Praticas 05c

O objetivo do programa eh testar as instruções básicas a seguir:

call scanf : rotina da biblioteca c para ler dados do teclado

saltos : instruções de desvios condicionais (if's) → jmp, jz, je, jne, jnz, jl, jle, jg, jge, jcxz e jecxz

loop : instrução de laço/repetição

Para gerar o executável, gere primeiro o objeto executando o seguinte comando:

```
as praticas_05c.s -o praticas_05c.o
```

e depois link dinamicamente com o seguinte comando:

```
ld praticas_05c.o -l c -dynamic-linker /lib/ld-linux.so.2 -o praticas_05c
```

==========
Codificação: Monte e teste o código, passo a passo, concatenando os trechos a seguir
==========

```
.section .data

pedido1:    .asciz      "\nTeste %d: Digite um valor inteiro => "
formato1:   .asciz      "%d"
mostra1:    .asciz      "Teste %d: Numero digitado = %d\n"
numero:     .int        0

pedido2:    .asciz      "\nTeste %d: Digite um caractere => "
formato2:   .asciz      "%c"
mostra2:    .asciz      "Teste %d: Caractere digitado = %c\n"
tecla:      .int        'A'

pedido3:    .asciz      "\nTeste %d: Digite uma string => "
formato3:   .asciz      "%s"
mostra3:    .asciz      "Teste %d: String digitada = %s\n"
frase:      .space      64

formatox:   .asciz      " %c" #despreza o <enter> e pega o próximo

pedido4:    .asciz      "\nTeste %d: Digite 2 numeros:\n N1 = "
pedido5:    .asciz      " N2 = "
n1:         .int        0
n2:         .int        0

mostra4:    .asciz      "Teste %d: Numeros lidos: n1 = %d  e n2 = %d\n"

mostra5:    .asciz      "Teste %d: n1 igual a n2\n"
mostra6:    .asciz      "Teste %d: n2 menor que n1\n"
mostra7:    .asciz      "Teste %d: n2 maior que n1\n"
mostra8:    .asciz      "Teste %d: Acabou as comparações!\n"

pedido6:    .asciz      "\nTeste %d: Quantos giros quer no loop? "
ngiros:     .int        0
```

```
mostra9:    .asciz      "Teste %d: Girando %d...\n"
mostra10:   .asciz      "Teste %d: Acabou o loop!\n\n"

.section .text

.globl      _start

_start:
```

1) lendo um caractere

```
      pushl $1
      pushl $pedido2
      call  printf

      pushl $tecla
      pushl $formato2
      call  scanf

      pushl tecla
      pushl $1
      pushl $mostra2
      call  printf
```

Coloque as instruções de finalização de programa para poder testar o programa até aqui. São elas:

```
      pushl $0
      call exit
```

Agora, insira no programa antes das instruções de finalização do programa, um a um, os trechos de códigos a seguir, de 2 adiante, mantendo os trechos anteriores já inseridos. Para cada trecho inserido, monte, link e execute o programa para observar os resultados. Depois insira o próximo.

2) lendo um número

```
      pushl $2
      pushl $pedido1
      call  printf

      pushl $numero
      pushl $formato1
      call  scanf

      pushl numero
      pushl $2
      pushl $mostra1
      call  printf
```

3) lendo uma string

```
      pushl $3
      pushl $pedido3
      call  printf

      pushl $frase
```

```
pushl $formato3
call  scanf

pushl $frase
pushl $3
pushl $mostra3
call  printf
```

ULTRA IMPORTANTE!: A função scanf é genérica e possibilita ler vários tipos de dados, por exemplo, caractere, número inteiro, número real (ponto flutuante) e string (cadeia de caracteres), entre outros. Para utilizá-la, devemos empilhar o endereço da variável (ou da região de memória) onde o dado lido será colocado e o endereço da string de formatação. A string de formatação informa o tipo do dado a ser lido, por exemplo “%c”, “%d”, “%f” e “%s”, entre outros. Ler um dado com scanf significa retirá-lo do buffer do teclado.

Os códigos de todas as teclas pressionadas ficam no buffer do teclado, controlado pelo Sistemas Operacional (SO). O SO é o responsável por obter o código de cada tecla pressionada, pela porta do teclado, e colocá-lo no buffer do teclado. O buffer mantém a sequência de todas as teclas digitadas e ainda não retiradas, até o limite máximo de seu tamanho.

Quando a aplicação solicita uma leitura (entrada) de dados, como por exemplo, usando a função scanf, o SO pega o dado do buffer e o coloca na área de memória associada a variável. A quantidade de caracteres que o SO irá retirar do buffer do teclado depende da especificação da função da aplicação.

