8. Runtime Environments

Eva Rose    Kristoffer Rose

NYU Courant Institute
Compiler Construction (CSCI-GA.2130-001)

April 6, 2015
1. Storage organization
2. Stack Allocation of Space
3. Non-local Data
4. Heap Management
5. Architectures
Sixth Compilation Phase

source program

Tokens

Lexical Analysis

Syntax Analysis

Semantic Analysis

Tree

Symbol Table

Intermediate Representation Generator

Tree

Optimizer

IR

Code Generator

IR

Assembly

target machine code

Machine-Dependent Code Optimizer
1 Storage organization

2 Stack Allocation of Space

3 Non-local Data

4 Heap Management

5 Architectures
**Static vs Dynamic**

**Static** means “at compile time.”

**Dynamic** means “at runtime.”

*Stack* has data local to call.
- Layout known statically
- Managed automatically by generated code

*Heap* has data that survives between calls.
- Layout not known statically
- Managed dynamically by garbage collector
**Static vs Dynamic**

**Static** means “at compile time.”

**Dynamic** means “at runtime.”

**Stack** has data local to call.
- Layout known statically
- Managed automatically by generated code

**Heap** has data that survives between calls.
- Layout not known statically
- Managed dynamically by garbage collector
Static vs Dynamic

**Static** means “at compile time.”

**Dynamic** means “at runtime.”

**Stack** has data local to call.
- **Layout known statically**
- Managed automatically by generated code

**Heap** has data that survives between calls.
- **Layout not known statically**
- Managed dynamically by garbage collector
Static vs Dynamic

Static  means “at compile time.”
Dynamic means “at runtime.”

Stack  has data local to call.
▶ Layout known statically
▶ Managed automatically by generated code

Heap  has data that survives between calls.
▶ Layout not known statically
▶ Managed dynamically by garbage collector
**Static vs Dynamic**

**Static** means “at compile time.”

**Dynamic** means “at runtime.”

**Stack** has data local to call.
- Layout known statically
- Managed automatically by generated code

**Heap** has data that survives between calls.
- Layout not known statically
- Managed dynamically by garbage collector
Static vs Dynamic

**Static** means “at compile time.”

**Dynamic** means “at runtime.”

**Stack** has data local to call.

- Layout known statically
- Managed automatically by generated code

**Heap** has data that survives between calls.

- Layout not known statically
- Managed dynamically by *garbage collector*
Storage Organization

- **Code area**: instructions, address pointers.
- **Static area**: global constants.
- **Heap**: objects, records, arrays, variable strings. \(^1\)
- **Runtime stack**: activation records/stack frames; objects, records, arrays, variable strings. \(^2\)

\(^1\) Dynamically allocated.
\(^2\) Statically allocated.
Storage Organization

**Code area** instructions, address pointers.
**Static area** global constants.
**Heap** objects, records, arrays, variable strings.\(^1\)
**Runtime stack** activation records/stack frames;
objects, records, arrays, variable strings.\(^2\)

\(^1\)Dynamically allocated.
\(^2\)Statically allocated.
Storage Organization

<table>
<thead>
<tr>
<th>Static</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Constants</td>
</tr>
<tr>
<td>Dynamic</td>
<td>Heap</td>
</tr>
<tr>
<td></td>
<td>Free Memory</td>
</tr>
<tr>
<td></td>
<td>Stack</td>
</tr>
</tbody>
</table>

lowest address

highest address
1. Storage organization
2. Stack Allocation of Space
3. Non-local Data
4. Heap Management
5. Architectures
Activation Tree

- Shows all calls at runtime.
- A call is a *child* of the callee node.
- Children are in *call order* from left to right.
Activation Tree Example

```
... void quicksort(int m, int n){
    int i;
    if (n > m){
        i = partition(m,n); /* picks "quicksort separator value" */
        quicksort(m, i-1);
        quicksort(i+1, n);
    }
}

int a[11];

void main() {
    readArray(); /* reads 9 integers into a[1],..,a[9] */
    a[0] = -9999; a[10] = 9999; /* sentinels */
    quicksort(1,9);
}
```

Check: Figure 7.4 (p.433).
The Stack

<table>
<thead>
<tr>
<th>Static</th>
<th>Code</th>
<th>lowest address</th>
</tr>
</thead>
<tbody>
<tr>
<td>Static</td>
<td>Static</td>
<td></td>
</tr>
<tr>
<td>Dynamic</td>
<td>Heap</td>
<td>Free Memory</td>
</tr>
<tr>
<td></td>
<td>↓</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Stack</td>
<td>highest address</td>
</tr>
</tbody>
</table>
Contents of Activation Record

