UNIVERSIDADE PRESBITERIANA MACKENZIE

Faculdade de Computação e Informática

Laboratório: Ferramentas de Depuração e Análise de Vazamento de Memória

Material de Estudo e Exercícios

Disciplina: Sistemas Operacionais

Prof. Lucas Cerqueira Figueiredo

1º Semestre de 2025

Sumário

1 Introdução

A depuração de problemas de memória é uma habilidade bem relevante para programadores, especialmente em linguagens como C, onde o gerenciamento de memória é realizado manualmente. Nesse laboratório vocês vão explorar o uso do Valgrind, uma ferramenta para detecção de problemas de memória em programas C/C++.

No contexto de Sistemas Operacionais, o gerenciar memória corretamente é crítico, pois o próprio kernel do sistema deve implementar mecanismos de alocação, rastreamento e liberação de recursos de memória para os processos. Um vazamento de memória no kernel pode comprometer todo o sistema, enquanto acessos inválidos podem causar falhas de segmentação (segmentation fault) e comprometer a estabilidade do sistema.

2 Fundamentos da Depuração de Memória

2.1 Problemas Comuns de Memória

1. **Vazamento de Memória (Memory Leak)**:
   - Ocorre quando alocamos memória dinamicamente, mas esquecemos de liberá-la
   - Causa esgotamento gradual da memória disponível
   - Em sistemas operacionais, pode levar a falhas por falta de recursos
2. **Acesso Inválido (Invalid Access)**:
   - Ler/escrever em posições de memória fora dos limites alocados
   - Acessar memória já liberada (use-after-free)
   - No contexto de SOs, pode levar a corrupção de dados críticos do sistema
3. **Liberação Dupla (Double Free)**:
   - Tentar liberar a mesma região de memória mais de uma vez
   - Pode corromper as estruturas internas do gerenciador de memória
4. **Uso de Variáveis Não Inicializadas**:
   - Utilizar o valor de variáveis sem inicialização prévia
   - Pode levar a comportamentos imprevisíveis e difíceis de rastrear

Relação com Sistemas Operacionais

Os problemas de memória são particularmente relevantes para o contexto de SO:

- O kernel precisa gerenciar a memória para todos os processos
- Vazamentos de memória no espaço do kernel não são automaticamente corrigidos quando um processo termina
- A memória do kernel é limitada e compartilhada por todo o sistema
- Falhas de memória no kernel podem comprometer a estabilidade de todo o sistema

3 Valgrind: Ferramenta de Análise Dinâmica

O Valgrind é um framework para ferramentas de análise dinâmica de código, sendo Memcheck seu componente mais utilizado para detecção de problemas de memória. Essa ferramenta foi criada para ajudar desenvolvedores a encontrar e corrigir problemas relacionados à memória que são difíceis de detectar com métodos tradicionais de depuração.

3.1 Como o Valgrind Funciona

1. Executa o programa em um ambiente controlado
2. Monitora todas as operações de memória (alocações, liberações, acessos)
3. Detecta problemas em tempo de execução
4. Gera relatórios detalhados sobre os problemas encontrados

3.2 Uso Básico do Valgrind

Para utilizar o Valgrind, você precisa:

1. Compilar o programa com flags de depuração:

```
gcc -g programa.c -o programa
```

2. Executar com Valgrind:

```
valgrind --leak-check=full --show-leak-kinds=all --track-origins=yes ./programa
```

Principais opções:

- `--leak-check=full`: Detalhamento completo dos vazamentos
- `--show-leak-kinds=all`: Mostra todos os tipos de vazamento
- `--track-origins=yes`: Rastreia a origem de valores não inicializados

3.3 Interpretando a Saída do Valgrind

O relatório do Valgrind possui seções importantes:

1. **Erros de memória** - Operações inválidas durante a execução
2. **Resumo de heap** - Estatísticas sobre alocação/liberação
3. **Vazamentos de memória** - Detalhes sobre memória não liberada

Tipos de vazamento reportados:

- **Definitivamente perdido**: Memória não liberada e sem referência
- **Indiretamente perdido**: Perdido devido a outro vazamento
- **Possivelmente perdido**: Valgrind não tem certeza se há referência
- **Ainda acessível**: Memória não liberada, mas ainda referenciada

