A Machine-Independent Debugger

David R. Hanson and Mukund Raghavachari

Princeton University

http://www.cs.princeton.edu/~drh/pubs/cdb.pdf
http://www.cs.princeton.edu/~drh/pubs/cdbtalk.pdf

Debuggers—Bottomfeeders of Systems Research

- **Notoriously machine-dependent; most depend on**

  architecture, operating system, compiler, linker, …

  arcane data formats, protocols, obscure/undocumented system calls

- **Building a debugger for *L* might take more work than building a compiler for *L***

gdb	**150,000+ lines (47,000 machine-dependent)**
`lcc`	**10,000+ lines (3,000 machine-dependent)**

- **Machine-independent/retargetable debuggers**

  Separate/isolate machine dependencies
  `ldb`’s PostScript symbol tables (Ramsey & Hanson, PLDI’92)

  Inject debugging code at the source-code level
  (Heymann, *SIGPLAN Notices*, 9/93)

  Inject debugging code at the intermediate-code level
  ML debugger (Tolmach & Appel, LFP’90)

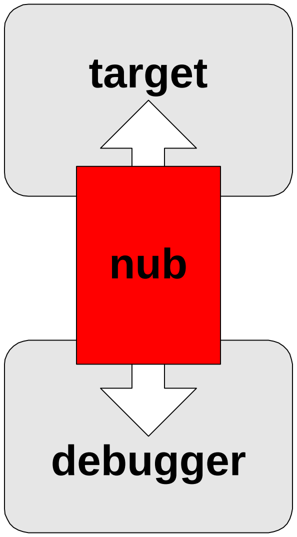
  Use an interpreter (e.g., Centerline’s ObjectCenter)

  Some of these approaches remain laborious; they don’t scale/extrapolate

- **Distill the good ideas, avoid *a priori* assumptions; cruise a huge design space**

Design

- **Embed a small ‘nub’ in the target, which communicates with the debugger**



- **Nub interface defines what debuggers can do to/with the target**
- **Interface must be small; implementation *<u>might</u>* depend on target, OS, compiler, …**

A Nub Interface

- **Types: source coordinates, program states, callbacks**

```
typedef struct {                       typedef struct {
    char file[32];                         char name[32];
    unsigned short x, y;                   Nub_coord_T src;
} Nub_coord_T;                             void *fp, *context;
                                       } Nub_state_T;

typedef void (*Nub_callback_T)(Nub_state_T state);
```

- **Read/write target’s ‘address spaces’**

```
int _Nub_fetch(int space, void *address, void *buf, int nbytes);
int _Nub_store(int space, void *address, void *buf, int nbytes);
```

- **Set/remove breakpoints; intercept faults**

```
void _Nub_init(Nub_callback_T startup, Nub_callback_T fault);
Nub_callback_T _Nub_set   (Nub_coord_T src, Nub_callback_T onbreak);
Nub_callback_T _Nub_remove(Nub_coord_T src);
void _Nub_src(Nub_coord_T src,
    void apply(int i, Nub_coord_T *src, void *cl), void *cl);
```

- **Inspect state**

```
int _Nub_frame(int n, Nub_state_T *state);
```

Nub Interface, cont’d

- **Address spaces**

  Code, data, symbol tables, stacks, registers, …

  Need not correspond to actual locations in the target

  Different implementations ⇒ different address spaces

- **Nub is just a conduit for *opaque data***

  Producers and consumers must agree on formats and interpretations

  Machine-independent manipulation of machine-dependent data

  Keeps interface and implementations small

- **Wide range of implementations and clients**

nub	***clients***
UNIX symbol tables **machine-specific executable files** **machine-specific instructions**	**gdb-style debugger**
⋮	
machine-dependent symbol tables **compiler-injected breakpoint hooks**	**cdb and `lcc`**

CDB—A Lean Debugger

- cdb's user interface is a frugal set of one-letter commands

b [*file*:]*line*[.*character*]	
	set a breakpoint at the specified source coordinate
c	continue execution
d [*n*]	move down the call stack 1 or *n* frames
f [*n*]	print everything about the current frame or about frame *n*
h	print this command summary
m [*n*]	move to frame 0 (the top frame) or to frame *n*
p	list the visible variables as p commands
p { [*file*:]*id* }	print the values of the listed identifiers
q	quit cdb and the target
r	remove the current breakpoint
r [*file*:]*line*[.*character*]	
	remove the breakpoint at the specified source coordinate
u [*n*]	move up the call stack 1 or *n* frames
w	display the call stack
!*cmd*	call the shell to execute *cmd*

