DCC510 – PROGRAMAÇÃO EM BAIXO NÍVEL

AULA 2

Carlos Bruno Oliveira Lopes

Engenheiro de Computação

Mestre em Ciência da Computação

Linguagem Assembly

Configuração do ambiente

- Sistema Operacional Linux
- NASM como compilador da linguagem Assembly
- GCC como compilador da linguagem C (Usaremos para gerar código Assembly a partir de programas C)
- GNU Make 4.0 como sistema de construção
- GDB como depurador
- O editor de texto de que você gosta.

Linguagem Assembly

NASM

- É um assembler 80x86 e x86-64 projetado para portabilidade e modularidade.
- Ele suporta uma variedade de formatos de arquivo de objeto, incluindo:
  - Linux e *BSD a.out, ELF, Mach-O, formato (OMF) .obj de 16 e 32 bits, COFF (incluindo suas variantes Win32 e Win64).
- Pode gerar saída arquivos binários simples, formatos Intel hex e Motorola S-Record.
- Sua sintaxe é projetada para ser simples e fácil para entender, semelhante à sintaxe no Manual do desenvolvedor de software da Intel com o mínimo de complexidade.
- Suporta todas as extensões de arquitetura x86 atualmente conhecidas e tem forte suporte para macros.

Linguagem Assembly

NASM

- Instalação:

+ no terminal:

sudo apt install nasm

nasm -v

- Compilador online:
  - https://www.jdoodle.com/compile-assembler-nasm-online/
- Plataforma alternativa:
  - https://dman95.github.io/SASM/english.html

Linguagem Assembly

GCC (GNU Compiler Collection ou GNU C Compiler)

- É uma distribuição integrada de compiladores para vários linguagem de programação.
- Essas linguagens são: C, C++, Objective-C, Objetive-C++, Fortran, Ada, D, Go e BRIG (DSAIL)

Linguagem Assembly

GNU Make

- É um utilitário para determinar automaticamente quais partes de um grande programa precisam ser recompiladas e emite comandos para recompilá-las.
- O programa make usa a base de dados makefile e os horários da última modificação dos arquivos para decidir quais arquivos precisam ser atualizados.
  - Para cada um desses arquivos, emite as receitas registradas na base de dados.

Linguagem Assembly

GNU GCC e Make

- Instalação:

+ no terminal:

sudo apt install build-essential

make -v

gcc --version

Linguagem Assembly

GDB

- É um depurador que permite o programador ver o que esta acontecendo dentro de outro programa que esta sendo executado.

+ no terminal:

sudo apt install gdb

gdb --version

Linguagem Assembly

Programa Hello, world!

- Um programa executará em um hardware.
- No caso do “Hello, world!”; ele deve mostrar caracteres na tela.
- No entanto, o programa não deve acessar os recursos diretamente sem que o sistema operacional coordene suas atividades.
  - O SO irá oferece um conjunto de rotinas para tratar a comunicação do programa com dispositivos externos, outros programas, sistema de arquivos, dentre outros.
- Portanto, um programa geralmente não pode ignorar o sistema operacional.
  - Ele usará as chamadas de sistema (system calls), que são rotinas oferecidas por um SO às aplicações de usuário

Linguagem Assembly

Programa Hello, world!

- O sistema Unix identifica um arquivo pelo seu descritor.
  - **Descritor**. É um valor inteiro.
- Um arquivo pode ser aberto pela chamada de sistema **open**.
- Quando um programa inicia, três arquivos são abertos: **stdin**, **stdout** e **stderr**. Seus descritores são 0, 1 e 2.
  - **stdin** é usado para tratar a entrada;
  - **stdout** para tratar a saída;
  - e **stderr** é utilizado para a saída de informações sobre o processo de execução do programa, mas não os seus resultados (como, erros e diagnósticos).

Linguagem Assembly

Programa Hello, world!

- Logo, “Hello, world!” deverá ser escrito em **stdout**.
  - Deve-se fazer uma chamada de sistema **write**.
    - Escreve-se uma dada quantidade de bytes da memória, começando em um dado endereço, em um arquivo com um dado descritor (Para o nosso caso, 1)
    - Os bytes conterão caracteres de string codificados, usando uma tabela predefinida (tabela ASCII)

Linguagem Assembly

Programa Hello, world!

