Modo Usuário e Chamadas de Sistema

Relatório da Tarefa 2

May 12, 2025

1 Introdução

Este relatório analisa as diferenças entre implementações em C e Assembly que utilizam chamadas de sistema para escrever mensagens na tela. O objetivo é compreender os mecanismos de transição entre o modo usuário e o modo kernel, bem como a importância das chamadas de sistema para operações aparentemente simples como a escrita em dispositivos de saída.

2 Implementação em C

2.1 Código Fonte

O programa em C utiliza a função `write()` da biblioteca padrão para imprimir uma mensagem no terminal:

```c
#include <unistd.h>

int main() {
    const char *message = "Hello␣from␣C␣program␣using␣write␣syscall\n";
    ssize_t result = write(STDOUT_FILENO, message, 38);

    return 0;
}
```

2.2 Análise com strace

A execução do programa com o comando `strace` revela as chamadas de sistema realizadas durante a execução:

```
execve("./write_c", ["./write_c"], 0x7fff9da3fc60 /* 63 vars */) = 0
brk(NULL)                               = 0x63db6abb3000
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (Arquivo ou diretorio
    inexistente)
openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 4
fstat(4, {st_mode=S_IFREG|0644, st_size=137951, ...}) = 0
mmap(NULL, 137951, PROT_READ, MAP_PRIVATE, 4, 0) = 0x7a1f119da000
close(4)                                = 0
openat(AT_FDCWD, "/usr/lib/libc.so.6", O_RDONLY|O_CLOEXEC) = 4
```

```
read(4, "\177ELF\2\1\1\3\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0‘v\2\0\0\0\0\0"..., 832)
    = 832
pread64(4, "\6\0\0\0\4\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0"...,
    840, 64) = 840
fstat(4, {st_mode=S_IFREG|0755, st_size=2014520, ...}) = 0
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0
    x7a1f119d8000
pread64(4, "\6\0\0\0\4\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0"...,
    840, 64) = 840
mmap(NULL, 2038904, PROT_READ, MAP_PRIVATE|MAP_DENYWRITE, 4, 0) = 0x7a1f117e6000
mmap(0x7a1f1180a000, 1511424, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_FIXED|
    MAP_DENYWRITE, 4, 0x24000) = 0x7a1f1180a000
mmap(0x7a1f1197b000, 323584, PROT_READ, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 4,
    0x195000) = 0x7a1f1197b000
mmap(0x7a1f119ca000, 24576, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|
    MAP_DENYWRITE, 4, 0x1e3000) = 0x7a1f119ca000
mmap(0x7a1f119d0000, 31864, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|
    MAP_ANONYMOUS, -1, 0) = 0x7a1f119d0000
close(4)                                = 0
mmap(NULL, 12288, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0
    x7a1f117e3000
arch_prctl(ARCH_SET_FS, 0x7a1f117e3740) = 0
set_tid_address(0x7a1f117e3a10)     = 28604
set_robust_list(0x7a1f117e3a20, 24) = 0
rseq(0x7a1f117e3680, 0x20, 0, 0x53053053) = 0
mprotect(0x7a1f119ca000, 16384, PROT_READ) = 0
mprotect(0x63db2b0bb000, 4096, PROT_READ) = 0
mprotect(0x7a1f11a37000, 8192, PROT_READ) = 0
prlimit64(0, RLIMIT_STACK, NULL, {rlim_cur=8192*1024, rlim_max=RLIM64_INFINITY})
     = 0
munmap(0x7a1f119da000, 137951)      = 0
write(1, "Hello from C program using write"..., 38) = 38
exit_group(0)                           = ?
+++ exited with 0 +++
```

Podemos observar uma série de chamadas de sistema relacionadas ao carregamento do programa, configuração do ambiente de execução e, finalmente, a chamada `write` que efetivamente imprime a mensagem no terminal.