- ***No*** expression evaluation, assignments, conditional breakpoints, single stepping, machine-level debugging, …

  Some features are done better by other programs, e.g., `ldb`'s expression server

  Some features are machine-dependent

Using CDB

- **Example: word frequency program**

```
% a.out <input
2    a
1    and
1    by
1    case
1    digits
1    followed
1    ignored
2    is
1    letter
1    letters
1    more
2    or
1    word
1    zero
```

- **Reads ‘words,’ builds a search tree of words and their counts, traverses tree**
- **Build `a.out`:**

```
% lcc -Wo-g4 wf.c lookup.c
% a.out <input
cdb>
```

Stopping Points

- **cdb can set breakpoints on *stopping points*: expressions, block entries/exits, …**

  in `wf.c`’s `getword()`:

```
16        while (◆(c = getchar()) != -1 && ◆isletter(c) == 0)
17            ◆;
18        for (◆s = buf; ◆(c = isletter(c)) != 0; ◆c = getchar())
19            ◆*s++ = c;
```

- **cdb accepts *incomplete coordinates*, uses `_Nub_src` to display those that ‘match’**

```
cdb> b 18
Sweep and send one of the following commands:
b wf.c:18.7
b wf.c:18.40
b wf.c:18.16
b lookup.c:18.11
```

- **Setting a breakpoint displays the command to remove it; in `lookup.c`’s `lookup()`**

```
cdb> b 17
Sweep and send one of the following commands:
b wf.c:17.3
b lookup.c:17.7
cdb> b lookup.c:17.7
To remove this breakpoint, sweep and send the command:
r lookup.c:17.7
```

Sample Program Extracts

- **From `wf.c`:**

```
static struct node *words = NULL;
…
int main(int argc, char *argv[]) {
    char buf[40];
    while (getword(buf))
        lookup(buf, &words)->count++;
    tprint(words);
```

- **From `lookup.c`:**

```
static struct node words[2000];
static int next = 0;
struct node *lookup(char *word, struct node **p) {
    if (*p) {
        int cond = strcmp(word, (*p)->word);
        if (◆cond < 0)
            return lookup(word, &(*p)->left);
        else if (cond > 0)
            return lookup(word, &(*p)->right);
        else
            return *p;
    }
    if (next >= sizeof words/sizeof words[0])
…
    return *p = &words[next++];
}
```

Printing Values

- **When target is continued, it stops at the breakpoint in `lookup()`**

  ```
  cdb> c
  stopped in lookup at lookup.c:17.7
  0   lookup(word=(char *)0Xeffffac0 "word",p=(struct node **)0X81a8)
  cdb>
  ```

  Control returns to cdb via its callback function passed to `_Nub_set`

  cdb uses the supplied `Nub_state_T` to print a synopsis of the top stack frame

  cdb prints values in source-language terms whenever possible

- **A breakpoint establishes a *focus*: a (coordinate, frame, function) triple**
- **Focus determines the *visible identifiers*, which a bare p command prints**

  ```
  cdb> p
  p cond
  p p
  p word
  p lookup.c:next
  p lookup.c:words
  p wf.c:words
  ```

  Victim uses the mouse to sweep and send the desired commands

- **GUIs would use point-and-click for most commands, e.g., `deet` (Hanson & Korn, USENIX’97)**

Specifying File-Scope Statics

- **Debugging is different than compiling:**

  Debugging focuses on *exploring* the entire target, not compiling its components

  Must be able to distinguish between file-scope statics with identical names

- **cdb permits filename prefixes, and a bare p command prints them that way**

```
cdb> p lookup.c:words
lookup.c:words={
    [0]={count=1,left=(struct node *)0X0,
        right=(struct node *)0X0,word=(char *)0X149a0 "a"}
    [1]={count=0,left=(struct node *)0X0,
        right=(struct node *)0X0,word=(char *)0X0}
    [1999]={count=0,left=(struct node *)0X0,
        right=(struct node *)0X0,word=(char *)0X0}
}
```