Exemplo de Saída do Valgrind

```
==12345== HEAP SUMMARY:
==12345==     in use at exit: 40 bytes in 1 blocks
==12345==   total heap usage: 1 allocs, 0 frees, 40 bytes allocated
==12345==
==12345== 40 bytes in 1 blocks are definitely lost in loss record 1 of 1
==12345==    at 0x4C2DB8F: malloc (in /usr/lib/valgrind/...)
==12345==    by 0x400584: funcao_com_vazamento (leak.c:5)
==12345==    by 0x4005A5: main (leak.c:14)
```

4 Exercícios Práticos

4.1 Exercício 1: Identificação de Vazamento de Memória

Código com problema:

```c
#include <stdlib.h>
#include <stdio.h>

void funcao_com_vazamento() {
    int *array = (int*) malloc(10 * sizeof(int));

    for (int i = 0; i < 10; i++) {
        array[i] = i * 10;
    }

    // Esqueceu de liberar a memória!
}

int main() {
    for (int i = 0; i < 5; i++) {
        funcao_com_vazamento();
    }
    printf("Programa executado com sucesso!\n");
    return 0;
}
```

Tarefas:

1. Compile o programa com flags de depuração
2. Execute-o com Valgrind e analise o relatório
3. Identifique quantos bytes foram vazados e em quantos blocos
4. Corrija o problema de vazamento

Contexto de SO: Este tipo de vazamento é similar ao que pode ocorrer em um sistema operacional quando um driver de dispositivo aloca memória para operações de E/S mas não a libera corretamente, levando a degradação gradual do desempenho do sistema.

4.2 Exercício 2: Uso de Memória Não Inicializada

Código com problema:

```c
#include <stdlib.h>
#include <stdio.h>

int main() {
    int *array = (int*) malloc(10 * sizeof(int));
    int soma = 0;

    // Não inicializamos o array!

    // Tentamos usar os valores não inicializados
    for (int i = 0; i < 10; i++) {
        soma += array[i];
    }

    printf("A soma dos elementos é: %d\n", soma);
```

```
    free(array);
    return 0;
}
```

Tarefas:

1. Compile e execute com Valgrind
2. Identifique o problema reportado
3. Corrija o código para eliminar o erro

Contexto de SO: Em sistemas operacionais, usar memória não inicializada pode revelar dados de outros processos (violação de segurança) ou levar a comportamentos imprevisíveis. É por isso que sistemas modernos como Linux, ao alocar memória para um novo processo, geralmente a preenche com zeros ou valores aleatórios por questões de segurança.

4.3 Exercício 3: Acesso Fora dos Limites (Buffer Overflow)

Código com problema:

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

int main() {
    int *array = (int*) malloc(5 * sizeof(int));

    for (int i = 0; i < 5; i++) {
        array[i] = i;
    }

    // Acesso fora dos limites!
    for (int i = 0; i <= 5; i++) {
        printf("array[%d] = %d\n", i, array[i]);
    }

    free(array);
    return 0;
}
```

Tarefas:

1. Execute com Valgrind e analise o relatório
2. Identifique o problema de acesso à memória
3. Corrija o código

Contexto de SO: Os buffer overflows são uma das vulnerabilidades mais comuns em sistemas operacionais e aplicações. Eles podem levar a corrupção de dados, crashes do sistema e até exploração por software malicioso. Muitos ataques históricos a sistemas operacionais exploraram esse tipo de vulnerabilidade.

4.4 Exercício 4: Liberação Dupla (Double Free)

Código com problema:

```c
#include <stdlib.h>
#include <stdio.h>

int main() {
    int *ptr = (int*) malloc(sizeof(int));
    *ptr = 42;

    printf("Valor: %d\n", *ptr);

    free(ptr);
    printf("Memória liberada uma vez\n");

    // Tentativa de liberar novamente!
    free(ptr);
    printf("Tentativa de liberar memória novamente\n");

    return 0;
}
```

Tarefas:

1. Execute com Valgrind
2. Identifique o erro de liberação dupla
3. Corrija o código para evitar o problema

Contexto de SO: Liberações duplas podem corromper as estruturas de dados internas do gerenciador de memória do sistema operacional. Se isso ocorrer no kernel, pode levar a falhas catastróficas no sistema. Muitos gerenciadores de memória modernos implementam proteções contra liberações duplas.

4.5 Exercício 5: Uso Após Liberação (Use-After-Free)

Código com problema:

```c
#include <stdlib.h>
#include <stdio.h>

int main() {
    int *ptr = (int*) malloc(sizeof(int));
    *ptr = 42;

    printf("Valor inicial: %d\n", *ptr);

    free(ptr);

