

Modo Usuário e Chamadas de Sistema

Relatório da Tarefa 2

May 12, 2025

1 Introdução

Este relatório analisa a interação entre programas e o sistema operacional através de chamadas de sistema, comparando implementações em C e Assembly que utilizam a chamada de sistema `write` para escrever na tela. As chamadas de sistema são o mecanismo fundamental que permite a transição segura entre o código de usuário (com privilégios limitados) e o código do kernel (com acesso completo ao hardware).

2 Implementação em C

O programa em C utiliza a função `write()` da biblioteca padrão:

```
1 #include <unistd.h>
2 int main() {
3     const char *msg = "Hello_from_C_program_using_write_syscall\n";
4     write(STDOUT_FILENO, msg, 38);
5     return 0;
6 }
```

Diretrizes à execução:

```
1 # Compilacao e execucao com strace (versao C)
2 $ gcc -static -o write_c write_c.c # Compilacao estatica
3 $ ./write_c                        # Execucao normal
4 $ strace ./write_c                 # Execucao com rastreamento
```

Análise com `strace` (compilação estática):

```
1 execve("./write_c", ["/write_c"], 0x7ffe070c1650 /* 63 vars */) = 0
2 brk(NULL) = 0x383333000
3 brk(0x383333d40) = 0x383333d40
4 arch_prctl(ARCH_SET_FS, 0x383333c0) = 0
5 set_tid_address(0x383333690) = 29421
6 set_robust_list(0x3833336a0, 24) = 0
7 mprotect(0x4a2000, 20480, PROT_READ) = 0
8 write(1, "Hello from C program using write"... , 38) = 38
9 exit_group(0) = ?
10 +++ exited with 0 +++
```

3 Implementação em Assembly

O programa em Assembly implementa diretamente a chamada de sistema:

```
1 .section .data
2 msg: .ascii "Hello_from_Assembly_program_using_syscall\n"
3     .set len, . - msg
4 .section .text
5 .global _start
6 _start:
7     mov $1, %rax    # write syscall
8     mov $1, %rdi    # stdout
9     lea msg(%rip), %rsi # msg ptr
10    mov $len, %rdx   # length
11    syscall
12    mov $60, %rax    # exit syscall
13    xor %rdi, %rdi   # status 0
14    syscall
```

Diretrizes à execução:

```
1 # Montagem, ligacao e execucao (versao Assembly)
2 $ as -o write_asm.o write_asm.s # Montagem do codigo
3 $ ld -o write_asm write_asm.o   # Ligacao do objeto
4 $ ./write_asm                   # Execucao normal
5 $ strace ./write_asm            # Execucao com rastreamento
```

Análise com `strace`:

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

4 Análise Comparativa

4.1 Papel da Instrução syscall em Assembly

A instrução `syscall` em arquiteturas x86-64 é responsável por:

- Interromper a execução normal do programa em modo usuário
- Salvar automaticamente o ponteiro de instrução (RIP) e flags (RFLAGS) na pilha do kernel
- Transferir o controle para o kernel, mudando para o modo kernel (Ring 0)
- Carregar o endereço da rotina de tratamento de `syscall` do kernel a partir do MSR `LSTAR`
- Executar o código do kernel correspondente a chamada especificada no registrador `RAX`
- Retornar o controle ao programa via `sysret`, voltando para o modo usuário (Ring 3)

Durante a transição do modo usuário para o kernel, ocorre uma sequência detalhada de operações:

- **Salvamento automático:** O processador salva RIP, RFLAGS, RSP, CS e SS
- **Seletores de segmento:** CS e SS recebem valores de kernel (0x08 e 0x10 respectivamente)
- **Troca de pilha:** RSP é atualizado para apontar para a pilha do kernel desse processo
- **Desabilitação de interrupções:** Interrupções são temporariamente desabilitadas
- **Chaveamento de contexto de memória:** CR3 pode mudar para o espaço de endereçamento do kernel
- **Atualização de estruturas:** A `task_struct` é marcada como `in_syscall=1`

Esta instrução é o mecanismo de baixo nível que permite a comunicação controlada entre programas de usuário e o kernel.

