Sistemas Operativos 2020-21
2º Guião Laboratorial
LEIC-A / LEIC-T / LETI
IST

Este guião pretende ajudar os alunos a ambientarem-se com o código base do projeto da disciplina. Antes de seguirem este guião, os alunos devem ler o enunciado geral do projeto, que está disponível na página da disciplina (no Fénix), na secção "Laboratórios".

Este guião pretende também guiar os alunos no contacto com a ferramenta **`gdb`** (*GNU debugger*), que possui uma grande importância no contexto do desenvolvimento de aplicações em ambiente UNIX. Serão apresentados vários exemplos de utilização. Será também revista a ferramenta **`valgrind`**.

Assume-se que os alunos já completaram o guião anterior.

Este exercício não será avaliado.

1. Análise do programa fornecido

Crie um diretório no seu computador e descarregue o arquivo **`lab2.zip`** que está disponível na página da disciplina (no fénix), na secção "Laboratórios". Para extrair os ficheiros contidos no arquivo, use o comando

```
unzip lab2.zip
```

Analise os ficheiros extraídos.[1] O arquivo contém uma diretoria **`fs`** com os ficheiros **`state.c`**, **`state.h`**, **`operations.c`** e **`operations.h`**. Estes constituem a base de implementação de um sistema de ficheiros (*file system*) simplificado, em modo utilizador. Os ficheiros **`state.c`** e **`state.h`** gerem *i-nodes*, que são estruturas de dados que podem representar um ficheiro ou uma diretoria (incluem atributos do ficheiro/diretoria e um apontador para o conteúdo do ficheiro/diretoria). Os ficheiros **`operations.c`** e **`operations.h`** implementam as operações[2] do sistema de ficheiros *tecnicofs*, permitindo criar, procurar, apagar e listar os ficheiros e as diretorias.

O arquivo contém também o programa **`main.c`**, que permite verificar o funcionamento do sistema de ficheiros. O programa reconhece os seguintes comandos: **c** ***name type*** (*create* **name** of *type* **f**=file or **d**=directory), **l** ***name*** (*lookup* **name**), **d** ***name*** (*delete* **name**). O formato dos comandos pode também ser consultado no enunciado geral. Quando o programa recebe um **ctrl+D** (*end of file*), termina e imprime o conteúdo da árvore de diretorias.

[1] Mas não compile e execute já o projeto! Como verá na secção seguinte, o código fornecido sofre de um *bug* e termina abruptamente.
[2] Na versão inicial do código base fornecido, apenas um subconjunto reduzido de operações é oferecido.

1. Identifique as diferentes operações disponibilizadas em **`operations.c`** e em **`state.c`**.

2. Analise o programa **`main.c`** e verifique a sintaxe dos vários comandos e como eles são executados.

3. Assuma a seguinte sequência de comandos:

   ```
   c /a d
   c /a/b f
   c /a/x/ d
   c /a/x/y f
   ```

   Esboce, em papel, o estado que o sistema de ficheiros deverá ter após executar estes comandos. Se tiver dúvidas, consulte o código dos ficheiros **`state.c`** e **`operations.c`**.

2. Utilização da ferramenta de depuração `gdb`

O **`gdb`** permite analisar o que está a acontecer dentro de um programa enquanto este está em execução ou o estado de um programa antes de este terminar abruptamente. A documentação completa da ferramenta de depuração **`gdb`** pode ser consultada em: http://www.gnu.org/software/gdb/documentation .

Para demonstrar as capacidades do **`gdb`** iremos usá-lo para identificar um *bug* existente no programa analisado no ponto anterior.

1. Analise a *Makefile* fornecida, que permite gerar o programa **`tecnicofs`**, e note o uso da flag **`-g`** na compilação. O uso desta flag é necessário para que o executável inclua a devida informação simbólica (nomes de variáveis, funções, etc), para facilitar o uso do **`gdb`**.

2. Gere o programa **`tecnicofs`** fazendo

   ```
   make
   ```

3. Execute o programa **`tecnicofs`** e introduza o comando indicado abaixo, que deverá criar o ficheiro */file1*.

   ```
   ./tecnicofs
   c /file1 f
   ```

4. O que sucedeu?

   Por vezes pode ser fácil identificar o problema que gerou o *segmentation fault*. Mas, geralmente, não é isso que se verifica. Nestes casos, o **`gdb`** é especialmente útil pois pode ser utilizado para analisar o programa após este terminar, como se fosse uma autópsia. Para ilustrar esta capacidade,

proceda do seguinte modo:

```
gdb ./tecnicofs core
```

NOTA IMPORTANTE: *Verifique previamente a existência do ficheiro* **`core`** *na directoria actual (use o comando* **`ls`***). Esse ficheiro é gerado quando o programa termina de modo anormal.*
Se o ficheiro **`core`** *não existir, utilize o comando* **`ulimit -c`** *para consultar o tamanho máximo permitido para os ficheiros* `core`*. Caso seja 0, para permitir que sejam gerados ficheiros com dimensão até 10MB, introduza:*

```
ulimit -c 10000000
```

