uma vez que como referido anteriormente, um nó sempre que é criado, é construído um novo processo associado a este nó à espera que escrevam para o seu respetivo pipe. Assim, até que o programa seja terminado ou o nó seja explicitamente removido (funcionalidade avançada), este continua a existir, sempre pronto a receber eventos.

3.2 Concorrências de escritas no mesmo nó

A rede poderá ter vários nós e várias conexões entre estes nós e é assegurado no entanto que ao fazer inject, seja de um evento ou de vários (através da leitura de um ficheiro de texto), que cada evento não é corrompido e chega ao destino correto, havendo apenas intercalamento de eventos de diferentes origens.

3.3 Nós sem saídas definida

Os nós sem saídas definidas, são executados de forma normal e imprimem no stdout. Se houver conexões, passa a enviar para os nós correspondentes. Isto permite que seja possível definir um componente que escreve num ficheiro de texto e assim as transformações que foram feitas antes, podem ser escritas no ficheiro em vez de ser tudo ignorado por não ter saída definida.

3.4 Especificação incremental de uma rede

É garantido que a rede é completamente flexível. É possível receber seja qual for o comando, seja quando for. Os nós podem ser criados, com o id e compontente desejado e várias conexões podem ser feitas, que nada afetará o sistema. Os nós são construídos em tempo real, pelo que a rede pode ser incrementada sem problemas adicionais.

Capítulo 4

Funcionalidades avançadas

4.1 Casos de Teste

Para facilitar o momento de avaliação, já temos redes definidas e prontas a serem carregadas, todas variadas e usando todos os componentes construídos. No momento de fazer o inject, é possível também injetar logo um ficheiro de texto.

4.2 Alteração de componentes

É possível alterar os componentes através da chamada do comando *change* $<id><args>$. Isto permite tornar a rede ainda mais flexível, uma vez que mesmo que um nó seja criado com um componente errado, ou seja mais conveninente a certa altura outro componente, tal alteração poderá ser realizada.

Tal é possível uma vez que se faça estas alterações na estrutura, tanto no processo de cada nodo envolvido, como no principal.

4.3 Remoção de nós

Os nós podem ser removidos sem problemas adicionais. Por exemplo, se forem criados nós com id's 1 2 3, e fizermos a conexão entre 1 e 2, 2 e 3, caso seja removido o nó 2, o 1 deixa de escrever para o 2 e passa a escrever para o nó que o nó 2 escrevia. Se fosse para o stdout, o nó 1 escrevia para o stdout, como no caso escreve para o nó 3, o nó 1 passava a escrever para o nó 3.

Tal é possível uma vez que se faça estas alterações na estrutura, tanto no processo de cada nodo envolvido, como no principal.

Capítulo 5

Testes

5.1 Componentes

5.1.1 Const

```
➜ ./cons 10
a:3:x:4
a:3:x:4:10
b:1:y:10
b:1:y:10:10
a:2:w:2
a:2:w:2:10
d:5:z:34
d:5:z:34:10
```

5.1.2 Filter

```
➜ ./filter 2 "<" 4
a:3:x:4
a:3:x:4
b:1:y:10
b:1:y:10
a:2:w:2
d:5:z:34
d:5:z:34
```

5.1.3 Window

```
→ ./window 4 avg 2
a:3:x:4
a:3:x:4:0
b:1:y:10
b:1:y:10:4
a:2:w:2
a:2:w:2:7
d:5:z:34
d:5:z:34:6
```

5.1.4 Spawn

```
→ ./spawn ls
10
1                       componentsCopy.h        rede.txt
2                       cons                    spawn
Makefile                consTeste               struct.c
README.md               enunciado-so-2016-17.pdf struct.h
basic                   exemplo.txt             struct.o
basic.c                 filter                  teste
components.c            grep                    teste.c
components.h            main                    teste2
components.o            main.c                  teste2.c
componentsCopy.c        main.o                  window
10:0
```

```
→ ./spawn notdefined
10
10:1
```

