CS 415: Lecture 16

- Runtime

Schedule

- HW2 due today
- Midterm Monday 3/27
- Project phase 3 due Monday 4/3
- HW3 due Monday 4/10
Where We Are

Compile Time vs. Run Time
Basic computer structure

CPU

Memory

memory bus

I/O bus

disk

Net interface

Memory Hierarchy

cpu

word transfer

cache

block transfer

main memory

page transfer

disks

- decrease cost per bit
- decrease frequency of access
- increase capacity
- increase access time
- increase size of transfer unit
Store-Program Machine

Connection with OS

OS multiplexing mechanisms give each process the illusion of running on a private machine.
Global Variables

Program begin
    x, y: integer
    call B
Procedure B
    y, z: real
    begin
        y = x + z
        call B
    end
Procedure C
    x: real
    begin
        call B
    end

What should we do with x and y?

What is a variable?

Insert x, y and appropriate attributes into the symbol table
One attribute is memory location “named” by x and y
   May be done during parse or in code generation phase when traversing the AST
Often, reserve an area in runtime memory for global variables

Can now replace “x” with address of x wherever x is used in the code - in this case, wherever the global region starts + 0
Local Variables

- What about y and z defined in procedure B and x defined in procedure C?
- 1st option you might think of is just to allocate some space in memory for these variables as well (as shown to the right) - we have already seen last time that this is not sufficient to support recursive calls to a procedure
- At runtime, allocate some space to store local variables
  - What do we need to do at compile time then?

Local Variable

Program

```
x, y: integer

Procedure B
y, z: real
begin
  ... y = x + z
  call B
end

begin
  ... call B
  ... call B
end
call B
... y = x + z
```

```
0
4
x
y
B:y
B:z
C:x
```

Convention is to allocate storage in a stack (often called the control stack)
Run Time Storage Organization

- **Code space**
  - Fixed size
  - Statically allocate

- **Data space**
  - Fixed size global variables may be statically allocated
  - Space to allocate variable size data at run time - heap

- **Control stack**
  - Fixed size local variables allocated on stack
  - Used to store "linkage" information as well

---

Run Time Storage Organization

- Each variable must be assigned a storage class
- Static or global variables
  - Allocated in globals region at compile-time
- Procedure/function local variables and parameters
  - Allocate dynamically on stack
- Storage dynamically allocated by program
  - Allocate from heap
  - Pointers may lead to non-local lifetimes
  - Usually explicit allocation
  - Explicit or implicit deallocation (garbage collection)
Data Access

- How to find data allocated dynamically on stack?
- By convention, designate one register as the stack pointer
- Stack pointer always point at current activation record
- Local variables and parameters are referenced as offsets from sp

Data Access

- What about non-local data?
- One approach is to use access links

```
procedure B
  x, y: integer

procedure C
  h, j: char

procedure D
  a, b: real
```
Accessing Non-Local Data

- Each variable is specified by a pair \((l, o)\) where
  - \(l\): the lexical level of the procedure/function's definition
  - \(o\): offset of variable within space reserved for local variable
    of the enclosing procedure/function

- Need to know current procedure/function level
  - Let current level be \(k\)
  - If \(k = l\), then offset off of \(sp\)
  - If \(k > l\), must find \(l\)'s activation record
    - Suppose \(k = 5\) and \(l = 3\), how do we find \(l\)'s activation record?
    - \(k < l\) cannot occur

Access Links

- Maintaining access links:
  - If procedure \(p\) is nested immediately within procedure \(q\), the
    access link for \(p\) points to the activation record of the most
    recent activation of \(q\)
    - Calling level \(k+1\) procedure – must be defined inside currently
      executing procedure
      - Pass my FP as access link
      - My backward chain will work for lower levels
    - Calling level \(l\) procedure where \(l \leq k\)
      - Find my link to level \(l-1\) and pass it as access link
      - Its backward chain will work for lower levels
Access Links

- **Access with access links:**
  - Each variable is described by \((l, o)\)
  - Assume access to \((l, o)\) at level \(k\)
  - Follow \(k-l\) access links starting in the current (top-most) frame.
  - Add offset to resulting pointer
  - Note that \(k-l\) is computed at compile time

- **What about globals?**

Activation Record

- Up to now we have only talked about how to allocate and access local and global variables
- What else do we need to do for a procedure call?
  - Parameters
  - Return value(s)
  - Return address
  - Registers
- Define a convention for what goes where in the activation record
Display

Access links seem to work great but there's one disadvantage ... what is it?

To improve run time access cost, use a display

- Table of access links for lower levels
- Lookup is indexed from known offset
- Takes slight amount of time at call
- Can have a single display (global) or one per frame (local)
- For level k procedure, need k slots

Access

- Variable described by (l,o)
- Assume access to (l,o) at level k
- Pointer to needed activation record is at DP + 4x(l-1), assuming that each pointer is 4 bytes
- Add offset to pointer

Maintenance - single global display

- Call from level k to level l
  - On entry to procedure at level l
    - Save the level l display value
    - Push FP of new frame into slot l on display
  - On return
    - Restore the level l display value
Display

- Individual frame-based displays
- Call from level $k$ to level $l$
  - If $l \leq k$
    - Copy $l-1$ display entries into child's frame
    - Push FP of new frame into level $l$ display slot
  - If $l > k$ (the only possible case)
    - Copy $k$ entries into child's frame
    - Push FP of new frame into level $k+1$ display slot
  - No work is required on return
  - Displays are deallocated with frames

Display vs. Access Links

- What’s the trade-off?
- Cost differences depend on
  - Frequency of non-local access
  - Average lexical nesting depth
  - Ratio of calls to non-local access
- (Sort of) conventional wisdom
  - Tight on registers $\rightarrow$ use access links
  - Lots of registers $\rightarrow$ use global display (and store in registers)
  - Shallow average nesting $\rightarrow$ frame-based display
Next Lecture

- Read ASU 7.5 and 8.1