

Software Básico

Aula #03

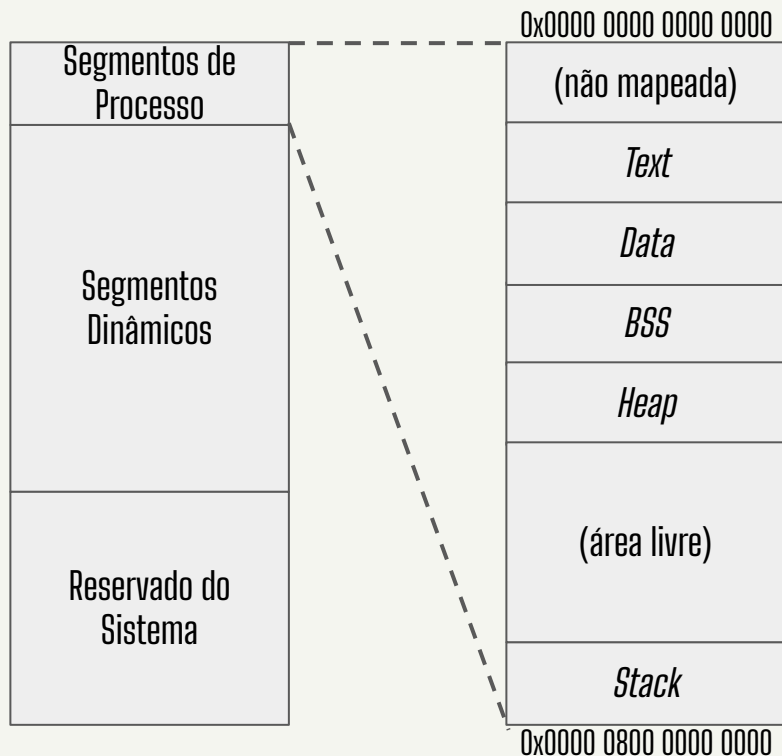
Código e Dados: Estrutura e Expressões Aritméticas

Como funciona as seções de código e dados? Como identificar programas em *assembly* (AMD64)? Como funcionam expressões aritméticas?

Ciência da Computação – BCC2 – 2023/02

Prof. Vinícius Fülber Garcia

Relembrando



Seções do segmento de processo:

- *Text*: código
- *Data*: dados globais inicializados
- *BSS*: dados não inicializados
- *Heap*: variáveis dinâmicas
- *Stack*: pilha de execução

Código e Dados: Escopo



O foco desta aula, e das próximas também, será explorar a seção de código (*text*) e dados (*data*).

- Atribuições e variáveis globais
- Expressões aritméticas
- Estruturas de repetição
- Estruturas condicionais
- Vetores

Tradução de programas em C para *assembly* (AMD64), gerando aplicações com funcionalidade equivalente.

Estrutura *Assembly* (AMD64)

```
int main(){  
    return 0;  
}
```

```
.section .text  
.global _start  
_start:  
    movq $60, %rax  
    movq $0, %rdi  
    syscall
```

Lembram do programa mais básico possível que criamos na última aula?

Vamos voltar a ele e ver uma **tradução deste (feita a mão) para código *assembly***.

Vamos analisar parte a parte!

Estrutura *Assembly* (AMD64)

```
int main(){  
    return 0;  
}
```

```
.section .text  
.global _start  
_start:  
    movq $60, %rax  
    movq $0, %rdi  
    syscall
```

DEFINIÇÃO DE SEÇÃO

As seções de um programa *assembly* são definidas utilizando a diretiva “.section” seguido de “.nome” (da seção), separados por espaço.

A seção *text* é obrigatória!

Estrutura *Assembly* (AMD64)

DEFINIÇÃO DE SÍMBOLOS GLOBAIS

```
int main(){  
    return 0;  
}
```

```
.section .text  
.global _start  
_start:  
    movq $60, %rax  
    movq $0, %rdi  
    syscall
```

Símbolos globais são referências em um código *assembly* e são definidos pela diretiva “.global”. Tais símbolos são reconhecidos em diferentes partes do programa *assembly*.

O símbolo *_start* é obrigatoriamente global!

Estrutura *Assembly* (AMD64)

```
int main(){  
    return 0;  
}
```

```
.section .text  
.global _start  
_start:  
    movq $60, %rax  
    movq $0, %rdi  
    syscall
```

DEFINIÇÃO DE REFERÊNCIA INICIAL

O símbolo `__start` define o ponto de início de execução do programa. Ou seja, é um símbolo único e necessário para a execução de um programa escrito em *assembly*.

O símbolo `__start` é obrigatório!

Estrutura *Assembly* (AMD64)

```
int main(){  
    return 0;  
}
```

```
.section .text  
.global _start  
_start:  
    movq $60, %rax  
    movq $0, %rdi  
    syscall
```

A INSTRUÇÃO *mov*

A instrução *mov* serve para mover um valor (primeiro argumento) para um local (segundo argumento).

- **movq**: valor com 64 bits
- **movl**: valor com 32 bits
- **movw**: valor com 16 bits
- **movb**: valor com 8 bits

Estrutura *Assembly* (AMD64)

```
int main(){  
    return 0;  
}
```

```
.section .text  
.global _start  
_start:  
    movq $60, %rax  
    movq $0, %rdi  
    syscall
```

O PRIMEIRO ARGUMENTO DO *mov*

O primeiro argumento indica o valor o qual deve ser copiado. Este pode ser uma constante (imediato - \$), estar contido em um registrador (reg. - %) ou estar contido em uma posição de memória (direto - 0x0.. ou rótulo de variável).

