Machine-Level Programming: Loops and Procedures

CMPU 224 – Computer Organization
Jason Waterman
Looping in C

• There are three looping constructs in C
  • while, do-while, and for loops

• Turn these loops into goto construct to implement in assembly

```c
while(test) {
  body;
}
do {
  body;
} while(test);
for (init; test; update) {
  body;
}
```
do-while loops

**do-while statement**

```
do
  body-statement
  while (test-expr);
```

**body-statement**

```
{
  Statement_1;
  Statement_2;
  ...
  Statement_n;
}
```

**C Goto Version**

```
loop:
  body-statement
  t = test-expr
  if (t)
    goto loop;
```
Looping: “do-while” example

• Count number of ones in argument \( x \)

• Use conditional branch to either continue looping or to exit loop

**do-while Version**

```c
long count_do(unsigned long x) {
    long result = 0;
    do {
        result += x & 0x1;
        x >>= 1;
    } while (x);
    return result;
}
```

**C Goto Version**

```c
long count_goto(unsigned long x) {
    long result = 0;
    loop:
    result += x & 0x1;
    x >>= 1;
    if(x) goto loop;
    return result;
}
```
“Do-While” Loop Compilation

Goto Version

long count_goto(unsigned long x)
{
    long result = 0;
    loop:
        result += x & 0x1;
        x >>= 1;
        if(x) goto loop;
    return result;
}

movl $0, %eax # result = 0
loop:
    movq %rdi, %rdx # x
    andl $1, %edx # t = x & 0x1
    addq %rdx, %rax # result += t
    shrq $1, %rdi # x >>= 1
    jne loop # if (x) goto loop
    ret

0000000000401106 <count_goto>:
  401106: b8 00 00 00 00 mov $0x0,%eax
  40110b: 48 89 fa mov %rdi,%rdx
  40110e: 83 e2 01 and $0x1,%edx
  401111: 48 01 d0 add %rdx,%rax
  401114: 48 d1 ef shr %rdi
  401117: 75 f2 jne 40110b <count_goto+0x5>
  401119: c3 retq

Register | Use(s)
---|---
%rdi | Argument x
%rax | result
General “While” Translation #1

- “Jump-to-test” translation
- A “do-while” loop with a jump to the first test
- Typically used with –Og

While version

```c
while (Test)  
  Body  
```

Goto Version

```c
  goto test;  
  loop:  
    Body  
  test:  
    if (Test)  
      goto loop;  
  done:
```
While Loop Example #1

C Code

```c
long count_while(unsigned long x) {
    long result = 0;
    while (x) {
        result += x & 0x1;
        x >>= 1;
    }
    return result;
}
```

Jump to Test Version

```c
long count_goto_jtt(unsigned long x) {
    long result = 0;
    goto test;
    loop:
    result += x & 0x1;
    x >>= 1;
    test:
    if(x) goto loop;
    return result;
}
```

- Initial goto starts loop at test
While Loop Example #2

**C Code**

```c
long count_while(unsigned long x) {
    long result = 0;
    while (x) {
        result += x & 0x1;
        x >>= 1;
    }
    return result;
}
```

```c
long count_goto_dw(unsigned long x) {
    long result = 0;
    if (!x) goto done;
    loop:
    result += x & 0x1;
    x >>= 1;
    if(x) goto loop;
    done:
    return result;
}
```

**Assembly**

```assembly
count_while:
    testq %rdi, %rdi          # Guard test
    je .L4                    # if (!x) goto done
    movl $0, %eax
.L3:
    movq %rdi, %rdx
    andl $1, %edx
    addq %rdx, %rax
    shrq %rdi
    jne .L3
    ret
.L4:
    movl $0, %eax
    ret
```

- Initial conditional guards the entrance to the loop
"For" Loop → While Loop

For Version

\[
\text{for (Init; Test; Update )}
\]
\[
\text{Body}
\]

While Version

\[
\text{Init;}
\]
\[
\text{while (Test ) {}
\]
\[
\text{Body}
\]
\[
\text{Update;}
\]
\[
\text{}}
\]
“For” Loop Form

General Form

\[
\text{for (Init; Test; Update) }
\]
\[
\text{Body}
\]

```c
long count_for(unsigned long x){
    size_t i;
    unsigned bit;
    long result = 0;
    for (i = 0; i < 64; i++){
        bit = (x >> i) & 0x1;
        result += bit;
    }
    return result;
}
```

