Laboratórios de Sistemas Operativos
Tutorial #2: Ferramentas de deteção de erros

Os tutoriais práticos de SO consistem num conjunto de exercícios práticos que permitem aos alunos familiarizarem-se com um determinado tema que será necessário para resolver os projetos da disciplina. Os tutoriais podem ser resolvidos individualmente ou em grupo. A sua resolução é recomendada mas não obrigatória. Não são avaliados.

Cada tutorial pressupõe que os exercícios são realizados numa interface de linha de comandos (shell) de um sistema Unix/Linux ou equivalente. Assume também que os alunos já resolveram os tutoriais anteriores.

1. GNU Debugger (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)
{
   if (p->left)
       list_tree(p->left);
   printf("%ld\n", p->key);
   if (p->right)
       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.

```
make
```

4. Execute o programa **`test`** e, no estado inicial (árvore vazia), introduza o novo comando. O que acontece?

```
./test
l
```

O facto do programa ter falhado com *segmentation fault* foi uma surpresa? Quando este tipo de erros acontece, nem sempre é óbvio identificar a causa (ou seja, o *bug*). Nestes casos, o **gdb** pode ser uma ferramenta muito útil. As seguintes alíneas mostram como.

Nota: Para usar o gdb nas alíneas seguintes, é necessário que o seu programa seja compilado com a opção -g. Verifique as ***CFLAGS*** *na* ***Makefile****.*

5. Execute de novo o programa **test**, mas agora sob o controlo do **gdb**.

```
gdb test
```

6. Dentro do **gdb**, inicie a execução do programa com o comando **run**, ou simplesmente **r**.
   Nota: caso quisesse passar argumentos de linha de comando ao programa, poderia especificá-los como argumentos do comando **run**.

```
(gdb) r
```

7. Tal como antes, introduza o comando l para listar o conteúdo da árvore (vazia). Como esperado, o programa irá de novo falhar com *segmentation fault.* No entanto, desta vez o gdb está a controlar a execução e apresentará a seguinte mensagem:

```
Program received signal SIGSEGV, Segmentation fault.
0x000055555555525d in list_tree (p=0x0) at test.c:8
8           if (p->left)
```

8. Neste momento, é como se a falha estivesse congelada, o que permite ao programador analisar o estado da execução do programa nesse exato momento para entender o que causou a falha.
   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  0x000055555555525d in list_tree (p=0x0) at test.c:8
#1  0x0000555555555444 in main () at test.c:47
(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`).

   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).

   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.

9. Com esta informação, já detetou o *bug*? Então saia do **`gdb`** (use o comando **`quit`** ou **`q`**) e corrija o *bug* no ficheiro **`test.c`**.

No exemplo anterior, analisou o estado de um programa no momento em que este falhou (com *segmentation fault*). Outra forma útil de entender o comportamento do programa é usando *breakpoints*. Por exemplo, quando se acrescenta uma nova função a um programa já existente, é usual usar-se *breakpoints* para acompanhar, passo a passo, a forma como esse novo código se comporta.
De seguida ilustramos usando a nova função **`list_tree`**, assumindo que já corrigiu o *bug* que detetou nos passos anteriores.

1. De novo, carregue o programa test no gdb:

```
gdb test
```

2. Utilize o comando **`break`** (abreviado **`b`**) para colocar um *breakpoint* na primeira instrução da nova função, **`list_tree`**:

```
(gdb) b list_tree
```

3. Para confirmar que o *breakpoint* foi definido, pode pedir para listar os *breakpoints* assim:

```
(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`**.

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

```
(gdb) r
```

5. A aplicação é executada normalmente ficando a aguardar *input*. Introduza algumas entradas na árvore e peça para listar o conteúdo da mesma, por exemplo assim:

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

6. 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
```

7. Acompanhe, passo a passo, a execução da função à medida que a árvore é recursivamente listada, usando as seguintes dicas:
    - Pode avançar pelo código, linha a linha, utilizando o comando **`next`** (abreviado **`n`**).
    - Sempre que a execução chega a uma linha em que é chamada uma função (por exemplo, quando **`list_tree`** é chamada recursivamente para uma das sub-árvores), pode decidir entrar dentro dessa chamada usando o comando **`step`** (abreviado **`s`**) em vez do **`next`**.
    - Em cada momento, pode pedir para ler o valor de qualquer variável que exista no contexto atual usando o comando **`print`** *nome da variável* (abreviado **`p`**).
      Nota: 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.

2. Sanitizadores de código

O gcc permite que o programa a compilar seja instrumentado com rotinas que verificam, em tempo de execução, se determinados comportamentos incorretos ocorrem. Estas opções de instrumentação chamam-se sanitizadores de código (*code sanitizers*).

Existem diferentes tipos de sanitizadores, cada um focado em detetar diferentes classes de erros. A lista completa pode ser encontrada no link seguinte. Note que nem todos os sanitizadores podem ser combinados num mesmo programa compilado.
https://gcc.gnu.org/onlinedocs/gcc/Instrumentation-Options.html

1. Entre outros, os seguintes sanitizadores pode ser especialmente úteis para os programas C que desenvolveremos em SO: *AddressSanitizer, UndefinedBehaviorSanitizer, ThreadSanitizer* (este apenas quando, mais tarde, aprendermos a fazer programas concorrentes).
   Que tipo de erros deteta cada um destes sanitizadores? Consulte na documentação do gcc (link acima).

2. Para usar cada sanitizador, basta usar a opção **`-fsanitize`** do **`gcc`** aquando da compilação do seu programa. Veja no link acima como usar esta opção para ativar cada tipo de sanitizador.

3. Experimente agora voltar ao programa que compôs na alínea 1.1, que tinha um *bug*. Edite a *Makefile* para, nas CFLAGS, incluir um ou mais sanitizadores de código à sua escolha.

4. Compile o programa (provavelmente terá de fazer *make clean*) e experimente de novo correr o programa. Se adicionou o sanitizador certo, então o seu programa será interrompido assim que o sanitizador deteta o *bug* e verá uma mensagem de erro que o ajudará a corrigir o *bug*.