Estrutura *Assembly* (AMD64)

```
int main(){  
    return 0;  
}
```

```
.section .text  
.global _start  
_start:  
    movq $60, %rax  
    movq $0, %rdi  
    syscall
```

O SEGUNDO ARGUMENTO DO *mov*

O segundo argumento define o destino para onde o valor (primeiro argumento) deve ser copiado. Este pode ser um registrador (reg. - %) ou posição de memória (direto - 0x0.. ou rótulo de variável).

Estrutura *Assembly* (AMD64)

IMPORTANTE!!

Uma instrução não pode realizar dois acessos à memória! Apenas um acesso pode ser previsto por instrução.

movq 0x00.. 0x01.. (não pode!)

Nesse caso, você pode, por exemplo, utilizar um registrador com sentinela do valor, executando duas operações de movimentação.

Estrutura *Assembly* (AMD64)

INTERESSANTE!!

A maioria das operações *assembly* do AMD64 tem dois operandos.

O primeiro operando é tipicamente uma fonte de dados e o segundo o destino do resultado da operação (podendo ou não ser uma fonte de dados também).

Estrutura *Assembly* (AMD64)

NECESSÁRIO!!

Dentro da nossa arquitetura, temos disponíveis dezesseis (16) registradores de propósito geral.

Reg.	%r0	%r1	%r2	%r3	%r4	%r5	%r6	%r7	%r8	%r9	%r10	%r11	%r12	%r13	%r14	%r15
Nome	%rax	%rcx	%rdx	%rbx	%rsp	%rbp	%rsi	%rdi	%r8	%r9	%r10	%r11	%r12	%r13	%r14	%r15

Estrutura *Assembly* (AMD64)

CURIOSO!!

Existe uma progressão histórica na disponibilidade e tamanho dos registradores.

ANO	MODELO	REGISTRADORES	TAMANHO	QUANTIDADE
1978	8086	AX, BX, ..., IP	16 bits	4-16 (8-8)
1982	80186	AX, BX, ..., IP	16 bits	4-16 (8-8)
1982	80286	AX, BX, ..., IP	16 bits	4-16 (8-8)
1985	80386	EAX, EBX, ..., EIP	32 bits	8-32
1989	80486	EAX, EBX, ..., EIP	32 bits	8-32
1993	Pentium	EAX, EBX, ..., EIP	32 bits	8-32
2004	Pentium	RAX, RBX, ..., RIP	64 bits	8-64
2008	I3, i5, i7	RAX, RBX, ..., RIP	64 bits	16-64

Estrutura *Assembly* (AMD64)

```
int main(){  
    return 0;  
}
```

```
.section .text  
.global _start  
_start:  
    movq $60, %rax  
    movq $0, %rdi  
    syscall
```

A PRIMEIRA OPERAÇÃO

Conforme o que já estudamos, podemos interpretar sintaticamente a primeira operação do nosso programa.

O que ela faz?

Estrutura *Assembly* (AMD64)

```
int main(){  
    return 0;  
}
```

```
.section .text  
.global _start  
_start:  
    movq $60, %rax  
    movq $0, %rdi  
    syscall
```

A PRIMEIRA OPERAÇÃO

Mas a grande questão é: **o que a operação significa?**

Quando o sistema lê o valor sessenta (60 - dec.) em %rax, o mesmo finaliza o processo.

Estrutura *Assembly* (AMD64)

```
int main(){  
    return 0;  
}
```

```
.section .text  
.global _start  
_start:  
    movq $60, %rax  
    movq $0, %rdi  
    syscall
```

A SEGUNDA OPERAÇÃO

A segunda operação também se dá pela instrução *mov*.

Quando finalizado o processo, o registrador `%rdi` armazena o valor de retorno do mesmo.

Estrutura *Assembly* (AMD64)

```
int main(){  
    return 0;  
}
```

```
.section .text  
.global _start  
_start:  
    movq $60, %rax  
    movq $0, %rdi  
    syscall
```

A TERCEIRA OPERAÇÃO

A terceira operação consiste em uma nova instrução: *syscall*

Em suma, essa instrução chama o sistema, que lê e aplica a operação que informamos em %rax.

Um Resumo: *Syscall*

Mas vamos entender um pouco melhor...

Como vimos anteriormente, um processo recebe um espaço de memória, com endereços virtuais, para trabalhar.

O processo não pode, de maneira alguma, acessar recursos fora do seu espaço particular.

Para utilizar tais recursos, o processo usa um intermediário: o SO!