Init
i = 0

Test
i < 64

Update
i++

Body
```c
{
    bit = (x >> i) & 0x1;
    result += bit;
}
```
For-While Conversion

long count_for_while(unsigned long x) {
    size_t i;
    unsigned bit
    long result = 0;
    i = 0;
    while (i < 64) {
        bit = (x >> i) & 0x1;
        result += bit;
        i++;
    }
    return result;
}

• While loop was converted to a do-while loop

count_for:
    movl $0, %edx # result = 0
    movl $0, %ecx # init: i = 0
.L7:
    movq %rdi, %rax # body
    shrq %cl, %rax # body: x >> i
    andl $1, %eax # body: (x >> i) & 1
    addq %rax, %rdx # body: result += bit
    addq $1, %rcx # update: i++
    cmpq $64, %rcx # test: i < 64
    jne .L7
    movq %rdx, %rax # return result
    ret
Mechanisms in Procedures

• Passing control
  • To beginning of procedure code
  • Back to return point

• Passing data
  • Procedure arguments
  • Return value

• Memory management
  • Allocate during procedure execution
  • Deallocate upon return

• Mechanisms are all implemented with machine instructions

• x86-64 implementation of a procedure uses only those mechanisms required

```c
int Q(int i)
{
    int t = 3*i;
    int v[10];
    // ... (remaining code)
    return v[t];
}
```
x86-64 Stack

• Region of memory for a process (running program) managed with stack discipline

• Grows toward lower addresses

• Register `%rsp` contains lowest stack address
  • address of “top” element

Stack Pointer: `%rsp`
Where Does the Stack Live?

Diagram not drawn to scale

- OS memory
- User stack (created at runtime)
- Memory mapped region for shared libraries
- Run-time heap (created by `malloc`)
- Read/write data
- Read-only code and data

Program start

Memory invisible to user code

printf function

Loaded from the executable file

2/16/2024

CMPU 224 -- Computer Organization
x86-64 Stack: Push

- `pushq Src`
  1. Fetch operand at `Src`
  2. Decrement `%rsp` by 8
  3. Write operand at address given by `%rsp`
x86-64 Stack: Pop

- **popq Dest**
  - Read *Value* at address given by \%rsp
  - Increment \%rsp by 8
  - Store value at Dest
Procedure Control Flow

• Use stack to support procedure call and return

• Procedure call: **call label**
  • Push return address on stack
  • Jump to **label**

• Return address:
  • Address of the next instruction right after call

• Procedure return: **ret**
  • Pop address from stack
  • Jump to address
Control Flow Example #1

```
00000000000400540 <multstore>:
  
  400544: callq 400550 <mult2>
  400549: mov %rax,(%rbx)
  
00000000000400550 <mult2>:
  400550: mov %rdi,%rax
  
  400557: retq
```

Stack

%rip 0x400544
%rsp 0x120

0x130
0x128
0x120

2/16/2024
CMPU 224 -- Computer Organization
Control Flow Example #2

000000000000400540 <multstore>:
  •
  •
400544: callq 400550 <mult2>
400549: mov %rax, (%rbx)
  •
  •

000000000000400550 <mult2>:
  •
400550: mov %rdi, %rax
  •
  •
400557: retq

Stack
0x130
0x128
0x120
0x118
0x118
0x130
%rip 0x400549
%rsp 0x118

%rip 0x400550
Control Flow Example #3

0000000000400540 <multstore>:
  
  400544: callq 400550 <mult2>
  400549: mov %rax, (%rbx)

0000000000400550 <mult2>:
  
  400550: mov %rdi, %rax
  
  400557: retq
Control Flow Example #4

00000000000400540 <multstore>:
•
•
400544: callq 400550 <mult2>
400549: mov %rax, (%rbx)
•
•

00000000000400550 <mult2>:
400550: mov %rdi, %rax
•
•
400557: retq

Stack

0x130
0x128
0x120
0x120

%rip 0x400549

%rsp 0x120
Procedure data flow

• First 6 arguments are stored in registers
  • Diane’s Silk Dress Cost $89

• Extra arguments are placed on the stack in the caller's frame in reverse order
  • Only allocate stack space when needed

