Programming Languages & Translators

RUN-TIME ENVIRONMENTS

Baishakhi Ray

Fall 2018

These slides are motivated from Prof. Alex Aiken: Compilers (Stanford)

- We have covered the front-end phases
  - Lexical analysis
  - Parsing
  - Semantic analysis

  } All the compilation errors are caught in this phase

- Next are the back-end phases
  - Code generation
  - Optimization

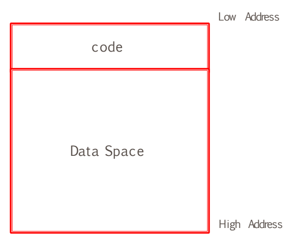
Run-time environments

- What are we trying to generate?
- How executable code is laid out?

Run-time Processes

- 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")

Memory Layout



- By tradition
  - Low address at the top
  - High address at the bottom
  - Lines delimiting areas for different kinds of data
- Simplified representation
  - Not all memory need be contiguous
- Compiler is responsible for:
  - Generating code
  - Orchestrating use of the data area

Code Generation Goals

- Two goals:
  - Correctness
  - Speed
- Most complications in code generation come from trying to be fast as well as correct

---

Assumptions about Execution

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

Activations

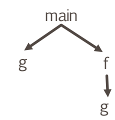
- An invocation of procedure P is an activation of P
- The lifetime of an activation of P is
  - All the steps to execute P
  - Including all the steps in procedures P calls
- The *lifetime* of a variable x is the portion of execution in which x is defined
  - Lifetime is a dynamic (run-time) concept
  - Scope is a static concept

Activation Trees

- Assumption (2) requires that when P calls Q, then Q returns before P does
- Lifetimes of procedure activations are properly nested
- Activation lifetimes can be depicted as a tree

- Example:

```
Class Main {
 int g() { 1 };
 int f() { g() };
 int main() { g(); f(); };
}
```



Example 2

```
Class Main {
 int g(){1};
 int f(int x){
   if(x == 0) g();
   else f(x-1);
 };
 int main() {f(3);};
}
```

Activation Trees

- The activation tree depends on run-time behavior
- The activation tree may be different for every program input
- Since activations are properly nested, a stack can track currently active procedures

Activation Trees

- Example:

```
Class Main {
 int g() { 1 };
 int f() { g() };
 int main() { g(); f(); };
}
```

main

Stack

main

Activation Trees

- Example:

```
Class Main {
 int g() { 1 };
 int f() { g() };
 int main() { g(); f(); };
}
```

Stack

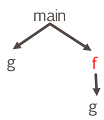
main

g

Activation Trees

- Example:

```
Class Main {
 int g() { 1 };
 int f() { g() };
 int main() { g(); f(); };
}
```

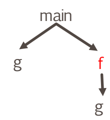


Stack
main
g f

Activation Trees

- Example:

```
Class Main {
 int g() { 1 };
 int f() { g() };
 int main() { g(); f(); };
}
```

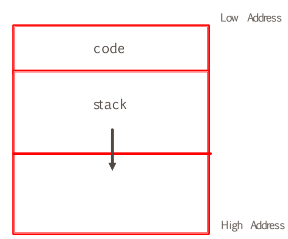


Stack
main
g f
g

Example 2

```
Class Main {
 int g(){1};
 int f(int x){
   if(x == 0) g();
   else f(x-1);
 };
 int main() {f(3);};
}
```

Revised Memory Layout



Activation Records

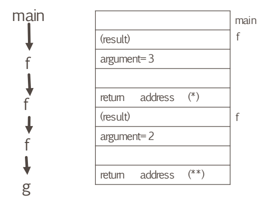
- The information needed to manage one procedure activation is called an activation record (AR) or frame.
- If procedure F calls G, then G's activation record contains a mix of info about F and G.
  - F is "suspended" until G completes, at which point F resumes.
  - G's AR contains information needed to resume execution of F.
  - G's AR may also contain:
    - G's return value (needed by F)
    - Actual parameters to G (supplied by F)
    - Space for G's local variables

The Contents of a Typical AR for G

- Space for G's return value
- Actual parameters
- Pointer to the previous activation record
  - The control link; points to AR of caller of G
- Machine status prior to calling G
  - Contents of registers & program counter
  - Local variables
- Other temporary values

