“Achei um bug, mas não é no código!”

E/S com funções do C em x86-64 e detalhes da Pilha

Paulo Ricardo Lisboa de Almeida

UDESC
UNIVERSIDADE
DO ESTADO DE
SANTA CATARINA

E/S em C

- Em C temos diversas funções para entrada e saída
  - scanf, printf, write, read, …
- As funções na verdade são *wrappers* para chamadas ao Sistema Operacional, que é o real responsável pela E/S

Write

- A função write é uma das mais simples funções de saída
  - Definida dentro de unistd.h
  - ssize_t write(int fd, const void *buf, size_t count);
    - fd é um número identificador do arquivo de saída
      - Em sistemas UNIX, o arquivo 1 representa STDOUT
        - Constante STDOUT_FILENO definida em unistd.h
    - *buf é o endereço de um vetor de caracteres
    - count é o número de caracteres total na string
    - Em caso de sucesso
      - A função retorna o número de caracteres impresso em fd
    - Em caso de erro
      - -1 é retornado
      - O erro é armazenado em errno

Write

- Para começar, crie o seguinte programa em C, compile e rode
  - Para compilar

    gcc programa.c -o programa

```c
#include<unistd.h>

int main(){
    char ola[] = "Ola Mundo\n";

    write(STDOUT_FILENO, ola, 10);
    return 0;
}
```

Segmento de dados somente leitura

- Em Assembly x86-64 do GAS, para instruir o compilador a inserir os dados na **seção de dados somente leitura**, utilize a diretiva
  .section .rodata
  - Observação
    - .rodata é mapeado para o segmento de texto do programa pelo montador em sistemas UNIX, pois esse segmento é somente leitura
      - Diferente do segmento de dados, que é leitura/escrita
  - A diretiva *.string* armazena uma string terminada por ‘/0’ (0) na memória
    - Exemplo com uma string rotulada de minha_str

```
minha_str: .string "minha string"
```

Argumentos

- Em x86-64, consideramos os argumentos da esquerda para a direita, e os colocamos nos seguintes registradores
  - Caso hajam mais parâmetros, esses são passados via pilha
    - São empilhados em ordem inversa
    - O primeiro parâmetro a ser empilhado é o mais a direita

Argumento	Registrador
Primeiro	rdi
Segundo	rsi
Terceiro	rdx
Quarto	rcx
Quinto	r8
Sexto	r9

- Observação
  - No modo 32 bits, todos os parâmetros são passados via pilha

Olá mundo em x86-64

```
        .section  .rodata             #dados somente leitura
ola_mundo:                            #nome da string
        .string "Ola Mundo\n"         #string terminada com \0
        .text                         #seção de texto
        .globl main                   #main visível globalmente
        .type  main, @function
main:                                 #inicio do main
        pushq  %rbp                   #salva stack frame na pilha
        movq %rsp, %rbp               #define novo stack frame
        movl $10, %edx                #terceiro argumento
        movl $ola_mundo, %esi         #segundo argumento
        movl $1, %edi                 #primeiro argumento
        call write                    #chama a função write
        movl $0, %eax                 #return 0
        movq %rbp, %rsp               #restaura a pilha
        popq %rbp                     #restaura o stack frame
        ret                           #retorna ao chamador
```

UDESC

Olá mundo em x86-64

Inserindo os parâmetros. Note que *movl $ola_mundo, %esi* carrega o **endereço** onde ola_mundo inicia na memória para %esi

```
        .section  .rodata          #dados somente leitura
ola_mundo:                         #nome da string
        .string "Ola Mundo\n"      #string terminada com \0
        .text                      #seção de texto
        .globl main                #main visível globalmente
        .type  main, @function
main:                              #inicio do main
        pushq  %rbp                #salva stack frame na pilha
        movq %rsp, %rbp            #define novo stack frame
        movl $10, %edx             #terceiro argumento
        movl $ola_mundo, %esi      #segundo argumento
        movl $1, %edi              #primeiro argumento
        call write                 #chama a função write
        movl $0, %eax              #return 0
        movq %rbp, %rsp            #restaura a pilha
        popq %rbp                  #restaura o stack frame
        ret                        #retorna ao chamador
```

