The Backend (continued)

David Notkin
Autumn 2008

Run-time storage layout

• Representation of
  – int, bool, etc.
  – arrays, records, etc.
  – procedures
• Placement of
  – global variables
  – local variables
  – parameters
  – results

Storage allocation strategies

• Given layout of data structure, where in memory to allocate space for each instance?
• Key issue: what is the lifetime (dynamic extent) of a variable/data structure?
  – Whole execution of program (e.g., global variables)
    • Static allocation
  – Execution of a procedure activation (e.g., locals)
    • Stack allocation
  – Variable (dynamically allocated data)
    • Heap allocation

Run-time memory

<table>
<thead>
<tr>
<th></th>
<th>Stack</th>
<th>Heap</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>down</td>
<td></td>
</tr>
<tr>
<td>static data</td>
<td></td>
<td></td>
</tr>
<tr>
<td>code/RO data</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

• Code/Read-only data area
  – Shared across processes running same program
• Static data area
  – Can start out initialized or zeroed
• Heap
  – Can expand upwards through (e.g. sbrk) system call
• Stack
  – Expands/contracts downwards automatically

Static allocation

• Statically allocate variables/data structures with global lifetime
  – Machine code
  – Compile-time constant scalars, strings, arrays, etc.
  – Global variables
  – static locals in C, all variables in FORTRAN
• Compiler uses symbolic addresses
• Linker assigns exact address, patches compiled code

Stack allocation

• Stack-allocate variables/data structures with LIFO lifetime
  – Data doesn’t outlive previously allocated data on the same stack
• Stack-allocate procedure activation records
  – Frame includes formals, locals, temps
  – And housekeeping: static link, dynamic link, …
• Fast to allocate and de-allocate storage
• Good memory locality
Stack allocation II

- What about variables local to nested scopes within one procedure?

```c
procedure P() {
    int x;
    for(int i=0; i<10; i++){
        double x;
        ...
    }
    for(int j=0; j<10; j++){
        double y;
        ...
    }
}
```

Stack allocation: constraints I

- No references to stack-allocated data allowed after returns
- This is violated by general first-class functions

```c
proc foo(x:int):
    proctype(y:int):int;
    begin
        return x + y;
    end bar;
    begin
        return bar;
    end
    var f:proctype(int):int;
    var g:proctype(int):int;
    f := foo(3);
    g := foo(4);
    output := f(5);
    output := g(6);
```

Stack allocation: constraints II

- Also violated if pointers to locals are allowed

```c
proc foo (x:int): *int;
    begin
        y := x * 2;
        return &y;
    end
    var w,z:*int;
    z := foo(3);
    w := foo(4);
    output := *z;
    output := *w;
```

Heap allocation

- For data with unknown lifetime
  - new/malloc to allocate space
  - delete/free or garbage collection to de-allocate
- Heap-allocate activation records of first-class functions
- Relatively expensive to manage
- Can have dangling reference, storage leaks
  - Garbage collection reduces (but may not eliminate) these classes of errors

Stack frame layout

- Formals, locals, housekeeping
  - Dynamic and static link
  - Saved registers, ...
- Dedicate registers to support stack access
  - FP - frame pointer: ptr to start of stack frame (fixed)
  - SP - stack pointer: ptr to end of stack (can move)

Key property

- All data in stack frame is at a fixed, statically computed offset from the FP
- This makes it easy to generate fast code to access the data in the stack frame
  - And lexically enclosing stack frames
- Can compute these offsets solely from the symbol tables
  - Based also on the chosen layout approach
Stack Layout

Accessing locals

• If a local is in the same stack frame then
  \[ t := *(fp + local\_offset) \]

• If in lexically-enclosing stack frame
  \[ t := *(fp + static\_link\_offset) \]

  \[ t := *(t + local\_offset) \]

• If in a further enclosing block
  \[ t := *(fp + static\_link\_offset) \]

  \[ t := *(t + static\_link\_offset) \]

  ...-

  \[ t := *(t + local\_offset) \]

At compile-time need to calculate

• Difference in nesting depth of use and definition
• Offset of local in defining stack frame
• Offsets of static links in intervening frames

Calling conventions

• Define responsibilities of caller and callee
  – To make sure the stack frame is properly set up
  and torn down
• Some things can only be done by the caller
• Other things can only be done by the callee
• Some can be done by either
• So, we need a protocol

Typical calling sequence

Caller
• Evaluate actual args
  – Order?
• Push onto stack
  – Order?
  – Alternative: First k args in registers
• Push callee's static link
  – Or in register? Before or after stack arguments?
• Execute call instruction
  – Hardware puts return address in a register

Callee
• Save bookkeeping information on stack
• Allocates space for locals, other data
  – \[ sp := sp - size\_of\_locals - other\_data \]
  – Locals stored in what order?
• Set up new frame pointer (fp := sp)
• Start executing callee's code

Typical return sequence

Callee
• Dealocate space for callee's static link, args
  – \[ sp := fp \]
• Continue execution in caller after call

Caller
• Dealocate space for local, other data
  – \[ sp := sp + size\_of\_locals + other\_data \]
• Restore caller's frame pointer, return address & other regs, all without
  losing addresses of stuff still needed in stack
• Execute return instruction