C Flow Control

David Chisnall

February 1, 2011

Outline

Disclaimer!

- These slides contain a lot of x86 assembly!
- If you don't understand the details, don't worry
- They are there to give you an idea of how the compiler works
- You aren't expected to remember the details

Machine Code Execution

1. CPU reads an instruction from address in program counter register
2. CPU increments program counter
3. CPU executes instruction
4. Repeat from step 1

Nonlinear Code Execution

- Instruction modifies program counter as part of operation
- Simplest case: jump instruction, just sets program counter value.
- Exposed in C as goto statement.

C goto Statement

```
// This loops forever
int main(void)
{
// Jump target
start:
    goto start;
    return 0;
}
```

- The label defines a location in the instruction stream
- The goto statement says 'jump to this label'

Compiling goto

```
.globl _main
_main:                          # Export main function
L2:                             # Define a label
    jmp L2                      # Jump to the label
```

- Structure of C is obvious in generated assembly
- Each C line maps to exactly one line of assembly (in this case)

Important Difference: The `jmp` instruction can take any value. The `goto` statement can only take a label defined in this *scope*.

Conditional Jumps

- Similar to a jump instruction
- Sets the program counter, but only if the condition is met

Conditional Jumps

```
int b = 0;
if (a > 0)
{
    b = a;
}
```

- `if` statement defines a conditional.
- The next statement is only executed if the condition is true
- The next 'statement' is often a block of code
- Another way of thinking about if: Jump over this statement if the condition if false

Compiling if

```
 movl    $0, -12(%ebp)   # Store 0 in b
 cmpl    $0, 8(%ebp)     # Compare 0 to a
 jle L2                  # Jump if <=
 movl    8(%ebp), %eax   # Load a into register
 movl    %eax, -12(%ebp)# Store register into b
L2:
```

- Compare instruction sets condition flags.
- Conditional jump sets program counter if some of these flags are set

Note: Assignment is two instructions, load value into a register, then store it into memory. C lets you avoid having to think about register allocation.

Loops

- `for`, `while`, and `do...while` loops
- All implemented as test-and-jump-to-start
- `switch` statements for choices
- Implemented jump table or as nested conditionals

Functions

- Decompose programs into subprograms
- One entry point, one or more exits
- Take parameters, return at most one value
- Java equivalent: Methods

How Functions Work

1. Save current instruction pointer
2. Jump to address of new function
3. Do stuff in that function
4. Load old instruction pointer

Emulating Functions With goto

Don't ever do this in real code!

```
    void *stack[1024]; // Call stack
    int stackDepth = 0; // Stack pointer
    stack[stackDepth++] = &&ret1; // Store
        return
    goto function;              // Call function
ret1:
    printf("Returned\n");
    return 0;
function:
    printf("Called Function\n");
    stack[stackDepth++] = &&ret2;
    goto function2;
ret2:
    goto *stack[--stackDepth];
    function2:
    printf("Called Function2\n");
    goto *stack[--stackDepth];
```

And This is *Why* `goto` is Considered Harmful!

- The `goto` statement reflects how the machine works
- Humans don't really think like that though...
- Structured programming makes it *much* easier to understand the program design

Function Syntax

```
// Returns int, takes two ints as arguments
int foo(int a, int b)
{
    return a + b;
}
// Returns int, takes nothing as an argument
int bar(void)
{
    return foo(1, 2);
}
```

Compiling a Function

```
_foo:                                   # Entry point
  pushl   %ebp                          # Store frame pointer
  movl    %esp, %ebp                    # Set frame pointer
  subl    $8, %esp                      # Allocate 8 bytes on
      stack
  movl    12(%ebp), %eax                # load a
  addl    8(%ebp), %eax                 # add b to a
  leave                                 # restore stack
  ret                                   # return to caller
_bar:
  pushl   %ebp                          # Store frame pointer
  movl    %esp, %ebp                    # Set frame pointer
  subl    $24, %esp                     # Allocate 24 bytes
  movl    $2, 4(%esp)                   # Store 2 on stack
  movl    $1, (%esp)                    # Store 1 on stack
  call    _foo                          # Call foo()
```

x86 Peculiarities

- Many architectures don't have `ret` and `leave` instructions.
- Stack is explicitly managed
- Return address pushed onto the stack (or stored in register, e.g. SPARC)
- `jmp` instruction used to return
- On some architectures, *leaf functions* are faster

Allocation on the Stack

- Very fast!
- Just change a value in a register
- Only for small amounts of data
- Typical stack size is under 1MB

Stack and Cache

- Reading data from main memory takes 150+ cycles
- Reading data from cache takes 2-10 cycles
- The top of the stack is (almost) always in cache

Stack Allocations and Scope

```
int foo(int arg)
{
    int a = 12;
    if (arg > a)
    {
        int b = arg;
        // c is not valid yet
        ...
        int c; // c is valid from here
        ...
    } // b and c are invalid after this point
} // a is invalid after here
```

Scope and Lifecycle

- Variables are bits of memory
- The memory for stack variables is freed when they go out of scope
- What about pointers?

Pointers are (still) Just Numbers

```
{
    // Allocate space for 100 ints
    int a* = malloc(100*sizeof(int));
} // a goes out of scope
```

- A pointer going out of scope frees the pointer
- It does not free the pointee
- This is an example of a *memory leak*

Another Example

```
int *a = NULL;
if (someCondition)
{
    // b is on the stack
    int b = someCalculation();
// a points to b
a = &b;
} // b goes out of scope
// a now points to an invalid pointer
```

- Freeing the pointee does not change the pointer
- This is an example of a *dangling pointer*

Summary

- All C flow control is a thin wrapper around machine instructions
- The stack is a hardware construct, used for storing return addresses and temporary state