UDESC

Olá mundo em x86-64

Chama a função write e empilha o endereço de retorno. Write é definido em “unistd.h”. O GCC vai ter que encontrar essa função no PATH do sistema e fazer a linkedição.

```
        .section  .rodata             #dados somente leitura
ola_mundo:                            #nome da string
        .string "Ola Mundo\n"         #string terminada com \0
        .text                         #seção de texto
        .globl main                   #main visível globalmente
        .type  main, @function
main:                                 #inicio do main
        pushq  %rbp                   #salva stack frame na pilha
        movq %rsp, %rbp               #define novo stack frame
        movl $10, %edx                #terceiro argumento
        movl $ola_mundo, %esi         #segundo argumento
        movl $1, %edi                 #primeiro argumento
        call write                    #chama a função write
        movl $0, %eax                 #return 0
        movq %rbp, %rsp               #restaura a pilha
        popq %rbp                     #restaura o stack frame
        ret                           #retorna ao chamador
```

UDESC

Montando o programa

- Monte o programa
  as olaMundo.s --gstabs -o olaMundo.o
- Faça a linkedição com a ajuda do GCC
  gcc olaMundo.o -no-pie -o olaMundo
  - -no-pie desabilita a geração de executáveis de posição independente
    - O pie permite o S.O. carregar as dependências em posições aleatórias da memória a cada rodada do programa
      - Recurso de segurança do S.O.
    - Não vamos conseguir usar devido a forma que estamos gerando nosso programa

Exercício

1. Monte e rode o programa do exemplo anterior

2. Rode passo a passo no GDB, e analise o que está acontecendo nos registradores e memória
   - Dicas do GDB
     - Para imprimir o conteúdo você pode usar a sintaxe do C. Exemplos:
       printf "%s", &ola_mundo (endereço do label)
       printf "%s", 0x400594 (passando um endereço de memória diretamente)
     - *si* executa a próxima instrução
       - Caso seja uma chamada a função, entra na função
     - *ni* faz o mesmo que *si*, **mas não entra na função**

Tornando mais legível

- A diretiva .equ pode ser utilizada para definir nomes a expressões
  .equ NOME, EXPRESSÃO
  - Na etapa de montagem, o montador substitui as ocorrências de NOME pelo valor de EXPRESSÃO
  - Similar a um #define em C

Tornando mais legível

Um ponto (.) significa “esse endereço”. Como o . está após a string, carregamos o end. final da string. Como o label ola_mundo marca o endereço de início da string, a conta sendo feita é endereçoFinal-endereçoInicial-1, o que dá o tamanho da string. O -1 é necessário pois a string é terminada com ‘\0’ por padrão com a diretiva .string.

```
        #constantes
        .equ STDOUT,1
        .section  .rodata             #dados somente leitura
ola_mundo:                            #nome da string
        .string "Ola Mundo\n"         #string terminada com \0
        .equ ola_mundoSz, .-ola_mundo-1
        .text                         #seção de texto
        .globl main                   #main visível globalmente
        .type  main, @function
main:                                 #inicio do main
        pushq  %rbp                   #salva stack frame na pilha
        movq %rsp, %rbp               #define novo stack frame
        movl $ola_mundoSz, %edx       #terceiro argumento
        movl $ola_mundo, %esi         #segundo argumento
        movl $STDOUT, %edi            #primeiro argumento
        call write                    #chama a função write
        movl $0, %eax                 #return 0
        movq %rbp, %rsp               #restaura a pilha
        popq %rbp                     #restaura o stack frame
        ret                           #retorna ao chamador
```

