The Procedure Abstraction
Part II: Symbol Tables, Storage

COMP 412
Fall 2005

Review

From last lecture

The Procedure serves as

- A control abstraction
- A naming abstraction
- An external interface

{ Access to system services, libraries, code from others …

We covered the control abstraction last lecture.

Today, we will focus on naming.

The Procedure as a Name Space

Each procedure creates its own name space

- Any name (almost) can be declared locally
- Local names obscure identical non-local names
- Local names cannot be seen outside the procedure
  - Nested procedures are "inside" by definition
- We call this set of rules & conventions "lexical scoping"

Examples

- C has global, static, local, and *block* scopes *(Fortran-like)*
  - Blocks can be nested, procedures cannot
- Scheme has global, procedure-wide, and nested scopes *(let)*
  - Procedure scope (typically) contains formal parameters

The Procedure as a Name Space

Why introduce lexical scoping?

- Provides a compile-time mechanism for binding "free" variables
- Simplifies rules for naming & resolves conflicts
- Lets the programmer introduce "local" names with impunity

How can the compiler keep track of all those names?

The Problem

- At point *p*, which declaration of *x* is current?
- At run-time, where is *x* found?
- As parser goes in & out of scopes, how does it delete *x*?

The Answer

- Lexically scoped symbol tables (see § 5.7.3)

Do People Use This Stuff ?

C macro from the MSCP compiler

```
#define fix_inequality(oper, new_opcode)                \
    if (value0 < value1)                                \
    {                                                   \
        Unsigned_Int temp = value0;                     \
        value0 = value1;                                \
        value1 = temp;                                  \
        opcode_name = new_opcode;                       \
        temp = oper->arguments[0];                      \
        oper->arguments[0] = oper->arguments[1];        \
        oper->arguments[1] = temp;                      \
        oper->opcode = new_opcode;                      \
    }
```

Declares a new name

Lexically-scoped Symbol Tables

§ 5.7 in EaC

The problem

- The compiler needs a distinct record for each declaration
- Nested lexical scopes admit duplicate declarations

The interface

- insert(*name, level*) – creates record for *name* at *level*
- lookup(*name, level*) – returns pointer or index
- delete(*level*) – removes all names declared at *level*

Many implementation schemes have been proposed (see § B.4)

- We'll stay at the conceptual level
- Hash table implementation is tricky, detailed, & fun

Symbol tables are compile-time structures that the compiler uses to resolve references to names. We'll see the corresponding run-time structures that are used to establish addressability later.

Example

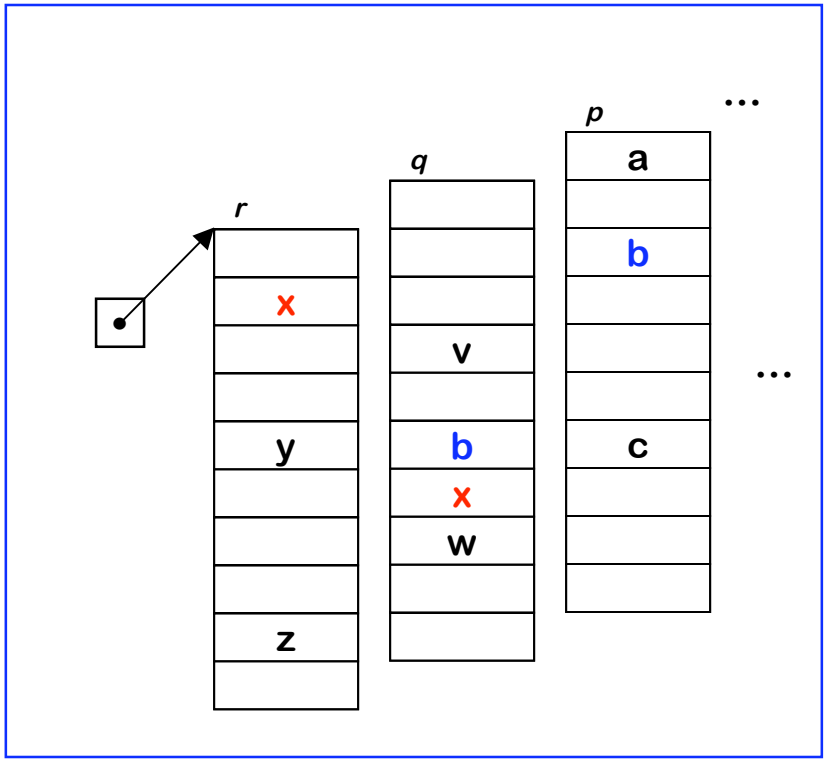
```
procedure p {
      int a, b, c
      procedure q {
            int v, b, x, w
            procedure  r {
                  int x, y, z
                  ....
            }
            procedure s {
                  int x, a, v
                  ...
            }
            ... r ... s
      }
     ... q ...
}
```

```
B0: {
            int a, b, c
B1:         {
                  int v, b, x, w
B2:               {
                        int x, y, z
                        ....
                  }
B3:               {
                        int x, a, v
                        ...
                  }
                  ...
            }
      ...
}
```