Um Resumo: *Syscall*

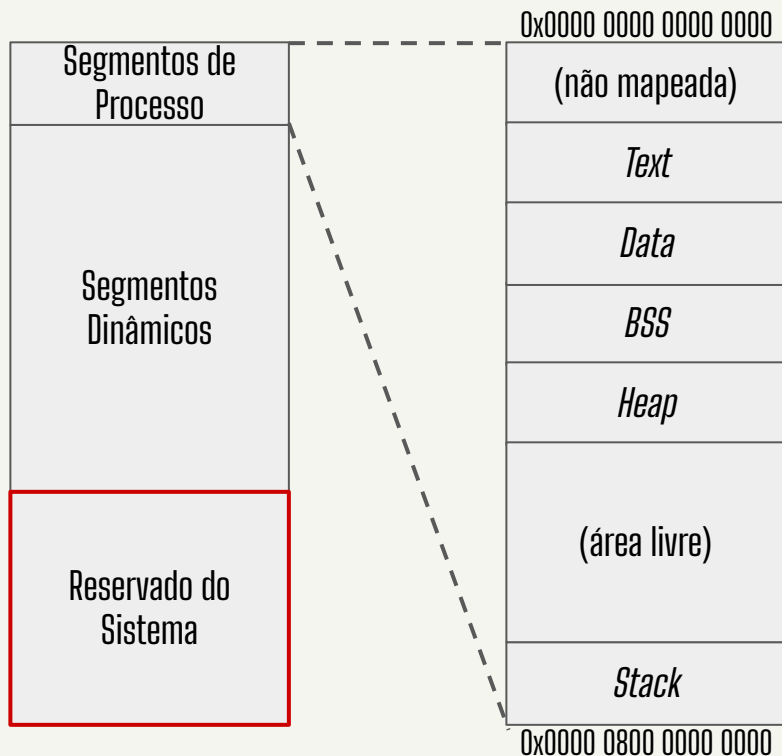
Exemplos de recursos externos:

- Finalização do processo
- Dados da memória secundária
- Operações em dispositivos externos

Existem protocolos específicos para cada tipo de operação de sistema: vamos analisar eles nas aulas futuras.

Por enquanto: a *syscall* serve para chamar o sistema!

Um Resumo: *Syscall*



Mas o que “chamar o sistema” significa na prática?

Todo o processo tem uma “janela” para o sistema operacional.

Desvio de fluxo de execução e, se precisar, retorno.

Do *Assembly* ao Executável

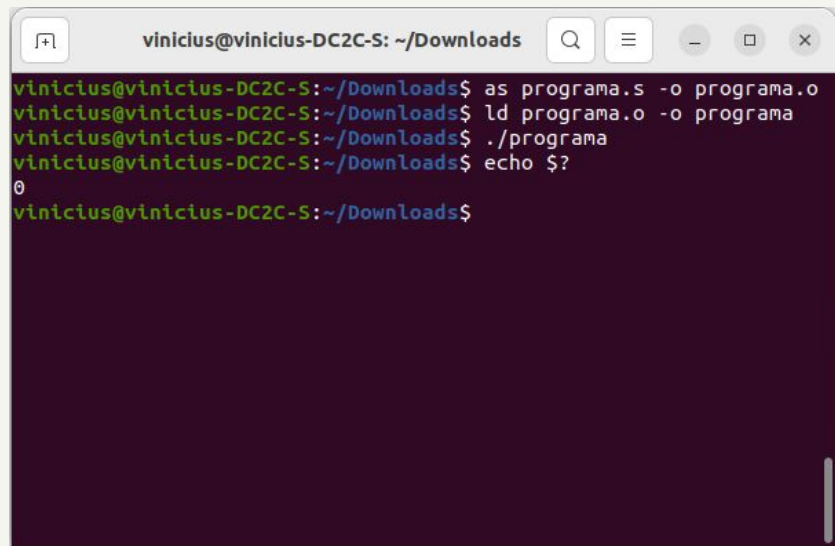
Legal! Já compreendemos o nosso programa escrito em *assembly*.

Mas como transformar ele em um executável?

Precisaremos utilizar um **montador** para gerar códigos-objeto e, em seguida, executar um **ligador** sobre os objetos para criarmos um executável.

No contexto do gcc, nosso montador se chama “**as**” e nosso ligador se chamada “**ld**”.

Do *Assembly* ao Executável



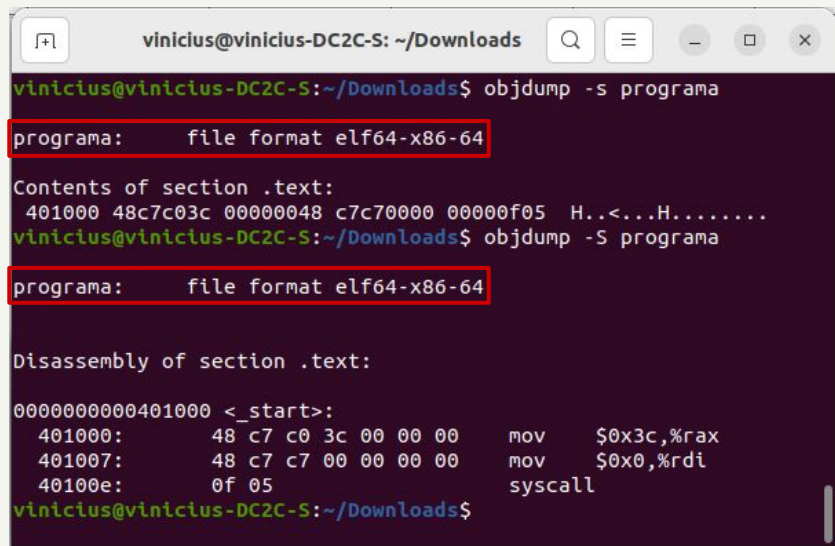
```
vinicius@vinicius-DC2C-S: ~/Downloads
vinicius@vinicius-DC2C-S:~/Downloads$ as programa.s -o programa.o
vinicius@vinicius-DC2C-S:~/Downloads$ ld programa.o -o programa
vinicius@vinicius-DC2C-S:~/Downloads$ ./programa
0
vinicius@vinicius-DC2C-S:~/Downloads$
```

Vamos considerar que salvamos o nosso programa escrito em assembly como “**programa.s**”.

