Insper

07 - Loops `while` e `for`

Sistemas Hardware-Software - 2019/1

Igor Montagner

Parte 1 - loops `while` e `for`

Exercício 1: veja o código assembly abaixo(arquivo `ex1.o`) e responda os itens.

```
Dump of assembler code for function soma_2n:
    0x0000 <+0>:     mov    $0x1,%eax
    0x0005 <+5>:     cmp    $0x1,%edi
    0x0008 <+8>:     jbe    0x11 <soma_2n+17>
    0x000a <+10>:    shr    %edi
    0x000c <+12>:    add    $0x1,%eax
    0x000f <+15>:    jmp    0x5 <soma_2n+5>
    0x0011 <+17>:    retq
```

1. Localize no código acima as instruções de saltos (`jmp` ou condicionais `j*`). Desenhe setas indicando para qual linha do código elas pulam.
2. Analise o fluxo de saltos do seu código. Existe um loop? Entre quais linhas?

3. Comece fazendo uma versão *C* usando somente `if-goto`

4. Transforme a construção que você fez acima em um código usando `while`.

Vamos agora exercitar o que vimos na aula expositiva.

Exercício 2: Leia o código assembly abaixo e responda.

```
Dump of assembler code for function soma_n:
    0x0000 <+0>:     mov    $0x0,%eax
    0x0005 <+5>:     mov    $0x0,%edx
    0x0000a <+10>:   cmp    %edi,%eax
    0x000c <+12>:    jge    0x19 <soma_n+25>
    0x000e <+14>:    movslq %eax,%rcx
    0x0011 <+17>:    add    %rcx,%rdx
    0x0014 <+20>:    add    $0x1,%eax
    0x0017 <+23>:    jmp    0xa <soma_n+10>
    0x0019 <+25>:    mov    %rdx,%rax
    0x001c <+28>:    retq
```

1. Desenhe as flechas indicando o destino de cada instrução de pulo (`jmp` ou `j*`).
2. Escreva abaixo o cabeçalho da função `soma_n`. **Dica**: procure por registradores que são lidos *antes* de serem escritos.

3. Faça a tradução do código acima para *C* usando somente `if-goto`

4. Converta o código acima para uma versão legível em *C*.

Parte 2 - variáveis locais

Como visto na expositiva, variáveis locais são armazenadas na pilha. O topo da pilha é armazenado em `%rsp` e ela cresce para baixo, ou seja, ao empilhar um dado o valor de `%rsp` diminui e ao desempilhar seu valor aumenta.

O compilador faz todo o possível para usar somente os registradores, porém em alguns casos é necessário guardar a variável na memória. Isso ocorre, em geral, quando usamos `&` para computar o endereço de uma variável. O exemplo mais comum nos códigos que já escrevemos é na leitura de valores usando `scanf`.

Funções que guardam variáveis na pilha seguem um padrão facilmente identificável. Primeiro elas subtraem um valor da pilha (`0x10` no exemplo abaixo) correspondente ao tamanho total de todas as variáveis usadas. Depois temos várias instruções usando endereços relativos a `%rsp` e por fim devolvemos o espaço usado somando `0x10` de volta a `%rsp`.

```
sub $0x10, %rsp
. . . // código da função aqui!
mov    0x8(%rsp),%eax
mov    %eax,%edx
add    0xc(%rsp),%edx
. . . // função continua
add $0x10, %rsp
ret
```

Um `lea` relativo a `%rsp` **nunca** é aritmético! Pense um pouco e entenda a razão disto antes de prosseguir.

Exercício 3: Um dos casos de uso mais comuns de variáveis na pilha é a criação de variáveis passadas para `scanf`. Vamos trabalhar na análise da função `exemplo2` do executável `ex3`.

```
Dump of assembler code for function exemplo2:
    0x1149 <+0>:     push   %rbx
    0x114a <+1>:     sub    $0x10,%rsp
    0x114e <+5>:     mov    %edi,%ebx
    0x1150 <+7>:     lea    0x8(%rsp),%rdx
    0x1155 <+12>:    lea    0xc(%rsp),%rsi
    0x115a <+17>:    lea    0xea3(%rip),%rdi        # 0x2004
    0x1161 <+24>:    mov    $0x0,%eax
    0x1166 <+29>:    callq  0x1040 <__isoc99_scanf@plt>
    0x116b <+34>:    mov    0x8(%rsp),%edx
    0x116f <+38>:    mov    0xc(%rsp),%eax
    0x1173 <+42>:    lea    (%rax,%rdx,2),%eax
    0x1176 <+45>:    add    %ebx,%eax
    0x1178 <+47>:    add    $0x10,%rsp
    0x117c <+51>:    pop    %rbx
    0x117d <+52>:    retq
```

1. Quanto espaço é reservado para variáveis locais?

2. Variáveis locais são acessadas usando endereços relativos a `%rsp`. Identifique quantas existem no código acima e quais seus tamanhos. Associe um nome de variável para cada endereço listado.

