Universidade do Minho

Departamento de Informática

Sistemas Operativos

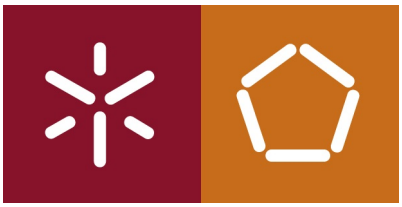
Relatório Projeto SO

Gil Cunha — Nuno Faria — Tânia Silva
Grupo 25

Resumo

Neste relatório iremos abordar a forma como executamos o trabalho prático de Sistemas Operativos. Mostraremos não só o procedimento tomado mas também as dificuldades encontradas e a forma como as superamos. Daremos também exemplos práticos da sua aplicação.

2 de Junho de 2017



Conteúdo

1 Introdução

Este relatório abordará a realização do projeto de Sistemas Operativos, que consiste na criação de uma rede de processamento de *inputs* - *Stream Processing*.

Efetuaremos essa rede com o auxílio de `fork`'s, `exec`'s, `signal`'s, entre outros. Procuraremos implementar todas as funcionalidades pretendidas, para depois termos uma maior liberdade na implementação de um exemplo prático.

2 Objetivo e enunciado

Este trabalho servirá para compreender melhor os conceitos de processos e comunicação entre estes (como por pipes ou sinais) e de que forma podemos usar os conhecimentos obtidos na disciplina para a resolução de problemas que beneficiem de programação concorrente.

O trabalho em si consiste na criação de uma rede que irá tratar informação recebida, possivelmente muita de cada vez. O programa terá os seguintes comandos:

- `const` - acrescentará uma determinada *string* no final de um *input*;
- `filter` - irá filtrar *input* de acordo com uma determinada condição;
- `window` - poderá executar diversas operações, sendo elas `min`, `max`, `avg`, `sum`;
- `spawn` - irá executar um determinado comando e colocar o resultado do `exit status` no final do *input*.

Poderá ainda executar alguns comandos UNIX já definidos (ex: `tee, cut, grep..`).

A rede será constituída por nodos que executam um determinado comando, nodos esses que poderão ser ligados a outros. É pretendido também que em qualquer momento seja possível acrescentar novos nodos, criar ou destruir ligações, ou até mesmo remover nodos, tendo o cuidado de não perder informação.

O tempo utilizado e a dificuldade deste projeto compensa depois na sua utilização, pois pode ser aplicada a diversos contextos sem ser preciso alterar uma linha de código.

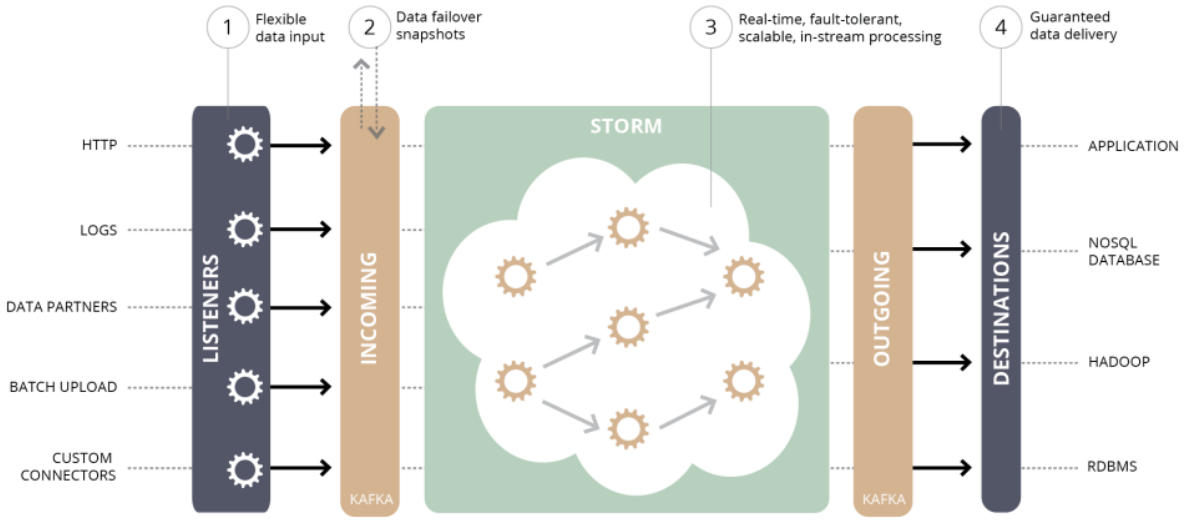


Figura 1: Esquematização de uma rede (source: `http://www.infochimps.com/`)