Primeiro, geramos o arquivo de código-objeto “**programa.o**” usando o “as”.

Então, geramos o executável “**programa**” com o “ld”.

Do *Assembly* ao Executável



```
vinicius@vinicius-DC2C-S: ~/Downloads
vinicius@vinicius-DC2C-S:~/Downloads$ objdump -s programa
programa:      file format elf64-x86-64

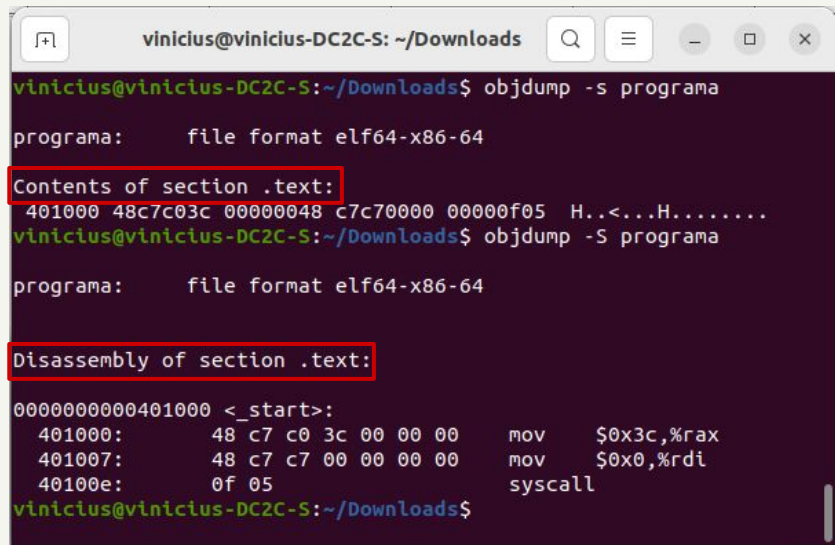
Contents of section .text:
 401000 48c7c03c 00000048 c7c70000 00000f05 H..<...H.....
vinicius@vinicius-DC2C-S:~/Downloads$ objdump -S programa
programa:      file format elf64-x86-64

Disassembly of section .text:
0000000000401000 <_start>:
 401000:  48 c7 c0 3c 00 00 00    mov     $0x3c,%rax
 401007:  48 c7 c7 00 00 00 00    mov     $0x0,%rdi
 40100e:  0f 05                  syscall
vinicius@vinicius-DC2C-S:~/Downloads$
```

E no arquivo executável, como essas operações se comportam em binário?

Analizando com *objdump* vamos primeiros observar que de fato temos um **ELF para arquitetura AMD64**.

Do *Assembly* ao Executável



```
vinicius@vinicius-DC2C-S: ~/Downloads
vinicius@vinicius-DC2C-S:~/Downloads$ objdump -s programa
programa:      file format elf64-x86-64

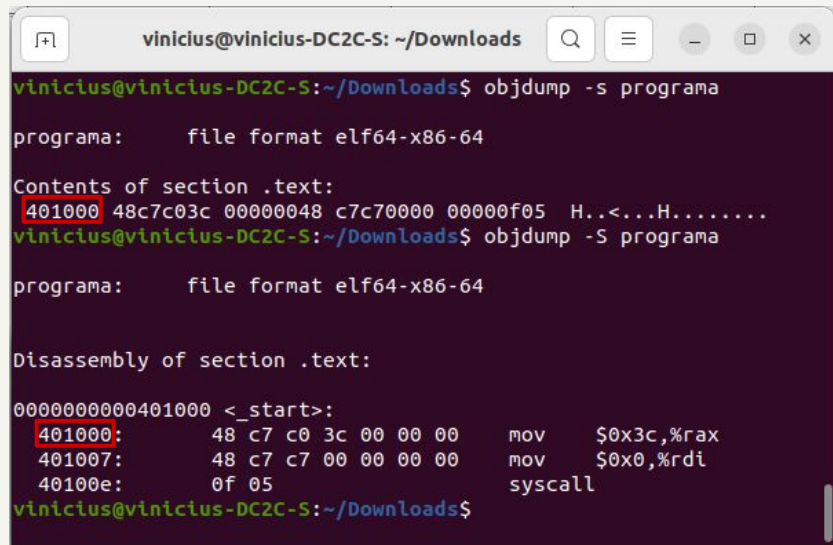
Contents of section .text:
 401000 48c7c03c 00000048 c7c70000 00000f05  H..<...H.....
vinicius@vinicius-DC2C-S:~/Downloads$ objdump -S programa
programa:      file format elf64-x86-64

Disassembly of section .text:
0000000000401000 <_start>:
 401000:    48 c7 c0 3c 00 00 00    mov     $0x3c,%rax
 401007:    48 c7 c7 00 00 00 00    mov     $0x0,%rdi
 40100e:    0f 05                  syscall
vinicius@vinicius-DC2C-S:~/Downloads$
```

Em seguida, podemos observar a **seção *text*** no arquivo.

Na primeira visualização (-s), temos apenas os bytes em hexadecimal; na segunda (-S), temos as operações em *assembly*.

