Relatório do Projeto de *SO*

António Sérgio (a78296) Hugo Brandão (a78582)
Tiago Alves (a78218)

Abril 2017

Grupo 2
Universidade do Minho
Departamento de Informática
Sistemas Operativos

Mestrado Integrado em Engenharia Informática

Conteúdo

1 Introdução

Este trabalho irá ser realizado no âmbito da unidade curricular *Sistemas Operativos*, no qual pretendemos construir um sistema de *stream processing*. Basicamente teremos que criar uma variedade de componentes que irão receber eventos e irão filtrá-los, modificá-los, entre outras funcionalidades.

Para conseguir implementar este tema iremos utilizar várias funcionalidades em **C** que fomos aprendendo ao longo do semestre nas aulas desta unidade curricular, tais como *pipes*, *fork*, *exec*, *read*, etc.

Após a leitura do enunciado do trabalho prático pensámos que as primeiras funções são relativamente acessíveis, todavia as funções do controlador como *connect*, *disconnect*, *node* e *inject*, causarão mais problemas visto que já requerem um maior cuidado na criação de processos e na abertura e fecho dos pipes. Contudo estamos motivados para realizar este trabalho.

Neste relatório iremos relatar todos os métodos utilizados para o bom funcionamento de cada função, os problemas que fomos encontrando e como os combatemos. Estamos à espera de uma jornada longa e com muitos contratempos, mas esperamos concretizar todos os objetivos por nós traçados.

2 Componentes

Esta parte do trabalho prático foi rapidamente concluída. Tivemos alguns contratempos mas céleremente superados, sem dúvida que esta foi a parte mais simples do trabalho, visto que apenas envolveu funcionalidades muito básicas que aprendemos nesta **UC**. Todas as implementações foram feitas como programas separados, isto é, leem do *stdin* e escrevem no *stdout*.

2.1 Const

Esta função simplesmente acrescenta um parametro passado como argumento no fim de cada linha lida, ou seja, quando fazemos *./const 10* todas as linhas a seguir escritas vão ser reescritras com um *:10* no fim.

Por exemplo:

```
./const 10
a:1:2
a:1:2:10
bind:2:slot:ok
bind:2:slot:ok:10
```

Esta função foi fácil de implementar, pois só utilizamos funcionalidades como o *read* e o *write*.

```
while((i = readln(0, buf, PIPE_BUF))>0){
  buf[i-1] = ':';
  strcat(buf, argv[1]);
    strcat(buf,"\n");
  i+=strlen(argv[1]);
  write(1, buf, i+1);
  memset(buf, 0, i);
}
```

2.2 Filter

A filter recebe como argumentos dois números de colunas e uma operação condicional, e vai aplicar essa operação às duas colunas, caso seja verdade imprime a linha, caso contrário não imprime nada.

No exemplo seguinte, caso a segunda coluna seja menor que a quarta, imprimimos, caso contrário, não imprimimos.

```
./filter 2 < 4
a:1:2:2
a:1:2:2
bind:2:slot:2
b:5:t:10
b:5:t:10
```

Foi uma função foi fácil de implementar. O único problema foi descobrir como separar as colunas, mas após alguma procura descobrimos uma implementação simples e terminamos rapidamente.

Aqui podemos ver um excerto de código, onde mostra como as colunas são processadas:

```
void  filter (void* buf, char* copy, int size, char* operation, int
     column, int column2){
    int c=1;
    int product1 , product2;
    char *num1, *num2, copy2[PIPE_BUF];
    strcpy(copy2, copy);

    for(num1 = strtok(copy, ":"); num1 != NULL && c<column; num1 =
        strtok(NULL, ":")){
        c++;
    }
    if (num1==NULL){
        perror("Missing columns!");
        exit(-1);
    }
    c=1;
    product1 = atoi(num1);
    for(num2 = strtok(copy2, ":"); num2 != NULL && c<column2; num2
        = strtok(NULL, ":")){
        c++;
    }
    if (num2==NULL){
        perror("Missing columns!");
        exit(-1);
    }
    else {
        product2 = atoi(num2);
        operate(buf, size, operation, product1, product2);

    }
}
```

2.3 Window

A window recebe como argumentos um número de uma coluna(c), uma função(f) e um número de linhas(l). Após receber uma linha, esta vai à coluna(c) e guarda-a. De seguida, cálcula a função(f) às últimas (l) linhas.