3 Desenvolvimento da solução

3.1 Componentes

A realização dos comandos foi a parte mais simples do projeto. Consistem apenas em manipulação de strings, algo que já sabíamos fazer sem os conhecimentos desta disciplina (à exceção do `spawn`, que terá que fazer um `fork/exec`). Cada um será um executável individual, podendo ser eventualmente utilizado por outros programas.

3.1.1 `Const`

Para este comando, apenas foi preciso alocar uma `string` como o tamanho do *input* mais o do argumento. De seguida faz-se um `strcat` e escreve-se no *output* a nova string.

3.1.2 `Filter`

Neste comando não será preciso alocar uma nova *string*. Contudo, terá que se dividir o argumento (para retirar as colunas a comparar e a forma de comparação), converter para inteiros e operações ($<, <=, >, >=, ==, !=$). Se determinado *input* verificar uma operação, escreve-se para o *output*.

3.1.3 `Window`

Este foi o comando mais complexo, visto que o resultado depende de *inputs* anteriores. Por isso foi criada uma pequena estrutura que irá guardar os resultados:

```
1     typedef struct values_{
2     int *values; // array com os valores
3     int current; // ultimo valor colocado
4     int used;    // numero de elementos usados
5     int size;    // tamanho do array
6     }*Values;
```

A cada operação, o resultado da coluna desejada é guardada no *array* `values`. Para gerir a memória, apenas serão guardados os elementos necessários. Por exemplo, se fizermos uma média dos 15 últimos elementos, apenas guardaremos os 15 últimos.

3.1.4 `Spawn`

Neste comando foi feito um `execvp` dos argumentos fornecidos, já com os `$` substituídos pela respetiva coluna do *input*. O *output* de realizar é ignorado. O *exit status* obtido é depois colocado no final da linha.

3.2 Controller

Esta é a parte principal do projeto, constituída por diferentes comandos, como por exemplo criação e ligação de nós.

3.2.1 Node

Este é o comando que cria e acrescenta um novo nó à rede. Primeiro, o nó irá abrir o *pipe* de entrada e criar um *filho*. Esse *filho* fará `dup2` do *pipe* de entrada com o seu *input*, executando de seguida o respetivo comando. Inicialmente, tínhamos apenas um `exec` (sem nenhum `fork`) que lia de um *pipe* e escrevia diretamente para a sua saída. Mas como não sabemos se terá descendência, não pode abrir diretamente o pipe de saída. Assim, será o *pai* a escrever para a saída e, quando for preciso redirecionar para o *pipe*, o *pai* recebe um sinal e faz um `dup2`, evitando aberturas de *pipes* de saída desnecessários.

3.2.2 Inject

Comando que servirá para enviar informação para um nó. O que simplesmente irá fazer é abrir dois pipes (entrada e saída). A abertura do pipe de saída é trivial, já para o de entrada irá executar um comando que abrirá um novo terminal, executando consequentemente um programa `client`, que irá ler do *stdin* e escrever para a entrada do `inject`.

3.2.3 Connect

Comando responsável pela ligação entre nós. No início estávamos a pensar deixar a parte de ligar um nó a outro para o processo `Node`, que ao executar o comando também tratava da entrada e saída. Contudo, isto pareceu-nos muito complicado, sendo por isso optado pela construção de processos intermédios que farão a ligação entre diversos nós.

Um `Connect` irá inicialmente abrir todas as ligações para onde irá escrever. De seguida irá terminar o processo que estava a fazer a ligação anteriormente, caso exista. Por último, irá ler da saída de um nó e escrever para todos os outros, num ciclo.

