DEBUG DE CÓDIGOS COM GDB E VALGRIND

DCE792 - AEDs II (Prática)

Atualizado em: 25 de agosto de 2023

Iago Carvalho

Departamento de Ciência da Computação

DEBUG DE CÓDIGOS

Debug (ou depuração) de códigos é o processo para descobrirmos erros de programação inseridos em nossos algoritmos

- Erros seus
- Erros de seus colegas
- Erros de atualização de *software*

Como vocês encontram erros em seus códigos?

- Usam um monte de printf para imprimir os valores das variáveis?
- Comentam partes do código onde suspeitam de erro?
  - Depois é só ir descomentando linha por linha e ver se o erro volta. Isso funciona bem?

COMO DETECTAR ERROS

Erros podem ser diminuídos utilizando TDD (*Test Driven Development*)

- Metodologia de desenvolvimento focada em testes

Em um nível mais baixo, erros podem ser localizados e corrigidos utilizando técnicas de depuração

- GDB (GNU Debugger)
- Valgrind

GDB: GNU Debugger

Valgrind Memcheck

GDB

GDB

É a ferramenta mais tradicional para fazer a depuração de códigos

- Suporta C, C++, D, Objective-C, Fortran, Java, OpenCL C, Pascal, Assembly, Modula-2 e Ada

Permite pausar a execução, continuar passo-a-passo, inspecionar ou alterar valor de variáveis

- Tudo em tempo de execução

Pode ser usado em linha de comando ou integrado em IDEs de desenvolvimento

- Por exemplo, no Visual Studio Code

UTILIZAÇÃO DO GDB

Para utilizarmos o GDB, é necessário compilar o código com o parâmetro -g

- Facilmente incluído em um *makefile*
- Existem diferentes níveis do parâmetro -g

Logo após, pode-se inicializar o debug chamando o GDB antes do executável

- gdb *executável*

INTERFACE DO GDB

```
iagoac@DESKTOP-BFIS0ST:~/github/dce792/makefile/terceiro_exemplo$ make
Construindo target usando o compilador GCC: source/helloWorld.c
gcc source/helloWorld.c -c -g -Wall -ansi -pedantic -o objects/helloWorld.o

Construindo target usando o compilador GCC: source/main.c
gcc source/main.c -c -g -Wall -ansi -pedantic -o objects/main.o

Construindo o binário usando o linker GCC: hello
gcc objects/helloWorld.o objects/main.o -o hello
Binário pronto!: hello

iagoac@DESKTOP-BFIS0ST:~/github/dce792/makefile/terceiro_exemplo$ gdb hello
GNU gdb (Ubuntu 12.1-0ubuntu1~22.04) 12.1
Copyright (C) 2022 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from hello...
(gdb)
```

VISUALIZANDO CÓDIGO COM GDB

Mostra o código do programa, posicionado na <*linha*>

- (gdb) list <*linha*>

Mostra o código do programa, posicionado na <função>

- (gdb) list <*função*>

BREAKPOINTS

Um *breakpoint* (ou ponto de parada) é um marcador posicionado em algum local onde queremos inspecionar o código

- Faz com que o código seja executado até o ponto mencionado
- Ao encontrar o ponto, o código para de executar
- É possível verificar valores de variáveis naquele exato momento

Podemos criar um *breakpoint* de três maneiras distintas

1. (gdb) break <*linha*>
2. (gdb) break <*arquivo*>:<*linha*>
3. (gdb) break <*função*>

CRIANDO BREAKPOINTS

1. (gdb) break <*linha*>
2. (gdb) break <*arquivo*>:<*linha*>
3. (gdb) break <*função*>

1. (gdb) break *3*
2. (gdb) break *helloWorld.c:4*
3. (gdb) break *helloWorld*

BREAKPOINTS CONDICIONAIS

Também é possível criar um *breakpoint* condicionado a algum valor de variável

- Útil para inspecionar alguma iteração específica de um loop

(gdb) break <*condição*>

Exemplo: (gdb) break *if i > 100*

MÉTODOS PARA LIDAR COM BREAKPOINTS

Listar todos os *breakpoints* existentes

- (gdb) info *breakpoint*

Apagar um *breakpoint*

- (gdb) delete #*breakpoint*

Habilitar ou desabilitar um *breakpoint*

- (gdb) enable #*breakpoint*
- (gdb) disable #*breakpoint*

NAVEGANDO ENTRE BREAKPOINTS

Continua a execução até o próximo *breakpoint* ou até o fim do programa

- (gdb) continue

Passa para a próxima linha

- (gdb) step
- (gdb) next

Os métodos *step* e *next* se diferenciam pela forma como tratam funções. Se a próxima linha for uma função, então

