Run-time Environments

Adapted from Lectures by
Profs. Alex Aiken and George Necula (UCB)
and Prof. Saman Amarasinghe (MIT)

Compiler Back-end Phases

- Code generation
- Optimization
- Before discussing code generation, we need to understand what we are trying to generate...
- Management of run-time resources
  - Memory, registers, CPU, etc
- Storage organization
  - Object layouts, stack layout, etc
- Correspondence between static (compile-time) and dynamic (run-time) structures

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")

Notes

- By tradition, pictures of machine organization have:
  - Low address at the top
  - High address at the bottom
  - Lines delimiting areas for different kinds of data are simplifications (e.g., not all memory contiguous)
- Compiler is responsible for:
  - Generating code for performing computations
  - Orchestrating use of the data area

Memory Layout

A Simple Example in OOPL

```java
class vector {
    int[] v;
    void add (int x) {
        int i = 0;
        while (i < v.length) {
            v[i] = v[i]*x;
            i = i+1;
        }
    }
}
```
Representing Arrays

- Items Stored Contiguously In Memory
- Length Stored In First Word
- Color Code
  - Red - generated by compiler automatically
  - Blue - program data or code
  - Magenta - executing code or data

Representing Vector Objects

- First Word Points to Class Information
  - Method Table
- Next Words Have Object Fields
  - For vectors, First Word is Reference to Array

Invoking Vector Add Method

```
vect.add(1);
```
- Create Activation Record
  - this onto stack
  - parameters onto stack
  - space for locals on stack
void add(int x) {
    int i;
    i = 0;
    while (i < v.length)
        v[i] = v[i]+x;
    i = i+1;
}

Executing Vector Add Method

void add(int x) {
    int i;
    i = 0;
    while (i < v.length)
        v[i] = v[i]+x;
    i = i+1;
}

Executing Vector Add Method

void add(int x) {
    int i;
    i = 0;
    while (i < v.length)
        v[i] = v[i]+x;
    i = i+1;
}

Executing Vector Add Method

void add(int x) {
    int i;
    i = 0;
    while (i < v.length)
        v[i] = v[i]+x;
    i = i+1;
}
void add(int x) {
    int i;
    i = 0;
    while (i < v.length)
        v[i] = v[i] + x;
    i = i + 1;
}

\[ \text{add method for vector} \]
Executing Vector Add Method

```java
void add(int x) {
    int i;
    i = 0;
    while (i < v.length)
        v[i] = v[i]+x;
    i = i+1;
}
```

Compiling Java Arithmetic Expressions

- Stack machine-based typed bytecodes
- Type coercion
- Instance vs class method

Java Example Eg1: Class method

```java
class Eg1 {
    static double f(double a, int i, int j) {
        return ((i + a) * j);
    }
    public static void main(String[] args) {
        System.out.println(f(0.0,1,2));
    }
}
```

Java Bytecodes for return expression in Eg1

```java
Method double f(double, int, int)  
  0 iload_2  
  1 i2d     
  2 dload_0 
  3 dadd    
  4 iload_3 
  5 i2d    
  6 dmul   
  7 dreturn
```
Java Example Eg2: Instance method

```java
class Eg2 {
    double f(double a, int i, int j) {
        return ((i + a) * j);
    }
    public static void main(String[] args) {
        System.out.println((new Eg2()).f(0.0, 1, 2));
    }
}
```

Java Bytcodes for return expression in Eg2

```java
Method double f(double, int, int)
0 iload_3
1 i2d
2 dload_1
3 dadd
4 iload 4
6 i2d
7 dmul
8 dreturn
```

Java Example Eg3: Formals and Locals

```java
class Eg3 {
    double f(double a, int i) {
        int j = 2;
        return ((i + a) * j);
    }
    public static void main(String[] args) {
        System.out.println((new Eg3()).f(0.0, 1));
    }
}
```

Java Bytcodes for body of f() in Eg3

```java
Method double f(double, int)
0 iconst_2
1 istore 4
3 iload_3
4 i2d
5 dload_1
6 dadd
7 iload 4
9 i2d
10 dmul
11 dreturn
```

Code Generation Goals

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

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
  - Note that
    - 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:
  - The activation tree may be different for every program input.
  - Since activations are properly nested, a stack can track currently active procedures.