4.2 Diferenças entre os Programas

- **Numero de syscalls:** C faz mais chamadas relacionadas a inicialização; Assembly usa apenas `write` e `exit`

Análise detalhada das chamadas:

- **execve:** Carrega o programa, substituindo o processo atual.
 - **brk:** Gerencia o heap, alocando/expandindo memória.
 - **arch_prctl:** Configura registrador FS para thread-local storage.
 - **set_tid_address:** Define endereço para ID da thread.
 - **set_robust_list:** Registra lista de mutexes do processo.
 - **mprotect:** Marca áreas de memória como somente leitura.
 - **write:** Nossa chamada proposital - escreve no stdout.
 - **exit_group:** Termina o processo com código 0.
- **Abstração:** C encapsula detalhes via `write()`; Assembly programa explicitamente registradores/syscall
 - **Controle:** Assembly controla diretamente a transição; C delega a `libc`
 - **Eficiência:** Assembly elimina overhead de camadas intermediárias

4.3 Necessidade de Chamadas de Sistema

Chamadas de sistema são necessárias devido à arquitetura de proteção entre modo usuário e kernel:

- **Segurança:** Impede acesso direto ao hardware/áreas críticas
- **Estabilidade:** Gerencia ordenadamente recursos compartilhados
- **Virtualização:** Abstrai dispositivos para múltiplos programas
- **Controle:** Define políticas de acesso a recursos

Para escrever na tela, o programa precisa acessar o terminal, que é um recurso compartilhado gerenciado pelo sistema operacional. Sem o controle centralizado via chamadas de sistema, haveria conflitos de acesso e possíveis instabilidades.

4.4 Transição Segura entre Modos

A transicao segura entre modos de execucao e garantida por:

1. **Instrucoes privilegiadas:** Apenas instrucoes especificas como `syscall` podem solicitar a mudanca para o modo kernel.
2. **Tabela de chamadas de sistema:** O kernel mantem uma tabela de funcoes permitidas (`sys_call_table` no Linux) que podem ser invocadas via chamadas de sistema.
3. **Validacao de parametros:** O kernel valida todos os parametros antes de executar operacoes.
4. **Hardware de protecao:** O MMU (Memory Management Unit) e os niveis de privilegio do processador (rings) impedem acesso nao autorizado.
5. **Troca de pilhas:** O kernel usa uma pilha separada para evitar que o modo usuario manipule a pilha do kernel.

Estado do processo e visao interna da transicao:

- **Registradores:** Todos os registradores de usuário são salvos na estrutura `pt_regs` na pilha do kernel
- **Estruturas de controle:**
 - `task_struct`: Contém todo o estado do processo (PID, prioridade, CPU, etc.)
 - `thread_info`: Armazena informações específicas da thread atual
 - Campo `flags`: Bits indicam `TIF_SYSCALL_TRACE` se strace estiver ativo
- **Memoria:** Espaco de usuario permanece mapeado mas protegido (kernel pode acessar memoria do usuario atraves de funcoes como `copy_from_user` com verificacoes)
- **Fluxo de execucao:**
 - Usuário → `syscall` → `entry_SYSCALL_64` → `do_syscall_64` → `sys_write`
 - Retorno: `sysret` restaura contexto armazenado, devolvendo controle ao userspace

Se o modo usuário tivesse acesso direto aos dispositivos:

1. Programas maliciosos poderiam modificar configuracoes criticas de hardware
2. Multiplos programas tentando controlar o mesmo dispositivo causariam conflitos
3. Nao haveria abstracao de hardware, complicando o desenvolvimento
4. Programas poderiam acessar dados sensiveis de outros programas
5. A estabilidade do sistema seria comprometida pela falta de coordenacao

5 Conclusão

A analise das implementacoes em C e Assembly demonstra a importancia das syscalls como interface controlada entre userspace e kernel. Esta separacao com transicoes controladas via `syscall` e fundamental para a seguranca e estabilidade dos sistemas modernos.

As transicoes envolvem salvamento e restauracao precisos do estado de execucao, incluindo:

- Conteudo dos registradores (RIP, RFLAGS, registradores de proposito geral)
- Ponteiro da pilha e seletores de segmento
- Estado de privilegio (mudanca entre Ring 3 e Ring 0)
- Atualizacao das estruturas de controle do kernel (`task_struct`, `thread_info`)
- Configuracao de protecoes de memoria para prevenir acesso nao autorizado

O processador x86-64 implementa otimizacoes especificas para syscalls, como as instrucoes dedicadas `syscall/sysret` e os registradores MSR (Model Specific Registers) que armazenam os pontos de entrada/saida do kernel. Esse hardware especializado torna as transicoes entre modo usuario e kernel muito mais eficientes do que interrupcoes genericas.

Todo esse mecanismo, mesmo parecendo excessivo para operacoes simples como escrever na tela, e essencial para manter o sistema operacional protegido e estavel. Ele permite que multiplos programas compartilhem recursos de forma segura e coordenada, balanceando facilidade de programacao (abordagem C) com eficiencia e controle preciso (abordagem Assembly).