Lexically-scoped Symbol Tables

High-level idea

- Create a new table for each scope
- Chain them together for lookup



"Sheaf of tables" implementation

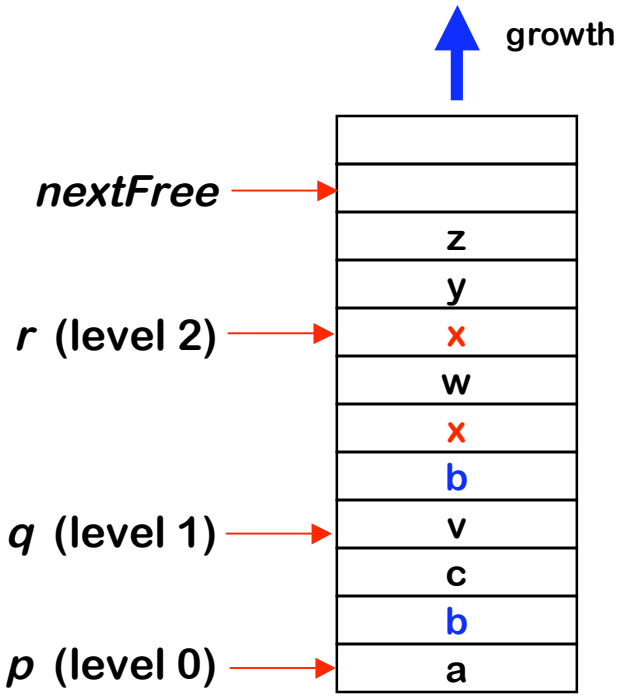
- *insert*() may need to create table
- it always inserts at current level
- *lookup*() walks chain of tables & returns first occurrence of name
- *delete*() throws away table for level *p*, if it is top table in the chain

If the compiler must preserve the table (*for, say, the debugger*), this idea is actually practical.

Individual tables can be hash tables.

Implementing Lexically Scoped Symbol Tables

Stack organization



Implementation

- *insert* () creates new level pointer if needed and inserts at nextFree
- *lookup* () searches linearly from nextFree–1 forward
- *delete* () sets nextFree to the equal the start location of the level deleted.

