Lecture 4

CIS 341: COMPILERS
Announcements

• HW2: X86lite
  – Available on the course web pages.
  – Due: Monday, February 2\textsuperscript{nd} at 11:59:59pm
  – Pair-programming:
    • There’s a pair-search survey on Piazza
    • Register the group on the submission page
    • Submission by any group member counts for the group

• Registration:
  – If you were on the wait list, you should have been contacted
  – If you are \textit{not} registered, please see me after class
Our current focus: X86 target language
See www.cis.upenn.edu/~cis341/15sp/hw/hw2/x86lite.shtml
Compilation & Execution

Source code

Compiler

Assembly Code

Assembler

Object Code

Linker

Library code

Fully-resolved machine Code

Loader

Executable image

foo.c

gcc -S

foo.s

as

foo.o

ld

foo

(Usually: gcc -o foo foo.c)
PROGRAMMING IN X86LITE
3 parts of the C memory model

• The code & data (or "text") segment
  – contains compiled code, constant strings, etc.
• The Heap
  – Stores dynamically allocated objects
  – Allocated via "malloc"
  – Deallocated via "free"
  – C runtime system
• The Stack
  – Stores local variables
  – Stores the return address of a function
• In practice, most languages use this model.
Local/Temporary Variable Storage

- Need space to store:
  - Global variables
  - Values passed as arguments to procedures
  - Local variables (either defined in the source program or introduced by the compiler)

- Processors provide two options
  - Registers: fast, small size (32 or 64 bits), limited number
  - Memory: slow, very large amount of space (4+ GB)

- In practice on X86:
  - Registers are limited (and have restrictions)
  - Divide memory into regions including the stack and the heap
Calling Conventions

• Specify the locations (e.g. register or stack) of arguments passed to a function
• Designate registers either:
  – Caller Save – e.g. freely usable by the called code
  – Callee Save – e.g. must be restored by the called code
• Protocol for deallocating stack-allocated arguments
  – Caller cleans up
  – Callee cleans up (makes variable arguments harder)
32-bit cdecl calling conventions

• “Standard” on X86 for many C-based operating systems (i.e. almost all)
  – Still some wrinkles about return values (e.g. some compilers use EAX and EDX to return small values)
  – This is evolving due to 64 bit (which allows for packing multiple values in one register)

• Arguments are passed on the stack in right-to-left order
• Return value is passed in EAX
• Registers EAX, ECX, EDX are caller save
• Other registers are callee save
  – Ignoring these conventions will cause havoc (bus errors or seg faults)

• Many other variants: fastcall, syscall, thiscall
x86-64 calling conventions

• Microsoft x64
  – Used by Visual C++ and Windows (but supported by gcc, intel C++, etc.)
  – 4 register arguments
  – 4-quad “shadow space”

• System V AMD64 ABI
  – Used by linux, bsd, Mac OSX
  – First six integer/pointer arguments are passed in registers:
    • rdi, rsi, rdx, rcx, r8, r9
    • Arguments seven and up, passed on the stack
  – Stack aligned on 16-byte boundaries
  – Callee save registers: rbp, rbx, r12—r15
  – Caller save register: everything else
  – Caller cleans up stack arguments
Call Stacks: Caller’s protocol

- **Function call:**
  
  \[ f(e_1, e_2, \ldots, e_n); \]

  1. Save caller-save registers:
     - all but rbp, rbx, r12-r15
  2. Evaluate \( e_1 \) to \( v_1 \), \( e_2 \) to \( v_2 \), \ldots, \( e_n \) to \( v_n \)
  3. Move \( v_1 \) .. \( v_6 \) into registers as on previous slide.
  4. Push \( v_7 \) to \( v_n \) onto the top of the stack.
  5. Use `call` to jump to the code for \( f \)
     - pushing the return address onto the stack.

- **Invariant:** returned value passed in `rax`

- **After call:**
  1. clean up the pushed arguments by popping the stack.
  2. Restore caller-saved registers

---

State of the stack just after the Call instruction:

`return addr.

\( v_7 \)

\( v_8 \)

\( \ldots \)

\( v_n \)

local variables`
Call Stacks: Callee’s protocol

• On entry:
  1. Save old frame pointer
     – rbp is callee save
  2. Create new frame pointer
     – movq rsp, rbp
  3. Allocate stack space for local variables.

• Invariants: (assuming quad-size values)
  – Function argument n > 6 is located at:
    rbp + (n-5) * 8
  – Local variable local\(_j\) is located at:
    rbp – (j – 1) * 8

• On exit:
  1. Pop local storage
  2. Restore rbp

State of the stack after Step 3 of entry.