Do *Assembly* ao Executável



```
vinicius@vinicius-DC2C-S: ~/Downloads
vinicius@vinicius-DC2C-S:~/Downloads$ objdump -s programa
programa:      file format elf64-x86-64

Contents of section .text:
401000 48c7c03c 00000048 c7c70000 00000f05 H..<...H.....
vinicius@vinicius-DC2C-S:~/Downloads$ objdump -S programa
programa:      file format elf64-x86-64

Disassembly of section .text:
0000000000401000 <_start>:
401000:  48 c7 c0 3c 00 00 00    mov     $0x3c,%rax
401007:  48 c7 c7 00 00 00 00    mov     $0x0,%rdi
40100e:  0f 05                  syscall
vinicius@vinicius-DC2C-S:~/Downloads$
```

Então, encontramos a posição de memória inicial da seção: **401000**

Note a diferença, em termos da exibição das posições de memória, entre as duas visualizações.

Do *Assembly* ao Executável

```
vinicius@vinicius-DC2C-S: ~/Downloads
vinicius@vinicius-DC2C-S:~/Downloads$ objdump -s programa
programa:      file format elf64-x86-64

Contents of section .text:
 401000 48c7c03c 00000048 c7c70000 00000f05 H..<...H.....
vinicius@vinicius-DC2C-S:~/Downloads$ objdump -S programa
programa:      file format elf64-x86-64

Disassembly of section .text:
0000000000401000 <start>:
 401000: 48 c7 c0 3c 00 00 00    mov     $0x3c,%rax
 401007: 48 c7 c7 00 00 00 00    mov     $0x0,%rdi
 40100e: 0f 05                  syscall
vinicius@vinicius-DC2C-S:~/Downloads$
```

E então podemos identificar a primeira instrução:

- 48: código para reg. 64
- C7 C0: move to %rax
- 3c 00 00 00: constante 60

Lembre-se, a arquitetura AMD64 é *little endian*!

Do *Assembly* ao Executável

```
vinicius@vinicius-DC2C-S: ~/Downloads
vinicius@vinicius-DC2C-S:~/Downloads$ objdump -s programa

programa:      file format elf64-x86-64

Contents of section .text:
 401000 48c7c03c 00000048 c7c70000 00000f05  H..<...H.....
vinicius@vinicius-DC2C-S:~/Downloads$ objdump -S programa

programa:      file format elf64-x86-64

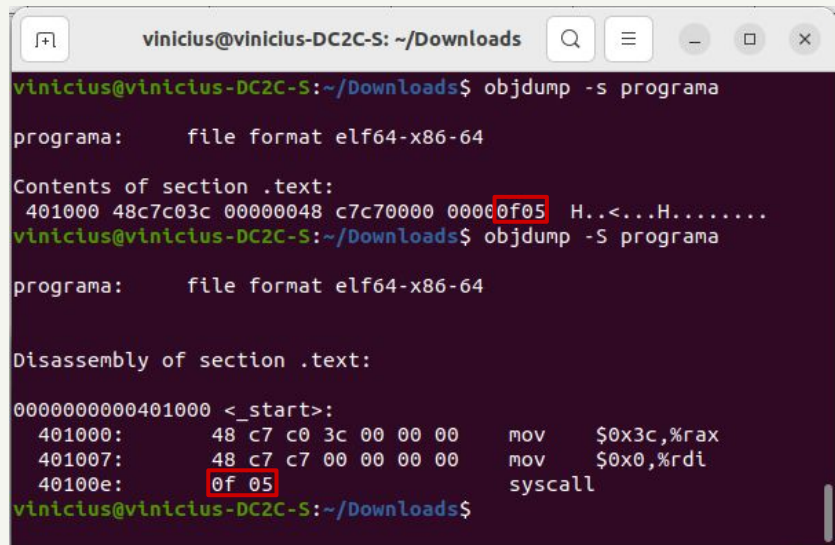
Disassembly of section .text:

0000000000401000 <_start>:
401000:  48 c7 c0 3c 00 00 00    mov     $0x3c,%rax
401007:  48 c7 c7 00 00 00 00    mov     $0x0,%rdi
40100e:  0f 05                  syscall
vinicius@vinicius-DC2C-S:~/Downloads$
```

Em seguida, a segunda instrução:

- 48: código para reg. 64
- C7 C7: move to %rdi
- 00 00 00 00: constante 0

Do *Assembly* ao Executável



```
vinicius@vinicius-DC2C-S: ~/Downloads
vinicius@vinicius-DC2C-S:~/Downloads$ objdump -s programa
programa:      file format elf64-x86-64

Contents of section .text:
 401000 48c7c03c 00000048 c7c70000 00000f05 H..<...H.....
vinicius@vinicius-DC2C-S:~/Downloads$ objdump -S programa
programa:      file format elf64-x86-64

Disassembly of section .text:
0000000000401000 <_start>:
401000:  48 c7 c0 3c 00 00 00    mov     $0x3c,%rax
401007:  48 c7 c7 00 00 00 00    mov     $0x0,%rdi
40100e:  0f 05                  syscall
vinicius@vinicius-DC2C-S:~/Downloads$
```