3 Implementação em Assembly

3.1 Código Fonte

O programa em Assembly implementa diretamente a chamada de sistema `write` sem usar a biblioteca C:

```
.section .data
message:
    .ascii "Hello␣from␣Assembly␣program␣using␣syscall\n"
    .set message_length, . - message

```

```
.section .text
.global _start
_start:
    # syscall write(int fd, const void *buf, size_t count)
    # rax=1 (syscall number for write)
    # rdi=1 (file descriptor: stdout)
    # rsi=pointer to message
    # rdx=message length
    mov $1, %rax        # syscall number for write
    mov $1, %rdi        # file descriptor: stdout (1)
    lea message(%rip), %rsi # pointer to the message
    mov $message_length, %rdx # message length
    syscall             # call kernel

    # syscall exit(int status)
    # rax=60 (syscall number for exit)
    # rdi=0 (exit status: 0)
    mov $60, %rax       # syscall number for exit
    mov $0, %rdi        # exit status 0
    syscall             # call kernel
```

3.2 Análise com strace

A saída do `strace` para o programa em Assembly é muito mais sucinta:

```
execve("./write_asm", ["./write_asm"], 0x7ffd182c4500 /* 63 vars */) = 0
write(1, "Hello from Assembly program usin"..., 42) = 42
exit(0)                                 = ?
+++ exited with 0 +++
```

Observamos apenas três chamadas de sistema: `execve` (executada pelo shell), `write` e `exit`.

4 Análise Comparativa

4.1 Papel da Instrução syscall em Assembly

A instrução `syscall` em Assembly desempenha um papel fundamental na transição entre o modo usuário e o modo kernel. Ela é uma instrução específica que:

1. Interrompe a execução normal do programa em modo usuário.
2. Transfere o controle para o kernel, mudando o processador do modo usuário para o modo kernel.
3. Executa o código do kernel correspondente à chamada de sistema especificada no registrador RAX.
4. Depois de concluída a operação, retorna o controle ao programa em modo usuário, mudando o processador de volta para o modo usuário.

Esta instrução é o mecanismo de baixo nível que permite a comunicação entre programas de usuário e o kernel, fornecendo uma interface controlada para solicitar serviços que exigem privilégios elevados.

4.2 Diferença na Execução entre o Programa em C e em Assembly

As principais diferenças observadas entre os dois programas são:

1. **Número de chamadas de sistema**: O programa em C realiza diversas chamadas de sistema relacionadas à inicialização do ambiente, carregamento de bibliotecas e configuração do processo, enquanto o programa em Assembly faz apenas as chamadas essenciais (write e exit).

2. **Camadas de abstração**: No programa em C, a função `write()` da biblioteca padrão encapsula a chamada de sistema, abstraindo os detalhes de como os argumentos são passados para o kernel. No Assembly, esses detalhes são explicitamente programados, com os argumentos sendo colocados nos registradores apropriados.

3. **Controle direto vs. indireto**: O programa em Assembly tem controle direto sobre a transição para o modo kernel através da instrução `syscall`, enquanto o programa em C delega essa responsabilidade à biblioteca C.

4. **Eficiência**: O programa em Assembly é mais eficiente em termos de número de instruções e chamadas de sistema executadas, já que elimina as camadas intermediárias presentes no programa em C.

O programa em C usa a biblioteca padrão (libc), que fornece uma camada de abstração para as chamadas de sistema, facilitando a programação mas adicionando overhead. Já o programa em Assembly interage diretamente com o kernel, sem camadas intermediárias.

4.3 Necessidade de Chamadas de Sistema para Tarefas Simples

As chamadas de sistema são necessárias para operações simples como imprimir na tela devido à arquitetura de proteção dos sistemas operacionais modernos, que se baseia na separação entre modo usuário e modo kernel. Esta separação é fundamental por vários motivos:

1. **Segurança**: O acesso direto aos dispositivos de hardware (como o terminal) poderia permitir que programas maliciosos ou com erros comprometessem o sistema inteiro.

2. **Estabilidade**: O kernel precisa gerenciar o acesso aos recursos compartilhados de forma ordenada, evitando conflitos entre programas que tentam usar o mesmo recurso simultaneamente.

3. **Virtualização de recursos**: O kernel abstrai os dispositivos físicos, permitindo que múltiplos programas acreditem estar usando o dispositivo exclusivamente.