• Return value stored $\%rax$

<table>
<thead>
<tr>
<th>1st</th>
<th>$%rdi$</th>
<th>Diane’s</th>
</tr>
</thead>
<tbody>
<tr>
<td>2nd</td>
<td>$%rsi$</td>
<td>Silk</td>
</tr>
<tr>
<td>3rd</td>
<td>$%rdx$</td>
<td>Dress</td>
</tr>
<tr>
<td>4th</td>
<td>$%rcx$</td>
<td>Cost</td>
</tr>
<tr>
<td>5th</td>
<td>$%r8$</td>
<td>8</td>
</tr>
<tr>
<td>6th</td>
<td>$%r9$</td>
<td>9</td>
</tr>
</tbody>
</table>
Stack Frames

• Holds state for a single procedure instantiation
  • From when called to procedure return
  • Callee returns before caller does

• Contents
  • Local variables (if needed)
  • Arguments 7th and above (if needed)
  • Return address

• Management
  • Space allocated when entering procedure
    • “Set-up” code
    • Includes push by call instruction
  • De-allocated when return
    • “Finish” code
    • Includes pop by ret instruction
Register Saving Conventions

• When procedure yoo calls who:
  • yoo is the caller
  • who is the callee

• Can registers be used for temporary storage?

  - Contents of register `%rdx` overwritten by who
  - This could be trouble → something should be done!
    • Need some coordination

```
  yoo:
    ...
    movq $224, %rdx
    call who
    addq %rdx, %rax
    ...
    ret

  who:
    ...
    subq $100, %rdx
    ...
    ret
```
Register Saving Conventions

• When procedure **yoo** calls **who**:
  • **yoo** is the **caller**
  • **who** is the **callee**

• How can registers be used for temporary storage?

• Conventions
  • **Caller Saved Registers**
    • Caller saves temporary values in its stack frame before the call
    • Contents are not preserved across function calls
  • **Callee Saved Registers**
    • Callee saves temporary values in its stack frame before using
    • Callee restores them before returning to caller
    • Contents are preserved across function calls
Caller Saved Register Conventions

• When procedure yoo calls who:
  • yoo is the caller
  • who is the callee
• The caller saves registers on the stack right before call, restores them right after function call
  • Callee doesn’t have to do anything

```
yoo:
  ...  
  movq $224, %rdx
  push %rdx
  call who
  pop %rdx
  addq %rdx, %rax
  ...  
  ret

who:
  ...  
  subq $100, %rdx
  ...  
  ret
```
Callee Register Saving Conventions

• When procedure yoo calls who:
  • yoo is the caller
  • who is the callee

• Callee saves registers on the stack at start of function, restores them right before returning
  • Caller doesn't have to do anything

```asm
yoo:
  . . .
  movq $224, %rbx
  call who
  addq %rbx, %rax
  . . .
  ret

who:
  push %rbx
  . . .
  subq $100, %rbx
  . . .
  pop %rbx
  ret
```
x86-64 Linux Register Usage Caller-saved

- Caller-saved registers
  - Must be saved by the caller if register value is needed after the call
  - Can be modified by procedure
- %rax
  - Return value
- %rdi, ..., %r9
  - Arguments
- %r10, %r11
  - Temporaries
x86-64 Linux Register Usage Callee-saved

- **Callee-saved registers**
  - **Callee must save & restore these registers if used**

- **%rbx, %r12, %r13, %r14, %r15**
  - Callee-saved temporaries

- **%rbp**
  - May be used as frame pointer

- **%rsp**
  - Special form of callee-saved register
  - Restored to original value upon exit from procedure
x86-64 Procedure Summary

• Current Stack Frame (“Top” to Bottom)
  • “Argument build”: Parameters for function about to call
  • Local variables if can’t keep in registers
  • Saved register context
  • Old frame pointer (optional)

• Caller Stack Frame
  • Return address
    • Pushed by call instruction
  • Arguments for this call

