Relatório de Atividade Prática
Aula 02 – Sistemas Operacionais (DCA3505)

Universidade Federal do Rio Grande do Norte

Gabriel S. N. Neto

21 de agosto de 2025

Exemplo de programa mínimo em Assembly

Escreva um código mínimo em linguagem de montagem que contenha apenas uma instrução para atribuir um valor a um registrador da CPU. Compile-o com flags apropriadas para evitar que o compilador adicione bibliotecas ou instruções extras. Ao executar o programa, você deverá observar uma falha de segmentação (*segmentation fault*).

```
.intel_syntax noprefix
.global _start

_start:
    mov eax, 5
```

Este código ASM possui apenas uma instrução: atribui o valor 5 ao registrador `EAX`.

Compilação

```
gcc -nostdlib -nostartfiles -static -Wl,-e,_start \
    -o prog1-intel codigo1-intel-sintaxe.s
```

Opções utilizadas:

- `-nostdlib` – não usar a biblioteca padrão
- `-nostartfiles` – não usar arquivos de inicialização padrão
- `-static` – gerar binário estático (sem dependências dinâmicas)
- `-Wl,-e,_start` – informar ao linker que o ponto de entrada é `_start`
- `-o prog1-intel` – nome do executável gerado
- `codigo1-intel-sintaxe.s` – arquivo fonte em assembly

Execução e falha

Ao executar o programa com `./prog1-intel`, a saída é: `Segmentation fault (core dumped)`.

Isso ocorre porque o programa não realiza uma chamada de sistema para encerrar sua execução (`exit`), de modo que o processador continua a executar instruções inválidas. Isso resulta em uma exceção de proteção, tratada pelo sistema operacional como falha de segmentação.

Código corrigido

```
.intel_syntax noprefix
.global _start

_start:
    mov rbx, 5      # coloca o valor 5 em RBX
    mov rax, 60     # cohdigo da syscall "exit" no Linux
    mov rdi, 0      # cohigo de retorno (exit code 0)
    syscall         # invoca a chamada de sistema
```

Explicação:

- `mov rbx, 5` – coloca o valor 5 no registrador `RBX`.
- `mov rax, 60` – define o número da chamada de sistema `exit`.
- `mov rdi, 0` – define o código de saída (retorno 0).
- `syscall` – executa a chamada de sistema, encerrando o programa.

Usando strace

```
gsnn@desktop-gsnn:~/so/aula02$ strace ./prog1-intel
execve("./prog1-intel", ["./prog1-intel"], 0x7ffe9223c110 /* 39 vars */) = 0
--- SIGSEGV {si_signo=SIGSEGV, si_code=SEGV_MAPERR, si_addr=0x5} ---
+++ killed by SIGSEGV (core dumped) +++
Segmentation fault (core dumped)

gsnn@desktop-gsnn:~/so/aula02$ strace ./prog2-intel
execve("./prog2-intel", ["./prog2-intel"], 0x7ffdb9817f20 /* 39 vars */) = 0
exit(0)                                 = ?
+++ exited with 0 +++
```

A saída do `strace` evidencia a diferença entre os dois programas: no primeiro caso, após a atribuição ao registrador, não há chamada de sistema para finalizar o processo, o que faz com que o programa termine em `segmentation fault`. Já no segundo, observa-se a execução explícita da chamada `exit(0)`, permitindo que o processo seja encerrado de forma correta e controlada pelo sistema operacional.

Discussão

Por que foi necessário incluir uma chamada de sistema para encerrar corretamente o programa?

A inclusão da chamada de sistema é necessária porque apenas o kernel tem autoridade para liberar os recursos do processo e marcar sua finalização. Sem isso, a CPU tenta executar instruções inválidas, resultando em falha de segmentação.

Quais seriam as implicações se programas de usuário pudessem ser executados livremente?

Se programas de usuário não dependessem do sistema operacional, haveria instabilidade, travamentos e falhas de segurança. A ausência de controle sobre memória e recursos permitiria que um processo comprometesse outros, além de colocar em risco o próprio kernel.