UDESC

Detalhe

- pushs e pops na pilha dependem do modo de operação do processador
  - No modo 64 bits, podemos fazer pushes e pops de 8 bytes (pushq e popq)
    - Mas **não podemos** fazer pushes e pops de 4 bytes (pushl e popl)

Detalhe

- Não podemos garantir que as funções externas que chamamos (exemplo: write) não vão alterar os registradores que estamos utilizando em nosso programa
  - Exemplo: rax, rbx, r12, …
  - Se esses registradores possuem algum valor necessário antes de chamar a função, **salve os valores na pilha e restaure depois**

Exercício

3. Faça as alterações no programa como no slide anterior. Veja que agora você pode modificar o conteúdo da string ola_mundo sem modificar nada a mais no programa.

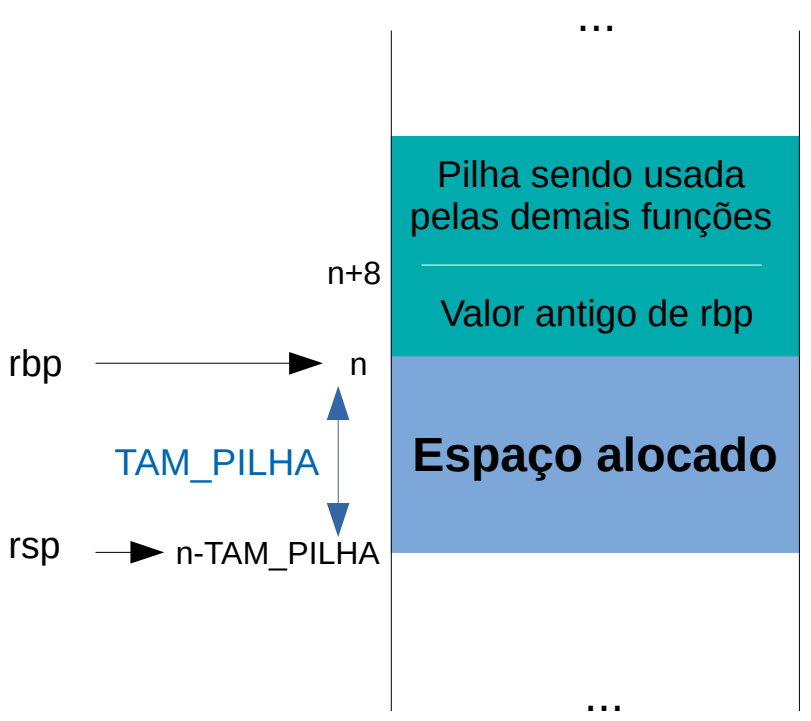
A pilha

- Segundo a Application Binary Interface AMD64 (ABI), o ponteiro de pilha (rbp) **sempre deve estar apontando para um endereço múltiplo de 16**
  - Necessário devido aos registradores SSE do x86-64
  - Uma forma simples de se fazer isso é ajustar a pilha de acordo com o tamanho necessário para as variáveis locais logo no início da chamada
  - Retorna para o estado original no final da chamada da função
    - **O ajuste da pilha deve ser feito logo após se estabelecer o frame pointer**

Prólogo e Epílogo

- Toda função (inclusive o main) vai ter então um **prólogo** (executado antes de tudo) e um **epílogo** (executado no fim), que será o seguinte:
  - **Prólogo**

    pushq %rbp #salvar o frame pointer na pilha
    movq %rsp, %rbp #estabelece o frame (base) pointer
    addq -TAM_PILHA, %rsp #ajusta o tamanho da pilha
  - De acordo com o Google
    - Prólogo: … a primeira parte da **tragédia**...



