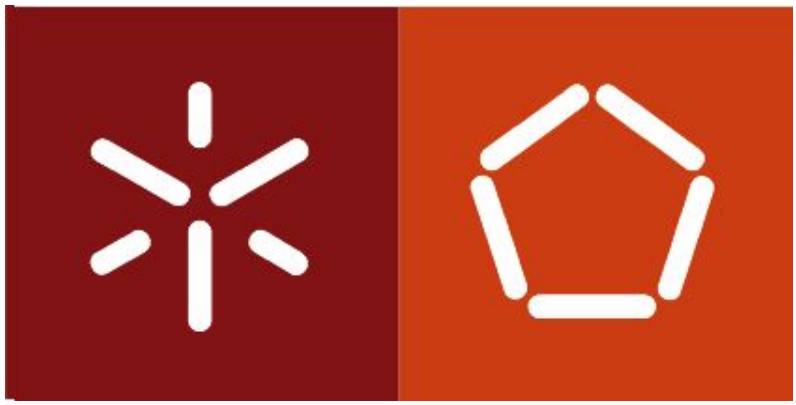


Universidade do Minho

Departamento de Informática

Mestrado Integrado em Engenharia Informática

Unidade Curricular de Sistemas Operativos

2015/2016

Backup Eficiente

André Geraldes a67673
Bruno Barbosa a67646
Tiago Cunha a67707

Resumo

Neste relatório são descritas as etapas mais relevantes do processo de desenvolvimento de um sistema eficiente de cópias de segurança. A eficiência passa pelo facto de os ficheiros originais serem comprimidos e guardados numa diretoria especifica apenas uma vez. Por exemplo, para dois ficheiros com o mesmo conteúdo existe uma ligação simbólica para o mesmo ficheiro comprimido. Para além das funcionalidades do sistema, serão também explicados os raciocínios por detrás da implementação bem como os vários métodos usados. O sistema é composto por dois componentes essenciais que correspondem ao servidor e cliente. O cliente tem como função enviar apenas comandos para o servidor. O servidor, por sua vez, fica continuamente a receber os comandos vindos do cliente, analisando-os e executando o respetivo pedido de acordo com o tipo do mesmo. O servidor tem a capacidade realizar 4 operações: cópia de segurança (*backup*), restauro de um ficheiro (*restore*), eliminar um ficheiro da diretoria de raiz de segurança e remover os ficheiros de segurança que não têm quaisquer referências para os mesmos (*gc*).

1. Desenvolvimento

Nesta secção vai-se explicar mais sucintamente cada pormenor do trabalho desenvolvido. Inicia-se com a estruturação da arquitetura usada, depois seguem-se especificações do cliente, servidor e das várias operações do sistema.

1.1. Arquitetura do Sistema

A Figura 1 ilustra a arquitetura do sistema. Basicamente, o sistema pode ser constituído por vários clientes e um servidor. Existe um *pipe* com nome (*FIFO*) que é criado pelo servidor e que é acedido pelos clientes de forma a enviarem os comandos do utilizador. O servidor comunica com os clientes através de sinais, indicando apenas o valor de retorno da operação: sucesso ou insucesso. Ou seja, os clientes escrevem no *FIFO* e o servidor lê o mesmo, criando um processo filho para executar cada pedido. Contudo, existe uma restrição que impede que hajam mais do que cinco pedidos a executar em paralelo, pelo que, caso isso aconteça, o servidor fica em modo de espera até que morra um desses descendentes. Em última instância, imediatamente após o cliente escreve o seu pedido no *FIFO*, este fica à espera da conclusão da tarefa no servidor. Isto é, permanece adormecido até à recepção do sinal de confirmação que indica se a operação foi, ou não, bem sucedida. O sinal *SIGUSR1* é utilizado para comunicar que a operação foi bem e o sinal *SIGUSR2* para o caso contrário.