- O *step* executa a primeira linha da função
- O *next* para no cabeçalho da função

NAVEGANDO ENTRE BREAKPOINTS

Continua a execução até o fim de um *loop*

- (gdb) until

Continua a execução até o fim da função atual

- (gdb) finish

WATCHPOINTS

Um *watchpoint* vigia o valor de uma variável

- Caso o valor da variável mude, o código é interrompido
- Desta forma é possível inspecionar o estado de seu algoritmo quando ocorrem troca de valor de uma variável

Define vigia sobre uma variável

- (gdb) watch <*variável*>

Lista todos os *watchpoints* definidos

- (gdb) info watchpoints

Remove um *watchpoint*

- (gdb) delete #*watchpoint*

INSPECIONANDO VARIÁVEIS

Imprime o valor de uma variável

- (gdb) print <*variável*>

Mostra o valor de uma variável a cada passo da execução

- (gdb) display <*variável*>

Lista dos displays ativos

- (gdb) info display

Remove um display

- (gdb) undisplay *#watchpoint*

ALTERANDO A PILHA DE PROGRAMA

Atribui valor a variável (do escopo)

- (gdb) set <*variável*> = <*valor*>

Chama uma função ou método

- (gdb) call <*função()*>

Desvia o fluxo de execução do programa

- (gdb) jump <*linha*>

ALTERANDO A PILHA DE PROGRAMA

Imprime a pilha de chamadas do programa, obtendo seus *frames*

- (gdb) backtrace

Troca o *frame* atual por um passado

- (gdb) frame *#frame*

Faz o *frame* atual retornar. Simula o fim de uma função

- (gdb) return
- (gdb) return *<valor>*

VALGRIND

VAZAMENTO DE MEMÓRIA

O *Valgrind* é um software utilizado para detectar vazamentos de memória

- Perda de espaço de memória
- Memória alocada dinamicamente e nunca desalocada

Em códigos pequenos e simples, isto não costuma ser um problema

- Toda memória alocada dinamicamente é liberada automaticamente com o fim do algoritmo

Passa a ser um problema em códigos longos, recursivos, ou que utilizem uma mesma função com vazamento de memória por diversas vezes

- Erros difíceis de serem encontrados e corrigidos
- Esgotamento (rápido) da RAM de seu dispositivo

COMO UTILIZAR O VALGRIND

Compilar o código com o parâmetro -g

- Similar ao realizado para o GDB

Recomenda-se também utilizar o parâmetro -Wall

O *Valgrind* é iniciado de forma simples

- valgrind –leak-check=yes *./executável*

INTERFACE DO VALGRIND

```
iagoac@DESKTOP-BFIS0ST:~/github/dce792/makefile/terceiro_exemplo$ valgrind --leak-check=yes ./hello
==19403== Memcheck, a memory error detector
==19403== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==19403== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==19403== Command: ./hello
==19403==
Hello World!
==19403==
==19403== HEAP SUMMARY:
==19403==     in use at exit: 0 bytes in 0 blocks
==19403==   total heap usage: 1 allocs, 1 frees, 1,024 bytes allocated
==19403==
==19403== All heap blocks were freed -- no leaks are possible
==19403==
==19403== For lists of detected and suppressed errors, rerun with: -s
==19403== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

O QUE O VALGRIND PODE FAZER?

As principais utilidades dele são

- Uso de memória não inicializada
- Acesso a memória desalocada
- Acesso a memória em espaço não alocado
- *Double free* / *Double delete*
- Troca de *free* por *delete* (e vice-versa)
- Perca de ponteiros

ACESSO A MEMÓRIA DESALOCADA