Esse programa faz uma chamada de sistema write com os argumentos corretos especificados nas linhas de 6 a 9.

```
1 global _start

2 section .data
3 message: db 'hello, world!', 10

4 section .text
5 _start:
6   mov rax, 1          ; system call number should be stored in rax
7   mov rdi, 1          ; argument #1 in rdi: where to write (descriptor)?
8   mov rsi, message    ; argument #2 in rsi: where does the string start?
9   mov rdx, 14         ; argument #3 in rdx: how many bytes to write?
10  syscall             ; this instruction invokes a system call
```

Linguagem Assembly

```
global _start

section .data
message: db 'hello, world!', 10

section .text
_start:
        mov     rax, 1          ; system call number should be stored in rax
        mov     rdi, 1          ; argument #1 in rdi: where to write (descriptor)?
        mov     rsi, message    ; argument #2 in rsi: where does the string start?
        mov rdx, 14             ; argument #3 in rdx: how many bytes to write?
syscall                         ; this instruction invokes a system call
```

Estrutura do programa

- Um programa Assembly, em geral, é dividido em **seções**. Onde, cada seção tem sua finalidade:
  - **.text** armazena instruções;
  - **.data** usado para variáveis globais;
- No programa resultante, todos os dados, correspondentes a cada seção, estarão reunidos em um só local.
- **Rótulos (labels)** utilizados para nomear valores numéricos de endereços.
  - Eles antecedem qualquer comando e, em geral, estão separados desses por dois pontos.
    - Linha 5: _start
- Em assembly é mais conveniente o uso do termo rótulos ou endereços. O conceito de variáveis e procedimento são sutis e pouco adequados.

Linguagem Assembly

```
global _start

section .data
message: db 'hello, world!', 10

section .text
_start:
        mov     rax, 1          ; system call number should be stored in rax
        mov     rdi, 1          ; argument #1 in rdi: where to write (descriptor)?
        mov     rsi, message    ; argument #2 in rsi: where does the string start?
        mov rdx, 14             ; argument #3 in rdx: how many bytes to write?
syscall                         ; this instruction invokes a system call
```

Estrutura do programa

- Um programa assembly poder ser dividido em vários arquivos.
  - O rótulo **_start** é usado como ponto de entrada e identifica a primeira instrução a ser executada.
- Comentários iniciam com um ponto e vírgula ‘**;**’ e se estendem até o final da linha.
- A linguagem Assembly é constituída de comandos (instruções) que são mapeados diretamente para código de máquina
- Diretivas (pseudoinstrução) são construções de linguagem que não são comandos, usadas para controle de processo de tradução.

Linguagem Assembly

```
global _start

section .data
message: db 'hello, world!', 10

section .text
_start:
        mov     rax, 1          ; system call number should be stored in rax
        mov     rdi, 1          ; argument #1 in rdi: where to write (descriptor)?
        mov     rsi, message    ; argument #2 in rsi: where does the string start?
        mov rdx, 14             ; argument #3 in rdx: how many bytes to write?
syscall                         ; this instruction invokes a system call
```

Estrutura do programa

- Um programa assembly poder ser dividido em vários arquivos.
  - O rótulo **_start** é usado como ponto de entrada e identifica a primeira instrução a ser executada.
- Comentários iniciam com um ponto e vírgula ‘**;**’ e se estendem até o final da linha.
- A linguagem Assembly é constituída de comandos (instruções) que são mapeados diretamente para código de máquina
- Diretivas (pseudoinstrução) são construções de linguagem que não são comandos, usadas para controle de processo de tradução.
  - **global**, **section** e **db** são diretivas.
    - No programa Hello world, o **db** é usado para criar bytes de dados.

Linguagem Assembly

```
global _start

section .data
message: db 'hello, world!', 10

section .text
_start:
        mov     rax, 1          ; system call number should be stored in rax
        mov     rdi, 1          ; argument #1 in rdi: where to write (descriptor)?
        mov     rsi, message    ; argument #2 in rsi: where does the string start?
        mov rdx, 14             ; argument #3 in rdx: how many bytes to write?
syscall                         ; this instruction invokes a system call
```

Estrutura do programa

- No geral, os dados são definidos usando uma das diretivas abaixo, diferenciando-se quanto ao formato dos dados:
  - **db** bytes;
  - **dw** palavras (words), que correspondem a 2 bytes;
  - **dd** palavras duplas (double words), que correspondem a 4 bytes;
  - **dq** palavras quádruplas (quad words), que correspondem a 8 bytes;

Linguagem Assembly

```
global _start

section .data
message: db 'hello, world!', 10

section .text
_start:
        mov     rax, 1          ; system call number should be stored in rax
        mov     rdi, 1          ; argument #1 in rdi: where to write (descriptor)?
        mov     rsi, message    ; argument #2 in rsi: where does the string start?
        mov rdx, 14             ; argument #3 in rdx: how many bytes to write?
syscall                         ; this instruction invokes a system call
```

