Modo Usuário e Chamadas de Sistema

Relatório da Tarefa 1

May 12, 2025

1 Introdução

Este relatório analisa o comportamento e as implicações da execução de programas em modo usuário e a necessidade de chamadas de sistema. Através de um experimento com código assembly minimalista, demonstramos a interação entre programas de usuário e o sistema operacional, com foco especial na ocorrência de falhas de segmentação e na importância das chamadas de sistema para encerramento de programas.

2 Código Mínimo e Falha de Segmentação

2.1 Código Inicial

O código assembly minimalista inicial continha apenas uma instrução para atribuir o valor 42 ao registrador EAX:

```
.section .text
.global _start
_start:
    movl $42, %eax
```

Este código foi compilado utilizando os seguintes comandos:

```
$ as -o minimal.o minimal.s
$ ld -o minimal minimal.o
```

Ao executar o programa resultante, ocorreu uma falha de segmentação:

```
$ ./minimal
Falha de segmentação (imagem do núcleo gravada)
```

2.2 Explicação da Falha

A falha de segmentação ocorre porque o programa não possui uma forma adequada de encerramento. Quando o processador chega ao final das instruções do programa, ele tenta continuar executando o que estiver na memória seguinte, que pode conter dados não-executáveis ou endereços de memória não mapeados.

Em programas normais, o encerramento é gerenciado pelo sistema operacional através de uma chamada de sistema específica (syscall). Sem essa chamada, o processador tenta

executar instruções em áreas de memória não autorizadas ou inexistentes, resultando na falha de segmentação.

A saída do comando `strace` confirma isso:

```
$ strace ./minimal
execve("./minimal", ["./minimal"], 0x7ffcef520df0 /* 63 vars */) = 0
--- SIGSEGV {si_signo=SIGSEGV, si_code=SEGV_MAPERR, si_addr=0x2a} ---
+++ killed by SIGSEGV (core dumped) +++
```

O valor `si_addr=0x2a` (decimal 42) indica que o processador tentou interpretar o valor atribuído ao registrador EAX como um endereço de instrução, o que causou a falha.

3 Correção do Código

3.1 Código Corrigido

Para corrigir a falha, foi necessário adicionar uma chamada de sistema para encerrar corretamente o programa:

```
.section .text
.global _start
_start:
    movl $42, %eax      # Assign value 42 to EAX register

    # Exit system call
    movl $1, %eax       # System call number for exit is 1
    movl $0, %ebx       # Return code 0 (success)
    int $0x80           # Invoke the system call
```

Este código adiciona a chamada de sistema `exit()` para encerrar o programa corretamente. Em sistemas Linux x86, a chamada de sistema é feita colocando o número da syscall no registrador EAX (1 para `exit`), os argumentos em outros registradores (EBX com o código de retorno 0), e executando a instrução `int $0x80` para transferir o controle para o kernel.

3.2 Resultado da Correção

Após a correção, o programa executa e termina normalmente, sem falhas de segmentação:

```
$ ./minimal_fixed
$ echo $?
0
```

4 Análise com strace

4.1 Antes da Correção

O resultado do `strace` antes da correção mostrou:

```
execve("./minimal", ["./minimal"], 0x7ffcef520df0 /* 63 vars */) = 0
--- SIGSEGV {si_signo=SIGSEGV, si_code=SEGV_MAPERR, si_addr=0x2a} ---
+++ killed by SIGSEGV (core dumped) +++
```

Observamos que apenas a chamada de sistema `execve` foi realizada (pelo shell para iniciar o programa), e em seguida ocorreu a falha de segmentação. Não houve chamada para encerramento normal do programa.

4.2 Depois da Correção

O resultado do `strace` após a correção:

```
execve("./minimal_fixed", ["./minimal_fixed"], 0x7fff7c885f10 /* 63 vars */) = 0
[ Process PID=27441 runs in 32 bit mode. ]
exit(0)                                 = ?
+++ exited with 0 +++
```

Agora podemos ver que, além da chamada `execve` inicial, há também a chamada de sistema `exit(0)` que permite o encerramento adequado do programa com código de retorno 0, indicando sucesso.

5 Necessidade da Chamada de Sistema para Encerramento

A chamada de sistema para encerramento (`exit`) é necessária pelos seguintes motivos:

1. **Liberação de recursos**: O sistema operacional precisa recuperar todos os recursos alocados ao processo, como memória, descritores de arquivo e outros recursos do sistema.
2. **Notificação de encerramento**: O processo pai (normalmente o shell) precisa ser notificado sobre o encerramento do processo filho e seu código de retorno.
3. **Atualização de tabelas internas**: O sistema operacional precisa atualizar suas tabelas internas para refletir que o processo não está mais em execução.
4. **Prevenção de falhas**: Sem um encerramento adequado, o processador tentaria executar instruções em áreas de memória não autorizadas, como demonstrado pelo experimento.

A chamada de sistema age como uma interface controlada entre o programa do usuário e o kernel do sistema operacional, permitindo a transição segura do modo usuário para o modo kernel durante operações privilegiadas como o encerramento do processo.

6 Implicações da Execução Livre

Se os programas de usuário pudessem ser executados livremente, sem recorrer ao sistema operacional para acessar recursos básicos, diversas implicações negativas surgiriam:

1. **Comprometimento da segurança**: Programas poderiam acessar livremente memória de outros processos ou do kernel, comprometendo a segurança e a privacidade dos dados.

2. **Instabilidade do sistema**: Programas mal-comportados poderiam facilmente travar todo o sistema, não apenas a si mesmos.

3. **Alocação descontrolada de recursos**: Sem o controle do sistema operacional, programas poderiam consumir todos os recursos disponíveis, como memória e CPU, impedindo que outros programas funcionassem adequadamente.

4. **Conflitos de hardware**: Múltiplos programas poderiam tentar controlar o mesmo hardware simultaneamente, causando comportamentos imprevisíveis.

5. **Ausência de isolamento**: Sem o isolamento provido pelo sistema operacional, um bug em qualquer programa poderia afetar todo o sistema.

6. **Dificuldade no compartilhamento justo**: O sistema operacional implementa algoritmos de escalonamento e compartilhamento de recursos que seriam impossíveis sem seu controle centralizado.

O modelo de separação entre modo usuário e modo kernel, com acesso a recursos críticos controlado por chamadas de sistema, é fundamental para a estabilidade, segurança e eficiência dos sistemas computacionais modernos. Este experimento simples demonstra claramente a importância dessa arquitetura, mesmo para operações aparentemente triviais como o encerramento de um programa.

7 Conclusão

Este experimento demonstrou de forma prática como funciona a interação entre programas em modo usuário e o sistema operacional através de chamadas de sistema. Vimos que mesmo a operação mais simples de encerramento de um programa requer uma chamada de sistema específica para garantir a estabilidade e segurança do sistema.

A separação entre modo usuário e modo kernel, com o controle de acesso a operações privilegiadas através de chamadas de sistema, é um princípio fundamental na arquitetura de sistemas operacionais modernos, permitindo o equilíbrio entre flexibilidade para os programas de usuário e proteção para o sistema como um todo.