Após este comando, volte a executar o programa tecnicofs e os comandos indicados no ponto 4, para gerar novo erro e gerar o ficheiro `core`*.*
Se ainda não existir nenhum ficheiro `core`*, é possível que a gestão desses ficheiros esteja a ser tratada por um programa chamado* `systemd`*. Pode lançar o depurador sobre o último* `core` *gerado com:*

```
coredumpctl gdb
```

---

5. Lançado o **`gdb`**, deve observar algo parecido com o indicado abaixo.

   Note que os endereços variam consideravelmente de máquina para máquina, por isso os endereços mostrados neste guião são apenas um exemplo.

```
 . . .
 . . .
Core was generated by `./tecnicofs'.
Program terminated with signal SIGSEGV, Segmentation fault.
#0  dir_add_entry (inumber=0, sub_inumber=1, sub_name=0x7ffd686e1710 "/file1")
    at fs/state.c:174
174          if (inode_table[inumber].data.dirEntries[i].inumber == FREE_INODE)
{
(gdb)
```

   Notar que a informação acima já dá pistas muito valiosas sobre onde o problema ocorreu (função `dir_add_entry`, que está no ficheiro `fs/state.c`, na linha 174), embora a causa possa estar noutro local.

6. Para saber qual o caminho percorrido pelo programa até chegar a esse ponto, use o comando **`backtrace`** (abreviado **`bt`**), o qual irá mostrar informação semelhante à seguinte.

```
(gdb) bt
#0  dir_add_entry (inumber=0, sub_inumber=1, sub_name=0x7ffd686e1710 "/file1")
    at fs/state.c:174
#1       0x0000563aa5e67d4a    in    create    (name=0x7ffd686e1810    "/file1",
nodeType=T_FILE)
    at fs/operations.c:116
```

```
#2  0x0000563aa5e68123 in applyCommands () at main.c:10
#3  0x0000563aa5e68327 in main (argc=1, argv=0x7ffd686e19f8) at main.c:76
(gdb)
```

O primeiro número em cada linha indica o nível em que essa função está, começando pela função onde o programa terminou. A função `dir_add_entry`, foi chamada pela função `create` existente em `fs/operations.c`, a qual foi chamada por `applyCommands` e assim por diante.

Observe como o **`gdb`** mostra os argumentos passados às funções. Por exemplo, a função `create` recebeu como argumentos `name` (que é um apontador e tem o endereço `0x7ffd686e1810`, onde se encontra a *string* `"/file1"`) e `nodeType=T_FILE`.

Nota: É frequente serem mostradas funções que são de sistema (embora não seja este o caso). Quando isso se verifica, obviamente, o que interessa é a última função que correu do nosso programa, pois será aí que está o erro e não nas funções de sistema (provavelmente as funções de sistema foram chamadas com valores inválidos, ponteiros a NULL ou ponteiros para os quais não foi alocada memória, ou algo similar).

7. Pode ver o código onde ocorreu o problema usando o comando **`list`** (abreviado **`l`**):

```
(gdb) l
```

Como pode verificar, na linha 174 é feito um acesso a:

```
inode_table[inumber].data.dirEntries[i].inumber
```

8. Pode ver o conteúdo das várias variáveis, mas antes necessita ir para esse contexto (*frame*), o que pode fazer usando o comando **`frame`** e indicando que é o contexto 0:

```
(gdb) frame 0
#0  dir_add_entry (inumber=0, sub_inumber=1, sub_name=0x7ffd686e1710 "/file1")
    at fs/state.c:174
174         if (inode_table[inumber].data.dirEntries[i].inumber == FREE_INODE)
{
(gdb)
```

Nota: Para examinar o contexto de chamada de uma dada função, anterior àquela em que o problema ocorreu, pode subir na pilha de chamadas (*call stack*), usando o comando **`up`**, ou ir directamente para o contexto desejado com o comando **`frame`**. Por exemplo, se quiser examinar a chamada de `dir_add_entry` pode usar `frame 1`.