Estrutura do programa

data_decl.asm

```
section .data
    example1: db 5, 16, 8, 4, 2, 1
    example2: times 999 db 42
    example3: dw 999
```

- `times n cmd` é uma diretiva para repetir `cmd n` vezes no código do programa
  - Ele funciona com instruções de CPU
- Note que é possível criar dados em qualquer seção, inclusive em **.text**

Linguagem Assembly

```
global _start

section .data
message: db 'hello, world!', 10

section .text
_start:
        mov     rax, 1          ; system call number should be stored in rax
        mov     rdi, 1          ; argument #1 in rdi: where to write (descriptor)?
        mov     rsi, message    ; argument #2 in rsi: where does the string start?
        mov rdx, 14             ; argument #3 in rdx: how many bytes to write?
syscall                         ; this instruction invokes a system call
```

Estrutura do programa

- Diretivas permitem definir vários objetos de dados
  - `message: db 'hello, world!', 10`
    - O código ASCII é usado para codificar e representar cada caractere da string “hello, world!”
    - Em seguida, é adicionado um byte igual a 10
      - Por que 10?

        Indica próxima linha, ou seja, para iniciar uma nova linha aplicamos o código 10

Linguagem Assembly

```
global _start

section .data
message: db 'hello, world!', 10

section .text
_start:
        mov     rax, 1          ; system call number should be stored in rax
        mov     rdi, 1          ; argument #1 in rdi: where to write (descriptor)?
        mov     rsi, message    ; argument #2 in rsi: where does the string start?
        mov rdx, 14             ; argument #3 in rdx: how many bytes to write?
syscall                         ; this instruction invokes a system call
```

Instruções Básicas

- **mov** usada para escrever uma valor no registrador ou na memória.
  - O valor pode ser obtidos de outro registrador ou da memória, ou pode ser um valor imediato.
  - No entanto:
    1. Não pode copiar dados da memória para a memória;
    2. os operandos de origem e de destino devem ter o mesmo tamanho;

Linguagem Assembly

```
global _start

section .data
message: db 'hello, world!', 10

section .text
_start:
        mov     rax, 1          ; system call number should be stored in rax
        mov     rdi, 1          ; argument #1 in rdi: where to write (descriptor)?
        mov     rsi, message    ; argument #2 in rsi: where does the string start?
        mov rdx, 14             ; argument #3 in rdx: how many bytes to write?
syscall                         ; this instruction invokes a system call
```

Instruções Básicas

- **syscall** usada para fazer chamadas de sistema (system calls) em sistemas *nix.
  - Quando há operações de entrada/saída que dependem de hardware, os programadores não têm permissão para controlá-lo diretamente, ignorando o sistema operacional.
  - Cada chamada de sistema tem um número único. Para executá-la:
    1. O registrador **rax** deve armazenar o número da chamada de sistema.
    2. Os registradores a seguir devem armazenar seus argumentos: **rdi**, **rsi**, **rdx**, **r10**, **r8** e **r9**. Uma chamada de sistema não pode aceitar mais do que seis argumentos.
    3. Execute a instrução **syscall**.

    (Não importa a ordem em que os registradores são inicializados)

Linguagem Assembly

```
global _start

section .data
message: db 'hello, world!', 10

section .text
_start:
        mov     rax, 1          ; system call number should be stored in rax
        mov     rdi, 1          ; argument #1 in rdi: where to write (descriptor)?
        mov     rsi, message    ; argument #2 in rsi: where does the string start?
        mov rdx, 14             ; argument #3 in rdx: how many bytes to write?
syscall                         ; this instruction invokes a system call
```

Instruções Básicas

- No programa “Hello, world!”, é utilizado uma syscall `write` simples. Ela aceita:
  1. Um descritor de arquivo;
  2. O endereço do buffer.
  3. A quantidade de bytes para escrever.

Linguagem Assembly

```
global _start

section .data
message: db 'hello, world!', 10

section .text
_start:
        mov     rax, 1          ; system call number should be stored in rax
        mov     rdi, 1          ; argument #1 in rdi: where to write (descriptor)?
        mov     rsi, message    ; argument #2 in rsi: where does the string start?
        mov rdx, 14             ; argument #3 in rdx: how many bytes to write?
syscall                         ; this instruction invokes a system call
```

Programa Hello, world!

