The Runtime Environment
Roadmap

Type checking
– Discussed a couple of points in the design space of type systems
– Showed how to infer and check the types of expressions in b
– Showed how to propagate type errors

Today
– Begin looking at how to “lower” the code down to the assembly-code level
Outline

• Talk about what a runtime environment is
• Discuss the “semantic gap”
  – The difference between the level of abstraction in source code and executables
• How memory is laid out in the address space of a machine
What Abstractions are We Missing?

- Loops
- Variables
- Scope
- Functions

- Flat list of opcodes
- Byte-addressable memory
What You See (in source code) Is Not What You eXecute

– We think in terms of high-level abstractions
– Many of these abstractions have no explicit representation in machine code
– [Complicates looking for bugs and security vulnerabilities in machine code]
Runtime Environment

Underlying software and hardware configuration assumed by the program

– May include an OS (may not!)
– May include a virtual machine
The Role of the Operating System

Program piggybacks on the OS

– Provides functions to access hardware
– Provides illusion of uniqueness
– Enforces some boundaries on what is allowed
Mediation is Slow

It is up to the compiler to use the runtime environment as best it can

– Limited number of very fast registers with which to do computation
– Comparatively large region of memory to hold data
– Some basic instructions from which to build more complex behaviors
Conventions

Assembly code enforces very few rules

- We’ll have to impose conventions on the way our program accesses memory

These conventions help to guarantee that separately developed code works together

- Allows modularity
- Increases (programmer) efficiency
Issues to Consider

Variables
– How are they stored?
– What happens when a variable’s value is needed?

How do functions work?
– What should happen when client code calls a function?
– What should happen when a function is entered?
– What should happen when a function returns?
General Memory Layout

• Think of program memory as a single array

• Individual memory cells can be accessed via their address
  – Represent address using a hex value
  – Represent contents using a decimal value

• Very common to represent program memory as a “tower”
  – Low addresses at the “top”
  – High addresses at the “bottom”
  – [I actually prefer high at the top]
How do We Divide up Memory?

Goals

– Flexibility
– Efficiency
– Speed
Memory Layout : Static Allocation

- Region for global memory
- 1 “frame” for each procedure of the program
  - Memory “slot” for each local, parameter
  - “slot” for caller
- Fast, but impractical
  - Why?
Memory: The Stack

Keep the procedure-frame idea, but allocate *per invocation*

- Also known as “activation records”
- We don’t know statically how many frames there might be
  - Fix a point in memory for the base; grow from there
  - By convention, grows from high addresses to low addresses
A Closer Look at Activation Records (ARs)

• Push a new frame on function entry
• Pop the frame on function exit
• To reduce the size, put static data in the global area
  – In particular, string constants
• Conceptually, allows infinite recursion
  – In practice, the stack can grow so large that it hits the global data

```c
foo(int arg1, int arg2)
{
    int local1 = arg1 - arg2;
    if (local1 > 0) { foo(arg1, 3); }
}
main()
{
    int main_local = 7;
    foo(5, 4);
}

Disclaimer:
High-level idea only
```
Activation Records: Dynamic Locals

• The stack can handle local variables whose size is unknown
  – Grow the frame as needed during its execution

• Consequently, the stack size is not known at compile time!
  – Store the previous frame’s boundaries in the current frame
  – In essence, there is a linked list of activation records
Activation Record: Summary

Items in the AR

- Local variables
- Info about the call made by the caller
  - Data context
    - Enough info to determine the boundaries of the frame in use when the current procedure was called
  - Control context
    - Enough info to know what code invoked the current procedure
Non-Local Dynamic Memory

• Surely we don’t want *all* data allocated in a function call to disappear on return
• Don’t know how much space we’ll need
  – Can allocate many such objects
  – The sizes can vary dynamically

```java
public makeList(){
    Node n = new Node();
    Node t = new Node();
    n.next = t;
    return n;
}
```
The Heap

- Region of memory independent of the stack
- Allocated according to calls in the program
- How do we give it back?
  - Programmer specifies when it will no longer be used (C)
  - Runtime environment can determine automatically when it could no longer be used (Java)
The Whole Picture

• The code resides in memory at addresses less than the region for the global variables
Function Calls

Where convention meets implementation

– Function calls are so common that their semantics are partially encoded into architecture
– Registers often have “nicknames” that hint at their purpose in representing ARs (fp, sp)
– Some instructions implement “shortcuts” for building up and breaking down ARs
When are We “In” a Function?

- $ip$ the instruction pointer tracks the line (address) of the code that is executing. It tracks “where we are at” in the program.

- If $ip$ points to code that was generated for some function, we’ll say we’re in that function.

```cpp
#1 int summation(int max){
#2   int sum = 1;
#3   for (int k = 1 ; k <= max ; k++){
#4     sum += k;
#5   }
#6   return sum;
#7 }
#8 void main(){
#9   int x = summation(4);
#10  cout << x;
#11 }
```

$ip$: #2
Caller / Callee relationship

Caller
– The function doing the invocation

Callee
– The function being invoked

Note that this is a per-call relationship
– main is the caller at line 5
– v is the callee at line 5

1. void v() {
2. }
3.
4. int main() {
5.     v();
6. }
How ARs are *Actually* Implemented

Two registers track the stack

– Frame pointer ($fp$) tracks the base of the stack
– Stack pointer ($sp$) tracks the top of the stack
Store the *caller-saved* registers in its own AR

Set up the actual parameters
- Set aside a slot for the return value
- Push parameters onto the stack

Copy return address out of $ip
- It is about to get overwritten

Jump to the first instruction of the callee
- Changes $ip
Function Entry: Callee Responsibilities

- Save $fp (because we need to restore it when the callee returns)
- Update the base of the new AR to be to end of the old AR
- Save callee-saved registers if necessary
- Make space for locals
Function Exit: Callee Responsibilities

- Set the return value
- Restore callee-saved registers
- Grab stored return address
- Restore *old $sp*: fixed (positive) offset from the current base of the stack
- Restore *old $fp*: also from stack
- Jump to the stored return address

Diagram:
- $ra$ After Call site
- $ip$ After Call site
- $sp$ After Call site
- $fp$ After Call site

Low memory addresses:
- Callee Locals
- Caller $fp$ (control link)
- Caller $ip$ after call
- Params
- Return value
- Caller-saved Registers
- Caller’s AR

High memory addresses:
- $sp$ $fp$ $sp$ $fp + X$
Function Exit: Caller Responsibilities

Pop the return value (or copy from register)
Restore caller-saved registers
The Stack = A Linked List of ARs

Linked List head

Linked List of ARs $fp$

with no space in-between list elements
Hardware Support for Functions

Calls
- JAL (Jump and Link): MIPS instruction that puts $ip in $ra, and then sets $ip to a given address
- Call: x86 instruction that pushes $ip directly onto the stack, then sets $ip to a given address

Return
- JR (Jump Return): MIPS instruction that sets $ip to $ra
- ret: x86 instruction that pops directly off the stack into $ip

SPARC “Sliding Windows”
- Crazy system where caller registers are automatically saved, new set of callee saved registers automatically exposed
Next Time

Variable accesses
- We’ve shown how to store variables
- How do we actually access them?
  - What about scopes?

Upcoming: MIPS
- We will fix a concrete runtime environment, not just a pseudocode machine