Universidade do Minho
Escola de Engenharia
Mestrado Integrado em Engenharia Informática

Sistemas Operativos

Ano Letivo de 2015/2016

Backup Eficiente

A27748 - Gustavo José Afonso Andrez

A74634 - Rogério Gomes Lopes Moreira

A67664 - Samuel Gonçalves Ferreira

20 Maio 2016

Índice

1. Introdução

O objetivo deste trabalho é, recorrendo às competências desenvolvidas nas várias disciplinas do curso, em particular, na disciplina de Sistemas Operativos, conceber uma aplicação que permita ao utilizador efetuar, de um modo eficiente, cópias de segurança (*backup*) de ficheiros ou pastas bem como restaurar (*restore*) ficheiros/pastas salvaguardados.

O sistema desenvolvido deverá apresentar características de privacidade dos dados.

2. Descrição dos ficheiros desenvolvidos

Neste ponto do trabalho pretendemos apresentar de um modo sintético mas completo, as funcionalidades desenvolvidas no projeto de forma a que o código gerado seja mais facilmente interpretado.

2.1. Inicialização (init.sh)

Para primeira utilização do programa é necessário executar o script de instalação através do comando:

```
sh init.sh
```

Com esta execução é criada no diretório `home/<user>/.` a pasta `.backup/` e, dentro desta, as pastas `data/` e `metadata/`. É também criado em `.backup/` o *pipe* que permite a comunicação entre o cliente e o servidor. O código é apresentado em seguida:

```
#!/bin/bash
mkdir ~/.backup
mkdir ~/.backup/data
mkdir ~/.backup/metadata
mkdir ~/.backup/pipe
```

2.2. Cliente (sobucli.c)

O cliente pode solicitar ao *servidor* um *backup*, *restore* ou *delete* de um ficheiro. Abaixo são explicadas as implementações destes pedidos por parte do cliente.

Estas solicitações têm de seguir os formatos:

```
sobucli backup <filePath(s)>
sobucli restore <filename(s)>
sobucli delete <filename(s)>
```

Neste programa, no caso de o número de argumentos fornecidos (`argc`) ser menor que 3 o pedido não está corretamente enunciado. Esta é a primeira verificação e, neste caso, é solicitado ao utilizador reintrodução do pedido no formato correto.

No caso em que o número de argumentos fornecidos (`argc`) for maior ou igual a três, mas com o segundo argumento inválido (diferente de '*backup*', '*restore*' e '*delete*'), o utilizador recebe a informação que o comando é inválido e também o formato correto do pedido.

A comunicação do servidor para o cliente é feita através dos seguintes sinais:

30 - sucesso
10 - ficheiro não encontrado
6 - já existe backup do ficheiro

Assim, o cliente está preparado para receber os três sinais da seguinte forma:

```
signal(30,hand);
signal(10,hand);
signal(6,hand);
```

A função *hand* recebe o sinal e imprime a informação correspondente.

2.2.1 Pedido de backup

Para testar se o pedido do utilizador corresponde a um *backup* é comparada a string "backup" com o segundo argumento fornecido (argv[1]).

As verificações e processos descritos em seguida são aplicadas a cada um dos argumentos restantes (os ficheiros que se pretende copiar).

A verificação da existência do ficheiro é feita através da variável `int status` que guarda o resultado do `stat` do ficheiro.

```
status = stat (argv[i],&st_buf);
```

Se `status==0`, o utilizador é informado que o ficheiro não existe e, portanto, não poderá ser feito o *backup*.

Caso contrário, é feito um teste para concluir se trata-se de uma diretoria ou de um ficheiro:

```
if (S_ISDIR (st_buf.st_mode))
```

Assim, no caso de ser uma diretoria, o utilizador é avisado que terá de fornecer um ficheiro e não um diretório.

No caso do argumento ser um ficheiro, o pedido de backup é enviado ao servidor (nesta altura já se tem a garantia da existência do ficheiro em questão).

O pedido de *backup* pelo cliente ao servidor é realizado através do *pipe* sendo enviada uma string com o seguinte formato:

```
"dirFicheiro" B "myPID"
```

em que:

`dirFicheiro` – caminho (path) do ficheiro a ser guardado;

`B` – informa que o pedido é de *backup*;

`myPID` – pid do processo do cliente que faz o pedido;

Depois de escrita a linha no *pipe*, o processo entra em espera (`pause()`) até receber um sinal do servidor – sinal este que indica o estado da operação. Os sinais possíveis na situação de backup são :

`s==30` – foi feito o *backup* corretamente;

`s==6` – não foi realizado *backup* por já existir um ficheiro com o mesmo nome;

2.2.2 Pedido de restore

Para testar se o pedido do utilizador corresponde a um *restore* é comparada a string "restore" com o segundo argumento fornecido (argv[1]).