- Para compilar o programa, salve o código em hello.asm e então execute os comandos a seguir no shell:

```
> nasm -felf64 hello.asm -o hello.o
> ld -o hello hello.o
> chmod u+x hello
```

- Para executar o programa, faça:

```
> ./hello
hello, world!
Segmentation fault
```

Linguagem Assembly

```
global _start

section .data
message: db 'hello, world!', 10

section .text
_start:
        mov     rax, 1          ; system call number should be stored in rax
        mov     rdi, 1          ; argument #1 in rdi: where to write (descriptor)?
        mov     rsi, message    ; argument #2 in rsi: where does the string start?
        mov rdx, 14             ; argument #3 in rdx: how many bytes to write?
syscall                         ; this instruction invokes a system call
```

Programa Hello, world!

> ./hello

hello, world!

Segmentation fault

- Observe que o programa fez o que queríamos, mas, parece que um erro ocorre em sua execução.
- O que há de errado?

Linguagem Assembly

```
global _start

section .data
message: db 'hello, world!', 10

section .text
_start:
        mov     rax, 1          ; system call number should be stored in rax
        mov     rdi, 1          ; argument #1 in rdi: where to write (descriptor)?
        mov     rsi, message    ; argument #2 in rsi: where does the string start?
        mov rdx, 14             ; argument #3 in rdx: how many bytes to write?
syscall                         ; this instruction invokes a system call
```

Programa Hello, world!

- O que há de errado?
  - O programa continua o seu trabalho, ou seja, após a `syscall`, a memória armazena alguns valores aleatórios nas próximas células.
  - No entanto, o processador não sabe se esses valores tem algum propósito (são instruções ou não).
    - Dessa forma, ele tentará interpretá-los pois o registrador **`rip`** aponta para eles.
    - Portanto, como é improvável que esses valores codifiquem instruções corretas, uma interrupção será gerada com código 6 (Instrução inválida).
- Então o que deve ser feito?

Linguagem Assembly

```
global _start

section .data
message: db 'hello, world!', 10

section .text
_start:
        mov     rax, 1          ; system call number should be stored in rax
        mov     rdi, 1          ; argument #1 in rdi: where to write (descriptor)?
        mov     rsi, message    ; argument #2 in rsi: where does the string start?
        mov rdx, 14             ; argument #3 in rdx: how many bytes to write?
syscall                         ; this instruction invokes a system call
```

Programa Hello, world!

- Então o que deve ser feito?
  - Utilizar a chamada de sistema `exit`, que encerra o programa de forma correta:

```
mov rax, 60    ; 'exit' syscall number
xor rdi, rdi
syscall
```

Linguagem Assembly

Programa Hello, world!

```
1 global _start

2 section .data
3 message: db 'hello, world!', 10

4 section .text
5 _start:
6   mov rax, 1          ; system call number should be stored in rax
7   mov rdi, 1          ; argument #1 in rdi: where to write (descriptor)?
8   mov rsi, message    ; argument #2 in rsi: where does the string start?
9   mov rdx, 14         ; argument #3 in rdx: how many bytes to write?
10  syscall             ; this instruction invokes a system call

11  mov rax, 60         ; 'exit' syscall number
12  xor rdi, rdi
13  syscall
```

Linguagem Assembly

Programa Hello, world!

- Então o que deve ser feito?
  - Utilizar a chamada de sistema `exit`, que encerra o programa de forma correta:

```
mov rax, 60    ; 'exit' syscall number
xor rdi, rdi
syscall
```

Linguagem Assembly – Questões

1. O que faz a instrução xor rdi, rdi?
2. Qual é o código de retorno do programa?
3. Qual é o primeiro argumento da chamada de sistema exit?

Linguagem Assembly – Prática

- Faça o download do código do programa “Exibe o valor de rax” que exibi o valor de `rax` em formato hexadecimal usando o link abaixo:
  - https://github.com/carlosbrunocb/DCC510_PROGRAMACAO_EM_BAIXO_NIVEL/tree/code/assembly_code
  - Acesse a pasta print_rax_value e baixe o arquivo **print_rax.asm**
  - Compile e rode execute o programa.
  - Então responda as questões abaixo:
    1. Qual é a diferença entre sar e shr? Consulte a documentação do Intel.
    2. Como podemos escrever números em diferentes sistemas numéricos, de modo que sejam compreensíveis ao NASM? Consulte a documentação do NASM.

RESUMO

- Linguagem Assembly
  - Programa “Hello World!”;
  - Estrutura do Programa;
  - Questões;