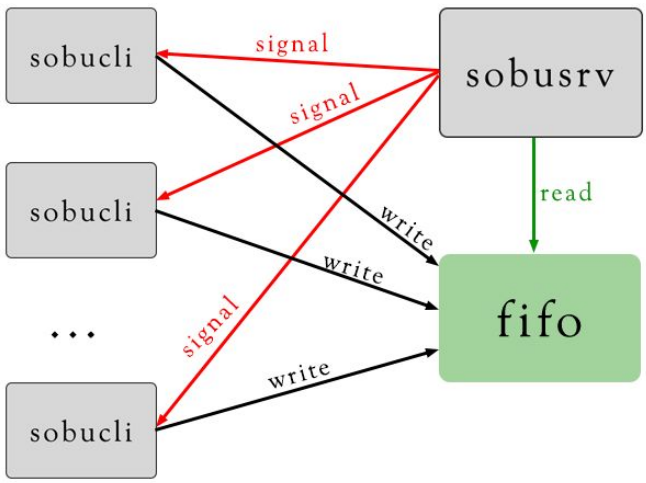


Figura 1. Arquitetura do Sistema

1.2. Cliente

Como já foi referido anteriormente, o programa do cliente trata de escrever no *FIFO* os comandos que serão usados no servidor. Todavia, antes disso, o cliente faz um conjunto de verificações de forma garantir que o comando está bem escrito respeitando devidamente a sintaxe para que depois seja mais fácil o seu tratamento no lado do servidor. Sendo assim, as verificações que o cliente realiza são as que se seguem e segundo a seguinte ordem:

1. Confirmar que o número de argumentos é maior ou igual a 2. No mínimo tem-se o nome do executável (*sobucli*) e a operação ([*backup* | *restore* | *delete* | *gc*]).
2. Tentar abrir o *FIFO* em modo de escrita. Uma vez que o *FIFO* é criado no servidor, caso este não tenha sido executado pelo menos uma vez, será apresentado um erro a avisar que não é possível abrir o *pipe*.
3. Verificar se a operação corresponde a [*backup* | *restore* | *delete* | *gc*].

Depois de realizadas todas estas verificações, o comando a enviar para o *FIFO* é concatenado numa única linha com um formato específico (*%d\t%s*) onde o inteiro corresponde ao *PID* do processo do cliente e a *string* à operação juntamente com os respetivos argumentos, ambos separados por espaços. Para terminar, o cliente fica adormecido até o servidor concluir a sua tarefa e enviar os signais de confirmação. O número de sinais a receber é igual ao número de argumentos do comando.

Na secção 1.3, em conjunto com a explicação das implementações das várias operações, vão ser dados alguns exemplos de como é realizado um pedido por parte do cliente.

1.3. Servidor

O servidor tem como primeira funcionalidade a criação do *pipe* com nome (*FIFO*) com as permissões 0666 na diretoria */home/user/.Backup*. Note que, de modo a saber a primeira parte do *path*, foi usada uma função que permite saber o valor de variáveis de ambiente, neste caso a variável *HOME*. O próximo passo é colocar o servidor a ler infinitamente do *FIFO*. Sempre que é lida uma linha do *FIFO* esta é dividida segundo o formato especificado na secção do cliente. Deste modo, conseguimos obter de imediato o *PID* do processo e o comando que lhe corresponde. Neste momento é criado um processo filho que se encarrega de processar o pedido. Atenção, que é nesta fase que faz a verificação de quanto processos filhos (ou pedidos) estão a correr no momento. Entenda-se que cada processo filho na *main* como sendo a execução de uma operação. Segundo o enunciado, o número de operações concorrentes está limitado a 5 com o intuito de não sobrecarregar o sistema. A seguir desta triagem vem a parte onde se distingue o tipo de operação, ou seja, em *backup*, *restore*, *delete* ou *gc*. Todas estas operações foram previamente implementadas através de *scripts* auxiliares, contudo, visto que não era esse o objectivo do trabalho, a solução foi traduzir as mesmas para código, recorrendo aos vários recursos disponíveis. Para cada operação foi criada uma função auxiliar que retorna 0 e 1 em caso de sucesso e insucesso, respetivamente. A partir deste valor, o servidor encarrega-se de enviar o sinal *SIGUSR1* ou *SIGUSR2* para o *PID* a que lhe diz respeito. Nos casos em que as operações são feitas ficheiro a ficheiro, é enviado um sinal por cada ficheiro concluído.