Em pseudocódigo:

```
1  sinalizar_node_in(abre pipe de saida)
2
3  for {id_1, id_2, ..., id_m}
4      adicionar_valor(in, id_i, "out")
5      adicionar_valor (id_i, in, "in")
6
7  for {out_1, out_2, ..., out_n}
8      abrir_entrada(out_i)
9
10 if (existe ligacao anterior) -> termina-la
11
12 while(read(in)){
13     ... //-> servira para verificar disconnects
14     for {out_1, out_2, ..., out_n}
15         write(out_i)
16 }
```

Por exemplo, se tivermos um `connect 1 2 3`, adicionamos o 2 e 3 na lista de saídas do 1 (e por sua vez o 1 na lista de entradas do 2 e 3). De seguida, iremos abrir não só

o *pipe* do 2 e 3, mas também todos os outros que já estavam ligados ao 1, terminado o processo anterior. Caso tenhamos um novo `connect`, a execução é repetida.

3.2.4 `Disconnect`

Comando responsável por desfazer a ligação entre dois nós. Como já tínhamos um processo que fazia a ligação entre um com vários nós (`Connect`), tornava-se mais fácil desfazer a sua ligação, visto que não tínhamos que alterar o Node. Podíamos fazer um novo `Connect` sem o nó que queríamos remover. Contudo, encontramos um forma melhor, através de sinais.

Precisamos apenas de dizer ao `Connect` que terá de parar de escrever para o *pipe* de um determinado nó, por isso, decidimos enviar os *bits* do respetivo número. A cada *bit 1*, enviaremos o sinal `SIGUSR1`, e para cada *bit 0* o `SIGUSR2`. Quando tivermos feitos todos os envios necessários enviaremos `SIGHUP`. A cada interação, o Connect irá construir o número a partir da quantidade de *bits* enviados e da função `pow`.

Quando tivermos o *id*, removemos o seu descritor (de entrada) da lista de escritas, completando assim o `disconnect`.

Parte que irá enviar os bits

```
//Enviar os bits do id a desligar
int id1_connect = info->node[findID(info, id1)].connect_pid;
for (aux = id2; aux > 0; aux = aux >> 1){
    if (aux % 2) kill(id1_connect, SIGUSR1); //Bit 1
    else kill(id1_connect, SIGUSR2); //Bit 0
    pause(); //Fica a espera para enviar um novo bit
}
```

Parte que irá receber os bits

```
switch(sig){
    case SIGUSR1 :{ //Bit 1
        disconnect_id += pow(2, disconnect_iterations);
        disconnect_iterations ++;
        break;
    }
    case SIGUSR2 :{ //Bit 0
        disconnect_iterations++;
        break;
    }
    case SIGHUP :{ //Envio completo
        disconnect_done = 1;
        disconnect_ids[disconnect_ids_size++] = disconnect_id;
        break;
    }
    ...
}
//Diz que pode receber um novo sinal
kill(getppid(), SIGUSR1);
```

Para garantir que não haverão *bits* perdidos, a parte que envia os bits irá esperar por um sinal enviado pela parte que irá receber, através de um `pause()`.

3.2.5 `Remove`

Este comando irá remover um nó da rede e ligar cada um dos nós que ligavam a ele a todos os nós que este ligava. Para isto, removemos o nó a desligar das listas de saída e entrada de todos os nós que ligavam/eram ligados a este, respetivamente. De seguida, fazemos para cada par de nós (entrada e saída) um `connect`. Por ultimo, enviamos um sinal para o `Node` a remover, dizendo que pode terminar a sua execução, e retiraremos a informação desse nó da estrutura.

3.2.6 `Client`

Este não é um comando do enunciado do projeto, mas apenas um auxílio ao comando `inject`. Irá receber como argumento um nome de um determinado *pipe*, donde terá que escrever o que lhe aparecer no *stdin*. Assim, é possibilitada a existência de múltiplos `injects`.

3.3 `StringProcessing`

Tanto os comandos como o controlador terão que manipular *strings*. Para isso, criamos um módulo que servirá para o auxílio destes programas. Salientando-se algumas funções:

```
1 char** divideString(char x[], char* divider);
2 char* strcatWithSpaces(char **c);
3 char* addCommandPrefix(char* cmd);
4 char* fifoName(int id, char* io);
5 int readline(int fildes, char *buffer, int size);
```

- `divideString` - irá dividir uma *string* em várias, separadas por um caractere *divider*;
- `strcatWithSpaces` - concatenará várias strings numa, separadas por espaço;
- `addCommandPrefix` - adicionará um "./"no início dos comandos do locais (ex: const - ./const);
- `fifoName` - criará uma *string* baseada num número e numa *string* "in"ou "out" (ex: fifoName(4, out) = fifo_4_out);
- `readLine` - lê uma linha de um descritor e coloca no `buffer`, retornando o número de caracteres lidos.

