Sistemas Operativos 2019-20

2º Guião Laboratorial

LEIC-A / LEIC-T / LETI

IST

Este documento pretende 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 tendo por base o código distribuído para o guião anterior (arquivo **`bst.zip`**). Será também revista a ferramenta **`valgrind`**.

A concluir, será explorada a criação de processos recorrendo à função de sistema **fork**.

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

Este exercício não será avaliado.

1. 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`** será usado o código distribuído na aula anterior. Crie um diretório no seu computador e descarregue o arquivo **`bst.zip`** (*binary search tree*) que está disponível na página da disciplina (no fénix), na secção "Laboratórios", e extraia os ficheiros lá existentes.

1. Adicione a seguinte função no início do ficheiro **`test.c`**.

```
void list_tree(node* p)
{
   list_tree(p->left);
   printf("%ld\n", p->key);
   list_tree(p->right);
}
```

2. Modifique a função `main` para oferecer o novo comando **`l`** que irá **listar** a árvore recorrendo à nova função `list_tree`.

3. Gere o programa **`test`** recorrendo à `makefile` fornecida.

   *Note que é obrigatório o uso da flag **`-g`** na compilação, para garantir que o executável inclui a devida informação simbólica (nomes de variáveis, funções, etc), para facilitar o uso do **`gdb`**.*

```
make
```

4. Execute o programa **`test`** e introduza os comandos indicados abaixo (note que é possível dar vários comandos numa mesma linha).

```
./test

a 20 a 15 a 30 a 22 a 10 a 29
p
l
```

5. O que sucedeu ao dar o comando **`l`** ?

   Provavelmente, no presente caso, será óbvio 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 test 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 test 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
```

---

6. 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 `./test'.
Program terminated with signal SIGSEGV, Segmentation fault.
#0  0x00005608d0e5b850 in list_tree (p=0x0) at test.c:11
11          list_tree(p->left);
(gdb)
```

7. 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  0x00005608d0e5b850 in list_tree (p=0x0) at test.c:11
#1  0x00005608d0e5b85c in list_tree (p=0x5608d13d78f0) at test.c:11
#2  0x00005608d0e5b85c in list_tree (p=0x5608d13d7870) at test.c:11
#3  0x00005608d0e5b85c in list_tree (p=0x5608d13d7830) at test.c:11
#4  0x00005608d0e5ba0a in main () at test.c:48
(gdb)
```

   O primeiro número em cada linha indica o nível em que essa função está, começando pela função onde o programa terminou abruptamente (neste caso, `list_tree`). Dado tratar-se de uma função que é chamada recursivamente, observa-se que o programa rebentou na 4ª chamada dessa função.

   Observe como o **`gdb`** mostra os argumentos passados às funções ao longo da pilha (neste caso, apenas o argumento **`p`**, sendo indicado o seu valor). Conclui-se assim que o erro ocorreu no ficheiro `test.c`, na linha 11, em que `list_tree` foi chamada com o apontador **`p`** igual a *null* (0x0).

   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.

8. Pode também examinar o contexto de chamada de uma dada função anterior àquela em que o problema ocorreu, subindo na pilha de chamadas (*call stack*) usando o comando **`up`**, ou ir para uma posição arbitrária com o comando **`frame`**. No presente caso, se quiser examinar a chamada inicial de `list_tree` deverá usar **`frame 4`**:

```
(gdb) frame 4
#4  0x00005608d0e5ba0a in main () at test.c:48
48              else if ( c == 'l' ) list_tree(root);
(gdb)#4  0x00005608d0e5ba0a in main () at test.c:48
(gdb)
```

9. Pode agora visualizar várias variáveis presentes nesse contexto, usando o comando **`p`**:

```
(gdb) p root
$1 = (node *) 0x5608d13d7830

(gdb) p *root
$2 = {key = 20, data = "empty", '\000' <repeats 14 times>,
  left = 0x5608d13d7870, right = 0x5608d13d78b0}

(gdb) p debug
$3 = 0

(gdb) p c
$4 = 108 'l'

(gdb) print k
$5 = 29
```

10. Para sair do **gdb** use o comando **q** (*quit*):

```
(gdb) q
```

Os passos descritos até agora apenas descrevem uma pequena parte das funcionalidades do **gdb**. No anexo no final deste guião encontra uma extensão a este guião que explica como analisar um programa em execução, passo a passo, definir *breakpoints*, etc.

2. 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 **test** e use a ferramenta **valgrind** para correr o programa.

```
make
valgrind --tool=memcheck --leak-check=yes ./test < tree1.txt
```

O programa pode ser corrido interactivamente ou, no caso acima, fornecendo-lhe dados a partir do ficheiro **tree1.txt**. Em qualquer dos casos, quando o programa termina, é mostrada informação semelhante à seguinte:

```
==19974== HEAP SUMMARY:
==19974==     in use at exit: 0 bytes in 0 blocks
==19974==   total heap usage: 10 allocs, 10 frees, 5,504 bytes allocated
==19974==
==19974== All heap blocks were freed -- no leaks are possible
==19974==
==19974== For counts of detected and suppressed errors, rerun with: -v
==19974== 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 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. Comente a linha de código "**`free_tree(root);`**" que se encontra na parte final do ficheiro **`test.c`**. Repita os comandos do ponto anterior e observe as indicações fornecidas pelo **`valgrind`**.