1.3.1. Backup

A primeira operação a implementar foi o *backup*. A Figura 2 corresponde à *script* que serviu de guião para a realização da mesma. Esta operação pode ser chamada pelo cliente das seguintes maneiras (vários tipos de exemplos):

./sobucli backup a.txt **ou** *./sobucli backup a.txt b.txt* **ou** *./sobucli backup *.txt*

Como podemos ver na Figura 2, o primeiro passo é criar o ficheiro comprimido através do comando *gzip -fk* que mantem o ficheiro original (não comprimido) e reescreve o ficheiro comprimido caso este já exista. Para tal, é criado um processo filho que se encarrega de executar este comando. De

seguida, simula-se a execução de um pipe de forma a guardar numa variável o resultado do comando *sha1sum*, que vai servir para renomear o ficheiro comprimido. Na simulação do pipe usam-se as habituais *system calls* para efetuar o redirecionamento do *STDIN* e do *STDOUT* dos vários comandos entre os vários processos, tais como fork (para executar os vários comandos), *pipe* (para permitir a comunicação entre processos) e *dup2* (para duplicar os descritores de ficheiros de modo a tornar o *STDOUT* de um processo no *STDIN* de outro). Com mais um processo filho, o ficheiro comprimido é movido para diretoria */data* já com o novo nome. De novo, recorre-se a um mais um processo filho para terminar esta operação criando uma ligação simbólica desde a diretoria */metadata* para o ficheiro comprimido na diretoria */data*.

```
1  #!/bin/bash
2
3  DIR=/home/$USER/.Backup
4  FILE=$1
5
6  gzip -f -k $FILE
7  fsha=$(sha1sum $FILE | awk '{print $1}')
8  mv $FILE.gz $DIR/data/$fsha
9  cd $DIR/metadata
10 ln -s -f ../data/$fsha $FILE
11
```

Figura 2. Script para executar a operação de backup.

1.3.2. Restore

A operação *restore* e *backup* são as funcionalidades básicas deste sistema. A Figura 3 ilustra a forma como estruturamos a sua implementação. Identicamente ao *backup* (neste caso existe uma pequena diferença pois o ficheiro que queremos restaurar tem que existir na diretoria */metadata*), a operação *restore* pode ser executada chamando-a da seguinte forma por parte do cliente:

./sobucli restore a.txt **ou** *./sobucli restore a.txt b.txt*

Portanto, esta operação pode resumir-se, em primeira instância, à obtenção da ligação para o ficheiro comprimido a partir da diretoria */metadata* através da execução de um pipe tal como é apresentado na linha 8 da Figura 2. A implementação do encadeamento de processos é semelhante à utilizada para o *backup*, no entanto, desta vez temos dois *pipes*, intercalando pelo meio a execução do comando *grep -w* para procurar exatamente pelo ficheiro passado pelo cliente. Assim consegue-se obter a ligação para o ficheiro comprimido. No sentido de concluir esta operação, é executado o comando *gunzip* que através de redirecionamento, recebe como *input* o *path* para o ficheiro comprimido a partir da diretoria */metadata* e que, por sua vez, redireciona o *output* para o ficheiro na diretoria local. De notar que caso o ficheiro a ser restaurado não esteja na diretoria local, ele acaba por ser criado. Em termos de código, esta última operação requer o uso da *flag -c* quando executamos o comando *gunzip*, bem como a abertura de um descritor para o ficheiro na diretoria local em modo de criação e escrita. Portanto, o *STDOUT* fica a "apontar" para o ficheiro a ser restaurado.

```
1  #!/bin/bash
2
3  LOCAL=$PWD
4  DIR=/home/$USER/.Backup
5  FILE=$1
6
7  cd $DIR/metadata
8  fn=$(ls -l | grep -w $FILE | awk '{print $11}')
9  gunzip < $fn > $LOCAL/$FILE
10
```