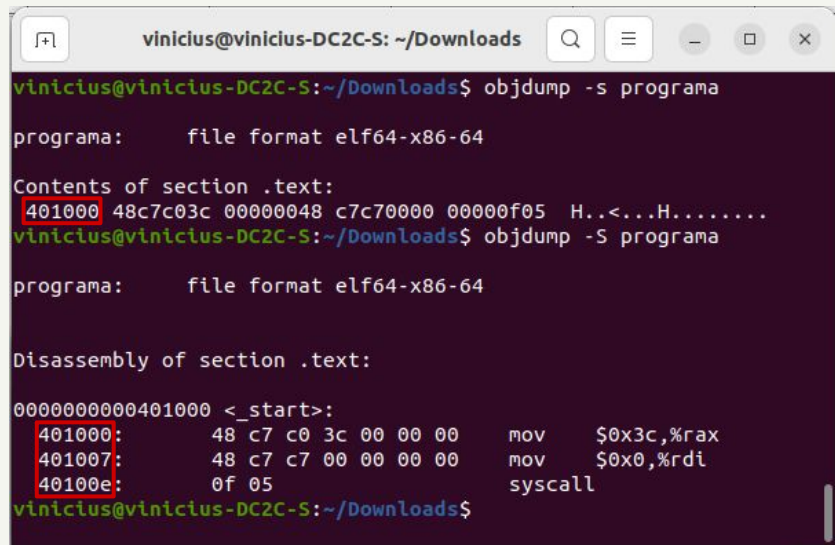
Por fim, a terceira instrução:

- 0f 05: código para *syscall*

Opa, apenas dois bytes nesta instrução?

Sim, estamos lidando com uma arquitetura **CISC**!

Do *Assembly* ao Executável



```
vinicius@vinicius-DC2C-S: ~/Downloads
vinicius@vinicius-DC2C-S:~/Downloads$ objdump -s programa
programa:      file format elf64-x86-64

Contents of section .text:
401000 48c7c03c 00000048 c7c70000 00000f05  H...<...H.....
vinicius@vinicius-DC2C-S:~/Downloads$ objdump -S programa
programa:      file format elf64-x86-64

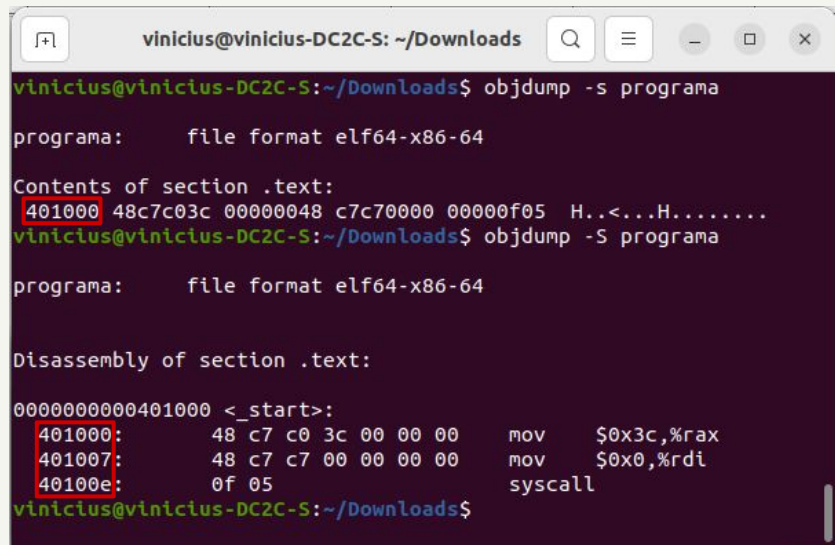
Disassembly of section .text:
0000000000401000 <_start>:
401000:  48 c7 c0 3c 00 00 00    mov     $0x3c,%rax
401007:  48 c7 c7 00 00 00 00    mov     $0x0,%rdi
40100e:  0f 05                  syscall
vinicius@vinicius-DC2C-S:~/Downloads$
```

Um ponto interessante é: os endereços já estão preparados para a região de memória do segmento do processo quando em execução!

$0x000\ 0040\ 0000 \equiv 0x000\ 0040\ 1000 \leq 0x800\ 0000\ 0000$
É válido!

Mas quem define esses endereços?
O **ligador** (ld)!

Do *Assembly* ao Executável



```
vinicius@vinicius-DC2C-S: ~/Downloads
vinicius@vinicius-DC2C-S:~/Downloads$ objdump -s programa
programa:      file format elf64-x86-64

Contents of section .text:
401000 48c7c03c 00000048 c7c70000 00000f05 H..<...H.....
vinicius@vinicius-DC2C-S:~/Downloads$ objdump -S programa
programa:      file format elf64-x86-64

Disassembly of section .text:
0000000000401000 <_start>:
401000:  48 c7 c0 3c 00 00 00      mov     $0x3c,%rax
401007:  48 c7 c7 00 00 00 00      mov     $0x0,%rdi
40100e:  0f 05                    syscall
vinicius@vinicius-DC2C-S:~/Downloads$
```

Quando nós executamos o programa, o carregador coloca as seções em memória (nem todas).

A CPU busca a primeira instrução (demarcada por `__start`), decodifica e executa a mesma

Expressões Aritméticas

```
int main() {  
    int a = 5, b = 10;  
    return (a + b);  
}
```

```
.section .data  
A: .quad 5  
B: .quad 10  
  
.section .text  
.globl _start  
_start:  
    movq A, %rdi  
    addq B, %rdi  
    movq $60, %rax  
    syscall
```

Considerando a versão do programa ao lado, vamos analisar a sua transcrição para *assembly*.

Agora as coisas já estão começando a ficar um pouco mais complexas!

Expressões Aritméticas

```
int main(){  
    int a = 5, b = 10;  
    return (a + b);  
}
```

```
.section .data  
A: .quad 5  
B: .quad 10  
.section .text  
.globl _start  
_start:  
    movq A, %rdi  
    addq B, %rdi  
    movq $60, %rax  
    syscall
```

DEFINIÇÃO DA SEÇÃO *DATA*

Além da seção *text*, onde as instruções são alocadas, agora também contamos com a seção *data*, para as nossas variáveis inicializadas!