Prólogo e Epílogo

- Toda função (inclusive o main) vai ter então um **prólogo** (executado antes de tudo) e um **epílogo** (executado no fim), que será o seguinte:
  - **Epílogo**

```
#SETAR UM VALOR DE RETORNO CASO NECESSÁRIO
movq %rbp, %rsp #retorna rsp para a posição original +8
popq %rbp       #carrega o valor salvo de rbp e rsp = rsp – 8
ret             #retornar ao chamador
```

Detalhe

- Toda variável na pilha **deve estar em um endereço que é múltiplo do seu tamanho**
  - Por exemplo, considerando que um inteiro ocupa 4 bytes
  - Ele pode estar no endereço -0, -4, -8, … **a partir do frame pointer**
  - Mas não pode estar no endereço -7 a partir do frame pointer

Read

- A função read é uma das mais simples funções de entrada
  - Definida dentro de unistd.h
  - ssize_t read(int fd, void *buf, size_t count);
    - fd é um número identificador do arquivo de saída
      - Em sistemas UNIX, o arquivo **0** representa **STDIN**
        - Constante STDOUT_FILENO definida em unistd.h
    - *buf é o endereço de um vetor de caracteres onde os valores serão gravados
    - count é o número de caracteres a ser lidos
    - Em caso de sucesso
      - A função retorna o número de caracteres lidos
    - Em caso de erro
      - -1 é retornado
      - O erro é armazenado em errno

Echo

- Vamos criar um programa que faz um echo de um caractere
  - O usuário digita um caractere, o programa o lê e escreve o mesmo caractere na tela
  - Detalhe
    - O usuário digita um caractere e enter (o enter injeta um \n)
      - Então tecnicamente estamos lendo dois caracteres
- Crie um programa chamado programaEcho.s

Echo

- Vamos usar a função read
  - Read armazena na memória o conteúdo lido
  - Vamos armazenar na memória local da função (pilha)
    - Quantos bytes precisamos liberar na pilha para ler os dois caracteres?

Echo

- Vamos usar a função read
  - Read armazena na memória o conteúdo lido
  - Vamos armazenar na memória local da função (pilha)
    - Quantos bytes precisamos liberar na pilha para ler os dois caracteres?
    - 2 bytes
    - Mas a pilha deve ser mantida em um endereço múltiplo de 16
    - **Então vamos liberar 16 bytes na pilha**
      - Restrição de alinhamento

Echo

STDIN é o arquivo 0 no UNIX

Pilha terá 16 bytes

Prólogo alocando 16 bytes.
add[bwlq] fonte, destino
destino = destino+fonte

```
#constantes
        .equ STDIN, 0
        .equ STDOUT, 1
#posições na STACK
        .equ aLetter, -16
        .equ localSize, -16

        .section .rodata
prompt:
        .string "Entre com o caractere: "
        #o tamamho da mensagem é o tamanho da string -1
        .equ promptSz,.-prompt-1
msg:
        .string "Voce entrou: "
        .equ msgSz,.-msg-1

        .text
        .globl main
        .type main, @function
main:
        pushq %rbp          #salvar o frame pointer do caller
        movq %rsp, %rbp     #setar o frame pointer do main
        addq $localSize, %rsp   #ajustar a pilha
```

O valor lido será armazenado 16 bytes a partir do stack pointer

UDESC

Echo - Continuação

Mensagem solicitando a entrada do teclado

Notação IMEDIATO(%reg) é base +deslocamento. Endereço base no registrador, e deslocamento no imediato.