**Example**

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

**Example 1**

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

**Revised Memory Layout**

```
Memory
  Code
  Stack

Low Address

Stack

High Address
```
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. So, $G$’s AR contains information needed to resume execution of $F$.

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
  - Arising during expression evaluation

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); (**)}; 
}
```

AR for $f$:
- return address
- control link
- argument
- result
- other temporary values

Stack After Two Calls to $f$

![Diagram showing stack after two calls to $f$]

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
    (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

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

```
Memory
  Code
  Static Data
  Stack

Low Address

High Address
```

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

Illegal C Example

```
typedef int (* proc)(void);
proc g (int x) {
    int f(void) {
        return x;
    }
    return f;
}
```

- Cf. Closures in Scheme, ML, etc
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

Notes (Cont.)

• 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

Memory Layout with Heap

Memory

Low Address

Code

Static Data

Stack

High Address

Heap

Alignment

• 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

Alignment (Cont.)

• Most machines have some alignment restrictions
  - Or performance penalties for poor alignment
• 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

Parameter Passing Mechanisms

• Different approaches
  - Call by value, Call by reference, Call by name, Call by value-result
• Different implementations
  - Via run-time stack, registers, or a combination
• Different division of labor between callee and caller regarding saving machine state (temporaries) across a procedure call
Details

Refer to SPIM manual
(MIPS conventions: for Interoperability)

Registers

- Return Address from a call
  - implicitly copied by jal and jalr instructions

<table>
<thead>
<tr>
<th>Register</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>zero hard-wired to zero</td>
</tr>
<tr>
<td>1</td>
<td>reserved for assembler</td>
</tr>
<tr>
<td>2-27</td>
<td>OS kernel</td>
</tr>
<tr>
<td>28-31</td>
<td>return address</td>
</tr>
</tbody>
</table>

- Frame pointer, Stack pointer, Pointer to global area, ...

<table>
<thead>
<tr>
<th>Register</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>zero hard-wired to zero</td>
</tr>
<tr>
<td>1</td>
<td>at reserved for assembler</td>
</tr>
<tr>
<td>2-3</td>
<td>v0-v1 expression eval and return of results</td>
</tr>
<tr>
<td>4-7</td>
<td>a0-a3 arguments 1 to 4</td>
</tr>
<tr>
<td>8-25</td>
<td>keep temporary values</td>
</tr>
<tr>
<td>26-27</td>
<td>OS kernel</td>
</tr>
<tr>
<td>28</td>
<td>pointer to global area</td>
</tr>
<tr>
<td>29</td>
<td>stack pointer</td>
</tr>
<tr>
<td>30</td>
<td>frame pointer</td>
</tr>
<tr>
<td>31</td>
<td>return address</td>
</tr>
</tbody>
</table>

- First four arguments to a call
  - Can use it for other purposes when args are dead
  - If more arguments ⇒ pass them via the stack

<table>
<thead>
<tr>
<th>Register</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>zero hard-wired to zero</td>
</tr>
<tr>
<td>1</td>
<td>at reserved for assembler</td>
</tr>
<tr>
<td>2-3</td>
<td>v0-v1 expression eval and return of results</td>
</tr>
<tr>
<td>4-7</td>
<td>a0-a3 arguments 1 to 4</td>
</tr>
<tr>
<td>8-27</td>
<td>callee saved temporary</td>
</tr>
<tr>
<td>28</td>
<td>pointer to global area</td>
</tr>
<tr>
<td>29</td>
<td>stack pointer</td>
</tr>
<tr>
<td>30</td>
<td>frame pointer</td>
</tr>
<tr>
<td>31</td>
<td>return address</td>
</tr>
</tbody>
</table>

- Rest are temporaries that need to be saved across a procedure call either by the caller, or the callee, or some combination of the two

<table>
<thead>
<tr>