Expressões Aritméticas

```
int main(){  
    int a = 5, b = 10;  
    return (a + b);  
}
```

```
.section .data  
A: .quad 5  
B: .quad 10  
  
.section .text  
.globl _start  
_start:  
    movq A, %rdi  
    addq B, %rdi  
    movq $60, %rax  
    syscall
```

DECLARAÇÃO E INICIALIZAÇÃO DE VARIÁVEIS

A declaração de variáveis começa pela definição de um rótulo para a mesma, este seguido por “:”.

Então define-se a quantidade de bits para a variável e o seu valor inicial.

Expressões Aritméticas

```
int main(){  
    int a = 5, b = 10;  
    return (a + b);  
}
```

```
.section .data  
A: .quad 5  
B: .quad 10  
.section .text  
.globl _start  
_start:  
    movq A, %rdi  
    addq B, %rdi  
    movq $60, %rax  
    syscall
```

DECLARAÇÃO E INICIALIZAÇÃO DE VARIÁVEIS

Os principais tamanhos de variáveis disponíveis são:

- .byte: 8 bits
- .word: 16 bits
- .long: 32 bits
- .quad: 64 bits

Expressões Aritméticas

```
int main() {  
    int a = 5, b = 10;  
    return (a + b);  
}
```

```
.section .data  
A: .quad 5  
B: .quad 10  
  
.section .text  
.globl _start  
_start:  
    movq A, %rdi  
    addq B, %rdi  
    movq $60, %rax  
    syscall
```

DEFINIÇÃO DA SEÇÃO *TEXT*

Neste ponto, não há nada novo. Definimos a seção *text* normalmente e definimos o seu ponto inicial para a execução.

Expressões Aritméticas

A PRIMEIRA INSTRUÇÃO

```
int main(){  
    int a = 5, b = 10;  
    return (a + b);  
}
```

```
.section .data  
A: .quad 5  
B: .quad 10  
  
.section .text  
.globl _start  
_start:  
    movq A, %rdi  
    addq B, %rdi  
    movq $60, %rax  
    syscall
```

A primeira instrução também não nos traz nada novo, consiste da operação *mov*.

A principal novidade é a utilização de endereçamento direto no primeiro argumento.

Expressões Aritméticas

```
int main(){  
    int a = 5, b = 10;  
    return (a + b);  
}
```

```
.section .data  
A: .quad 5  
B: .quad 10  
.section .text  
.globl _start  
_start:  
    movq A, %rdi  
    addq B, %rdi  
    movq $60, %rax  
    syscall
```

A SEGUNDA INSTRUÇÃO

A segunda instrução é nova! Ela consiste na operação *add*.

A operação *add* recebe dois parâmetros tal qual a operação *mov* (mesmas possibilidades e restrições).

Expressões Aritméticas

A SEGUNDA INSTRUÇÃO

```
int main() {  
    int a = 5, b = 10;  
    return (a + b);  
}
```

```
.section .data  
A: .quad 5  
B: .quad 10  
  
.section .text  
.globl _start  
_start:  
    movq A, %rdi  
    addq B, %rdi  
    movq $60, %rax  
    syscall
```

A segunda instrução é nova! Ela consiste na operação *add* (operação de soma).

A operação *add* recebe dois parâmetros tal qual a operação *mov* (mesmas possibilidades, restrições e resultados).

Expressões Aritméticas

```
int main() {  
    int a = 5, b = 10;  
    return (a + b);  
}
```

```
.section .data  
A: .quad 5  
B: .quad 10  
  
.section .text  
.globl _start  
_start:  
    movq A, %rdi  
    addq B, %rdi  
    movq $60, %rax  
    syscall
```

A TERCEIRA INSTRUÇÃO E A QUARTA INSTRUÇÃO

Bloco que ajusta a requisição e realiza a *syscall* para o encerramento do processo.

Do *Assembly* ao Executável

```
vinicius@vinicius-DC2C-S: ~/Downloads
vinicius@vinicius-DC2C-S:~/Downloads$ objdump -s programa
programa:          file format elf64-x86-64

Contents of section .text:
 401000 488b3c25 00204000 48033c25 08204000  H.<%. @.H.<%. @.
 401010 48c7c03c 0000000f 05          H..<.....
Contents of section .data:
402000 05000000 00000000 0a000000 00000000  .....
vinicius@vinicius-DC2C-S:~/Downloads$ objdump -S programa
programa:          file format elf64-x86-64

Disassembly of section .text:

0000000000401000 <_start>:
 401000:      48 8b 3c 25 00 20 40      mov     0x402000,%rdi
 401007:      00
 401008:      48 03 3c 25 08 20 40      add     0x402008,%rdi
 40100f:      00
 401010:      48 c7 c0 3c 00 00 00      mov     $0x3c,%rax
 401017:      0f 05                    syscall
vinicius@vinicius-DC2C-S:~/Downloads$
```

Analizando o **executável gerado após a montagem (as) e ligação (ld)** do nosso programa podemos notar a presença de uma nova seção: *data*.

Esta seção define o **espaço de memória e os valores iniciais** das nossas variáveis.