• Important Points
  • Stack is the right data structure for procedure call / return
    • If P calls Q, then Q returns before P
  • Recursion (& mutual recursion) handled by normal calling conventions
    • Can safely store values in local stack frame and in callee-saved registers
    • Put overflow function arguments at top of stack
    • Result returned in %rax
/* Recursive count */
long count_r(unsigned long x) {
    if (x == 0)
        return 0;
    else
        return (x & 1) + count_r(x >> 1);
}
Recursive Function Terminal Case

```c
/* Recursive count */
long count_r(unsigned long x) {
    if (x == 0)
        return 0;
    else
        return (x & 1) + count_r(x >> 1);
}
```

```
count_r:
  movl $0, %eax
  testq %rdi, %rdi
  je .L6
  pushq %rbx
  movq %rdi, %rbx
  andl $1, %ebx
  shrq %rdi
  call count_r
  addq %rbx, %rax
  popq %rbx
.L6:
  ret
```

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>x</td>
<td>Argument</td>
</tr>
<tr>
<td>%rax</td>
<td>Return value</td>
<td>Return value</td>
</tr>
</tbody>
</table>
/* Recursive count */
long count_r(unsigned long x) {
    if (x == 0)
        return 0;
    else
        return (x & 1) + count_r(x >> 1);
}

count_r:
    movl $0, %eax
    testq %rdi, %rdi
    je .L6
    pushq %rbx
    movq %rdi, %rbx
    andl $1, %ebx
    shrq %rdi
    call count_r
    addq %rbx, %rax
    popq %rbx
.L6:
    ret

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>x</td>
<td>Argument</td>
</tr>
</tbody>
</table>

2/16/2024
CMPU 224 -- Computer Organization
/* Recursive count */
long count_r(unsigned long x) {
    if (x == 0)
        return 0;
    else
        return (x & 1) + count_r(x >> 1);
}

count_r:
    movl $0, %eax
    testq %rdi, %rdi
    je .L6
    pushq %rbx
    movq %rdi, %rbx
    andl $1, %ebx
    shrq %rdi
    call count_r
    addq %rbx, %rax
    popq %rbx
    .L6:
    ret

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>x &gt;&gt; 1</td>
<td>New argument</td>
</tr>
<tr>
<td>%rbx</td>
<td>x &amp; 1</td>
<td>Callee-saved</td>
</tr>
</tbody>
</table>
/* Recursive count */
long count_r(unsigned long x) {
    if (x == 0)
        return 0;
    else
        return (x & 1) + count_r(x >> 1);
}

count_r:
    movl    $0, %eax
    testq   %rdi, %rdi
    je      .L6
    pushq   %rbx
    movq    %rdi, %rbx
    andl    $1, %ebx
    shrq    %rdi
    call    count_r
    addq    %rbx, %rax
    popq    %rbx
.L6:
    ret

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rbx</td>
<td>x &amp; 1</td>
<td>Callee-saved</td>
</tr>
<tr>
<td>%rax</td>
<td>Recursive call return value</td>
<td></td>
</tr>
</tbody>
</table>
/* Recursive count */
long count_r(unsigned long x) {
    if (x == 0)
        return 0;
    else
        return (x & 1) + count_r(x >> 1);
}

count_r:
    movl $0, %eax
    testq %rdi, %rdi
    je .L6
    pushq %rbx
    movq %rdi, %rbx
    andl $1, %ebx
    shrq %rdi
    call count_r
    addq %rbx, %rax
    popq %rbx
    .L6:
    ret

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rbx</td>
<td>x &amp; 1</td>
<td>Callee-saved</td>
</tr>
<tr>
<td>%rax</td>
<td>Return value</td>
<td></td>
</tr>
</tbody>
</table>
/* Recursive count */
long count_r(unsigned long x) {
    if (x == 0)
        return 0;
    else
        return (x & 1) + count_r(x >> 1);
}

count_r:
    movl $0, %eax
    testq %rdi, %rdi
    je .L6
    pushq %rbx
    movq %rdi, %rbx
    andl $1, %ebx
    shrq %rdi
    call count_r
    addq %rbx, %rax
    popq %rbx
  .L6:
      ret

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rax</td>
<td>Return value</td>
<td>Return value</td>
</tr>
</tbody>
</table>

register Use(s)
Observations About Recursion

• Handled Without Special Consideration
  • Stack frames mean that each function call has private storage
    • Saved registers & local variables
    • Saved return pointer
  • Register saving conventions prevent one function call from corrupting another’s data
  • Stack discipline follows call / return pattern
    • If P calls Q, then Q returns before P
    • Last-In, First-Out

• Also works for mutual recursion
  • P calls Q; Q calls P