A função scanf faz uma varredura (“escaneamento”) no buffer do teclado, do caractere mais antigo em direção ao mais recente digitado, a fim de agrupá-los no dado a ser lido. Scanf descarta espaços em branco e/ou o caractere <enter> que estiverem no início do buffer do teclado, antes do dado a ser lido, mas deixa no buffer do teclado tudo que estiver depois do dado a ser lido, seja espaço em branco, digitos numéricos ou alfabéticos, teclas <enter>, <capslock>, <shift> etc. Assim sendo, a leitura com o scanf deixa no buffer, **no mínimo**, um caractere **<enter>** que precisou ser digitado no final da leitura, e por isso, normalmente, ele é desprezado na próxima leitura com scanf.

No caso da leitura de números, observa-se que um espaço em branco ou qualquer caractere **não numérico** serve como delimitador do dado numérico a ser lido e permanecerá no buffer juntamente com os caracteres seguintes posicionados após o dado numérico lido, ou seja, somente os caracteres numéricos que formam o número são retirados do buffer, o restante seguinte permanece.

No caso da leitura de strings, estas não podem conter espaços em branco (caractere de separação), no início, pois são desprezados, e nem no meio, porque são encarados como delimitadores; nesse caso, a string lida será “cortada” no primeiro espaço em branco encontrado e o restante ficará no buffer do teclado. Somente a parte que precede o “espaço em branco” será colocada na região de memória informada.

Mas leitura de caracteres, a função scanf trabalha diferente. Nesse caso, ela lê o primeiro caracter e pronto, sem qualquer descarte anterior, independente de qual seja o caractere, e deixa o restante no buffer do teclado. Esta característica pode causar sérios problemas para os

programadores inexperientes, pois o caractere lido será normalmente o **<enter>**, **espaço em branco, tabulação** ou qualquer outro resíduo deixado pela digitação/leitura anterior. O resíduo é capturado de imediato, pois já está disponível, e o processamento pode prosseguir sem a ncessidade de esperar a leitura do caractere de interesse propriamente dito.

Mesmo que tenhamos usuários sérios e cuidadosos, e que os dados sejam digitados corretamente, apenas pra o propósito desejado, ao final de uma entrada de dados, seja durante a leitura de um número, caractere ou string, a tecla **<enter>** sempre será digitada para finalizar a entrada. Para testar esse fato, execute o trecho a seguir e veja o resultado.

4) lendo um caractere depois de número e de string usando **"%c"**.

```
pushl $4
pushl $pedido1
call  printf

pushl $numero
pushl $formato1
call  scanf

pushl numero
pushl $4
pushl $mostra1
call  printf

pushl $4
pushl $pedido2
call  printf

pushl $tecla
pushl $formato2
call  scanf

pushl tecla
pushl $4
pushl $mostra2
call  printf

pushl $4
pushl $pedido3
call  printf

pushl $frase
pushl $formato3
call  scanf

pushl $frase
pushl $4
pushl $mostra3
call  printf

pushl $4
pushl $pedido2
call  printf
```

```
pushl $tecla
pushl $formato2
call  scanf

pushl tecla
pushl $4
pushl $mostra2
call  printf
```

Truque! A string de formatação **“ %c”** (com um espaço em branco antes do %c) funciona quando se deseja desprezar o primeiro caractere do buffer, caso ele seja **<enter>**, **espaço em branco ou tabulação**. Se existir outros tipos de caracteres além destes antes do caractere que se deseja ler, este truque não funcionará. Programadores cuidosos implementam rotinas para esvaziar todo o buffer do teclado antes da próxima leitura, para evitar/impedir usuários brincalhões ou rackers.

5) lendo um caractere depois de número e de string usando **“ %c”**.

```
pushl $5
pushl $pedido1
call  printf

pushl $numero
pushl $formato1
call  scanf

pushl numero
pushl $5
pushl $mostra1
call  printf

pushl $5
pushl $pedido2
call  printf

pushl $tecla
pushl $formatox
call  scanf

pushl tecla
pushl $5
pushl $mostra2
call  printf

pushl $5
pushl $pedido3
call  printf

pushl $frase
pushl $formato3
call  scanf

pushl $frase
pushl $5
pushl $mostra3
call  printf
```

```
      pushl $5
      pushl $pedido2
      call  printf

      pushl $tecla
      pushl $formatox
      call  scanf

      pushl tecla
      pushl $5
      pushl $mostra2
      call  printf
```