Do *Assembly* ao Executável

```
vinicius@vinicius-DC2C-S: ~/Downloads
vinicius@vinicius-DC2C-S:~/Downloads$ objdump -s programa

programa:      file format elf64-x86-64

Contents of section .text:
 401000 48b3c25 00204000 48033c25 08204000  H.<%. @.H.<%. @.
 401010 48c7c03c 0000000f 05          H..<.....
Contents of section .data:
402000 05000000 00000000 0a000000 00000000  ....
vinicius@vinicius-DC2C-S:~/Downloads$ objdump -S programa

programa:      file format elf64-x86-64

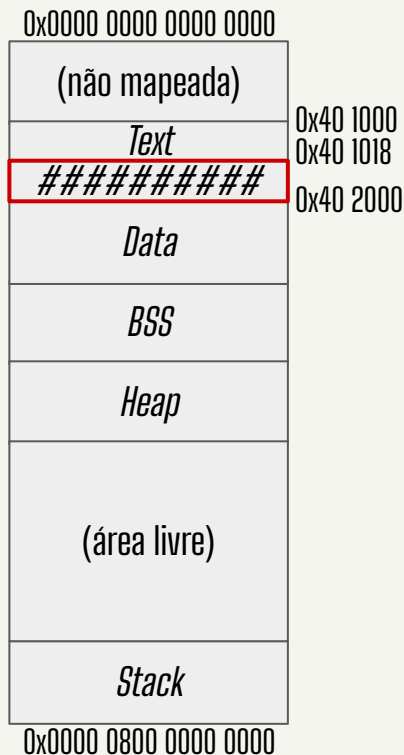
Disassembly of section .text:

0000000000401000 <_start>:
 401000:      48 8b 3c 25 00 20 40      mov     0x402000,%rdi
 401007:      00
 401008:      48 03 3c 25 08 20 40      add     0x402008,%rdi
 40100f:      00
 401010:      48 c7 c0 3c 00 00 00      mov     $0x3c,%rax
 401017:      0f 05                    syscall
vinicius@vinicius-DC2C-S:~/Downloads$
```

Verificando os endereços de memória disponíveis podemos deduzir algumas coisas:

- A seção *data* inicia no endereço 0x402000
- A seção *data* ocorre depois da seção *text*
 - 4096 bytes a frente

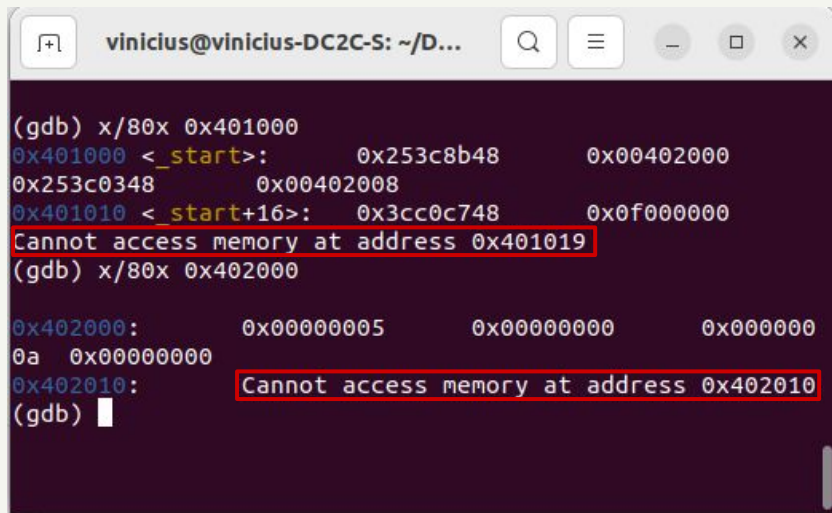
Do *Assembly* ao Executável



Ok! Mas se a seção *text* começa em 0x401000 e contém apenas 24 bytes, e a seção *data* começa apenas em 0x402000, o que tem no espaço vago?

A resposta é: **nada!** Essa parte da memória tem seu acesso bloqueado para impedir injeções de código.

Do *Assembly* ao Executável



```
vinicius@vinicius-DC2C-S: ~/D...  
(gdb) x/80x 0x401000  
0x401000 <_start>:      0x253c8b48      0x00402000  
0x253c0348      0x00402008  
0x401010 <_start+16>:  0x3cc0c748      0x0f000000  
Cannot access memory at address 0x401019  
(gdb) x/80x 0x402000  
  
0x402000:      0x00000005      0x00000000      0x00000000  
0a 0x00000000  
0x402010:      Cannot access memory at address 0x402010  
(gdb)
```

Notem o exame do binário com o gdb:

- Na seção *text*, o exame dos dados é suspenso no endereço 0x401019
- Na seção *data*, o exame dos dados é suspenso no endereço 0x402010

Outras Operações Aritméticas

Naturalmente, o assembly x86_64 fornece outras operações aritméticas além da adição (`add`) vista anteriormente. Entre as operações disponíveis, destacam-se:

- `add arg1 arg2`
- `sub arg1 arg2`
- `mul arg1`
- `div arg1`

Exercício #02

```
.section .data
A: .quad 5
B: .quad 10
.section .text
.globl _start
_start:
    movq A, %rdi
    addq B, %rdi
    movq $60, %rax
    syscall
```

Utilize o último programa estudado na aula de hoje (ao lado) e faça a sua montagem (*as*, criando um arquivo de código-objeto) e ligação (*ld*, criando um arquivo executável) e responda:

- Qual é a diferença nos **endereços de memória indicados nos arquivos**? Na sua opinião, qual é o **motivo dessa diferença**? (Dica: verifique através da aplicação objdump)

Obrigado!

Vinícius Fülber Garcia
inf.ufpr.br/vinicius/
viniciusfulber@ufpr.br