Uma vez que o cliente não tem acesso ao diretório do *backup*, cabe ao servidor testar a existência do ficheiro solicitado, então, é simplesmente escrita no *pipe* uma string com o seguinte formato:

```
"nomeFicheiro" R "myPID"
```

em que:

`nomeFicheiro` – nome do ficheiro a ser guardado;
`R` – informa que o pedido é de *restore*;
`myPID` – pid do processo do cliente que faz o pedido;

Depois de feita a escrita no *pipe*, o processo entra em espera (`pause()`) até receber um sinal do servidor – sinal este que indica o estado da operação. Os sinais possíveis na situação de restore são:

`s==30` – foi feito o *restore* corretamente;
`s==10` – não foi realizado *restore* por não existir o ficheiro solicitado;

2.2.3 Pedido de delete

A implementação do pedido de *delete* é em tudo idêntica ao pedido de *restore* sendo a informação enviada ao *servidor* pelo *pipe*:

```
"nomeFicheiro" D "myPID"
```

em que:

`nomeFicheiro` – nome do ficheiro a ser apagado;
`D` – informa que o pedido é de *delete*;
`myPID` – pid do processo do cliente que faz o pedido;

2.3. Servidor (sobusrv.c)

Neste programa são declaradas diversas variáveis necessárias para o armazenamento temporário e/ou comunicação de dados entre o processo principal e os seus processos-filho.

Como já referido, a informação que é enviada do cliente para o servidor tem o seguinte formato:

```
"dirFicheiro" X "myPID"
```

em que:

`dirFicheiro` – nome do ficheiro a ser guardado/restaurado
`X` – corresponde ao caracter 'B' (*backup*), 'R' (*restore*) ou 'D' (*delete*)
`myPID` – pid do pedido

Após inicializado o servidor, este entra num ciclo (`while (1))` de leitura do *pipe*.

De modo a que possam ser executadas operações em simultâneo (no máximo 5) de instâncias diferentes do programa cliente, foi criado um *array* (`int pidsFilhos[5]`) inicializado a zero. Este *array* irá guardar os *pid* dos processos-filho que estão a processar uma operação.

Assim, quando é lida uma linha do *pipe*, é feita uma atualização do `pidsFilhos` conforme o estado dos processos presentes no array. Isto é, o *array* é percorrido e para cada um dos *pid*, a partir da função `waitpid`, com a opção *WNOHANG*, conclui-se se o processo em questão já terminou e, caso afirmativo, é alterada essa posição do *array* para zero.

Em seguida é feita uma verificação se existem menos de 5 processos-filho ativos (procurar posições igual a zero no *array*). Caso o *array* contenha todas as suas posições ocupadas (diferente de zero), o programa aguarda que um dos processos-filho termine sendo libertada posteriormente a posição correspondente no *array* `pidsFilhos`.

Neste fase de execução existe a garantia que existe uma posição do *array* livre, ou seja, é possível agora, sem comprometer o limite de 5 processos concorrentes, criar um processo-filho que execute a operação pretendida.

É então criado um processo-filho sendo o seu *pid* armazenado na primeira posição livre do *array* `pidsFilhos`.

Após criado o processo-filho, existem dois contextos de execução:

- processo principal – não tem mais instruções até ao final do ciclo *while*, ou seja, retorna à leitura do *pipe*, repetindo o processo descrito, desde a entrada no ciclo `while(1)`, até que o utilizador termine o programa, emitindo o sinal de SIGINT (Ctrl+C).

- processo-filho – processa a linha lida do *pipe* pelo processo principal, executando as operações descritas em seguida.

Em primeiro lugar, esta linha é desmembrada e guardada nas seguintes variáveis:

`char dir` – dirFicheiro
`char op` – X ('B', 'R' ou 'D')
`int pid` – PID

Consoante o valor da variável `op`, é executada uma das operações descritas abaixo.

2.3.1 *Backup*

Este processo é iniciado no caso de `op=='B'`.

Em seguida é calculado o *digest* do ficheiro recorrendo ao programa *sha1sum* e, através do comando *cut*, é retirada apenas a informação necessária (*digest*). Este procedimento é feito através do comando:

```
sha1sum "dirFicheiro" |  cut -d ' ' -f1
```

Para testar se já existe um ficheiro com a mesma designação na diretoria *metadata/*, é executado o seguinte comando:

```
find "metadata" -name "nomeFicheiro"
```

Em caso positivo, é enviado o sinal “6” para o processo com o *pid* respetivo ao cliente que solicitou a operação. Este sinal corresponde à informação de que já existe um ficheiro com o mesmo nome.

```
kill(pid,6)
```

Uma vez enviado o sinal, o processo-filho termina a sua execução através da função `_exit(1).`

Em caso negativo, é feita a verificação se já existe *backup* do ficheiro no diretório *data*. Esta verificação é feita através da função:

```
temBackup (int digest)
```

Esta função recebe uma String com um digest e o seu retorno permite concluir se já existe algum backup do ficheiro em questão.