  Prints the complete innards of array, structures, and unions

  Omits 2nd and succeeding array elements with equal values

```
cdb> p wf.c:words
wf.c:words=(struct node *)0X81a8 {count=1,left=(struct node *)0X0,
        right=(struct node *)0X0,word=(char *)0X149a0 "a"}
```

Exploring the Stack

- `w` command displays the call stack; u, d, and `m` commands move the focus

```
cdb> c
    ...       (6 times)
cdb> c
stopped in lookup at lookup.c:17.7
0   lookup(word=(char *)0Xf7fffac0 "letter",p=(struct node **)0X8b90)
cdb> w
*0  lookup(word=(char *)0Xf7fffac0 "letter",p=(struct node **)0X8b90)
1   lookup(word=(char *)0Xf7fffac0 "letter",p=(struct node **)0X8b84)
2   lookup(word=(char *)0Xf7fffac0 "letter",p=(struct node **)0X81a8)
3   main(argc=1,argv=(char **)0Xf7fffbac)
cdb> d 2
2   lookup(word=(char *)0Xf7fffac0 "letter",p=(struct node **)0X81a8)
cdb> u
1   lookup(word=(char *)0Xf7fffac0 "letter",p=(struct node **)0X8b84)
cdb> m
0   lookup(word=(char *)0Xf7fffac0 "letter",p=(struct node **)0X8b90)
```

- `f` command displays locals; compare display of `buf` with display of `word`

```
cdb> f
0   lookup(word=(char *)0Xf7fffac0 "letter",p=(struct node **)0X8b90)
        cond=3
cdb> f3
3   main(argc=1,argv=(char **)0Xf7fffbac)
        buf={"letter"}
```

Implementation

- `lcc` emits machine-independent data and code that cooperates with a ***machine-independent*** nub

  Symbol tables: initialized C data structures for symbols, types, strings, …

  Breakpoint 'hooks:' code at each stopping point tests if a breakpoint is set

    Injected code is at the ***intermediate-code*** level
    like `lcc`'s profiling code (Fraser & Hanson, *SIGPLAN Notices*, 10/91)
    Easier than injecting source code; no implementation-defined behaviors

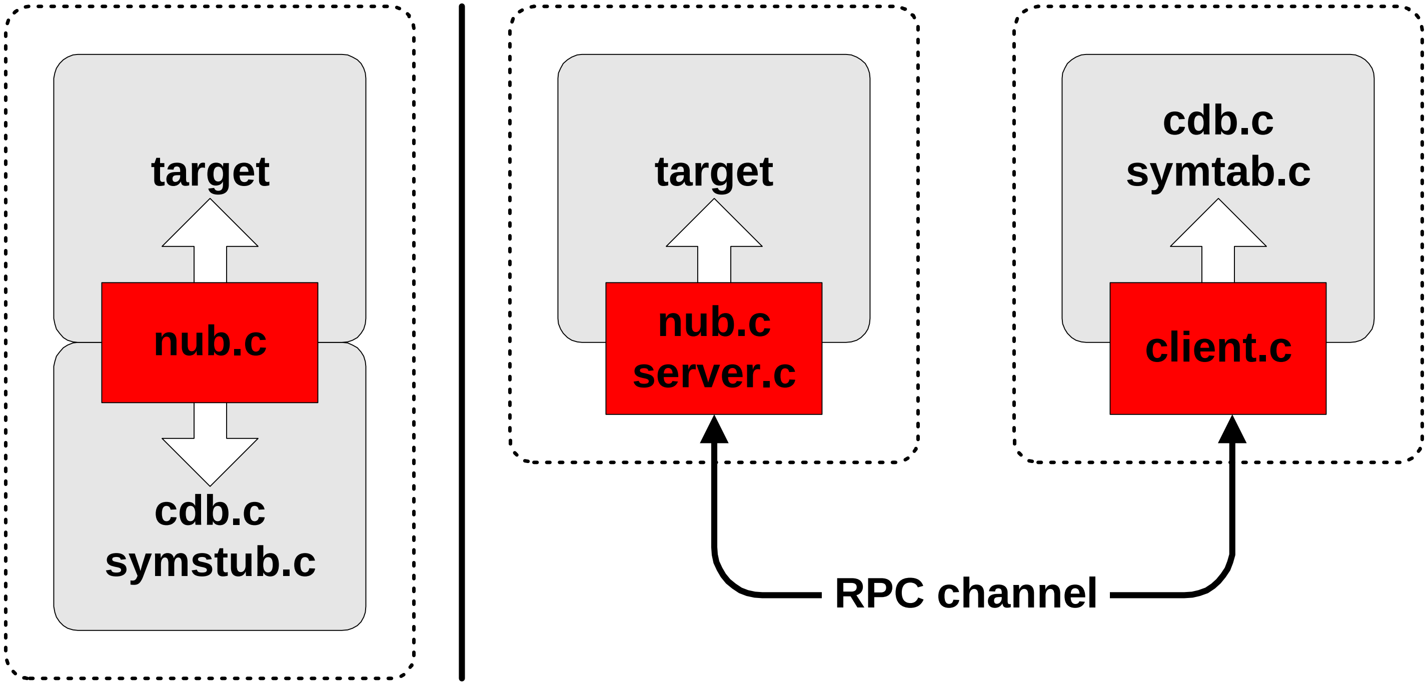
  Logical 'address spaces' are all part of the target