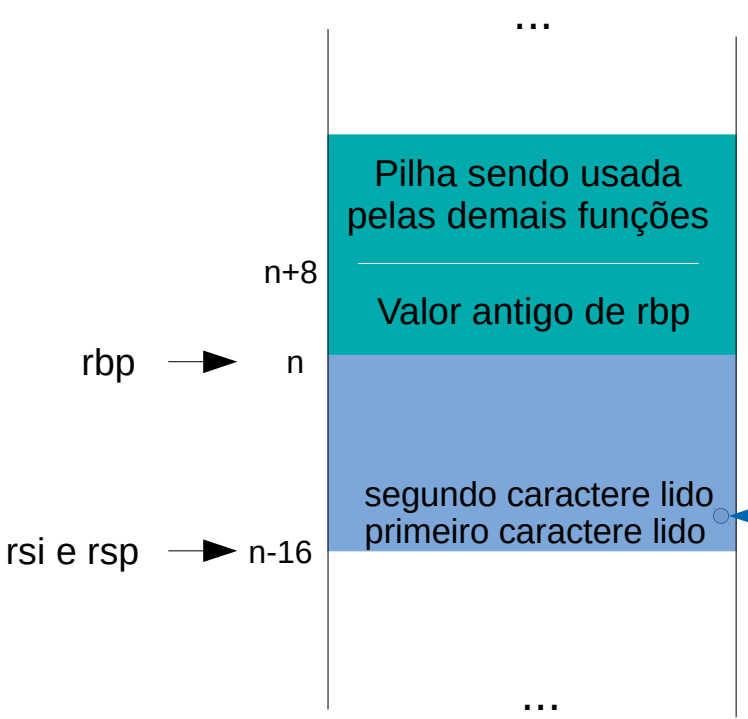
```
movl $promptSz, %edx    #tamanho da string em edx
movl $prompt, %esi      #endereço da string em esi
movl $STDOUT, %edi      #stdout em edi
call write              #pula para função write
```

```
movl $2, %edx           #ler 1 caractere + \n
leaq aLetter(%rbp), %rsi #calcular o end. de aLetter e armazenar em rsi
movl $STDIN, %edi       #ler de STDIN
call read               #lê e armazena no endereço apontado por rsi
```

lea[lq] DESL(%regBase), %dest calcula o endereço efetivo utilizando base+deslocamento e armazena no registrador destino

UDESC

Echo - Continuação




```
movl $promptSz, %edx        #tamanho da string em edx
movl $prompt, %esi          #endereço da string em esi
movl $STDOUT, %edi #stdout em edi
call write                  #pula para função write

movl $2, %edx           #ler 1 caractere + \n
leaq aLetter(%rbp), %rsi    #calcular o end. de aLetter e armazenar em rsi
movl $STDIN, %edi       #ler de STDIN
call read               #lê e armazena no endereço apontado por rsi
```

Echo - Continuação

Mensagem “Você entrou: ”

```
movl $msgSz, %edx
movl $msg, %esi
movl $STDOUT, %edi
call write
```

O read armazenou os caracteres digitados 16 bytes a partir do stack pointer rbp

```
movl $2, %edx
leaq aLetter(%rbp), %rsi
movl $STDOUT, %edi
call write
```

Epílogo

```
movl $0, %eax        #return 0
movq %rbp, %rsp      #remover as variáveis locais
popq %rbp            #restaurar o frame pointer
ret
```

UDESC

Exercício

4. Execute o programa “programaEcho” dos slides anteriores passo a passo no GDB, e analise as alterações sendo feitas na memória e nos registradores.
   - Pesquise sobre como ver a pilha no GDB

5. Crie um programa que lê uma palavra com exatamente 3 caracteres, e depois lê outra palavra com exatamente 2 caracteres. Depois o programa deve exibir a palavra de 2 caracteres, e a palavra de 3 caracteres na tela, **nessa ordem**.
   - Dica. Ao invés de fazer duas chamadas a write para escrever as strings, tente fazer uma única chamada que escreve as duas ao mesmo tempo.

Referências

- Bob Plantz. **Introduction to Computer Organization: A Guide to X86-64 Assembly Language and GNU/Linux.** 2011.
- **Intel® 64 and IA-32 Architectures Software Developer's Manual.** Intel, 2019.
- D. Patterson; J. Henessy. **Organização e Projeto de Computadores**: **A Interface Hardware / Software.** 4a Edição. Elsevier Brasil, 2014.
- STALLINGS, W. **Arquitetura e Organização de Computadores.** 9 ed. Prentice Hall. São Paulo, 2012.
- M. Matz, J. Hubička, A. Jaeger, M. Mitchell. **System V Application Binary Interface AMD64 Architecture Processor Supplement.** 2014.