Programação de Computadores

Aula #11

Depuração de Código

O que é depuração de código? O que é e como utilizar o GDB? Como funciona a depuração de memória e o *profiling* de código?

Ciência da Computação – BCC e IBM – 2024/01
Prof. Vinícius Fülber Garcia

Depuração de Código

Depurar significa **“purificar”** ou **“limpar”**. A depuração de código é muito útil para resolver erros de execução, de alocação/liberação de memória ou de desempenho.

Existem muitas ferramentas e técnicas que podem ser utilizadas para depurar um programa, seja ele escrito em C ou não.

Depuração de Código

Porém, quando falamos em depuração de um programa escrito em C, precisamos realizar tomar algumas medidas para incluir símbolos de depuração durante o processo de compilação:

- Nomes de variáveis
- Linhas de código
- ...

```
gcc meu_programa.c -g -o meu_programa.h
```

Depuração de Execução

Realizando a compilação com os símbolos de programação incluídos (-g), podemos executar uma depuração de execução do programa.

Para a linguagem C, o depurador padrão é o GNU *Debugger* (GDB)

O GDB é uma ferramenta de depuração em modo texto com diversas funcionalidades. Para acessar essas funcionalidades, devemos executar o programa com o GDB:

```
gdb meu_programa
```

Depuração de Execução

O GDB inclui uma lista bastante grande de comandos; os principais são:

Comando	Utilidade	Exemplo
r	Executa o programa	r
l	Lista linhas de código	l 1
b	Cria um ponto de parada (breakpoint)	b 10
c	Continua a execução	c
s	Avança para a próxima linha	s
n	Avança para a próxima linha (sem função)	n

Depuração de Execução

O GDB inclui uma lista bastante grande de comandos; os principais são:

Comando	Utilidade	Exemplo
p	Mostra o valor de uma variável	p var
watch	Informa a mudança de valor de uma variável	watch v
disp	Informa o valor de uma variável a cada pausa	disp v
set variable	Ajusta o valor de uma variável	set variable var = 1
bt	Mostra a posição atual do programa	bt
frame	Seleciona a hierarquia de função para análise	frame 2

Depuração de Execução

Existe um **documento de referência** (disponível **AQUI**) com diversas operações disponíveis dentro do depurador.

Além disso, cabe ressaltar que existem interfaces diferentes para o GDB, como a NCurses, CGDB, DDD e Nemiver.

Ainda, é possível utilizar o GDB integrado a algumas IDEs, como o Code::Blocks e Eclipse.

Depuração de Memória

Existe uma infinidade de potenciais problemas de memória:

- *Buffer overflow*
- Uso de ponteiros não inicializados
- *Memory leak*
- ...

Existem algumas ferramentas que podem ser utilizadas nesse contexto para detectar tais erros, antes ou durante a compilação de um programa

Depuração de Memória

Algumas ferramentas analisam o código de forma estática, independente do compilador. No contexto da linguagem C, podemos citar como exemplos:

- cppcheck (cppcheck --enable=all meu_programa.c)
- splint *(splint meu_programa.c)*

Também é possível utilizar algumas flags de compilação para realizar a depuração de memória, exemplos em C são:

- -Wall

Depuração de Memória

Também, é possível utilizar depuradores de memória para detectar possíveis problemas de alocação e acesso à memória após a compilação, em tempo de execução:

- Mtrace
- Valgrind

Nesse contexto, a solução mais utilizada, sem dúvidas, é o **Valgrind**

Depuração de Memória

O Valgrind conta com uma miríade de ferramentas para a depuração de memória; as mais comuns são:

- memcheck: análise de acesso à memória (valgrind --tool=memcheck ./meu_programa)
- cachegrind: análise de uso das caches do sistema (valgrind --tool=cachegrind ./meu_programa)
- exp-sgcheck: análise de variáveis globais e de pilha de memória (valgrind --tool=exp-sgcheck ./meu_programa)
- leak-check: análise de vazamento de memória (valgrind --leak-check=full ./meu_programa)

Tracers

Tracers são muito úteis para **analisar chamadas de sistema e outras operações** realizadas por processos genéricos.

A principal vantagem na utilização de *tracers* é a não necessidade de instrumentação de código ou de acesso ao código fonte de um programa, além de poderem iniciar sua operação em um processo já em execução.

strace (strace ./meu_programa) e ltrace (ltrace ./meu_programa)

Profilers

Profilers se destacam por analisarem o **desempenho de um programa**, principalmente no que se refere a aspectos temporais.

Em C, o mais conhecido *profiler* é o gprof, sendo capaz de:

- Determinar o tempo gasto em cada função
- Gerar um grafo de chamadas de função
- Definir o número de chamadas de uma função

Profilers

Para o uso do gprof, entretanto, é necessária a instrumentação do binário gerado após o processo de compilação:

```
gcc -g -pg meu_programa.c -o meu_programa
```

Após a execução do programa, será gerado um **arquivo de *profiling* chamado gmon.out**, este será então utilizado pelo gprof:

```
gprof meu_programa gmon.out
```

Rubber Duck

Debugar um código não é uma tarefa trivial!!

Mesmo com acesso a todas as ferramentas citadas anteriormente, podemos ainda assim ter problemas para determinar erros e problemas lógicos em um código fonte.

A estratégia *rubber duck* (patinho de borracha) consiste em simplesmente explicar seu código para alguém na esperança de que um erro oculto se torne evidente no processo.

Exercício #11

```c
#include <stdlib.h>
#include <stdio.h>
int main (void){
 int *vetor_int, i;
 short *vetor_short;
 vetor_int = (int*) malloc(10 * sizeof(int));
 vetor_short = (short*) malloc (4 * sizeof(short));
 for (i = 0; i < 4; i++){
     vetor_int[i] = i;
     vetor_short[i] = i;
 }
 for (; i < 14; i++){
     vetor_int[i] = i;
 }
 printf("Shorts:[%d][%d][%d][%d]\n", //
   vetor_short[0], vetor_short[1], //
   vetor_short[2], vetor_short[3]);
 return 0;
}
```

Considere o programa ao lado; **compile e verifique o resultado de execução**. Em seguida, explore o mesmo através das ferramentas apresentadas para determinar, **tecnicamente**, o que está acontecendo.

Dica: GDB e Valgrind podem ser muito úteis!

Obrigado!

Vinícius Fülber Garcia
inf.ufpr.br/vinicius
vinicius@inf.ufpr.br