Por exemplo, *./window 3 max 2* vai guardando a terceira coluna de cada linha e faz o *max*(máximo) da terceira coluna das últimas 2 linhas.

```
./window 3 max 2
a:3:3
a:3:3:0 //visto que ainda nao ha colunas guardadas, o maximo e zero
b:5:6:7
b:5:6:7:3 //visto que so escrevemos uma linha, o maximo e essa
    linha
c:3:5
c:3:5:6 // a partir de agora faz o maximo das ultimas duas linhas,
    como as colunas guardadas sao "3" e "6", o maximo e 6
```

Apesar de parecer uma função mais complicada, não foi muito difícil de implementar, visto que já sabiamos como separar as colunas e as operações pedidas eram de fácil implementação.

Aqui podemos ver um excerto de código, onde mostramos como a implementação e muito parecida à da filter:

```c
void window(void* buf, char* copy, int size, char* operation, int
    coluna){
    int c=1;
    int product;
    char* num;
    for(num = strtok(copy, ":"); num != NULL && c<coluna; num =
        strtok(NULL, ":")){
        c++;
    }
    if (num==NULL){
        perror("missing columns");
        exit(-1);
    }
    else {
        product = atoi(num);

        operate(buf, size, operation, product);
    }
}
```

2.4 Spawn

A spawn recebe como argumento um comando e seus argumentos. De seguida executa o comando com os argumentos passados e guarda o seu exit status. Após isso vai processar linhas e escreve o *exitstatus* no fim da cada linha.

Por exemplo, se fizermos *./spawn ls -l*, caso esta operação corra bem acrescenta (exit status = 0):

```
./spawn ls -l
a:1:2:2
a:1:2:2:0
b:5:t:10
b:5:t:10:0
```

Devido ao exec esta função causou alguns problemas. No entanto, após algum trabalho com forks, a função ficou rapidamente operacional.

Aqui podemos ver um excerto de código, onde mostra como o comando é executado e como guardamos o *exit status*.

```c
void spawn(char* buf, char* copy, char** argv, int sizeBuf){
    int status;
    char execResult[4];
    pid_t p;
    p = fork();
    if(!p){
        execvp(argv[1], &argv[1]);
        _exit(-1);
    }
    else{
        wait(&status);
        sprintf(execResult, "%d", WEXITSTATUS(status));
        writeOP(buf, execResult, sizeBuf);
    }
}
```

3 Controlador

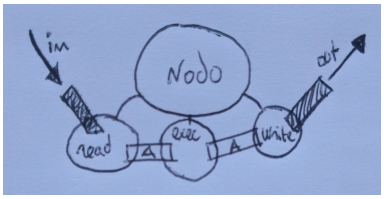
Esta parte do trabalho foi a mais longa, pois já começou a envolver gestão de processos e pipes, tanto anónimos como com nome. Ocorreram pequenos problemas que nos ocuparam horas para resolver, visto que a maneira como abrimos e fechamos os pipes, e como estes comunicam entre si em processos diferentes tem de ser meticulosamente cuidada.

3.1 Node

A node recebe um número, e um componente com os seus argumentos. O número(n) recebido serve para identifcar este node. Este vai executar o componente com os argumentos, e passar um input recebido no *fifo de entrada* desse node como input desse componente e, de seguida, guarda o output gerado e vai redirecioná-lo para o *fifo de saída* desse node.

Um nodo é composto por 3 processos-filho. Um que lê constantemente do *fifo de entrada* e escreve no pipe anónimo de entrada. Um que redireciona o *stdin* para o pipe anónimo de entrada e o *stdout* para o pipe anónimo de saída através de *dups* e executa o comando dado como argumento através de um exec. O último lê permanentemente do pipe anónimo de saída e escreve o que leu para o *fifo de saída*.

A metodologia que utilizamos para a identificação dos pipes usa a concatenação de strings para uma fácil identificação. Por exemplo, o *fifo de entrada* do nodo 1 será o "in1"e o do nodo 77 será o "in77", tal como o *fifo de sáida* do nodo 1 será o "out1"e o do nodo 77 "out77".



Estrutura de um nodo.