4 Estrutura de dados

Para tornar a realização do trabalho mais simples, colocamos a informação relativa a cada nó numa estrutura.

```
1 //Estrutra de um nodo
2 typedef struct controllerNode{
3     int id; ///< id do nodo
4     char* cmd; ///< string com o comando a ser executado
5     char* args; ///< string com os argumentos do comando
6     int* in_ids; ///< ids de todas a ligacoes ao no
7     int size_in; ///< tamanho de in_ids
8     int used_in; ///< quantidade de in_ids
9     int* out_ids; ///< ids a quem o no liga
10    int size_out; ///< tamanho de out_ids
11    int used_out; ///< quantidade de out_ids
12    char* pipeIn_name; ///< nome do pipe de entrada
13    char* pipeOut_name; ///< nome do pipe de saida
14    int f_in; ///< descritor do pipe de entrada
15    int f_out; ///< descritor do pipe de saida
16    int pid; ///< id do processo que cria o no
17    int connect_pid; ///< id do connect de saida
18 }*ControllerNode;
```

Teremos também uma outra estrutura que guardará não só toda a informação de cada nó mas também outros inteiros que indicam tamanhos.

```
1 //Estrutura principal
2 typedef struct controllerInfo{
3     int size; ///< tamanho da estrutura
4     int used; ///< numero de posicoes usadas
5     int nnodes; ///< numero de nodes ate ao momento
6     ControllerNode node; ///< array de nodos
7 }*ControllerInfo;
```

Em `ControllerNode` guardaremos o ID para sabermos que nó estamos a aceder no momento, encontrar a posição ou se existe na estrutura (os nós não são guardados por ID's mas por ordem de chegada); os comandos para sabermos o que vamos executar; os *in_ids* e *out_ids* para sabermos quais os `connects` que temos de fazer (tanto a fazer `connect` como `remove`); o nome dos pipes e os descritores para poderem ser abertos por outros processos (como `connect`); os `process ids` para poder ser feita a comunicação entre processos.

Já em `ControllerInfo` guardamos o `nnodes` para gestão de nomes de *pipes*.

Com isto garantimos uma maior facilidade na manipulação dos dados.

5 Exemplos

5.1 Controlo de tamanho de peças industriais

Consideremos uma industria que se encarrega de produzir um determinado tipo de peças. O tamanho das peças produzidas por essa empresa tem que estar num intervalo específico. Da produção diária, podemos ter até cerca de 3% de peças defeituosas, sendo que caso o número seja superior, terá que haver uma manutenção das máquinas.

Podemos especificar a rede da seguinte forma:

```
node 1 const 300:350:id
node 2 window 2 sum 200000
node 3 filter 1 < 3
node 4 filter 1 >= 3
node 5 filter 1 > 4
node 6 filter 1 <= 4
node 7 tee defective.txt
node 8 tee accepted.txt
node 9 window 1 sum 6000
node 10 const 6000
node 11 filter 7 == 8
node 12 const 0
node 13 const 1
node 14 window 7 sum 10
node 15 const 10
node 16 filter 8 == 9
node 17 spawn shutdown -h now
node 18 const accepted
node 19 const defective
node 20 tee log.txt
connect 1 2
connect 2 3 4
connect 4 5 6
connect 6 8
connect 8 18
connect 3 7
connect 5 7
connect 7 9 13 19
connect 9 10
connect 10 11
connect 11 17
connect 8 12
connect 12 14
connect 13 14
connect 14 15
connect 15 16
connect 16 17
connect 18 20
connect 19 20
```

Teremos que injetar no nó um `const 1`, para poder saber a identificação de cada peça. Por sua vez, esse nó irá colocar a informação sobre o intervalo do tamanho das peças ([300,350]).