<th>Register</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>zero hard-wired to zero</td>
</tr>
<tr>
<td>1</td>
<td>reserved for assembler</td>
</tr>
<tr>
<td>2-3</td>
<td>v0-v1 expression eval and return of results</td>
</tr>
<tr>
<td>4-7</td>
<td>a0-a3 arguments 1 to 4</td>
</tr>
<tr>
<td>8-25</td>
<td>keep temporary values</td>
</tr>
<tr>
<td>26-27</td>
<td>OS kernel</td>
</tr>
<tr>
<td>28</td>
<td>pointer to global area</td>
</tr>
<tr>
<td>29</td>
<td>stack pointer</td>
</tr>
<tr>
<td>30</td>
<td>frame pointer</td>
</tr>
<tr>
<td>31</td>
<td>return address</td>
</tr>
</tbody>
</table>

Question:

- What are the advantages/disadvantages of:
  - Callee saving of registers?
  - Caller saving of registers?

- What do the following registers signify:
  - a0-a3, b0-b3, t0-t15
  - k0-k7
  - ra
  - sp
  - fp
  - gp
Stack

- Keeps parameters and local variables
  - Each invocation gets a new copy
- Caller needs to save
  - Any caller-save registers that have a live value
  - Any parameters that are passed
  - Return address (when branch instruction is taken)
- Callee needs to save
  - Previous stack pointer address
  - Previous frame pointer and global area pointer
  - Any callee-save registers that may be used

- Address of the n-th argument is \(n \times 4 + \text{fp}\)
- Local variables are a negative constant off \(\text{fp}\)

- When calling a new procedure, caller should:
  - Push any t0-t9 that has a live value on the stack
  - Put arguments 1-4 in register a0-a3
  - Push rest of the arguments on the stack
  - Do a jal or jalr

- In a procedure call, the callee at the beginning:
  - Push $fp on the stack
  - Copy $sp to $fp
  - Push $ra on the stack
  - If any s0-s7 is used in the procedure save it on the stack
  - Create space for local variables on the stack

- In a procedure call, after executing the callee, at the end:
  - Put return values in v0,v1
  - Restore callee saved registers from stack
  - Update $sp using $fp to \((\text{fp}+4) \times \ldots\)
In a procedure call, the callee at the end:
- put return values on v0,v1
- restore the callee saved registers from stack
- update $sp using $fp ($fp-4)
- restore $ra from stack
- restore $fp from stack
- execute jr $ra and return to caller

On return from a procedure call, the caller:
- update $sp to ignore arguments
- restore the caller saved registers
- Continue...

**Runtime Environments : Overview**

- Fully static (e.g., FORTRAN 77)
- Stack-based
  - without local procedures (e.g., C)
  - Dynamic link
  - with local procedures (e.g., Pascal, Ada)
    - Static and dynamic links
      - To access non-local variables with static scoping
      - Display registers
    - Efficient access to non-local variables
  - with local procedure parameters (e.g., Pascal)
- Fully dynamic (e.g., Scheme, ML)

**Additional Issues**

- In the implementation of block-structured languages (such as Pascal), chains of static / dynamic links or displays are used to access non-local variables.

**Example of LINK and UNLK**

```c
int CallingFunction(int x) {
    int y;
    CalledFunction(1,2);
    return (5);
}

void CalledFunction(int param1, int param2) {
    int local1, local2;
    local1 = param2;
}
```

```assembly
* Reserving space for local variable y (4 bytes)
  LINK Ab, -4
  CalledFunction(L2)
  * Pushing the second parameter on the stack
  MOVE.L #2, -(A7)
  * Pushing the first parameter on the stack
  MOVE.L #1, -(A7)
  * Calling the CalledFunction()
  JSR _CalledFunction
  * Pop out the parameters after return
  ADDQ.L #8, A7
  return (5)
  * Copy the returned value 5 into D0
  MOVEQ.L #5, D0
  * Freeing up the stack space taken by local variables
  UNLK A6
  * Return back to the calling function
  RTS
```
void CalledFunction(int param1, int param2) {
    int local1, local2;
    // Reserving space for locals (8 bytes)
    LINK A6, #-8

    local1 = param2;
    MOVE.L 12(A6), -4(A6)
    // Freeing stack space of locals
    UNLK A6
    // Return back to the calling function
    RTS
}