4. **Compatibilidade**: A interface de chamadas de sistema proporciona uma API estável que permanece consistente, mesmo que o hardware subjacente mude.

5. **Controle de acesso**: O kernel pode implementar políticas de permissão para determinar quais processos podem acessar quais recursos.

Mesmo para uma operação aparentemente simples como escrever na tela, é necessário passar pelo kernel porque o terminal é um recurso compartilhado que precisa ser gerenciado centralmente para evitar colisões de escrita e garantir que a saída de diferentes programas seja coordenada.

4.4 Transição Segura entre Modos de Execução

A transição segura entre os modos de execução é garantida por vários mecanismos:

1. **Instruções privilegiadas**: As instruções `syscall` e similares são as únicas formas permitidas para um programa de usuário solicitar a mudança para o modo kernel. Tentativas de executar instruções privilegiadas diretamente resultam em exceções.

2. **Tabela de chamadas de sistema**: O kernel mantém uma tabela de funções permitidas que podem ser invocadas via chamadas de sistema, limitando o que o código de usuário pode solicitar ao kernel.

3. **Validação de parâmetros**: O kernel valida todos os parâmetros fornecidos pelo código de usuário antes de executar qualquer operação, evitando manipulações maliciosas.

4. **Mapeamento de memória**: O hardware de gerenciamento de memória (MMU) protege a memória do kernel, impedindo o acesso direto pelo código em modo usuário.

5. **Níveis de privilégio do processador**: Processadores modernos suportam diferentes níveis de privilégio (anéis de proteção), onde o kernel opera no nível mais privilegiado (Ring 0) e os aplicativos de usuário em um nível menos privilegiado (Ring 3).

4.5 Implicações do Acesso Direto aos Dispositivos

Se o modo usuário tivesse acesso direto aos dispositivos, diversas consequências negativas surgiriam:

1. **Corrupção do sistema**: Programas mal escritos ou maliciosos poderiam modificar configurações críticas de hardware, corrompendo o funcionamento de todo o sistema.

2. **Conflitos de acesso**: Múltiplos programas tentando controlar o mesmo dispositivo simultaneamente resultariam em comportamentos imprevisíveis e possível travamento do sistema.

3. **Ausência de abstração**: Cada programa precisaria implementar seus próprios drivers para cada dispositivo, tornando o desenvolvimento extremamente complexo e propenso a erros.

4. **Escalonamento de privilégios**: Programas maliciosos poderiam facilmente obter controle total sobre o sistema, comprometendo a segurança e a privacidade dos dados.

5. **Instabilidade**: A falta de coordenação central levaria a frequentes travamentos e comportamentos inesperados, uma vez que os programas poderiam interferir uns nos outros.

6. **Monitoramento complexo**: Seria difícil para o sistema operacional monitorar e registrar o uso de recursos, dificultando a depuração e a resolução de problemas.

5 Conclusão

A análise dos programas em C e Assembly que utilizam chamadas de sistema para escrever no terminal revela a importância da separação entre modo usuário e modo kernel nos sistemas operacionais modernos. A instrução `syscall` em Assembly representa o mecanismo fundamental que permite a transição controlada entre esses modos, garantindo que operações privilegiadas sejam executadas com segurança.

A diferença na forma de execução entre os programas destaca como as bibliotecas padrão proporcionam abstrações úteis, embora ao custo de alguma eficiência. O programa em Assembly, por outro lado, demonstra como é possível interagir diretamente com o kernel sem camadas intermediárias, mas exigindo um conhecimento mais detalhado sobre os mecanismos de baixo nível.

A necessidade de chamadas de sistema para tarefas simples como imprimir na tela é um exemplo claro de como a arquitetura dos sistemas operacionais prioriza a segurança, estabilidade e compartilhamento de recursos sobre a eficiência bruta. Essa abordagem, que pode parecer excessiva para operações triviais, é crucial para o funcionamento confiável e seguro dos sistemas computacionais modernos, onde múltiplos programas compartilham os mesmos recursos físicos.