3. A chamada em `exemplo2+29` é um `scanf`, que recebe como primeiro parâmetro a string de formato a ser lido (aquela com os `%d`). Use o `gdb` para mostrá-la e escreva abaixo.

4. Com base nos itens acima, escreva a chamada para o `scanf` feita em `exemplo2`.

5. O `lea` pode ser usado tanto para a operação *endereço de* (`&`) como para cálculos simples. Escreva ao lado de cada ocorrência acima se o uso é para `&` ou para aritmética.
6. Com todas essas informações em mãos, faça uma tradução da função acima para *C*

Parte 3 - Exercícios avançados

Exercício 4: Considerando o arquivo *ex4*, responda as perguntas abaixo.

```
Dump of assembler code for function ex4:
    0x1139 <+0>:     mov    $0x0,%ecx
    0x113e <+5>:     mov    $0x0,%r8d
    0x1144 <+11>:    jmp    0x114a <ex4+17>
    0x1146 <+13>:    add    $0x1,%rcx
    0x114a <+17>:    cmp    %rdi,%rcx
    0x114d <+20>:    jge    0x1161 <ex4+40>
    0x114f <+22>:    mov    %rcx,%rax
    0x1152 <+25>:    cqto
    0x1154 <+27>:    idiv   %rsi
    0x1157 <+30>:    test   %rdx,%rdx
    0x115a <+33>:    jne    0x1146 <ex4+13>
    0x115c <+35>:    add    %rcx,%r8
    0x115f <+38>:    jmp    0x1146 <ex4+13>
    0x1161 <+40>:    mov    %r8,%rax
    0x1164 <+43>:    retq
```

1. Quantos argumentos a função acima recebe? Quais seus tipos? **Dica**: não se esqueça de buscar por registradores que são lidos antes de serem escritos.

2. A função retorna algum valor? Se sim, qual seu tipo?

3. A função acima combina loops e condicionais. Desenhe setas para onde as instruções de `jmp` apontam.
4. Com base no exercício anterior, entre quais linhas o loop ocorre? E a condicional?

5. O loop acima tem uma variável contadora. Ela está em qual registrador? Qual seu tipo?

6. Revise o funcionamento da instrução `idiv`. Em qual registrador é armazenado o resultado da divisão? E o resto?

7. Qual a condição testada na condicional?

8. Escreva uma versão do código acima usando somente `if-goto`.

9. Escreva uma versão legível do código acima

Exercício 5: Considerando o arquivo *ex5* (função `main` abaixo), responda as perguntas.

```
Dump of assembler code for function main:
   0x1149 <+0>:     sub    $0x18,%rsp
   0x114d <+4>:     lea    0xc(%rsp),%rsi
   0x1152 <+9>:     lea    0xeab(%rip),%rdi        # 0x2004
   0x1159 <+16>:    mov    $0x0,%eax
   0x115e <+21>:    callq  0x1040 <__isoc99_scanf@plt>
   0x1163 <+26>:    cmpl   $0x0,0xc(%rsp)
   0x1168 <+31>:    js     0x1180 <main+55>
   0x116a <+33>:    lea    0xe9f(%rip),%rdi        # 0x2010
   0x1171 <+40>:    callq  0x1030 <puts@plt>
   0x1176 <+45>:    mov    $0x0,%eax
   0x117b <+50>:    add    $0x18,%rsp
   0x117f <+54>:    retq
   0x1180 <+55>:    lea    0xe80(%rip),%rdi        # 0x2007
   0x1187 <+62>:    callq  0x1030 <puts@plt>
   0x118c <+67>:    jmp    0x1176 <main+45>
```

1. Começaremos examinando as chamadas em `main+40` e `main+62`. Elas são para a função `puts`. Veja sua documentação (procure por *C puts.*) e explique abaixo o quê ela faz e quais são seus argumentos.

2. Examine os argumentos passados para `puts` usando o *gdb* e escreva-os abaixo. (**Dica**: você usará o comando `x`)

3. Agora olharemos as variáveis locais. Quanto espaço é reservado para elas? Liste abaixo as que você encontrou e dê um nome para cada uma.

4. Vamos agora olhar a chamada `call` em `main+21`. Quais são seus argumentos? Use o *gdb* para ver o valor do primeiro deles (usando o comando `x`). O segundo deve ser familiar dos exercícios anteriores.

5. Finalmente, faça uma versão em *C* do código acima. Se necessário faça uma versão intermediária usando `if-goto`.