Projeto de Sistemas Operativos
Grupo 21

Diogo Braga A82547 João Silva A82005
Ricardo Caçador A81064

31 de Janeiro de 2019

Resumo

Este documento apresenta o projeto de Sistemas Operativos, do curso de Engenharia Informática da Universidade do Minho.

O projeto baseia-se na criação de um sistema para processamento de *notebooks*, que misturam fragmentos de código, resultados de execução, e documentação.

Conteúdo

1 Introdução

Este trabalho tem por base a construção de um programa para processamento de notebooks. Aliado a tal está o desafio de procurar o melhor algoritmo para resolução do problema proposto usando os conhecimentos adquiridos na Unidade Curricular de Sistemas Operativos.

A Secção 2 indica a abordagem ao problema proposto no projeto, a Secção 3 aborda as diferentes execuções utilizadas para cada género de linha presente nos ficheiros, a Secção 4 apresenta as diferentes abordagens às interrupções que podem acontecer durante a execução do programa, e a Secção 5 aborda o momento final de transferência de dados do ficheiro auxiliar para o original. A Secção 6 mostra um exemplo de teste realizado com sucesso. O relatório termina com conclusões na Secção 7, onde é também apresentada uma análise crítica dos resultados obtidos.

2 Abordagem ao problema

2.1 Parser do ficheiro

De modo a iniciar a análise do ficheiro carregamos o seu conteúdo através da função ***read_all_lines***.

Esta utiliza inicialmente uma função ***read_line*** que lê linha a linha o ficheiro até ocorrer *EOF*. O final de cada linha é dado pelo aparecimento do carácter $\backslash n$, sendo que nestes momentos é dado como lida uma linha e esta é retornada.

É aberto um **ficheiro auxiliar**, para leitura e escrita, onde vão ser colocados tanto os fragmentos de código a ser executados como os resultados dessas mesmas execuções. Este ficheiro denominado **auxiliar.nb** é colocado na pasta **tmp** com o intuito de não influenciar por exemplo comandos como o **ls** que listam todos os ficheiros da pasta, neste caso também listaria o ficheiro auxiliar, que não é suposto estar vísivel. É também alocada memória para uma estrutura auxiliar denominada **POSICOES** criada para armazenar um vetor de **POSICAO**, que contêm o início da posição de um resultado, assim como o tamanho, em bytes, desse mesmo resultado. Com esta estrutura temos armazenadas todas as posições do ficheiro auxiliar onde estão os resultados das execuções de comandos executados até ao momento.

Com o auxílio da system call ***lseek*** é então lido todo o ficheiro, linha a linha.

```
typedef struct posicao{
        int posicao_inicio;
        int num_bytes;
} *POSICAO;

typedef struct posicoes{
        POSICAO* pos;
        int pos_count;
} *POSICOES;
```

Figura 1: Struct **POSICOES**

3 Execução de programas

Todo este processo é comandado pela função ***execution***.

Primeiramente, no processo de execução, é identificada a natureza da linha através da função ***line_nature***, que consoante os caracteres iniciais diferentes de cada linha atribui também diferentes execuções, nomeadamente:

- Comando normal;
- Comando com pipe, em que o stdin é o resultado comando anterior;
- Comando com pipe, em que o stdin é o resultado do n-ésimo comando anterior;
- Início do resultado;
- Fim do resultado;
- Durante o resultado;
- Linha de texto/Documentação.

Antes da execução de qualquer comando, é realizada a separação do mesmo. Inicialmente, e já depois de atribuído o tipo de execução a realizar, são retirados os caracteres que mostram esta diferença. Depois, com o auxílio da função ***strtok*** realizamos a divisão da string, sendo os delimitadores o espaço e o \n. Estas strings resultantes, que serão o comando e respetivos argumentos, são armazenados num array de strings, contendo NULL como último elemento, com isto temos tudo preparado para a execução com um **execvp**. O mesmo processo é realizado para as restantes linhas.

De modo a mostrar que estamos num momento de impressão de resultado, tanto a execução normal como a execução com pipe, coloca no ficheiro auxiliar os caracteres que indicam início de impressão do resultado, e no fim os caracteres que indicam final da impressão do resultado.

3.1 Execução linha de texto

Neste tipo de execução apenas é realizado o ***write*** da linha para o ficheiro auxiliar em causa.

3.2 Execução comando normal

Na execução do comando normal é criado um processo-filho que redireciona o seu stdout para o ficheiro auxiliar e depois realiza o **execvp** do array de char* **comando**. Desta forma, o resultado da execução é escrita no ficheiro auxiliar em causa.

De modo a manter atualizada a estrutura auxiliar que estamos a utilizar, guardamos o início e o final do resultado para que este seja adicionado à referida estrutura, algo realizado com as capacidades da system call ***lseek***.

3.3 Execução comando com pipe

A execução com pipe pode ser formatada de duas formas distintas. Um caso em que a finalidade é usar o comando anterior como resultado anterior, e outro caso em que a finalidade é usar o n-comando anterior como resultado anterior. Esta decisão é diferenciada através do parâmetro passado na invocação desta função.

Caso seja para analisar o n-comando anterior, a função antes de inicializar o processo de execução absorve este parâmetro para o usar.

A partir deste momento, a execução é igual para ambos os modos, porque o valor do comando anterior, neste caso 1, também está presente em n, no caso do n-comando anterior.