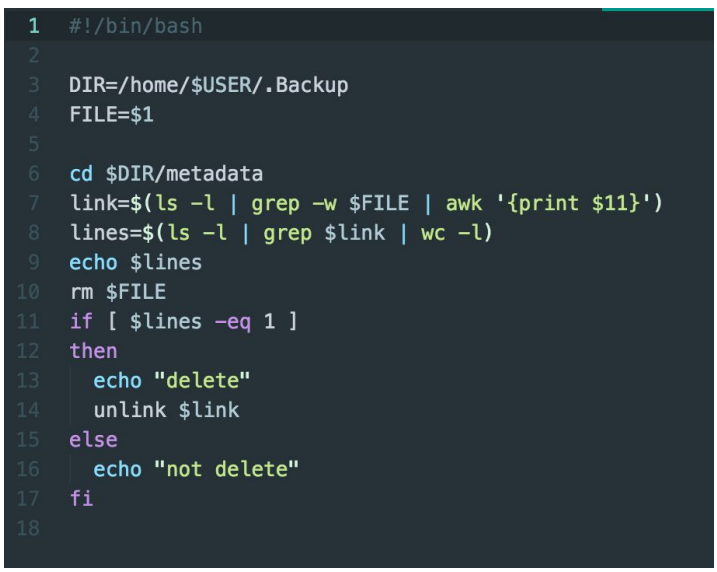
Figura 3. Script para executar a operação de restore.

1.3.3. Delete

Comparativamente às implementações das operações anteriores, a operação *delete* introduz um pouco mais de complexidade na medida em que são utilizados dois encadeamentos de processos, tal como é demonstrado na Figura 4. O cliente pode pedir para efetuar uma operação *delete* através da seguinte maneira:

./sobucli delete a.txt **ou** *./sobucli delete a.txt b.txt*

Não é que os encadeamentos se tratem de uma novidade, porque até então, já foram utilizados outros encadeamentos idênticos. Neste caso a implementação passou, infelizmente, pela replicação de código já usado mas com as devidas alterações e com a garantia de estar a funcionar corretamente. Traduzindo a *script*, a partir da diretoria */metadata* guarda-se uma variável com resultado do encadeamento da linha 7, ou seja, a ligação para */data* do ficheiro que queremos remover. De seguida, no segundo encadeamento faz-se uma procura dessa mesma ligação na diretoria atual, com a intenção de saber quantas vezes ela aparece. Nesse sentido, o *output* deste corresponde ao número de ocorrências da ligação, pelo que se for apenas 1, remove-se a ligação e o ficheiro comprimido na diretoria */data*, caso contrário, significa que existe mais do que uma ligação para o respetivo ficheiro comprimido, logo este não deverá ser removido.



```
1  #!/bin/bash
2
3  DIR=/home/$USER/.Backup
4  FILE=$1
5
6  cd $DIR/metadata
7  link=$(ls -l | grep -w $FILE | awk '{print $11}')
8  lines=$(ls -l | grep $link | wc -l)
9  echo $lines
10 rm $FILE
11 if [ $lines -eq 1 ]
12 then
13   echo "delete"
14   unlink $link
15 else
16   echo "not delete"
17 fi
18
```

Figura 4. Script para executar a operação de delete.