3. Criação de processos filho

Nesta secção serão estudadas a criação de processos filho e algumas das suas particularidades. Recorde a noção de processo dada na aula teórica e consulte o livro/slides sobre este tópico.

1. No programa **`test.c`**, adicione um novo comando **`f`** que executa o seguinte bloco de código:

```
pid_t pid;
pid = fork();
if(pid > 0)
  wait(NULL);
```

Se tentar compilar, verá que o código acima precisa de alguns *includes* em falta. Consulte as *man pages* das funções **`fork`** e **`wait`** para descobrir como corrigir este problema.

2. Antes de executar o programa, discuta com o seu colega de grupo qual o comportamento esperado do novo comando **`f`**. Consulte as *man pages* das funções **`fork`** e **`wait`** sempre que tiver dúvidas.

3. Compile e corra o programa, introduzindo os seguintes comandos

```
./test
a 20 a 15 a 30 a 22 a 10 a 29
p
```

4. Carregue **`ctrl+z`** para suspender o processo. De seguida, execute o comando **`ps`** (pesquise: **`man ps`**) na linha de comandos. Como esperado, deverá observar apenas um processo a executar o programa **`test`**.

```
ctrl+z
ps
  PID TTY          TIME CMD
29796 pts/3    00:00:00 bash
29862 pts/3    00:00:00 test
29868 pts/3    00:00:00 ps
```

5. Introduza **`fg`** na linha de comandos para continuar o processo que acabou de suspender.

```
fg
```

6. De volta ao nosso programa, introduza agora o comando **`f`**.

```
f
```

7. Repita os passos 4 e 5. Desta vez, deverá observar dois processos ativos que executam o programa **`test`**. Qual o processo pai e qual o processo filho?

8. Introduza os comandos seguintes, tendo noção que está a interagir com o processo filho.

```
p                                ← print tree (igual à do pai)
a 10
r 15
p                                ← print tree
q                                ← quit (termina processo filho)
```

9. Introduza o comando seguinte, sendo que agora está a interagir com o processo pai. Por que razão as alterações feitas na alínea anterior não são apresentadas no processo pai?

```
p
```

10. De seguida, introduza os comandos seguintes (alterar árvore, criar novo processo filho, visualizar árvore dentro do filho). Discuta o que observa.

```
a 10
r 15
p
f
p
```

Anexo: Utilização da ferramenta de depuração `gdb` (cont.)

Em seguida ilustra-se a definição de *breakpoints* e outros comandos que permitem analisar um programa em execução.

1. Carregue o programa test no gdb:

```
gdb test
```

2. Utilize o comando **`break`** (abreviado **`b`**) para colocar um *breakpoint* na intrução localizada na linha 44 do ficheiro **`bst.c`**:

```
(gdb) b bst.c:44
```

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

```
(gdb) b list_tree
```

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 é executada normalmente ficando a aguardar *input*. Introduza a seguinte linha:

```
> a 20 a 15 a 30
```

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 função **`insert`** (linha 44). **NOTA:** A linha onde pára ainda não foi executada!

8. Avance pelo código, linha a linha, utilizando o comando **`next`** (abreviado **`n`**), podendo visualizar as diversas variáveis. 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) n
(gdb) n
(gdb) p debug
(gdb) p *root
(gdb) n
(gdb) n
```

9. Pode continuar a execução sem interrupções até ao próximo *breakpoint* utilizando o comando **`continue`** (abreviado **`c`**).

```
(gdb) c
```

10. Outra maneira de navegar pelo programa é usando o comando **`step`** (abreviado **`s`**), que funciona como o **`next`** mas entra dentro das funções que executa. Existe também o comando **`finish`** (abreviado **`fin`**) que executa a função actual até à sua conclusão. Experimente o que se segue:

```
(gdb) s              ← Step; entra na função new_node
(gdb) s              ← Entra na função malloc
(gdb) fin            ← Executa malloc até ao fim
(gdb) c              ← Avança até ao próximo breakpoint
(gdb) c              ← Avança e test vai ler novo comando
>
```

11. Introduza o comando **`p`** (referente ao programa **`test`**) para imprimir a árvore entretanto criada e em seguida introduza o comando **`l`** para listar essa mesma árvore.

```
> p
    15
  20
```

```
    30
> l
```

12. O programa pára no *breakpoint* definido em **`list_tree`**. Use o comando **`n`** (**`next`**) para ir avançando no programa e observe os valores do apontador **`p`**:

```
(gdb) n
(gdb) n
(gdb) n
```

13. Saia do gdb com **`quit`** (abreviado **`q`**) ou premindo **`Ctrl-D`**:

```
(gdb) q
```

14. Corrija a função **`list_tree`** e volte a compilar e testar o programa.