Por exemplo, quando fazemos *./node 1 ./const 10*, este vai criar um node que é identificado com o número 1, e todo o input recebido irá ser recebido durante execução do componente const, e todo o output irá sair com um *:10* no final.

```
1 dupexecp = fork();
2       if(!dupexecp){
3         close(nonamepipein[WRITE]);
4         close(nonamepipeout[READ]);
5         dup2(nonamepipein[READ], 0);//dup do std in para o pipe in
6         dup2(nonamepipeout[WRITE], 1);//dup do std out para o pipe
              out
7         execv(argv[2], &argv[2]);
```

```
8           execvp(argv[2], &argv[2]); //no caso do comando ser um pre
                definido o exec anterior falhara, executando entao um
                execvp
9       }
```

Aqui está um excerto de código utilizado que mostra como conseguimos colocar o input recebido na node, como input para o componente a ser executado. Usámos *fork* porque necessitamos de um processo apenas para o *execv*, visto que este vai substituir o processo onde está inserido, e utilizamos os *dup2* para "substituir"o stdin e o stdout por *pipes anónimos*, que iram fornecer informação ao componente e guardar o seu output.

A teoria da implementação foi, razoavelmente, fácil de criar, no entanto a sua implementação foi mais difícil, posto que criámos muitos processos e há muitos pipes anónimos e com nome a comunicarem entre si, logo tivemos de ser muito cuidadosos com o que faziamos em cada processo.

3.2 Connect

A connect recebe números como argumentos. Estes números representam nodes diferentes, e a connect vai fazer com que estes comuniquem entre si, ou seja, tudo que sair do node passado primeiro como argumento (estiver no *fifo de saída*), vai entrar nos nodes passados a seguir nos argumentos (entrar no *fifo de entrada*).

Por exemplo, *./connect 1 2* faz com que tudo que entra que esteja na saída do *node 1* entre no *node 2*.

```
1   int id = atoi(argv[1]), i;
2   if(nodes[id]==0){
3     perror("nodo␣nao␣existe");
4     return 0;
5   }
6   if(connects[id]){
7     kill(connects[id], SIGTERM);
8   }
9   for(i=2; i<argc; i++){
10    if(nodes[atoi(argv[i])]==0){
11      perror("nodo␣nao␣existe");
12      return 0;
13    }
14    connections[id][atoi(argv[i])] = 1;
15  }
```

No excerto de código anterior mostra como controlamos todas as possibilidades. Para isso utilizamos dois arrays e uma matriz. Caso o node passado como primeiro argumento, ou seja, onde vamos buscar informação, já tiver conexões com outros nodes, o array *connects* guarda *pid* desse processo, e matamos esse processo pois iremos fazer um processo que faça todas as conexões desse node. Porque fazemos isto ? Para controlar o número de processos criados, visto que não podemos ter um grande número de processos ativos simultaneamente. Esta foi uma das implementações que utilizamos para gerir processos.

O array *nodes* guarda todos os nodes já criados, logo se quisermos conectar nodes que não existam, ou pelo menos um deles não exista, temos de avisar que tal é impossível e retornar um erro.

Já a *matriz* serve para saber que nodes estão conectados, e atualizar essas conexões. Assim, após matarmos (caso exista) a conexão antiga entre o primeiro node (de onde vai sair a informação) e os restantes nodes (que vão receber a informação), vamos atualizar a matriz e de seguida através desta vamos criar um processo que ligue esse node aos nodes que já estava ligado e aos novos nodes.

Esta foi a função mais demorada a concluir, pois existia sempre erros minímos que nos adulteravam os resultados. Passámos algumas horas de volta desta, mas concluímo-la com sucesso.

3.3 Disconnect

A *disconnect* (tal como percebemos pelo nome) faz o contrário da connect. Esta recebe uma lista de nodes e vai desconectar o primeiro dos restantes. Esta função foi rapidamente concluída, visto que a maneira que a implementámos é semelhante à maneira como implementámos a connect, simétrica, até, na prática. A única diferença é que em vez de colocar um **1** na matriz (significa que estão conectados), colocámos um **0** (significa que estão desconectados), tudo o resto é igual, ou seja, vai verificar se os nodes existem, se existem conexões.

Após destruir o processo vai criar os processos todos de novo que existiam e não tenham sido desconectados para aquele node. No entanto esta parte já estava implementada na connect também.

3.4 Inject

A inject recebe como argumentos um número, um cmd, e argumentos para o cmd. O número representa o node para o qual queremos mandar o output que o cmd dá ao fim de executar com os seus argumentos.