- Compared with other debuggers, cdb is tiny:

lines	***file***	***purpose***
31	`cdbld`	linking script, per target (but only 4 lines change)
565	`stab.c`	symbol table and breakpoint code emitter (loaded w/`lcc`)
249	`nub.c`	the nub
191	`client.c`	RPC stub for the debugger
202	`server.c`	RPC stub for the nub
794	`cdb.c`	cdb's user interface and command processor
80	`symtab.c`	symbol table and type management, e.g., caching
15	`symstub.c`	`symtab` stubs for single-process debugger

Configuration

- cdb can be loaded with the target or run in separate process



RPC code is the minimal needed for cdb

Could use a generic, architecture-neutral RPC package

Modules

- **For each module (i.e., `.c` file), `lcc` emits an initialized instance of**

```
struct module {
    union scoordinate *coordinates;
    char **files;
    struct ssymbol *link;
};
```

`coordinates`	**points to an array of source-coordinate data**
`files`	**points to an array of file names**
`link`	**points to the ‘link’ symbol**

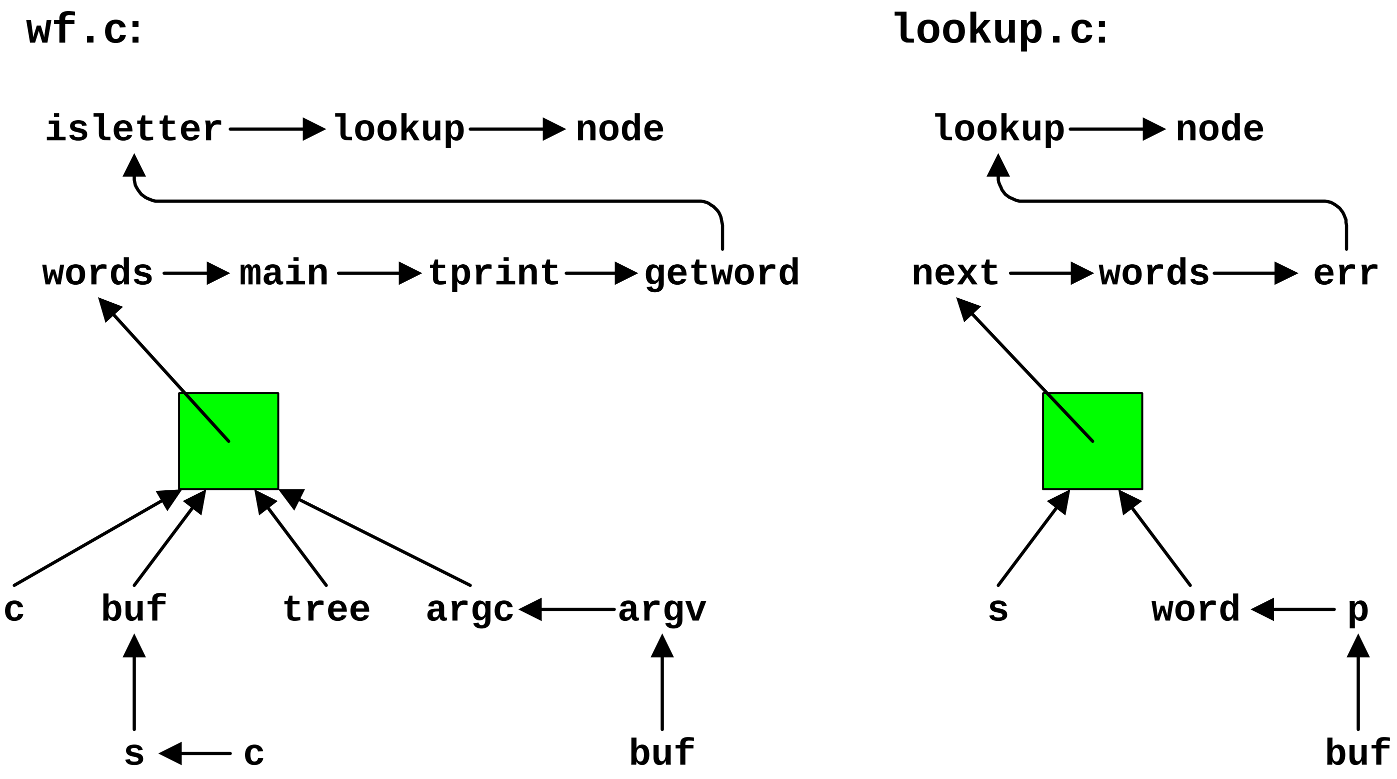
- **For each symbol (e.g., variable, type, tag, …), `lcc` emits an initialized instance of**

```
struct ssymbol {
    int offset;
    void *address;
    …
    struct stype *type;
    struct ssymbol *uplink;
};
```

`offset`	**shadow stack offset for automatic locals and parameters**
`address`	**address of globals and statics**
`type`	**points to a description of the symbol’s type**
`uplink`	**points to another symbol in the same or enclosing scope**

Symbol Tables

- A module’s symbol table is an ***inverted*** tree of initialized `ssymbol` structures



Visible symbols are those on the path from a leaf to the root

`context` fields in `_Nub_state_Ts` hold pointers to `ssymbols`

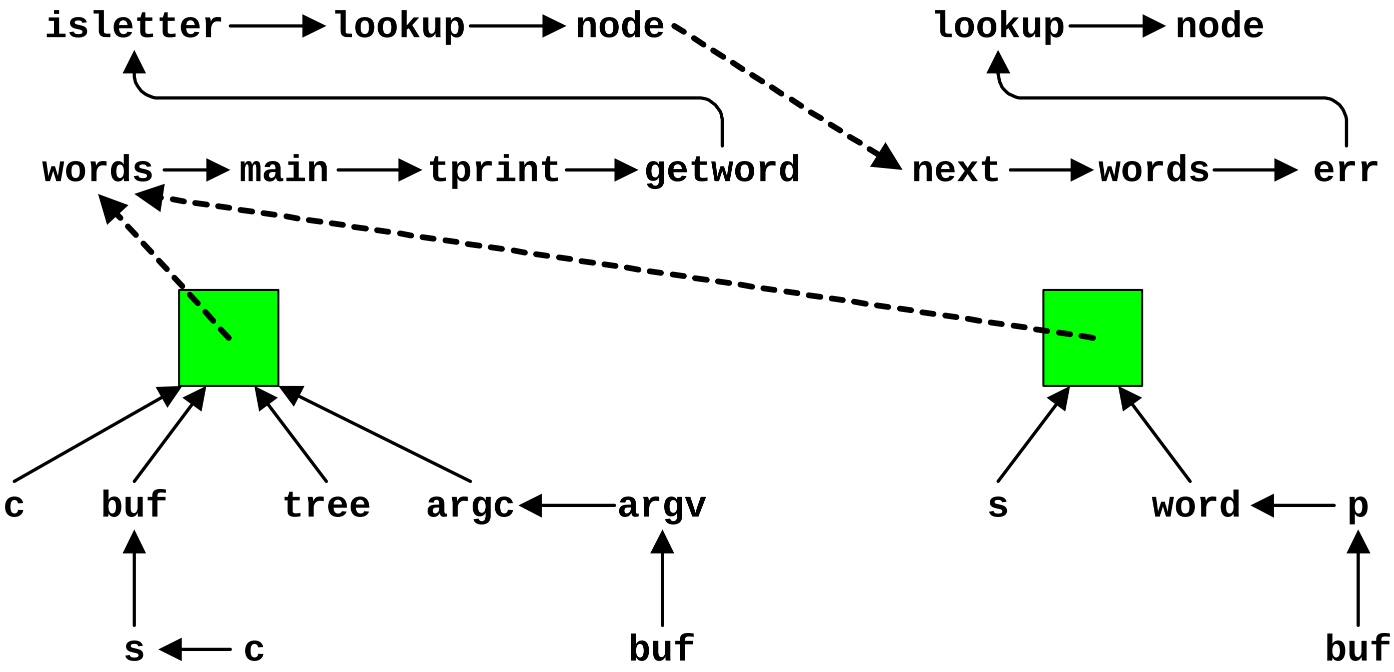
- Debuggers may use/cache more efficient structures, e.g., `symtab.c` uses hash tables