1. Temporary values that do not fit in registers.
2. Local data for called procedure.
3. Saved machine state (notably return address).
5. Control link (caller’s activation record).
6. Space for return value.
7. Actual parameters (not in registers).
Activation record or “frame”

<table>
<thead>
<tr>
<th>Actual Parameters</th>
<th>higher addresses</th>
</tr>
</thead>
<tbody>
<tr>
<td>Return Values</td>
<td></td>
</tr>
<tr>
<td>Control Link</td>
<td></td>
</tr>
<tr>
<td>Access Link</td>
<td></td>
</tr>
<tr>
<td>Saved Machine State</td>
<td></td>
</tr>
<tr>
<td>Local data</td>
<td>lower addresses</td>
</tr>
<tr>
<td>Temporaries</td>
<td></td>
</tr>
</tbody>
</table>
Calling Sequence

1. Caller stores what it needs to communicate.
2. Callee allocates what extra space it needs.
3. Callee runs.
4. Callee cleans up any local material.
5. Callee restores Caller’s state (which includes “jumping back”).
Calling Sequence

1. Caller stores what it needs to communicate.
2. Callee allocates what extra space it needs.
3. Callee runs.
4. Callee cleans up any local material.
5. Callee restores Caller’s state (which includes “jumping back”).
Calling Sequence

1. Caller stores what it needs to communicate.
2. Callee allocates what extra space it needs.
3. Callee runs.
4. Callee cleans up any local material.
5. Callee restores Caller’s state (which includes “jumping back”).
Calling Sequence

1. Caller stores what it needs to communicate.
2. Callee allocates what extra space it needs.
3. Callee runs.
4. Callee cleans up any local material.
5. Callee restores Caller’s state (which includes “jumping back”).
Calling Sequence

1. Caller stores what it needs to communicate.
2. Callee allocates what extra space it needs.
3. Callee runs.
4. Callee cleans up any local material.
5. Callee restores Caller’s state (which includes “jumping back”).
Where Is My Activation Record?

- Keep a **Frame Pointer** in a register.
- Keep track of **Local Stack Size**.
  - Variable-length data on stack.
<table>
<thead>
<tr>
<th></th>
<th>Storage organization</th>
<th>Stack Allocation of Space</th>
<th>Non-local Data</th>
<th>Heap Management</th>
<th>Architectures</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Storage organization</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Stack Allocation of Space</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Non-local Data</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Heap Management</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Architectures</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Easy: C

- Global and static data goes in static section.
- Everything else on the stack.
- Has **nested procedures**.
- Uses *access link* or *display*.
1. Storage organization
2. Stack Allocation of Space
3. Non-local Data
4. Heap Management
5. Architectures
# The heap

<table>
<thead>
<tr>
<th>Static</th>
<th>Code</th>
<th>lower addresses</th>
</tr>
</thead>
<tbody>
<tr>
<td>Static</td>
<td>Static</td>
<td></td>
</tr>
<tr>
<td>Dynamic</td>
<td>Heap</td>
<td>.Free Memory</td>
</tr>
<tr>
<td></td>
<td>Stack</td>
<td>higher addresses</td>
</tr>
</tbody>
</table>
The heap

Heap "objects" may live indefinitely.
"Life span" is managed automatically or from program (memory manager).

<table>
<thead>
<tr>
<th>Allocation</th>
<th>Java</th>
<th>C</th>
<th>C++</th>
</tr>
</thead>
<tbody>
<tr>
<td>Deallocation</td>
<td>new</td>
<td>malloc</td>
<td>new</td>
</tr>
<tr>
<td></td>
<td>garbage c.</td>
<td>free</td>
<td>delete</td>
</tr>
</tbody>
</table>
Memory Manager

**Allocation** get contiguous chunk of bytes.

**Deallocation** release previously allocated chunk for reuse.

Space efficiency Use the available memory well.

Program efficiency Make access to the allocated memory fast.

Low overhead Reduce administrative cost of allocation.
Memory Manager

Allocation get contiguous chunk of bytes.
Deallocation release previously allocated chunk for reuse.

Space efficiency Use the available memory well.
Program efficiency Make access to the allocated memory fast.
Low overhead Reduce administrative cost of allocation.
Memory Hierarchy

Registers  $10^3$ bytes @ 0.2 ns.
L1 cache  $10^5$ bytes @ 5 ns.
L2 cache  $10^7$ bytes @ 20 ns.
Physical Memory  $10^{10}$ bytes @ 100 ns.
Flash Storage  $10^{12}$ bytes @ 1 ms.
Disk  $10^{14}$ bytes @ 10 ms.
Locality

- Memory is loaded into cache in *chunks*.
- Multiple access to same chunk improves cache rate.
- Reorganizing data can drastically improve performance.
  - Allocator and garbage collector can help!
1. Storage organization
2. Stack Allocation of Space
3. Non-local Data
4. Heap Management
5. Architectures
Target Architectures

High-end (expensive) architectures:
- ARM: Tablets, mobile phones, ultrabooks, Raspberry Pi.
- amd64: Intel (and AMD) PCs and servers.
- SPARC: Oracle servers.
- Power PC: IBM servers.