É realizado um acesso à estrutura POSICAO para ter a informação do resultado do n-comando anterior. Este processo é realizado através da função ***le_resultado***, que através do início do resultado e do número de bytes a ler, retorna num apontador alocado o resultado lido do ficheiro auxiliar.

Na execução, significado real do âmbito disciplinar, é criado um pipe ***pd*** através da system call ***pipe***.

- No processo-filho é duplicado o stdout para o ***descritor de escrita*** do pipe através da system call ***dup2***.

É escrito o resultado no stdout, que por consequência da mudança anterior, foi redirecionado para o pipe.

- No processo-pai o stdin foi redirecionado para o ***descritor de leitura*** do pipe;
- Procedeu-se também ao redirecionamento do stdout do processo-pai para o ***ficheiro auxiliar***.

Desta forma, o ***execvp*** de **comando** vai usar como ponto de partida não o stdin habitual, mas sim o descritor de leitura do pipe. Esta execução é escrita no ficheiro auxiliar, também devido às duplicações anteriormente referidas. De referir que o processo-filho é depois morto.

De modo a manter atualizada a estrutura auxiliar que estamos a utilizar, guardamos o início e o final do resultado para que este seja adicionado à referida estrutura, algo realizado com as capacidades da system call ***lseek***.

4 Deteção de erros e interrupção de execução

Como é pedido no enunciado caso algum dos comandos não consiga ser executado, ou caso ocorra a interrupção de um processamento em curso, o notebook deve permanecer inalterado.

Esta situação pode acontecer em vários pontos do processamento, e desta forma existem também diferentes formas de proceder, nomeadamente:

- Caso a situação ocorra no ***open*** do ficheiro original, este é fechado e o processo é morto, segundo o sinal SIGQUIT.
- Caso a situação ocorra durante a **leitura e execução do programa**, o ficheiro auxiliar é removido, é fechado o ficheiro original e o processo é morto, assim o ficheiro original fica inalterado.
- Caso a situação ocorra **depois do processamento do notebook**, ou seja, depois do ficheiro original ser truncado e aberto para escrita, é copiado todo o conteúdo do auxiliar para o original. Posteriormente o auxiliar é apagado e o original fechado.
- Caso a situação ocorra na **copia** do ficheiro auxiliar para o original, o auxiliar é apagado e é fechado o original. O processo seguidamente é morto.
- Caso a situação ocorra na **remoção** do ficheiro auxiliar, o original é fechado e o processo seguidamente é morto.

5 Finalização do processo

Depois de todas as execuções ocorridas no ficheiro auxiliar, este é totalmente copiado para o ficheiro original, que anteriormente fora truncado.

O ficheiro auxiliar é depois apagado através da realização da system call ***execlp*** com os parâmetros ***("rm","rm","/tmp/auxiliar.nb", NULL)***.

6 Exemplo de teste

```
Este comando lista os ficheiros:
$ ls
Agora podemos ordenar estes ficheiros:
$| sort
Escolher os 5 primeiros do ls:
$2| head -5
Tail do anterior
$| tail -1
Word count do ls
$4| wc -l
```

Figura 2: Ficheiro original no ponto inicial

```
Este comando lista os ficheiros:
$ ls
>>>
Makefile
execution.c
file_string.c
main.c
notebook
obj
parser.c
<<<
Agora podemos ordenar estes ficheiros:
$| sort
>>>
Makefile
execution.c
file_string.c
main.c
notebook
obj
parser.c
<<<
```

Figura 3: Ficheiro auxiliar a meio do processo

```
Este comando lista os ficheiros:
$ ls
>>>
Makefile
execution.c
file_string.c
main.c
notebook
obj
parser.c
<<<
Agora podemos ordenar estes ficheiros:
$| sort
>>>
Makefile
execution.c
file_string.c
main.c
notebook
obj
parser.c
<<<
Escolher os 5 primeiros do ls:
$2| head -5
>>>
Makefile
execution.c
file_string.c
main.c
notebook
<<<
Tail do anterior
$| tail -1
>>>
notebook
<<<
Word count do ls
$4| wc -l
>>>
       7
<<<
```

Figura 4: Ficheiro original no ponto final

7 Conclusão

A escolha desta abordagem relatada no relatório deve-se a vários fatores:

- Usamos um ficheiro auxiliar para processar o original porque desta forma não estamos limitados a um espaço de memória alocado que possa ser curto para o necessário no programa.
- Como estamos a realizar as execuções no ficheiro auxiliar, caso aconteça uma interrupção ou um erro, é fácil apagar este ficheiro sem influenciar o original, que não vai ser alterado.
- Com as posições de todos resultados guardados numa estrutura é facil voltar atrás até ao n-ésimo resultado sem ter que executar várias vezes os mesmos comandos tal como pedido no enunciado.

Importante referir que, com a nossa implementação, conseguimos realizar o re-processamento de um notebook, podendo este ser alterado e reprocessado.

O grupo pensou em várias formas de aplicar execução concorrente, mas com a estrutura do projeto previamente estabelecida percebemos que poderia ser de difícil implementação e dessa forma nao compensaria reformular uma grande parte do código.

Seria interessante se o executável pudesse receber mais que um argumento, pois implicaria uma possível decisão sobre um escalonador de processos, ou mesmo uma execução concorrente no processamento de notebooks.

Finalizando, gostaríamos de frisar que concluímos todas as funcionalidades básicas propostas, bem como uma funcionalidade avançada. Tendo em conta os objetivos definidos, o grupo considera que os resultados principais foram obtidos.