Caso ainda não exista, é feita a compressão do ficheiro através do programa *gzip* e o ficheiro *.gz* gerado é guardado na diretoria *data*, através do comando:

```
gzip -k -c "dirFicheiro" > /home/<user>/.backup/data/"digestFicheiro".gz
```

O ficheiro é guardado tendo como nome o *digest* gerado.

Em seguida é criada uma ligação para o ficheiro com o nome original do ficheiro e esta é guardada na diretoria *metadata*.

```
ln -s /home/<user>/.backup/data/"digestFicheiro".gz
    /home/<user>/.backup/metadata/"nomeFicheiro"
```

No caso de já existir um *backup* na diretoria data de um ficheiro com o mesmo *digest*, apenas é criado o ficheiro ligação para esse ficheiro.

Em seguida é enviado o sinal “30” para o processo com o *pid* respetivo ao cliente que solicitou a operação, sinal que corresponde à informação de que o backup foi executado com sucesso:

```
kill(pid,30)
```

Uma vez enviado o sinal, o processo-filho termina a sua execução através da função `_exit(1)`.

Caso não o pedido não seja de *backup*, terá de ser necessariamente de *restore* ou *delete*. Em ambos os casos terá de ser feita uma verificação se o ficheiro existe. Assim, é feita uma verificação se o nome do ficheiro existe na diretoria `metadata/` através do comando:

```
find /home/<user>/.backup/metadata -name "nomeFicheiro"
```

No caso em que o ficheiro não existir, não é possível fazer o *restore* ou *delete* e é enviado o sinal “10” para o processo com o *pid* respetivo ao cliente que solicitou a operação, sinal que corresponde à informação que o ficheiro pretendido não existe:

```
kill(pid,10)
```

Uma vez enviado o sinal, o processo-filho termina a sua execução através da função `_exit(1)`.

No caso em que o ficheiro existe é guardada na variável `char digest` o caminho para o ficheiro correspondente na pasta `data/`. Este processo é realizado através do comando:

```
ls -l /home/<user>/.backup/metadata | grep "nomeFicheiro"  |
                    cut -d ' ' -f11
```

Nesta altura é feita a verificação se se trata de um pedido de *restore* (op='R'). Em caso negativo, o pedido será de *delete*.

2.3.2 *Restore*

No caso em que a string lida tem como segundo parâmetro o caracter 'R', é então iniciado o processo de *restore*.

É realizada a descompressão do ficheiro para a diretoria do cliente através do comando:

```
gunzip < /home/<user>/.backup/metadata/"digestFicheiro" >
                    ./"nomeFicheiro
```

É enviado o sinal "30" para o processo com o *pid* respetivo ao cliente que solicitou a operação, sinal que corresponde à informação de que o restore foi executado com sucesso:

```
kill(pid,30)
```

Uma vez enviado o sinal, o processo-filho termina a sua execução através da função `_exit(1)`.

2.3.3 *Delete*

No caso em que a string lida tem como segundo parâmetro o caracter 'D', é então iniciado o processo de *delete*.

Neste caso a ligação na diretoria `.backup/metadata/` será removida e, caso seja a única ligação para o ficheiro com o nome do digest do ficheiro, a cópia de segurança também será apagada. Assim, é necessário calcular inicialmente quantas ligações existem para a cópia ("digest".gz) em `.backup/data/`. Este cálculo é feito através do comando:

```
ls -l /home/<user>/.backup/metadata | grep "digest".gz  | wc -l
```

No caso em que existe apenas uma ligação para a cópia de segurança, a cópia é removida através do comando:

```
rm /home/<user>/.backup/data/"digestFicheiro".gz
```

Em seguida, independentemente do resultado do teste anterior, procede-se à remoção da ligação na diretoria `.backup/metadata/` através do comando:

```
rm /home/<user>/.backup/metadata/"nome"
```

Em seguida é enviado o sinal “30” para o processo com o *pid* respetivo ao cliente que solicitou a operação, sinal que corresponde à informação de que o delete foi executado com sucesso:

```
kill(pid,30)
```

Uma vez enviado o sinal, o processo-filho termina a sua execução através da função `_exit(1)`.

2.4. Makefile

Apresenta-se em seguida o código da makefile:

```
SRCS = $(wildcard *.c)
CFLAGS= -Wall -pedantic
PROGS = $(patsubst %.c,%,$(SRCS))
all: $(PROGS)
%: %.c
	$(CC) $(CFLAGS)  -o $@ $<
```

2.5. Comentários

Tanto no servidor como no cliente está definida a função `readln`, desenvolvida nas aulas práticas da UC, para ler uma linha de um dado file descriptor.