5.1.5 Grep

```
→ ./grep 2 2
10:10

2:2
2:2
```

5.1.6 Cat

```
→ ./cat
10:20
10:20
10123:ads:20
10123:ads:20
1:w:d:2:a
1:w:d:2:a
```

5.2 Controlador

```
→ ./main
node 1 cons 1
node 2 cons 2
node 3 cons 3
connect 1 2
connect 2 3
inject 1
11
11:1:2:3
12
12:1:2:3
Leaving inject...
remove 2
inject 1
13
13:1:3
20
20:1:3
Leaving inject...
node 2 cons 10
connect 3 2
inject 1
1000
1000:1:3:10
Leaving inject...
```

Com carregamento de rede:

```
node 1 cons 1
node 2 spawn ls
node 3 spawn notcommand
connect 1 2
connect 2 3
node 4 filter 1 > 2
connect 3 4
node 5 cat
node 6 tee tee.txt
```

```
→ ./main
rede rede.txt
inject 1
10
10:1:0:1
20
20:1:0:1
Leaving inject...
inject 5
10
10
20
20
Leaving inject...
node 7 cons 10
connect 4 7
inject 1
10
10:1:0:1:10
Leaving inject...
```

Capítulo 6

Makefile

```
default: controlador grep const window spawn filter tee cat

run: controlador
./controlador

controlador: controlador.o struct.o
gcc -o controlador controlador.o struct.o

controlador.o: controlador.c struct.h
gcc -c -Wall controlador.c

struct.o: struct.c
gcc -c -Wall struct.c

grep: grep.o
gcc -o grep grep.o

grep.o: grep.c
gcc -c -Wall grep.c

const: const.o
gcc -o const const.o

const.o: const.c
gcc -c -Wall const.c

window: window.o
gcc -o window window.o

window.o: window.c
gcc -c -Wall window.c

spawn: spawn.o
gcc -o spawn spawn.o

spawn.o: spawn.c
gcc -c -Wall spawn.c

filter: filter.o
gcc -o filter filter.o

filter.o: filter.c
gcc -c -Wall filter.c

tee: tee.o
gcc -o tee tee.o

tee.o: tee.c
gcc -c -Wall tee.c

cat: cat.o
gcc -o cat cat.o

cat.o: cat.c
gcc -c -Wall cat.c

clean:
-rm -f controlador
-rm -f controlador.o
-rm -f struct.o
-rm -f const.o grep.o window.o filter.o spawn.o tee.o
-rm -f const grep window filter spawn tee cat
```

A nossa **Makefile**, tem como objectivo principal compilar primeiro todos os ficheiros *.c* para os seus respetivos *.o* (ficheiros objecto) e, em seguida, fazer o link entre os *.o* obtidos criando o nosso "controlador" (executável do programa principal) e os executáveis dos vários componentes. A compilação é realizada com a FLAGS --Wall. Introduzindo o comando *make* são compilados todos os *.c* que tenham sido alterados ou que ainda não estejam compilados e, posteriormente, criados os respetivos executáveis.

Possui também, o comando *make clean* que elimina todos os ".o" e executáveis gerados. Com *make run* é permitido correr o programa (./controlador).

Capítulo 7

Conclusão

Com o término deste trabalho podemos concluir que todos os principais requisitos necessários para que o programa de execução de comandos em fluxo, proposto pelos docentes se encontre sólido, fundamentado e completamente funcional, foram atingidos com sucesso. Aquilo que o grupo identificou como a principal falha do programa desenvolvido e portanto do projeto no geral foi a incapacidade do controlador de receber e processar mais do que um evento de cada vez, já que a "janela" estabelecida foi de uma unidade. Deste modo a função window desenvolvida torna-se inútil. Tal só foi reparado no final do trabalho, no entanto, com mais algum tempo seria possível termos o projeto concluído na totalidade como era sugerido. Tirando este aspeto reiteramos que o projeto atingiu a grande maioria dos seus objetivos, incluindo o de incentivar os alunos a explorar e melhor compreender os conceitos dados nas aulas da unidade curricular. Damos portanto este projeto como concluído, uma vez que o programa foi implementado com sucesso e a aprendizagem verificada, após termos sido capazes de superar as necessidades e desafios encontrados durante o desenvolvimento do mesmo.