Por exemplo, *./inject 1 ./const 10*, vai executar o componente **./const 10** e redirecionar o seu output para o node 1.

```
1  pipe(nonamepipeout);
2      p = fork();
3      if(!p){
4          close(nonamepipeout[0]);
5          dup2(nonamepipeout[1], 1);
6          close(nonamepipeout[1]);
7          execv(argv[2], &argv[2]);
8          execvp(argv[2], &argv[2]); //no caso do comando ser um pre
               definido o exec anterior falhara, executando entao um
               execvp
9      }
10     close(nonamepipeout[1]);
11     int infd = open(id, O_WRONLY);
12     i = readln(nonamepipeout[0], buf, PIPE_BUF);
13   write(infd, buf, i);
14   fsync(infd);
```

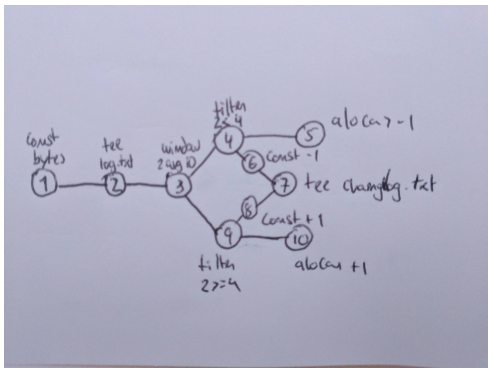
Aqui podemos ver um excerto do código utilizado, onde vemos como implementamos, mais uma vez utilizamos um *fork* para utilizar um *exec* e usamos um *dup2* para conseguir guardar todo o output gerado após a execução do exec. Após isto abrimos o *fifo de entrada* do node id, e escrevemos todo o output lá.

Foi uma função feita sem grandes problemas, pois a base da implementação já tinha sido feito noutras funções do controlador.

4 Caso de Teste

Para melhor testar o nosso trabalho, criamos um caso de teste que, ainda que muito simplificado, tenta solucionar um problema real através de stream processing.

Imaginemos então que existe um software totalmente alocado numa *cloud*, e que necessita de servidores para gerir os *data sockets* que recebe e envia. Numa situação ideal em que o número de servidores é infinito, mas não existe maneira de o gerir, a seguinte rede poderá servir como um possível gestor de alocação de servidores, de maneira a aumentar a eficiência do sistema e a quantidade de servidores alocados num determinado momento ser ótima.



Rede de teste.

Num caso teórico em que existisse um comando *alocar* em que alocava +1 ou -1 servidor dependendo do argumento, esta rede recebe os dados no formato id:numberofbytes, por exemplo, 7:4096. A rede guarda um entrylog.txt todos os sockets que recebe e filtra os dados em dois grupos, os que tem um número de bytes maior que a média dos últimos 10 e aqueles com menor. Guarda também os sockets que causaram uma mudança no número de servidores alocados. Assim, um possível ficheiro de configuração poderá ser:

```
node 1 const bytes
node 2 tee entrylog.txt
node 3 window 2 avg 2
node 4 filter 2 < 4
node 5 alocar -1
node 6 const -1
node 7 tee changelog.txt
node 8 const +1
node 9 filter 2 >= 4
node 10 alocar +1
connect 1 2
connect 2 3
```

```
connect 3 4 9
connect 4 5 6
connect 6 7
connect 8 7
connect 9 8 10
```

5 Conclusão

O trabalho foi realizado com sucesso apesar de ter dado alguns problemas. Tivemos bastante dificuldade na parte de gerir processos e pipes, pois às vezes uns abriam e outros não, havia menos processos que o suposto, entre outros problemas. Tivemos algumas dor de cabeça até encontrar o porquê do problema e a sua solução.

Achamos, no entanto, que este trabalho nos ajudou bastante a perceber todas as funcionalidades aprendidas nesta **UC**, pois tivemos uma abordagem prática e por conta própria, que nos fez procurar soluções e métodos para desenvolver estas.

Houve aspetos e funcionalidades que não fomos capazes de concluir, como por exemplo a remoção de nodos ou a substituição das suas componentes, mas mesmo assim acreditamos que o resulado foi satisfatório e, acima de tudo, enriquecedor.

Após a conclusão deste trabalho, pensamos que todos os elementos deste grupo saem com uma percepção mais realista e clara da fragilidade dos processos quando não são bem geridos. Percebemos também que a maneira como abrimos e comunicamos entre os pipes tem uma enorme importância para termos o resultado esperado.