Foi também criado um script – `clean.sh` - com a funcionalidade de limpar os diretórios de *backup* (remover todos os ficheiros em `data/` e `metadata/`).

Para não tornar a leitura exaustiva, não foram descritos pormenorizadamente os processos para a execução de comandos como cut, grep, find, entre outros. Abaixo apresentamos uma descrição pormenorizada para alguns deste processos:

Exemplo 1:

```
sha1sum "dirFicheiro" |  cut -d ' ' -f1
```

Para execução deste comando, o programa cria um filho, que por sua vez cria um filho. O segundo filho executa o programa sha1sum com o diretório do ficheiro redirecionando o seu output para um *pipe* utilizado entre este processo e o seu pai (primeiro filho do processo principal). O primeiro filho executa o comando `cut -d ' ' -f1` tendo previamente redirecionado o seu stdin para a saída do *pipe* e o seu stdout para a entrada de um *pipe* utilizado entre este processo e o seu pai (programa principal). Por fim, o programa principal lê da saída do *pipe*, guardando o conteúdo recebido numa variável digest que irá ser usada nas próximas operações.

Exemplo 2:

```
ls -l /home/<user>/.backup/metadata | grep "digest".gz  | wc -l
```

Para execução deste comando, o programa cria um filho, que por sua vez cria um filho, que por sua vez cria um filho. O terceiro filho executa o programa `ls -l` com a diretoria da pasta `metadata/`, previamente redirecionado o seu stdout para a entrada de um *pipe* utilizado entre este processo e o seu pai. O segundo filho executa o programa `grep` com o digest do ficheiro (calculado anteriormente) redirecionando o seu stdout para um *pipe* utilizado entre este processo e o seu pai (primeiro filho do processo principal) e redirecionando o stdin para a saída do *pipe* de comunicação com o seu filho. O primeiro filho executa o comando `wc -l` tendo previamente redirecionado o seu stdin para a saída do *pipe* e o seu stdout para a entrada de um *pipe* utilizado entre este processo e o seu pai (programa principal). Por fim, o programa principal lê da saída do *pipe*, convertendo o conteúdo recebido para um inteiro e guardando-o numa variável que irá ser usada para toma de decisões.

Exemplo 3:

```
gunzip < /home/<user>/.backup/metadata/"digestFicheiro".gz >
                    ./"nomeFicheiro
```

Para execução deste comando, o programa cria um filho.

O filho começa por abrir, apenas para leitura, o ficheiro que se pretende descomprimir, localizado em `/home/<user>/.backup/metadata/"digestFicheiro".gz`, e abrir, ou criar caso não exista, para escrita, o ficheiro `./"nomeFicheiro"`. Redireciona o seu stdin para o *file descriptor* do primeiro ficheiro e o seu *stdout* para o *file descriptor* do segundo ficheiro. Por fim o filho executa o programa `gunzip`:

```
execlp("gunzip", "gunzip", NULL)
```

2.6. Resultados

Uma das características desejáveis do sistema desenvolvido era a eficiência respeitante ao espaço dedicado para o *backup*. Este ponto foi cumprido pois, antes de os ficheiros serem salvaguardados, estes são comprimidos com recurso ao programa *gzip*. Não menos importante, quando o utilizador solicita o *backup* de um ficheiro que já tenha sido alvo de *backup*, mas com um nome diferente, é criada uma ligação para o *backup* já existente, não havendo por isso duplicação de dados.

Cada vez que um ficheiro é apagado, é executada uma remoção dessa informação do *backup*, ou seja, o sistema elimina da diretoria `metadata/.` a ligação desse ficheiro e faz-se a verificação se existem outras ligações ativas para a cópia correspondente. Caso haja outras ligações a cópia é mantida em `data/.`, caso contrário é eliminada. Deste modo nunca existem ligações sem cópias associadas ou vice-versa.

3. Conclusão

Como conclusão salientamos os seguintes aspetos:

- o sistema funciona como previsto apresentando funcionalidades de backup, restore e delete de ficheiros;
- toda a informação é guardada no diretório .backup/;
- a privacidade é garantida pois o cliente não tem acesso direto à informação salvaguardada;
- o cliente escreve no *pipe* para fazer pedidos ao servidor. O servidor comunica com o cliente através do uso de sinais;
- a implementação é eficiente (ao nível do espaço ocupado em disco) pois não existem cópias duplicadas nem cópias que não tenham ligações na diretoria `metadata/`.;
- devido ao modo como foi implementado o comando delete, descrito em Resultados, não faz sentido a implementação de um comando `gc`;
- o programa criado não permite operações sobre pastas. No entanto, quando é fornecido um diretório como argumento, esse caso é distinguido sendo o utilizador informado que apenas é permitida a operação sobre ficheiros.

Pelo referido, consideramos que no geral os objetivos do trabalho foram cumpridos.