Lecture 15 Outline

- Run time environments
- Correspondence between static (compile time) and dynamic (run-time) structures
- Static storage management
- Stack-based storage management

Reading Assignment:

Section 7.1-7.4

Finished with front end of compiler -
- lexical analysis
- syntax analysis
- semantic analysis

Next come the back end of compiler - machine dependent
- optimization
- target code generation

Will discuss code generation first

Must understand how program executes - what kind of code we must generate

Execution of a program is initially under the control of the operating system

When a program is invoked, several things happen:
- the operating system allocates space for the program
- the code is loaded into part of the space
- jump to entry point of the code - the main program

<table>
<thead>
<tr>
<th>Program Space</th>
<th>Low address</th>
</tr>
</thead>
<tbody>
<tr>
<td>Code</td>
<td></td>
</tr>
<tr>
<td>Other space</td>
<td>High address</td>
</tr>
</tbody>
</table>

Note: not necessary for all program space to be contiguous
What is the purpose of "other space"?
- Holds all data that the program needs and creates
- Other space = Data space for program

Compiler is responsible for:
- generating the code
- orchestrating/managing the use of the data area

Main concerns in code generation are:
- correctness
- efficiency – speed and memory

Many complications in code generation are motivated by efficiency concerns.
Assume the following about program execution:
- Execution is sequential: control moves from one point in the program to another
- in a well-defined order
- When a procedure is called, control eventually returns to the point immediately
- after the point of call

Will relax restrictions later.

As program executes, must allocate and deallocate storage to data area.
Run time routines are required to manage storage of code & data.

The routines should organize storage in such a fashion that the semantics of the
language can be implemented.

The type of data organization/management scheme depends upon the features in the
language:
- data structures – static vs. dynamic
- control structures
  - procedures - tasks - blocks - methods
  - scoping rules for these structures – static vs. dynamic
  (note sometimes even loops define a new scope)

Based upon the features present, we use the following types of management
schemes:
- Static: variables stored in statically allocated area example: global variables
- Stack: dynamically allocates storage for procedure activations
  example: recursive procedures
- Heap: objects that cannot be stacked, i.e., last created is not the first
  destroyed
  example: coroutines, tasks, pointer-objects

Combinations are also used - static, stack and heap.
Procedures major factor

- An invocation of procedure P is an activation of P - creates an instance of procedure P.
- The lifetime of an activation of P:
  - all steps taken in executing P and
  - all steps in procedures that P calls.
- The lifetime of a variable x is the portion of execution in which x is defined.

Note that:
- Lifetime is a dynamic (run-time) concept.
- Scope is a static concept.
- Storage for procedure allocated when a procedure is made as it creates a new instance of the procedure.
- Storage deallocated when procedure has finished executing & return is made.

Assumption made requires that when P calls Q, then Q returns before P.

If P1 and P2 are two procedure activations then the lifetimes are either disjoint or nested.

Activation lifetimes can be depicted as a tree.

Example

```java
Class Main {
    g(): Int { 1; }
    f(): Int { g(); }
    main(): Int { g(); f(); }
}
```

Main
Example 2

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

What is the activation tree for this example?

Notes

- The activation tree depends on run-time behavior
- The activation tree may be different for every program input
- Need to keep track of procedure activations during execution

Activation Record

The information needed to manage one procedure activation is called an activation record or frame.
Contains information to execute the procedure itself
Also contains information to implement call/return

- If procedure F calls G then G's activation record contains information about F and G
  - G is suspended until G completes, at which point F resumes. G's activation record must contain 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.
Contents of a Typical AR for G:

- **Local data** - storage for data declared in the procedure - layout as we discussed
- **temporaries** - arising during evaluation of expressions
- **actual parameters** - note: caller will attempt to pass these parameters in registers
- **return value** - note: attempt is made to return it in a register
- **control link** - points to AR of the caller
- **access link** - point to static parent - used to access non local and non global data
- **saved machine status** - return address, register values (some machines will provide multiple sets of registers to avoid the cost of saving)

Storage allocation strategies

**Static Allocation**
- layout storage for all data objects at compile time, e.g. names are bound to storage as the program is compiled
- bindings do not change, i.e., every time a procedure is activated its names are bound to the same storage allocation
- values of variables can be retained across procedure activations (typically - depends on language)

We already know how to compute offset from the base of the AR
- Once we decide the order in which ARs are stored we can compute the offset from the base of the entire AR
- if we know where each variable is and its position never changes
- Even if variables are shared by different program modules, we can find the locations - no indirection is needed/no dynamic information needed

**Limitations of static**
- cannot implement recursion/functions are not reentrant
- maximum required storage must be allocated

**Advantages:**
- fast
- easy to implement

**Stack based storage management scheme**
- enables recursion
- supports block structured languages - storage is allocated when it is needed - i.e., when procedure is called, block is entered
- termination causes deallocation of storage i.e., when return from procedure, exit from block

Consider 2 types of stack allocation schemes
- Flat level of nesting = C locals and globals
- Full block structured language = locals and non-locals and globals
Revise storage picture: Storage divided into areas for
- code
- static data
- stack for procedure activation
- PC points to currently executing instructions
- TOP points to top of stack
- Stack is organized in contiguous storage

How is information for a procedure structured?

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:

<table>
<thead>
<tr>
<th>result</th>
<th>argument</th>
<th>control link</th>
<th>return address</th>
</tr>
</thead>
</table>

Stack After Two Calls to f

Main

f
  (result)
  3
  (*)
  (result)
  2
  (**)

  f

  (***)
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.

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!

Example

The picture shows the state after the call to 2nd invocation of f returns

Main

f

f

(result)
3
(*)
1
2
(**)
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.

We need to know

• which AR is active – frame pointer
• where locals are – current AR
• where globals are – static area
• where to put temporaries – stack pointer – first available location on top of stack.

Assume

P calls
Q

We know

FP
TOP

Caller's responsibility

Callee's responsibility

1. caller evaluates actual parameters
2. caller stores return address & old FP in callee's AR - control link - caller sets FP to its new position
3. caller saves registers and other status info
4. callee initializes its own data & begins execution
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

Memory Layout with Static Data

<table>
<thead>
<tr>
<th>Low Address</th>
<th>High Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>Code</td>
<td>Memory</td>
</tr>
<tr>
<td>Static Data</td>
<td>Stack</td>
</tr>
</tbody>
</table>

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
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

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 the grow towards each other

Memory Layout with Heap
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
- 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

Alignment (Cont.)

- 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