

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*).

```
1 .intel_syntax noprefix
2 .global _start
3
4 _start:
5     mov eax, 5
```

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

Compilação

```
1 gcc -nostdlib -nostartfiles -static -Wl,-e,_start \
2     -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`), o que faz com que o registrador de instrução (PC/RIP) continue apontando para a próxima instrução, fazendo com que o processador execute instruções que provavelmente serão inválidas — instruções que ou não existem, ou que existem, mas muito provavelmente tentarão acessar ou modificar alguma área de memória não destinada ao processo. Isso gera uma interrupção de hardware, que é tratada pelo sistema operacional como falha de segmentação.

Código corrigido

```
1 .intel_syntax noprefix
2 .global _start
3
4 _start:
5     mov rax, 5          # coloca o valor 5 em RBX
6     mov rax, 60         # código da syscall "exit" no Linux
7     mov rdi, 0          # código de retorno (exit code 0)
8     syscall             # invoca a chamada de sistema
```

Explicação:

- `mov rax, 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

```
1 gsnn@desktop-gsnn:~/so/aula02$ strace ./prog1-intel
2 execve("./prog1-intel", ["/prog1-intel"], 0x7ffe9223c110 /* 39 vars
   */) = 0
3 --- SIGSEGV {si_signo=SIGSEGV, si_code=SEGV_MAPERR, si_addr=0x5} ---
4 +++ killed by SIGSEGV (core dumped) +++
5 Segmentation fault (core dumped)
6
7 gsnn@desktop-gsnn:~/so/aula02$ strace ./prog2-intel
8 execve("./prog2-intel", ["/prog2-intel"], 0x7ffdb9817f20 /* 39 vars
   */) = 0
9 exit(0)                                     = ?
10 +++ 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.