```
#include <stdio.h>
#include <iostream>

int main()
{
    //Declarando um ponteiro para tipo 'int'
    int *usarEndDesalocado;
    //Alocando memoria de tamanho 'int' e gravando seu endereco no ponteiro
    usarEndDesalocado = new int();
    //Inicializando memoria com o valor 1111
    *usarEndDesalocado = 1111;
    //Desalocando memoria
    delete usarEndDesalocado;
    //Tentando imprimir o valor de memoria previamente desalocada: Comportamento indefinido!
    printf ("%i\n", *usarEndDesalocado);

    return 0;
}
```

ACESSO A MEMÓRIA EM ESPAÇO NÃO ALOCADO

```c
#include <stdio.h>

int main()
{
    //Declarando um ponteiro para tipo 'int'
    int *usarEndForaAlocacao;
    //Alocando memoria de tamanho 'int' e gravando seu endereco no ponteiro
    usarEndForaAlocacao = new int();
    //Inicializando memoria com o valor 1111
    *usarEndForaAlocacao = 1111;
    //Tentando imprimir o valor de memoria alem do espaco alocado: Comportamento indefinido!
    printf ("%i\n", *(usarEndForaAlocacao+1));

    return 0;
}
```

ACESSO A MEMÓRIA EM ESPAÇO NÃO ALOCADO

```c
#include <stdio.h>

int main()
{
    //Declarando um ponteiro para tipo 'int'
    int *usarEndForaAlocacao;
    //Alocando memoria de tamanho 'int' e gravando seu endereco no ponteiro
    usarEndForaAlocacao = new int();
    //Inicializando memoria com o valor 1111
    *usarEndForaAlocacao = 1111;
    //Tentando imprimir o valor de memoria alem do espaco alocado: Comportamento indefinido!
    printf ("%i\n", *(usarEndForaAlocacao+1));

    return 0;
}
```

ACESSO A MEMÓRIA EM ESPAÇO NÃO ALOCADO

```c
#include <stdio.h>

int main()
{
    //Declarando um vetor de 5 posicoes 'int'
    int intArray[5] = {1,2,3,4,5};
    //Tentando acessar a 6a. posicao: Comportamento indefinido!
    printf ("%i\n", intArray[6]);

    return 0;
}
```

DOUBLE FREE/DOUBLE DELETE

```
#include <stdio.h>

int main()
{
    //Declarando um ponteiro para tipo 'int'
    int *duplodelete;
    //Alocando memoria de tamanho 'int' e gravando seu endereco no ponteiro
    //Utilizando o operador NEW
    duplodelete = new int();
    //Inicializando memoria com o valor 1111
    *duplodelete = 1111;
    //Liberando endereco de memoria
    delete duplodelete;
    //Liberando o mesmo endereco de memoria: PROBLEMA!
    delete duplodelete;

    return 0;
}
```

TROCA DE *FREE* POR *DELETE* (E VICE-VERSA)

```
#include <stdlib.h>

int main()
{
    //Declarando um ponteiro para tipo 'int'
    int *int_usando_new;
    //Alocando memoria de tamanho 'int' e gravando seu endereco no ponteiro
    //Utilizando NEW
    int_usando_new = new int();
    //Inicializando memoria com o valor 1111
    *int_usando_new = 1111;
    //Liberando endereco de memoria
    //Utilizando FREE: Nao utilizacao do par certo para desalocar memoria
    //New -> Delete
    //Malloc -> Free
    free (int_usando_new);

    return 0;
}
```

PERCA DE PONTEIROS

```
#include <stdio.h>

int main()
{
    //Declarando um ponteiro para tipo 'int'
    int *valor_inteiro_ptr;
    //Alocando memoria de tamanho 'int' e gravando seu endereco no ponteiro
    valor_inteiro_ptr = new int();
    //Inicializando memoria com o valor 1111
    *valor_inteiro_ptr = 1111;
    //Nova alocacao de memoria, endereco para a antiga alocacao sera perdido!
    //Perdemos o endereco anteriormente alocado
    valor_inteiro_ptr = new int();
    //Desalocando endereco mais recentemente alocacao
    //Endereco alocado anteriormente nao foi desalocado: MEMORY LEAK!
    delete valor_inteiro_ptr;

    return 0;
}
```

PRÓXIMA AULA:

LISTAS