9. Após o comando **`frame 0`** pode ver o conteúdo das variáveis existente nesse contexto usando o comando **`print`** (abreviado **`p`**). Notar que pode indicar qualquer tipo de variável, incluindo apontadores, conteúdo de apontadores, elementos de um array, estruturas, etc.

```
(gdb) p inumber
$1 = 0

(gdb) p inode_table[inumber]
$2 = {nodeType = T_DIRECTORY, data = {fileContents = 0x0, dirEntries = 0x0}}

(gdb) p inode_table[inumber].data.dirEntries
$3 = (DirEntry *) 0x0

(gdb)
```

O conteúdo de `inode_table[inumber]` dá pistas sobre o problema, pois indica que o apontador `data` tem o valor `0x0` (*NULL*), ou seja, aparenta que não foi alocada memória para ele. Obviamente, irá ocorrer um erro ao tentar aceder a `inode_table[inumber].data.dirEntries` pois seria esperado ter um apontador para uma zona de memória que conteria um array de estruturas `dirEntry` e afinal ele contém *NULL* (logo não aponta para nenhuma zona de memória válida).

10. Em síntese, pode-se concluir que `inode_table[0]` não foi devidamente inicializada, devendo então procurar-se a causa dessa falha ou erro. Para tal, iremos passar ao contexto anterior/acima (função `create`) e analisar o que se passou antes de ser chamada a função `dir_add_entry`. Execute e analise os comandos indicados a seguir:

```
(gdb) frame 1
#1       0x0000563aa5e67d4a    in    create    (name=0x7ffd686e1810    "/file1",
nodeType=T_FILE)
    at fs/operations.c:157
157    if (dir_add_entry(parent_inumber, child_inumber, child_name) == FAIL) {

(gdb) list
[…]
149      /* create node and add entry to folder that contains new node */
150      int child_inumber = inode_create(nodeType);
[…]
157      if (dir_add_entry(parent_inumber, child_inumber, child_name) == FAIL)
{
[…]
```

Como se observa, antes `dir_add_entry` foi chamada a função `inode_create`, cujo nome sugere que seja a responsável pela alocação de memória para o *i-node*, já que o cria. Para prosseguir sugere-se agora seguir outra estratégia: correr o programa sob o controlo do **`gdb`** e analisar o que ocorre nessa função. Para tal, na próxima secção, iremos aprender a colocar pontos de paragem (*breakpoints*), para rapidamente chegar a um dado ponto do código, e a correr o

programa instrução a instrução ou a executar uma função completa.

Saia da sessão actual do **gdb** usando o comando com **quit** (abreviado **q**) ou premindo **Ctrl-D**.

```
(gdb) q
```

3. Execução de um programa sob o controlo do gdb

1. Corra o programa **tecnicofs** dentro do **gdb**:

```
gdb ./tecnicofs
```

2. Utilize o comando **break** (abreviado **b**) para colocar um *breakpoint* na instrução localizada na linha 29 do ficheiro **main.c**:

```
(gdb) b main.c:29
```

3. Também pode colocar um *breakpoint* na primeira instrução de uma função, bastando indicar o seu nome:

```
(gdb) b create
```

4. Pode visualizar os *breakpoints* que foram definidos usando o comando:

```
(gdb) info b
```

   Notar que um *breakpoint* pode ser *disabled*, *enabled* ou apagado usando respectivamente os comandos **disable n**, **enable n** e **delete n**, em que **n** representa o número do *breakpoint* indicado pelo comando **info b**.

5. Execute o programa usando o comando **run** (abreviado **r**):

```
(gdb) r
```

6. A aplicação inicia a sua execução, ficando a aguardar *input*. Introduza o seguinte comando:

```
c /file1 f
```

7. Quando o programa chega a um *breakpoint*, é interrompido pelo **gdb**, aparecendo no ecrã a linha de código onde o programa parou. Pode ver em mais detalhe o código onde se encontra utilizando o

comando **`list`** (abreviado **`l`**):

```
(gdb) l
```

O programa parou na linha 29, que chama a função **`printf`**.

NOTA: A linha onde o **`gdb`** pára, que é mostrada no ecrã, **ainda não foi executada**!

8. Podem-se visualizar variáveis com o comando **`print`** (abreviado **`p`**) e avançar pelo código, linha a linha, utilizando o comando **`next`** (abreviado **`n`**), ou executando passo a passo com o comando **`step`** (abreviado **`s`**). Notar também os comandos **`until`** e **`advance`**.