Example 2

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



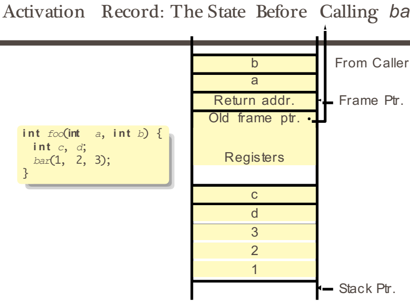
Discussion

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

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

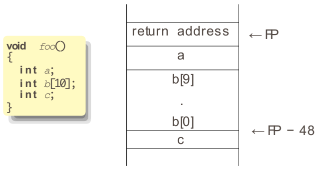
Thus, the AR layout and the code generator must be designed together.

An Activation Record: The State Before Calling *bar*



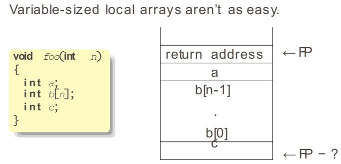
Allocating Fixed-Size Arrays

Local arrays with fixed size are easy to stack.



Allocating Variable-Sized Arrays

Variable-sized local arrays aren't as easy.

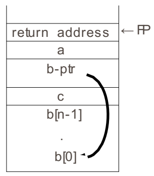


Doesn't work: generated code expects a fixed offset for c.
Even worse for multi-dimensional arrays.

Allocating Variable-Sized Arrays

As always:
add a level of indirection

```
void foo(int n)
{
  int a;
  int b[n];
  int c;
}
```

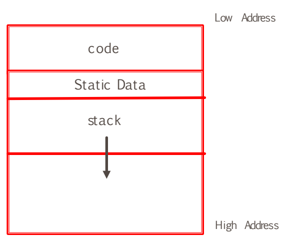


Variables remain constant offset from frame pointer.

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

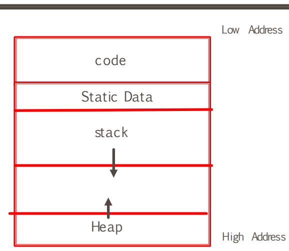
Revised Memory Layout



Heap Storage

- A value that outlives the procedure that creates it cannot be kept in the AR .
- Eg. 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

Revised Memory Layout



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
- 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

Data Layout

- Low-level details of machine architecture are important in laying out data for correct code and maximum performance
- Chief among these concerns is alignment

Alignment

- Most modern machines are (still) 32 bit
  - 8 bits in a byte
  - 4 bytes in a word
  - Machines are either byte or word addressable
- Data is word aligned if it begins at a word boundary
- Most machines have some alignment restrictions or performance penalties for poor alignment
  - SPARC and ARM prohibit unaligned accesses
  - MIPS has special unaligned load/store instructions
  - x86, 68k run more slowly with unaligned accesses
- Example: A string "Hello" Takes 5 characters (without a terminating \0)
  - To word align next datum, add 3 "padding" characters to the string -
  - The padding is not part of the string, it's just unused memory

Padding

- To avoid unaligned accesses, the C compiler pads the layout of unions and records.
- Rules:
  - Each n-byte object must start on a multiple of n bytes (no unaligned accesses).
  - Any object containing an n-byte object must be of size m*n for some integer m (aligned even when arrayed).

```
struct padded {
        int x;    /* 4 bytes */
        char z;   /* 1 byte */
        short y;  /* 2 bytes */
        char w;   /* 1 byte */
};
```

x	x	x	x
y	y		z
			w

```
struct padded {
        char a;   /* 1 byte */
        short b;  /* 2 bytes */
        short c;  /* 2 bytes */
};
```

b	b		a
		c	c

Unions

- A C struct has a separate space for each field; a C union shares one space among all fields

```
union intchar {
 int i;  /* 4 bytes */
 char c; /* 1 byte */
};
```

i	i	i	i/c

```
union twostructs {
 struct {
      char c; /* 1 byte */
      int i; /* 4 bytes */
   } a;
 struct {
      short s1; /* 2 bytes */
      short s2; /* 2 bytes */
  } b;
}
```

			c
i	i	i	i

or

s2	s2	s1	s1