Low-end (cheap) architectures:
- 6505: 8-bit architecture...
The Stack

Calling Conventions generally speaking:

A “static storage allocation contract” between caller and callee, which is guaranteed to hold at runtime.

- size of allocated entities (actual parameters and return values, e.g. based on static type info),
- where address information is kept (registers or stack).
## ARM Calling Conventions

<table>
<thead>
<tr>
<th>Register</th>
<th>On Entry</th>
<th>On Return</th>
</tr>
</thead>
<tbody>
<tr>
<td>r0–3</td>
<td>parameter or unused</td>
<td>return value or unused</td>
</tr>
<tr>
<td>r4–11</td>
<td>preserved</td>
<td>same as on entry</td>
</tr>
<tr>
<td>r12</td>
<td>undefined</td>
<td>undefined</td>
</tr>
<tr>
<td>r13 ‘sp’</td>
<td>stack pointer</td>
<td>same as on entry</td>
</tr>
<tr>
<td>r14 ‘lr’</td>
<td>return address</td>
<td>(unconstrained)</td>
</tr>
<tr>
<td>r15 ‘pc’</td>
<td>program counter</td>
<td>return address</td>
</tr>
</tbody>
</table>

## ARM Calling Conventions

<table>
<thead>
<tr>
<th>Register</th>
<th>On Entry</th>
<th>On Return</th>
</tr>
</thead>
<tbody>
<tr>
<td>r0–3</td>
<td>parameter or unused</td>
<td>return value or unused</td>
</tr>
<tr>
<td>r4–11</td>
<td>preserved</td>
<td>same as on entry</td>
</tr>
<tr>
<td>r12</td>
<td>undefined</td>
<td>undefined</td>
</tr>
<tr>
<td>r13 ‘sp’</td>
<td>stack pointer</td>
<td>same as on entry</td>
</tr>
<tr>
<td>r14 ‘lr’</td>
<td>return address</td>
<td>(unconstrained)</td>
</tr>
<tr>
<td>r15 ‘pc’</td>
<td>program counter</td>
<td>return address</td>
</tr>
</tbody>
</table>

ARM Instructions

- **MOV B, A** – “Move”
  copy what is in A to B.

- **BL label** – “Branch with Link”
  set lr to next instruction address then jump to label.

- **STMFD sp!, {regs}** – “Store Multiple/Fully Descending”
  push all the listed regs on stack (that grows downwards).

- **LDMFD sp!, {regs}** – “Load Multiple/Fully Descending”
  pop all the listed regs from stack (downwards).
ARM Call and Return Code

Caller:

MOV r0, <param1>
MOV r1, <param2>
MOV r2, <param3>
MOV r3, <param4>
BL Callee

Callee:

STMFD sp!,{r4-r11,lr}

Body with parameters in r0–r3...

MOV r0, <return value>
LDMFD sp!,{r4-r11,pc}

Use return value in r0
AMD64 (Intel) Call and Return Code

Caller:

*put actuals on stack/registers*

call Callee

Callee:

*push %rbp*

*move %rbp, %rsp*

*sub %rsp, <frame size>*

Body with parameters in [%rbp-...]...

*move %rax, <return value>*

*move %rsp, %rbp*

*pop %rbp*

*ret*

*remove actuals from stack*

*Use return value in %rax*
Other stack topics

- saving registers across calls,
- variable-sized data on stack,
- accessing non-local stack variables.
ARM Call and Return Code Example

Caller:

```
MOVR0, #1
MOVR1, #1
BL Callee
```

Callee:

```
STMFDsp!,{r4-r11,lr}
ADD R0, R0, R1
LDMFDsp!,{r4-r11,pc}
```

Continue...
ARM Call and Return Code Example

Caller:
MOV r0, #1
MOV r1, #1
BL Callee

Callee:
STMFD sp!,{r4-r11,lr}
ADD r0, r0, r1
LDMFD sp!,{r4-r11,pc}

Continue...
ARM Call and Return Code Example

Caller:
- MOV r0, #1
- MOV r1, #1
- BL Callee

Callee:
- STMFD sp!,{lr}
- ADD r0, r0, r1
- LDMFD sp!,{pc}

Continue...
ARM Call and Return Code Example

Caller:

MOV r0, #1
MOV r1, #1
BL Callee

Callee:

ADD r0, r0, r1
MOV pc, lr

Continue...
Questions?