Enviaremos depois para nós que irão determinar o tamanho da peça, colocando-a num ficheiro `accepted.txt`, que indica que esta se encontra nos limites pretendidos, ou num `defective.txt`, que indica o contrário.

Cada uma das peças será etiquetada com a sua condição (`accepted/rejected`) e colocado num ficheiro `log.txt`.

Para verificarmos a qualidade da produção, iremos contar todos os casos de peças defeituosas. Caso seja superior a 3% do 200000 produzidos (= 6000), faremos um `spawn shutdown -h now` para terminar a produção daquela determinada máquina. Outra verificação foi a contagem das ultimas peças. Na eventualidade de encontrarmos dez peças defeituosas seguidas (o que numa situação normal seria muito improvável), isto dar-nos-à a indicação que algo correu mal naquele determinado instante, procedendo assim à interrupção da produção (evitando prejuízos nas próximas peças).

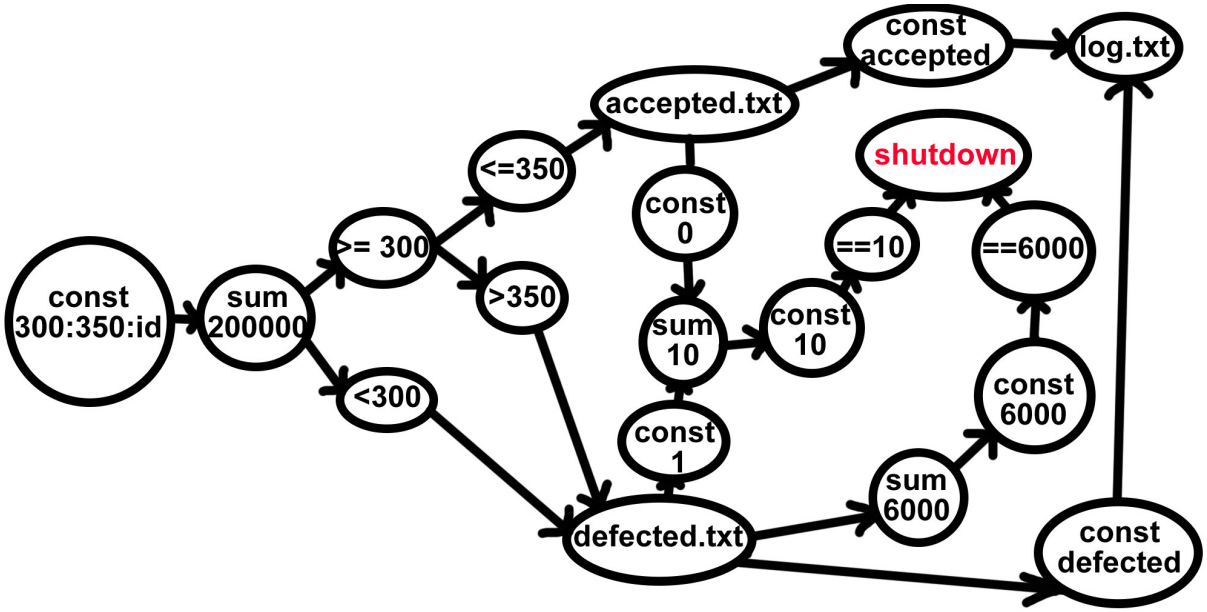


Figura 2: Esquematização da rede de controlo de tamanho de peças

Exemplo de um `log.txt`

```
330:1:300:350:id:0:accepted
332:1:300:350:id:1:accepted
351:1:300:350:id:2:defective
328:1:300:350:id:3:accepted
344:1:300:350:id:4:accepted
298:1:300:350:id:5:defective
301:1:300:350:id:6:accepted
300:1:300:350:id:7:accepted
323:1:300:350:id:8:accepted
348:1:300:350:id:9:accepted
355:1:300:350:id:10:defective
325:1:300:350:id:11:accepted
```

5.2 Processamento de notas de alunos

Consideremos os alunos do 2º ano do 2º semestre de MIEI. Queremos processar as notas automaticamente, separando-as por UC.