Advantage

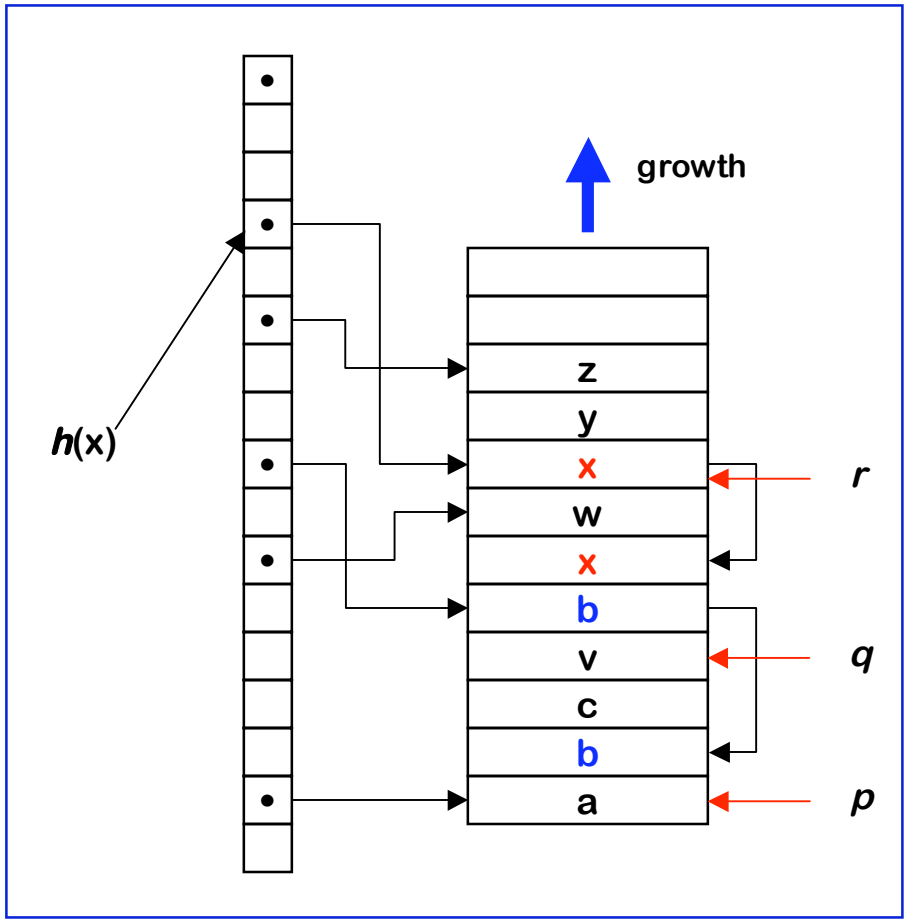
- Uses much less space

Disadvantage

- Lookups can be expensive

Implementing Lexically Scoped Symbol Tables

Threaded stack organization



Implementation

- *insert* () puts new entry at the head of the list for the name
- *lookup* () goes direct to location
- *delete* () processes each element in level being deleted to remove from head of list

Advantage

- lookup is fast

Disadvantage

- delete takes time proportional to number of declared variables in level

The Procedure as an External Interface

OS needs a way to start the program's execution

- Programmer needs a way to indicate where it begins
  - The "main" procedure in most languages
- When user invokes "grep" at a command line
  - OS finds the executable
  - OS creates a process and arranges for it to run "grep"
  - "grep" is code from the compiler, linked with run-time system
    - Starts the run-time environment & calls "main"
    - After main, it shuts down run-time environment & returns
- When "grep" needs system services
  - It makes a system call, such as fopen()

UNIX/Linux specific discussion

Where Do All These Variables Go?

Automatic & Local

- Keep them in the procedure activation record or in a register
- Automatic ⇒ lifetime matches procedure's lifetime

Static

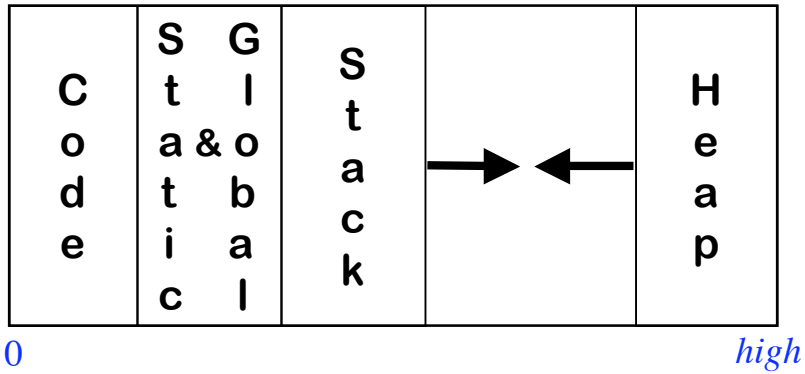
- Procedure scope ⇒ storage area affixed with procedure name
  - &_p.x
- File scope ⇒ storage area affixed with file name
- Lifetime is entire execution

Global

- One or more named global data areas
- One per variable, or per file, or per program, …
- Lifetime is entire execution

