

“Se parece fácil demais, algo deve estar errado.”

# Utilizando printf, scanf e fazendo syscalls no x86-64

Paulo Ricardo Lisboa de Almeida

# Printf e Scanf

- Vamos usar printf e scanf para ler e escrever valores formatados
- Crie um programa chamado echoInt.s

# EchoInt

A pilha terá 16 bytes. O valor lido é um inteiro de 4 bytes, que será deslocado 4 bytes a partir do stack pointer (rbp)

```
#posições na stack
.equ inteiro1, -4
.equ localSize, -16
```

```
.section .rodata
prompt:
.string "Digite um inteiro: "
scanFormat:
.string "%i"
printFormat:
.string "Voce digitou %i\n"
```

```
.text
.globl main
.type main, @function
main:
```

```
pushq %rbp    #salvar rbp
movq %rsp,%rbp #estabelecer novo frame pointer
addq $localSize, %rsp    #alocar espaço na pilha
```

Prólogo ajustando o rbp e alocando espaço na pilha

# EchoInt

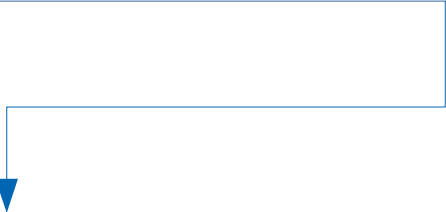
- Desejamos fazer o seguinte agora:  
`printf("Digite um inteiro: ");`
- Lembrando que definimos  
prompt:  
`.string "Digite um inteiro: "`
- E a convenção dos argumento é a seguinte

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

- Como esse trecho fica em x86-64?

# Echolnt

```
movq $prompt, %rdi #endereço da string em edi
movq $0, %rax      #0 em rax para ausência de argumentos em ponto flutuante
call printf        #jump and link para printf
```

- 
- Note o 0 movido para rax, e que rax não é um registrador para passagem de parâmetros de acordo com a convenção.
  - De acordo com a ABI, **o número de parâmetros** passados em registradores SSE deve ser especificado em rax.
  - Em x86-64, valores em ponto flutuante são passados nos registradores SSE para o printf/scanf. Como não passamos pontos flutuante, podemos mover 0 para rax.

# EchoInt

- Vamos inserir o trecho a seguir  
scanf("%i", &inteiro1)

- Lembrando que definimos

```
.equ inteiro1, -4
```

```
scanFormat:  
    .string "%i"
```

- **Como fica?**

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

# EchoInt

```
leaq inteiro1(%rbp), %rsi #endereço do inteiro na pilha para rsi
movq $scanFormat, %rdi #endereço da string em edi
movq $0, %rax          #sem ponto flutuante
call scanf #jump and link para scanf
```

# EchoInt

printf("Voce digitou %i\n", inteiro1)  
Lembre-se que o inteiro lido tem 4 bytes. Então utilizamos movl para carregar o registrador esi (e não rsi)

movl inteiro1(%rbp), %esi #copia o inteiro da pilha para rsi  
movq \$printfFormat, %rdi #endereço da string de formatação em edi  
movq \$0, %rax #sem ponto flutuante  
call printf #jump and link para printf

movl \$0, %eax #return 0  
movq %rbp, %rsp #retornar a pilha para posição inicial + 8  
popq %rbp #retornar rbp para a posição inicial e pilha-8  
ret

Epílogo



# Exercícios

1. Execute o programa anterior no GDB e analise as alterações nos registradores e na pilha.
2. Modifique o programa anterior para que sejam solicitados dois inteiros do teclado, e seja exibido o resultado da soma desses inteiros na tela.

# Interfaceando com o S.O.

- No momento estamos utilizando algumas funções das bibliotecas padrão do C
  - read, write, printf, scanf, ...
  - Essas funções organizam os dados internamente, e **chamam o Sistema Operacional para realizar a entrada e saída**
  - São **wrappers**
  - Somente o Sistema Operacional pode se comunicar com os sistemas de entrada e saída