6) testando je, jl e jmp. Outros: je, jne, jnz, jle, jg, jge, jcxz, jecxz

```
      pushl $6
      pushl $pedido4
      call  printf

      pushl $n1
      pushl $formato1
      call  scanf

      pushl $6
      pushl $pedido5
      call  printf

      pushl $n2
      pushl $formato1
      call  scanf

      pushl n2
      pushl n1
      pushl $6
      pushl $mostra4
      call  printf

      movl  n2, %ebx  # %eax e %ecx sao alterados no printf. %ebx nao.
      cmpl  n1, %ebx
      je    saoiguais  #aqui também serve o jz
      jl    n2menorn1
      jmp   n1menorn2

saoiguais:

      pushl $6
      pushl $mostra5
      call  printf
      jmp   fim

n2menorn1:

      pushl $6
      pushl $mostra6
      call  printf
      jmp   fim
```

```
n1menorn2:

      pushl $6
      pushl $mostra7
      call  printf
      jmp   fim

fim:

      pushl $6
      push  $mostra8
      call  printf
```

Explicação: A instrução lógica **cmpl** compara os dois operandos de 32 bits. Mais precisamente, ela compara o segundo operando com o primeiro, fazendo o mesmo que a instrução **subl**. As versões **cmpw** e **cmpb** também são aceitas para 16 e 8 bits. O resultado lógico da comparação (**igual**, **maior** ou **menor**) é colocado no registrador de flag da arquitetura (**EFLAGS**), de uso interno, pois não pode ser manipulado diretamente pelo programador, muito embora ele possa ser lido, analisado e modificado por meio de instruções que o fazem de forma implícita ou explícita. Por exemplo, as instruções **pushfd/popfd** podem empilhar/desempilhar os 32 bits do EFLAGS, permitindo que o programador possa recuperá-lo/alterá-lo em um registrador normal.

A instrução cmpl possui a seguinte sintaxe:

cmpl op1, op2

onde op1 e op2 podem ser memória ou registrador, mas não simultaneamente memória.

As instruções de saltos (desvios) condicionais (também chamados de jumps condicionais ou branches) **je, jz, jl, jne, jnz, jle, jg, jge, jcxz, jecxz** analisam o conteúdo do EFLAGS para tomar uma decisão de qual caminho direcionar o controle de execução. Somente o **jmp** não é condicional, pois necessariamaente salta para o destino informado pelo operando. Qualquer instrução aritmética (add, mul, div, sub ...), após executada,altera o estado do EFLAGS. Alguns jumps condicionais mais conhecidos são mostrados a seguie, mas existem muitos outros. Maiores detalhes podem ser encontrados no livro Professional Assembly Language página 137.

```
      je = salta se o resultado da comparação for “igual”
      jne = salta se o resultado da comparação for “não igual”
      jz = salta se o resultado da operação for “zero”
      jnz = salta se o resultado da operação for “não zero”
      jl = salta se o resultado da operação for “menor ou igual a zero”
      jle = salta se o resultado da operação for “menor que zero”
      jg = salta se o resultado da operação for “maior que zero”
      jge = salta se o resultado da operação for “maior ou igual a zero”
      jcxz = salta se o registrador %cx for zero
      jecxz = salta se o registrador %ecx for zero
      jo =  salta se uma operação resultou em overflow (estouro de
            capacidade em numeros com sinais)
      jc =  salta se uma operação resultou em carry (“vai um”, estouro de
            capacidade em numeros sem sinal)


#7) testando loop. ele utiliza o %ecx para decremento automatico ateh 0
```

```
      pushl $7
      pushl $pedido6
      call  printf

      pushl $ngiros
      pushl $formato1
      call  scanf
      movl  ngiros, %ecx


volta2:

      movl  %ecx, %ebx  # backup de %ecx, pois ele eh alterado no printf
      pushl %ecx
      pushl $7
      pushl $mostra9
      call  printf
      movl  %ebx, %ecx
      loop  volta2

      pushl $7
      push  $mostra10
      call  printf
```

Explicação: A instrução **loop** trabalha associada implicitamente ao registrador %ecx. Cada vez que a instrução loop é executada, ela decrementa o %ecx em 1 unidade e salta o controle de execução para o rótulo informado, nesse exemplo, volta2. Quando o %ecx atingir zero, o controle de execução não saltara mais e prosseguirá na execução da instrução seguinte a loop. Cuidado para não deixar o %ecx negativo, pois a instrução loop podera “girar indefinidamente”, provavelmente até causar segmentation fault (core dump).

DESAFIO: Faça um programa em assembly que leia N pares de números inteiros do teclado. Para cada par, o programa deve dizer se o primeiro número é menor, maior ou igual ao segundo.