Placing Run-time Data Structures

Classic Organization



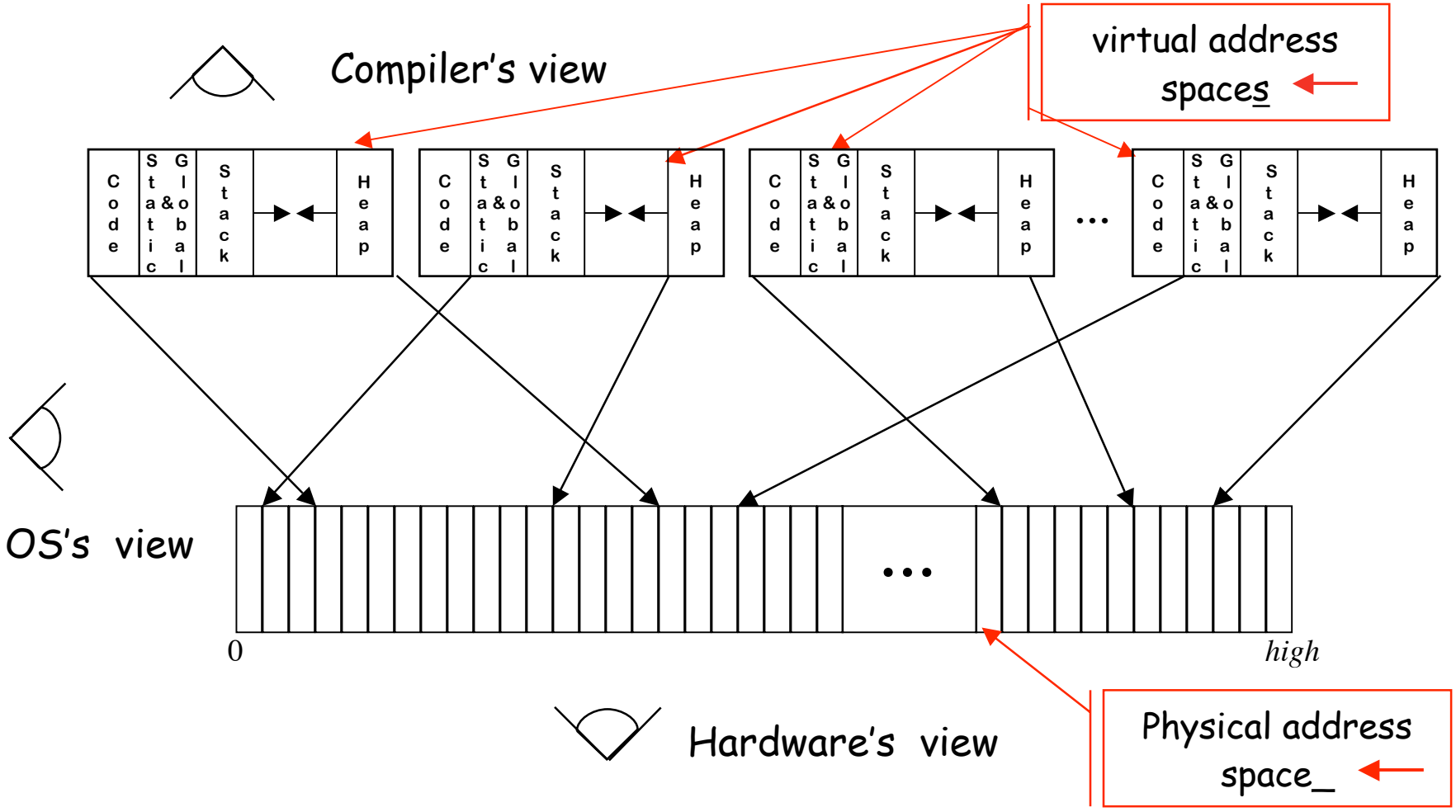
Single Logical Address Space

- Better utilization if stack & heap grow toward each other
- Very old result (Knuth)
- Code & data separate or interleaved
- Uses address space, not allocated memory

- Code, static, & global data have known size
  - → Use symbolic labels in the code
- Heap & stack both grow & shrink over time
- This is a *virtual* address space

How Does This Really Work?

The Big Picture



Where Do Local Variables Live?

A Simplistic model

- Allocate a data area for each distinct scope
- One data area per "sheaf" in scoped table

What about recursion?

- Need a data area per invocation (or activation) of a scope
- We call this the scope's activation record
- The compiler can also store control information there!

More complex scheme

- One activation record (AR) per procedure instance
- All the procedure's scopes share a single AR *(may share space)*
- Static relationship between scopes in single procedure

Used this way, "static" means knowable at *compile time* (and, therefore, fixed).