1.3.4. GC

A quarta e última operação implementada foi a *gc*. Esta operação tem como função eliminar todos os ficheiros na diretoria *data/* que não estejam a ser usados por nenhuma das entradas em *metadata/*. Para execução deste método, o cliente pode fazer a sua chamada através de:

./sobucli gc

O desenrolar desta operação inicia com a seleção de todos os ficheiros da diretoria */data*. Depois, um a um, verifica-se se existe alguma ligação na diretoria */metadata* para o mesmo, contabilizando o número de ocorrências. Caso não existam correspondências então remove-se o ficheiro comprimido. A nível de código, a linha 6 da Figura 5 é equivalente a ter um *pipe* anónimo onde se reencaminha o resultado do comando *ls* na diretoria */data* para o comando *xargs*. Desta forma, ficamos com todos os ficheiros da diretoria */data* numa única linha separados por espaços, o que simplifica posteriormente a procura desses mesmos ficheiros na diretoria */metadata* através de um ciclo sobre os vários *tokens* (ficheiros) da linha. Exatamente como na *script*, se o número de correspondências for 0 então remove-se o respetivo ficheiro uma vez que não existe nada que dependa dele.

```
1  #!/bin/bash
2
3  DIR=/home/$USER/.Backup
4
5  cd $DIR/data
6  for file in *
7  do
8    lines=$(ls -l ../metadata | grep -w $file | wc -l)
9    if [ $lines -eq 0 ]
10   then
11     rm $file
12   fi
13 done
14
```

Figura 5. Script para executar a operação de gc.

2. Considerações Finais

Com a realização deste trabalho prático foi possível, antes de mais, colocar em prática todo o conhecimento adquirido nesta unidade curricular de Sistemas Operativos. Desde a acessos a ficheiro, gestão de processos, execução de programas, passando por redirecionamentos, pipes anónimos e com nome até aos sinais. Todos esses conceitos foram necessários para a concepção deste trabalho.

Relativamente ao trabalho desenvolvido existem algumas funcionalidades propostas no enunciado, mais concretamente na parte de Valorização, que não foram implementadas. Por exemplo, a possibilidade de indicar uma diretoria base para cópia ou restauro, em vez de ficheiros isolados. Outro exemplo, foi o de experimentar ter vários utilizadores a aceder em simultâneo por controlo remoto e testar o caso sobrecarga do sistema.

Contudo, apesar destas funcionalidades não terem sido realmente implementadas, foram estudadas e testadas possíveis abordagens para as mesmas. Na situação de comprimir uma diretoria a ideia seria utilizar o seguinte comando *tar -zcvf folder.tar.gz folder/*, alterando o nome do ficheiro comprimido para o resultado da execução do comando *sha1sum* sobre si mesmo. O restante procedimento seria tal e qual como se fosse para um ficheiro individual. Em relação ao facto de simular a concorrência, os testes realizados consistiram em reduzir o número máximo de operações em simultâneo a 1. Dado que o envio de sinais do servidor para o cliente tinham um atraso de 1 segundo (para garantir que todos os sinais chegavam ao cliente), foi possível comprovar o controlo de sobrecarga, por exemplo, com um processo (num terminal) que fazia *backup* de 7 ficheiros e outro processo (outro terminal) que fazia *backup* apenas de 1 ficheiro. Enquanto que o programa fazia o *backup* dos tais 7 ficheiros, o outro processo ficava em modo de espera até que o atual terminasse e só depois executava. A Figura 6 mostra um excerto de código da *main* onde é feito esse controlo.

```
507  while(1) {
508    fd = open(fifo,O_RDONLY);
509    while((r = read(fd,&buffer,SIZE)) > 0) {
510      write(1,&buffer,r);
511      sscanf(buffer,"%d\t%[^\n]s",&pid,command); // PID and command are OK
512      // server handles at maximun 5 request simultaneously
513      if(running < FORKS) {
514        running++;
515        if(fork() == 0) {
516          // creates a child process to process request
517          processRequest(pid,command);
518          _exit(1);
519        } else if(running >= FORKS) {
520          wait(0);
521          running--;
522        }
523      }
```

Figura 6. Controlo de operações em paralelo

3. Instalação

Juntamente com o código foi desenvolvida uma *Makefile* com várias opções que permite a instalação e/ou configuração do sistema. Neste sentido, não foi criada nenhuma *script* de instalação. Na primeira instalação basta fazer *make all* no terminal que são imediatamente criadas as diretorias de *backup* e os executáveis.