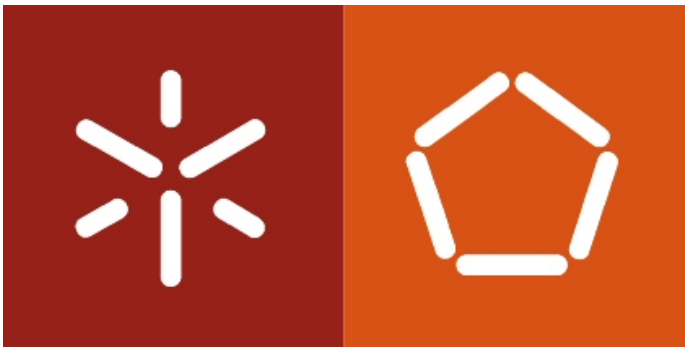

Processamento de Notebooks

2.º ano de MIEInf
Departamento de Engenharia Informática

Inês Alves - A81368
João Lopes - A80397
Shahzod Yusupov - A82617

Unidade Curricular: Sistemas Operativos

Grupo n.º 55

Junho de 2018

Conteúdo

1 Introdução

No âmbito da Unidade Curricular de Sistemas Operativos, do Mestrado Integrado em Engenharia Informática da Universidade do Minho, foi-nos proposta a implementação de um sistema para processamento de *notebooks*.

Foi essencial para a execução deste projeto o conhecimento adquirido ao longo do semestre nas aulas práticas desta UC, a utilização dos manuais das chamadas ao sistema (*System Calls*) e ainda um pouco de pesquisa e aprofundamento de conteúdos pelos motores de busca já por todos conhecidos e diariamente utilizados.

No entanto, conseguimos perceber desde muito cedo que, devido à grande quantidade de processos que teríamos que executar, iria ser essencial o domínio da *System Call fork()*:

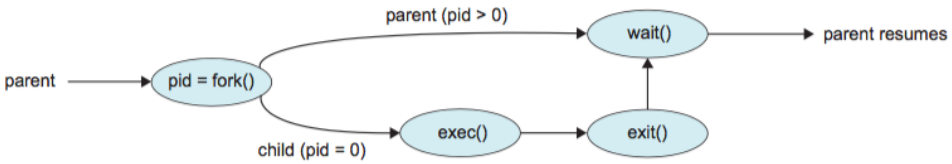

Figura 1: Criação de processos usando a *System Call* fork()

Neste relatório será descrito claramente o que o programa faz, com exemplos de testes que foram feitos pelo grupo, assim como solicitado pelos docentes.

2 Descrição do Problema

Como já referido, nesta UC, o objetivo era a criação de um sistema para processamento de *notebooks*.

No contexto do nosso projeto, um *notebook* é um ficheiro de texto que, após ser processado, é modificado de modo a incorporar resultados da execução de código ou comandos nele embebidos.

Posto isto, o objetivo principal deste projeto era a interpretação e execução dos comandos embebidos num determinado ficheiro, sendo que o resultado desta execução seria acrescentado ao ficheiro, tal como clarificado no enunciado do projeto.

Para além disso, podemos dividir os comandos que pretendemos executar em dois tipos:

- **Tipo 1: Caso em que a linha onde se encontra o comando começa apenas por $**
  Estas linhas são interpretadas como comandos que serão executados, sendo o resultado produzido inserido imediatamente a seguir, delimitado por >>>e <<<.

- **Tipo 2: Caso em que a linha onde se encontra o comando começa por $|**
  Estas linahs executam comandos que têm como *standard input* o *output* do comando anterior. Nestes também o resultado é delimitado por >>>e <<<.

  Para realizar com sucesso este tipo de comandos, foi necessário "jogar"muito bem com os redirecionamentos de leitura e de escrita, usando para isto pipes anónimos, estudados por nós nas aulas desta UC.

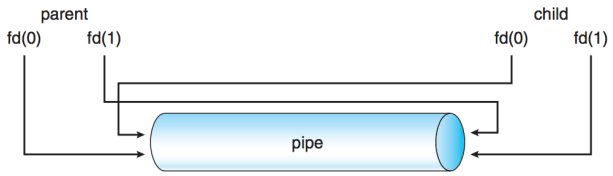

Figura 2: Descritores de ficheiro para um pipe anónimo

Sendo assim, o programa deve reconhecer que somente estes casos possuem comandos à sua frente possíveis de serem executados pelo mesmo.

Tendo em conta a diversidade de opções que possuimos para realizar este sistema, houve inicialmente uma partilha de ideias entre os elementos do grupo com vista a ponderar todas as vantagens e desvatagens que teriamos com as diferentes abordagens possíveis, sendo que, o nosso principal objetivo, foi sempre chegarmos à melhor e mais eficiente implementação.

3 Conceção da Solução

Após discutido pelo grupo, chegamos à conclusão que, para a elaboração do programa, teriamos três partes(funções) distintas:

1. **Função responsável pela execução de comandos**

   Inicialmente, esta função estava dividida em duas funções: uma que executava comandos do Tipo 1 e outra que executava comandos do Tipo 2, ambos referidos e explicados na secção anterior.

   No entanto, foi decidido em grupo que não seria necessária esta distinção, uma vez que, para além dos problemas que estávamos a ter para guardar os *outputs* dos comandos que seriam necessários para a execução de outros comandos, a mesma podia ser feita de outra forma. A nova e definitiva forma de realizar a execução de comandos foi então passar como argumento a esta função a linha de uma matriz com os comandos que pretendíamos executar (matriz esta preenchida pela função *parse*).