# Realizando Syscalls

- Os registradores utilizados para realizar chamadas ao S.O., e os códigos de chamadas podem ser vistos no manual
  - Digite
    - `man syscall`
    - `man syscalls`
    - [filippo.io/linux-syscall-table](http://filippo.io/linux-syscall-table)
    - Códigos das syscalls em `usr/include/asm/unistd_64.h`  
`cat /usr/include/asm/unistd_64.h | grep NOME_SYSCALL`
  - O código da syscall deve ser **carregado em eax**
    - O S.O. lê o valor nesse registrador para definir o que fazer
    - Mesma lógica que usamos no MIPS ao carregar o código da Syscall em `$v0`

# Alguns códigos

- Alguns códigos que usaremos
  - read 0
  - write 1
  - exit 60
- Os parâmetros são passados de acordo com a ordem que aparece no manual
  - man read
  - man write
  - man exit

# Linkedição sem o GCC

- Vamos criar um programa que solicita um caractere, e o imprime na tela utilizando syscalls
  - Crie um arquivo chamado programaEcho.s
- **Convenção**
  - O rótulo do ponto inicial do programa deve ser `__start`
    - Dois underscores sem espaços entre eles
    - Quando usávamos a convenção do C, o ponto inicial era o *main*

# Para montar o programa

- Podemos montar o programa normalmente utilizando o GAS
  - `as nomePrograma.s --gstabs -o nomeObjetoSaida.o`
- A linkedição agora pode ser feita com o Linkeditor GNU, sem precisar passar pelo GCC
  - `ld nomeObjeto.o -e __start -o nomeBinarioFinal`  
-e especifica o rótulo onde o programa inicia

# Echo

```
#constantes
.equ STDIN,0
.equ STDOUT,1
.equ READ,0
.equ WRITE,1
.equ EXIT,60

#posições na stack
.equ aLetter,-16
.equ localSize,-16
```

```
.section .rodata
prompt:
.string "Digite um caractere: "
.equ promptSz,.-prompt-1

msg:
.string "Voce digitou: "
.equ msgSz,.-msg-1
```

Rótulo onde o programa inicia

```
.text
.globl __start
__start:
pushq %rbp
movq %rsp,%rbp
addq $localSize,%rsp #final do prólogo
```

# Echo

- Como fica a syscall para escrever a frase “Digite um Caractere”?
  - Código do write é 1 (em eax)
- Ordem dos argumentos

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

- Digite *man write*



# Echo

A única diferença para a chamada do wrapper de C que fizemos é o código da syscall em eax

```
movq $promptSz,%rdx #tamanho da string em rdx
movq $prompt,%rsi #endereço da string em rsi
movq $STDOUT,%rdi #id arquivo saída em rdi
movl $WRITE, %eax #código syscall em rax
syscall #chama o S.O.
```

# Exercício

## 3. Termine o programa

- Faça a syscall para ler 2 caracteres
  - O caractere digitado + \n que o usuário vai inserir
- Faça a syscall para escrever a mensagem “Você digitou: ”
- Faça a syscall para escrever os dois caracteres na tela
- Faça o epílogo
- Faça a syscall para terminar o programa (exit) retornando 0 ao S.O.

# Resposta

```
movq $2,%rdx #ler um caractere+\n
leaq aLetter(%rbp),%rsi #endereço onde salvar
movq $STDIN,%rdi #id do arquivo de entrada em rdi
movl $READ,%eax #código da syscall em rax
syscall #chama o S.O.
```

```
movq $msgSz,%rdx #tamanho da string em rdx
movq $msg,%rsi #endereço da string em rsi
movq $STDOUT,%rdi #id do arquivo de saída em rdi
movl $WRITE,%eax #código da syscall em rax
syscall #chama o S.O.
```

```
movq $2,%rdx #escrever 2 caracteres
leaq aLetter(%rbp),%rsi #endereço dos caracteres
movq $STDOUT,%rdi #id do arquivo de saída em rdi
movl $WRITE,%eax #código da syscall em rax
syscall #chama o S.O.
```

```
movq %rbp,%rsp #restaura a pilha
popq %rbp #restaura o stack pointer
movq $0,%rdi #valor que será retornado ao S.O.
movl $EXIT,%eax #código da syscall exit em rax
syscall #syscall para terminar o programa
```

# Exercício

4. Crie um programa que lê um caractere do teclado (+\n) e imprime esse caractere na tela em maiúsculo. Chame o programa de upperSyscall.s
- Não utilize as funções do C. Utilize apenas syscalls.

# 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.