

Run-time Environments

Lecture 14

Prof. Necla CS 164 Lecture 14

1

Status

- We have covered the front-end phases
 - Lexical analysis
 - Parsing
 - Semantic analysis
- Next are the back-end phases
 - Optimization
 - Code generation
- We'll do code generation first . . .

Prof. Necla CS 164 Lecture 14

2

Run-time environments

- Before discussing code generation, we need to understand what we are trying to generate
- There are a number of standard techniques that are widely used for structuring executable code

Prof. Necla CS 164 Lecture 14

3

Outline

- Management of run-time resources
- Correspondence between static (compile-time) and dynamic (run-time) structures
- Storage organization

Prof. Necla CS 164 Lecture 14

4

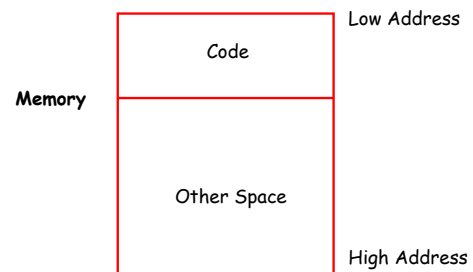
Run-time Resources

- Execution of a program is initially under the control of the operating system
- When a program is invoked:
 - The OS allocates space for the program
 - The code is loaded into part of the space
 - The OS jumps to the entry point (i.e., "main")

Prof. Necla CS 164 Lecture 14

5

Memory Layout



Prof. Necla CS 164 Lecture 14

6

Notes

- Our pictures of machine organization have:
 - Low address at the top
 - High address at the bottom
 - Lines delimiting areas for different kinds of data
- These pictures are simplifications
 - E.g., not all memory need be contiguous
- In some textbooks lower addresses are at bottom

Prof. Necula CS 164 Lecture 14

7

What is Other Space?

- Holds all data for the program
- Other Space = Data Space
- Compiler is responsible for:
 - Generating code
 - Orchestrating use of the data area

Prof. Necula CS 164 Lecture 14

8

Assumptions about Execution

1. Execution is sequential; control moves from one point in a program to another in a well-defined order
2. When a procedure is called, control eventually returns to the point immediately after the call

Do these assumptions always hold?

Prof. Necula CS 164 Lecture 14

9

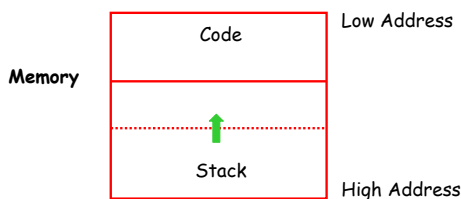
Activation Frames

- We must data for each function invocation:
 - Storage for local variables and actual arguments
 - Return address: to resume execution of its caller
 - We store such data in an activation frame
- Activation frames are linked
 - Control link: pointer to caller's activation frame
- A function invocation terminates before the invocation of its caller terminates
 - Activation frames form a stack.
 - Top of the stack is the current activation frame

Prof. Necula CS 164 Lecture 14

10

Revised Memory Layout



- On many machines the stack starts at high-addresses and grows towards lower addresses

Prof. Necula CS 164 Lecture 14

11

Example 2, Revisited

```
Class Main {
  g() : Int { 1 };
  f(x: Int): Int { if x=0 then g() else f(x - 1)(**)fi };
  main(): Int {{f(3); (*) }};
```

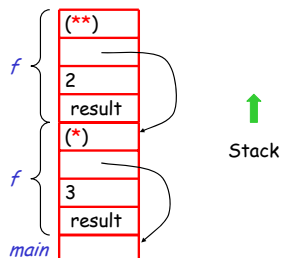
AR for f:

return address
control link
argument
space for result

Prof. Necula CS 164 Lecture 14

12

Stack After Two Calls to f



Prof. Neucula CS 164 Lecture 14

13

Notes

- `main` has no argument or local variables and its result is never used; its AR is uninteresting
- `(*)` and `(**)` are return addresses of the invocations of `f`
 - The return address is where execution resumes after a procedure call finishes
- This is only one of many possible AR designs
 - Would also work for C, Pascal, FORTRAN, etc.

Prof. Neucula CS 164 Lecture 14

14

The Main Point

The compiler must determine, at compile-time, the layout of activation records and generate code that correctly accesses locations in the activation record

Thus, the AR layout and the code generator must be designed together!

Prof. Neucula CS 164 Lecture 14

15

Discussion

- The advantage of placing the return value 1st in a frame is that the caller can find it at a fixed offset from its own frame
 - The caller must write the return address there
- There is nothing magic about this organization
 - Can rearrange order of frame elements
 - Can divide caller/callee responsibilities differently
 - An organization is better if it improves execution speed or simplifies code generation

Prof. Neucula CS 164 Lecture 14

16

Discussion (Cont.)

- Real compilers hold as much of the frame as possible in registers
 - Especially the method result and arguments

Prof. Neucula CS 164 Lecture 14

17

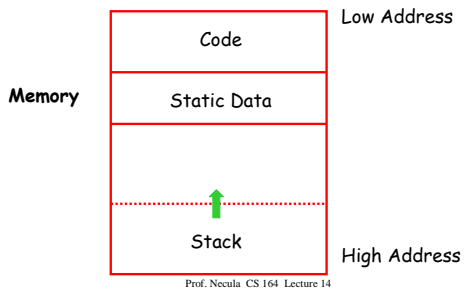
Globals

- All references to a global variable point to the same object
 - Can't store a global in an activation record
- Globals are assigned a fixed address once
 - Variables with fixed address are "statically allocated"
- Depending on the language, there may be other statically allocated values

Prof. Neucula CS 164 Lecture 14

18

Memory Layout with Static Data



Notes

- The code area contains object code
 - For most languages, fixed size and read only
 - The static area contains data (not code) with fixed addresses (e.g., global data)
 - Fixed size, may be readable or writable
 - The stack contains an AR for each currently active procedure
 - Each AR usually fixed size, contains locals
 - Heap contains all other data
 - In C, heap is managed by *malloc* and *free*
- Prof. Necula CS 164 Lecture 14 21

Heap Storage

- A value that outlives the procedure that creates it cannot be kept in the AR
 - `method foo() { new Bar }`
 - The `Bar` value must survive deallocation of `foo`'s AR
 - Languages with dynamically allocated data use a heap to store dynamic data
- Prof. Necula CS 164 Lecture 14 20

Notes (Cont.)

- Both the heap and the stack grow
 - Must take care that they don't grow into each other
 - Solution: start heap and stack at opposite ends of memory and let them grow towards each other
- Prof. Necula CS 164 Lecture 14 22

Memory Layout with Heap