2. **Função responsável por terminar os processos**

   Esta função foi desenvolvida com o intuito de realizar o requisito 2.1.3 do enunciado do projeto. Quando o processo filho termina, um sinal (*SIGCHLD*) é enviado ao pai deste processo. Resumidamente, o objetivo deste envio de sinal é dar a conhecer ao pai se ocorreu algum erro na execução dos comandos e, caso tenha ocorrido, em qual dos processos ocorreu esse erro, sendo então possível que a execução de nenhum dos processos seja feita, tal como pedido no enunciado.

   Para controlo de erros foi também utilizada diversas vezes a função *perror* que imprime, se for o caso, uma mensagem de erro para o *Standard Error*.

3. **Função parse**

   Por fim, a função *parse* é a principal função deste projeto. Como referido em cima, é esta função que preenche a matriz com os comandos que irão (ou não, no caso de existir

um comando inválido) ser executados. Posto isto é então a base para a realização do requisito 2.1.1, auxiliada pela primeira função aqui referida.

Para além disso também permite dar resposta ao requisito 2.1.2, não escrevendo o conteúdo que está entre os limites já falados: >>>e <<<no ficheiro original. Esta torna assim possível alterar os comandos que queremos executar no ficheiro, sem comprometer a eficácia do projeto, cumprindo o requisito mencionado.

Todas estas funções se encontram no mesmo ficheiro: parse.c. No entanto, este projeto conta com a implementação de mais ficheiros essenciais à boa execução do programa, sendo eles: um ficheiro que contém a função *main*, um ficheiro auxiliar onde são gerados os resultados da execução para posterior cópia para o ficheiro originalmente dado, e, obviamente, a makefile para ser possível a compilação de todo o código desenvolvido pelo grupo.

```
HEADERS = parser.h

default: main

main.o: main.c $(HEADERS)
  gcc -c main.c -o main.o

parser.o: parser.c parser.h
  cc -c main.c inc.h

main: main.c parser.h
  cc -o notebook main.c parser.c

clean:
  -rm -f main.o
  -rm -f main
```

Figura 3: Makefile

4 Resultados do Projeto

Já explicado o modo de conceção da solução deste programa, vejamos agora o mesmo a funcionar.

Consideremos que queremos executar os comandos presentes neste ficheiro:

```
Este comando lista todos os processos do sistema:
$ ps
Este comando lista os ficheiros:
$ ls
Agora podemos ordenar estes ficheiros:
$| comando
E escolher o primeiro:
$| head -1
E ainda contar o número de linhas, palavras e bytes existente no output deste último comando:
$| wc
```

Figura 4: Ficheiro a executar com comando inválido

Note-se que o terceiro comando é um comando que não existe, logo o programa vai permanecer inalterado.

No entanto, se todos os comandos forem comandos válidos, vai ser executado sem qualquer problema. Considere-se o seguinte exemplo:

```
Este comando lista todos os processos do sistema:
$ ps
Este comando lista os ficheiros:
$ ls
Agora podemos ordenar estes ficheiros:
$| sort
E escolher o primeiro:
$| head -1
E ainda contar o número de linhas, palavras e bytes existente no output deste último comando:
$| wc
```

Figura 5: Ficheiro a executar com todos os comandos válidos

Resultado da execução deste ficheiro:

```
Este comando lista todos os processos do sistema:
$ ps
>>>
  PID TTY           TIME CMD
 2578 ttys000    0:00.03 /Applications/iTerm.app/Contents/MacOS/iTerm2 --server login -fp ines
 2580 ttys000    0:00.07 -bash
 2984 ttys000    0:00.00 ./notebook exemplo.tex
<<<
Este comando lista os ficheiros:
$ ls
>>>
Makefile
exemplo.tex
ficheiroAux.c
latex
main.c
notebook
parser.c
parser.h
<<<
Agora podemos ordenar estes ficheiros:
$ sort
>>>
Makefile
exemplo.tex
ficheiroAux.c
latex
main.c
notebook
parser.c
parser.h
<<<
E escolher o primeiro:
$ head -1
>>>
Makefile
<<<
E ainda contar o número de linhas, palavras e bytes existentes no output deste último comando:
$ wc
>>>
       1       1       9
<<<
```

Figura 6: Ficheiro executado com sucesso

Vejamos agora o que acontece se qualquer um dos comandos for alterado:

```
Este comando lista os ficheiros:
$ ls
>>>
Makefile
exemplo.tex
ficheiroAux.c
latex
main.c
notebook
parser.c
parser.h
<<<


Este comando lista todos os processos do sistema:
$ ps
>>>
  PID TTY           TIME CMD
 2578 ttys000    0:00.03 /Applications/iTerm.app/Contents/MacOS/iTerm2 --server login -fp ines
 2580 ttys000    0:00.10 -bash
 3147 ttys000    0:00.01 ./notebook exemplo.tex
<<<


Agora podemos escolher o segundo:
$ head -2
>>>
  PID TTY           TIME CMD
 2578 ttys000    0:00.03 /Applications/iTerm.app/Contents/MacOS/iTerm2 --server login -fp ines
<<<


E obter informação sobre a diretoria onde me encontro:
$ pwd
>>>
/Users/ines/Desktop/novo/SO
<<<
```

Figura 7: Ficheiro alterado e executado com sucesso

5 Conclusão

Este projeto, foi um dos trabalhos mais exigentes até ao momento. Contudo, foi também um trabalho muito bom para pôr em prática os conteúdos lecionados nas aulas práticas, exigindo de nós um estudo contínuo e sólido ao longo do semestre.

Apesar das dificuldades encontradas, consideramos que a realização deste projeto foi elaborada com sucesso, sendo que conseguimos dar resposta a todas as funcionalidades básicas requeridas.

No entanto, o grupo consegue reconhecer que, com uma melhor gestão do tempo, teria sido possível realizar mais funcionalidades do que as que foram apresentadas e até melhorar as funcionalidades feitas até ao presente.