    // Uso após liberação!
    *ptr = 100;
    printf("Valor após liberação: %d\n", *ptr);
```

```
    return 0;
}
```

Tarefas:

1. Execute com Valgrind
2. Identifique o problema de uso após liberação
3. Corrija o código

Contexto de SO: Vulnerabilidades do tipo use-after-free são comuns em drivers de dispositivos e no próprio kernel dos sistemas operacionais. Quando a memória é liberada, ela pode ser realocada para outro propósito, mas se o código original continuar a usá-la, pode interferir com o novo uso, causando comportamentos imprevisíveis.

4.6 Exercício 6: Implementação de Lista Ligada com Gerenciamento de Memória

Implemente uma função de gerenciamento simplificado de memória para uma lista ligada:

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

typedef struct Node {
    int value;
    struct Node* next;
} Node;

// Função para criar um novo nó
Node* createNode(int value) {
    Node* newNode = (Node*)malloc(sizeof(Node));
    if (newNode == NULL) {
        printf("Erro: falha na alocação de memória\n");
        exit(1);
    }
    newNode->value = value;
    newNode->next = NULL;
    return newNode;
}

// Função para adicionar um nó no final da lista
void appendNode(Node** head, int value) {
    Node* newNode = createNode(value);

    if (*head == NULL) {
        *head = newNode;
        return;
    }

    Node* current = *head;
    while (current->next != NULL) {
        current = current->next;
    }
```

```
    current->next = newNode;
}

// Função para imprimir a lista
void printList(Node* head) {
    Node* current = head;
    while (current != NULL) {
        printf("%d -> ", current->value);
        current = current->next;
    }
    printf("NULL\n");
}

// TODO: Implemente a função para liberar toda a lista
void freeList(Node* head) {
    // Sua implementação aqui
}

int main() {
    Node* list = NULL;

    // Adiciona 10 nós
    for (int i = 0; i < 10; i++) {
        appendNode(&list, i * 10);
    }

    printList(list);

    // TODO: Libere a memória da lista

    return 0;
}
```

Tarefas:

1. Implemente a função `freeList` para liberar corretamente toda a memória da lista
2. Utilizando Valgrind, verifique se sua implementação está livre de vazamentos
3. Teste sua solução com diferentes tamanhos de lista

Contexto de SO: Listas ligadas são estruturas de dados fundamentais em sistemas operacionais, usadas para implementar tabelas de processos, gerenciamento de memória e filas de E/S. O gerenciamento correto destas estruturas é crucial para a estabilidade do sistema.

5 Dicas e Melhores Práticas

1. **Sempre inicialize variáveis** antes de usá-las
2. **Para cada malloc(), deve haver um free()** correspondente
3. **Verifique o retorno de malloc()** para garantir que a alocação foi bem-sucedida
4. **Use NULL após liberar um ponteiro** para evitar uso acidental após liberação:

```
    free(ptr);
    ptr = NULL;
```

5. **Verifique os limites de arrays** antes de acessá-los
6. **Compile com flags de depuração** (-g) para obter relatórios mais detalhados

6 Entrega

Para cada exercício, entregue:

1. O código corrigido (arquivo .c)
2. A saída do Valgrind para o código original (arquivo .txt)
3. A saída do Valgrind para o código corrigido (arquivo .txt)

Formato de Entrega

Organize os arquivos da seguinte forma:

```
lab_valgrind_RAdoAluno/
  |-- exercicio1/
  |     |-- corrigido.c
  |     |-- valgrind_original.txt
  |     |-- valgrind_corrigido.txt
  |-- exercicio2/
  |     |-- ...
```

Compacte a pasta e envie pelo Moodle até a data limite.

7 Bibliografia Recomendada

- Documentação oficial do Valgrind: http://valgrind.org/
- TANENBAUM, A. S. *Sistemas Operacionais Modernos*. 3ª ed. São Paulo: Pearson, 2010.
- SILBERSCHATZ, A., GALVIN, P.B, GAGNE, G. *Fundamentos de Sistemas Operacionais*. 8ª. ed. São Paulo: LTC, 2010.
- ARPACI-DUSSEAU, R. H.; ARPACI-DUSSEAU, A. C. *Operating Systems: Three Easy Pieces*. Arpaci-Dusseau Books, versão 1.10, novembro de 2023.