Pode-se fazer a seguinte representação:

```
node 1 const Notas
node 2 const MediaTotal
node 3 window 3 avg 400
node 4 tee Notas.txt
node 5 const 1:2:3:4:5:6:10
node 6 filter 3 >= 12
node 7 filter 3 < 12
node 8 filter 2 == 6
node 9 filter 2 == 7
node 10 filter 2 == 8
node 11 filter 2 == 9
node 12 filter 2 == 10
node 13 filter 2 == 11
node 14 const Passou
node 15 const NaoPassou
node 16 const Fim
node 17 tee EE.txt
node 18 tee CP.txt
node 19 tee SO.txt
node 20 tee POO.txt
node 21 tee LI.txt
node 22 tee Opcao.txt
node 23 const 18
node 24 filter 3 >= 13
node 25 const ProvaOral
node 26 tee ProvaOral.txt
node 27 spawn gedit Notas.txt
connect 1 2 5 27
connect 2 3
connect 3 4
connect 5 6 7
connect 6 14 23
connect 7 15
connect 14 16
connect 15 16
connect 16 8 9 10 11 12 13
connect 8 17
connect 9 18
connect 10 19
connect 11 20
connect 12 21
connect 13 22
connect 23 24
connect 24 25
connect 25 26
```

A rede irá certificar-se de separar os *input* de acordo com a UCs (um inteiro de 1 a 6) indicando se um determinado aluno passou ou não. Selecionamos ainda aqueles que têm uma nota superior a 18 valores e colocamos numa lista de alunos que terão que fazer uma prova oral.

Também calcularemos a média geral de todas as notas.

No final iremos obter 8 ficheiros de texto, onde 6 deles terão a nota de todos os alunos a uma determinada cadeira (`LI.txt, SO.txt, POO.txt, EE.txt, Opção.txt, CP.txt`), outro conterá os que terão que fazer uma prova oral (`ProvaOral.txt`) e por fim um que terá todas as notas de todas as UCs (`Notas.txt`), indicando a média total.

Exemplo de um ficheiro de uma determinada UC (3)

```
1  Hugo_Costa:3:15:Aluno:Notas:(...):Passou:Fim
2  Marco_Abreu:3:8:Aluno:Notas:(...):NaoPassou:Fim
3  Rui_Peixoto:3:10:Aluno:Notas:(...):Passou:Fim
4  Laura_Martins:3:18:Aluno:Notas:(...):Passou:Fim
5  Diana_Silva:3:7:Aluno:Notas:(...):NaoPassou:Fim
6  Joao_Sousa:3:20:Aluno:Notas:(...):Passou:Fim
7  Maria_Gomes:3:17:Aluno:Notas:(...):Passou:Fim
8  Rafael_Oliveira:3:6:Aluno:Notas:(...):10:NaoPassou:Fim
9  Filipa_Cunha:3:9:Aluno:Notas:(...):NaoPassou:Fim
10 Carlos_Mendes:3:19:Aluno:Notas:(...):Passou:Fim
```

Exemplo do ficheiro `ProvaOral.c`

```
1 Laura_Martins:3:18:Aluno:Notas:(...):ProvaOral
2 Joao_Sousa:3:20:Aluno:Notas:(...):ProvaOral
3 Carlos_Mendes:3:19:Aluno:Notas:(...):ProvaOral
```

6 Conclusão

A realização deste trabalho permitiu-nos adquirir métodos de resolução de problemas onde a programação concorrente torna-os muito mais fáceis de resolver.

A parte mais difícil deste trabalho foi começar. Não a parte de fazer os comandos como o `const`, mas como é que iremos organizar o projeto. `Como é que um nó depois de criado iria controlar para onde escrevia? Se não sabemos se um nó terá saída, como é que abrimos o pipe?` Estas e outras dúvidas foram resolvidas ao olhar para o problema e simplificar o mais que podíamos (como por exemplo criar um processo auxiliar que trata das conexões). Também foi muito importante a utilização de sinais para a comunicação de processos, um conceito simples (e muito limitado, visto que existem apenas cerca de 30 sinais) mas muito prático.

Se não tivéssemos conhecimentos de programação concorrente, a forma de como iriamos implementar a rede das peças seria de forma sequencial, e quando se tinha uma bifurcação na rede, apenas podíamos processar um lado de cada vez, e isso com grandes quantidades de informação em tempo real tornaria-se muito custoso.