   Note que não pode observar o valor de variáveis que ainda não foram definidas, tendo de esperar até estar na linha seguinte à da definição para poder inspecionar o seu valor. Também não pode observar variáveis declaradas num contexto diferente daquele em que se encontra.

```
(gdb) p type                ← mostra valor da variável type
(gdb) n                     ← executa a próxima linha (printf)
(gdb) s                     ← Step; entra na função create
(gdb) l                     ← list code
(gdb) p name                ← mostra valor do parâmetro name
(gdb) n                     ← next
(gdb) n                     ← next
(gdb) until 135             ← executa até à linha 135
(gdb) n                     ← next; executa inode_get
(gdb) p pdata               ← pdata = 0x0; problema ocorreu antes
(gdb) p inode_table[0]      ← confirma-se que data já é 0x0
                      Há que procurar quando inode_table[0] é criado
                      Aqui o problema já ocorreu.
(gdb) kill                  ← mata programa; responder y

(gdb) b init_fs             ← breakpoint na função init_fs
(gdb) r                     ← run; e vai parar em init_fs
(gdb) s                     ← entra em inode_table_init
(gdb) n                     ← next
(gdb) n                     ← next
 . . .                      ← está num ciclo...
(gdb) finish                ← conclui função actual; evita
                                        repetir ciclo até ao fim
(gdb) s                     ← step; entra em inode_create
(gdb) n                     ← next
```

```
(gdb) n                        ← next
(gdb) n                        ← next
(gdb) n                        ← next; vai alocar memória
(gdb) n                        ← next; atribui memória ao inode raiz (0)
(gdb) p inode_table[0]         ← ver que data aponta para memória
                                     Até aqui, tudo bem!
(gdb) n                        ← next; vai executar um ciclo
(gdb) list                     ← ver código para ver onde termina ciclo
(gdb) until 54                 ← fim do ciclo; próxima instrução é:
                        inode_table[inumber].data.fileContents = NULL;
                 o que não pode estar correcto!!! Acima foi atribuída
                 memória ao inode e agora atribui-se NULL ???

                 Analisar código e corrigir problema!

(gdb) q                        ← quit; sai do gdb
```

Nota: Caso se queira continuar com a execução do programa até ao próximo *breakpoint* usar o comando **`continue`** (abreviado **`c`**).

4. Utilização da ferramenta de verificação `valgrind`

A ferramenta **`valgrind`** permite detectar fugas de memória (*memory leaks*) e outras incorrecções no código. A par do **`gdb`**, é uma ferramenta fundamental para identificar eventuais problemas existentes nos nossos programas.

Para utilizar o **`valgrind`** é necessário usar a *flag* **`-g`** quando compila o código com o **`gcc`**, à semelhança do que se verifica quando se pretende usar o **`gdb`**.

1. Gere de novo o programa **`tecnicofs`** e use a ferramenta **`valgrind`** para correr o programa.

```
make
valgrind --tool=memcheck --leak-check=yes ./tecnicofs
```

Experimente submeter alguns comandos válidos e, no final, **ctrl+D** para terminar o programa ordeiramente. Quando o programa termina, a ferramenta **`valgrind`** mostrará informação semelhante à seguinte:

```
==6091==
==6091== HEAP SUMMARY:
```

```
==6091==     in use at exit: 0 bytes in 0 blocks
==6091==   total heap usage: 5 allocs, 5 frees, 11,360 bytes allocated
==6091==
==6091== All heap blocks were freed -- no leaks are possible
==6091==
==6091== For counts of detected and suppressed errors, rerun with: -v
==6091== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Neste caso, está tudo bem. Mas poderá não ser esse o caso. Ao usar o programa, esteja particularmente atento a mensagens como **`Invalid read`** e **`Invalid write`** que indicam estar a tentar ler ou escrever fora da área de memória reservada por si. O **`valgrind`** também detecta a utilização de variáveis não inicializadas dentro de expressões condicionais. Nesse caso receberá a mensagem **`Conditional jump or move depends on uninitialised value(s)`**.

Por fim, o **`valgrind`** fornece informação sobre a quantidade de memória alocada e libertada na *heap*, indicando quando essas duas quantidades não são iguais, o que aponta para existirem *memory leaks*.

2. Na função **`main`**, comente a linha de código "**`destroy_fs();`**" que se encontra na parte final do ficheiro **`main.c`**. Repita os comandos do ponto anterior e observe as indicações fornecidas pelo **`valgrind`**.

```
==7489==
==7489== HEAP SUMMARY:
==7489==     in use at exit: 6,240 bytes in 3 blocks
==7489==   total heap usage: 5 allocs, 2 frees, 11,360 bytes allocated
==7489==
==7489== LEAK SUMMARY:
==7489==    definitely lost: 0 bytes in 0 blocks
==7489==    indirectly lost: 0 bytes in 0 blocks
==7489==      possibly lost: 0 bytes in 0 blocks
==7489==    still reachable: 6,240 bytes in 3 blocks
==7489==         suppressed: 0 bytes in 0 blocks
==7489== Reachable blocks (those to which a pointer was found) are not shown.
==7489== To see them, rerun with: --leak-check=full --show-leak-kinds=all
==7489==
==7489== For counts of detected and suppressed errors, rerun with: -v
==7489== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```