Linking

- For a ***program***, cdbld (the linking script) generates an array of pointers to modules

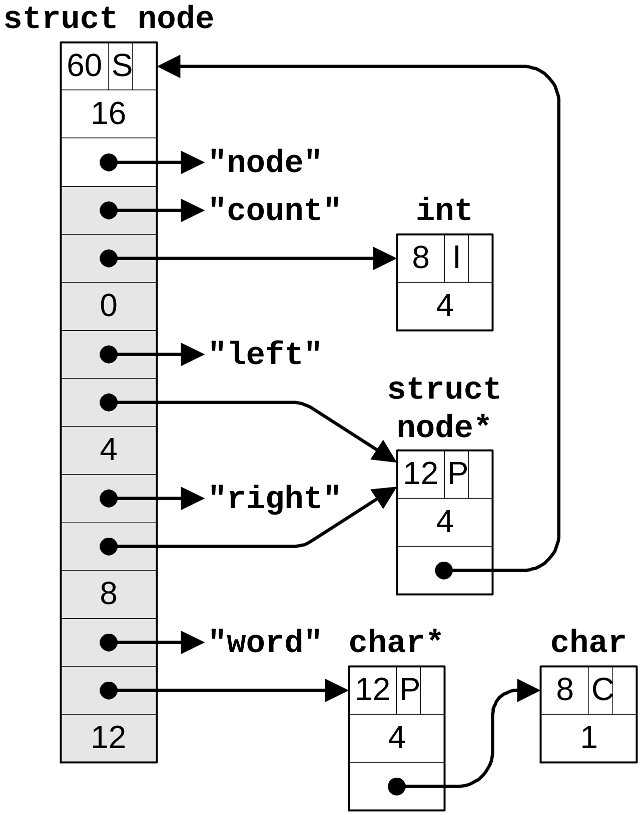
```
struct module *_Nub_modules[] = {
    &__module__V309159f22d5b,
    &__module__V309159f12d59,
    0
};
```

- Machine-independent start-up code collects all file-scope identifiers into a ***single list*** emanating from each module's ***link*** symbol:



Types

- For each type, `lcc` emits an initialized instances of, essentially, an AST for the type



```
struct node {
    int count;
    struct node *left;
    struct node *right;
    char *word;
};
```

- Types guide traversal of values
- Debuggers may use/cache more efficient/compact representations, e.g., `symtab.c` prefetches types

Breakpoints

- **Each module has an array of source coordinates**

```
union scoordinate {
    int i;
    struct { unsigned int y:16,x:10,index:5,flag:1; } le;
    struct { unsigned int flag:1,index:5,x:10,y:16; } be;
};
```

`flag` is always the sign bit; endianness is determined on-the-fly

- **For each stopping point at an expression *e*, `lcc` emits the equivalent of**

`(`*module*`.coordinates[`*n*`].i < 0 && _Nub_bp(`*n*, *tail*`), `*e*`)`

n **is the index of the source coordinate**
tail **points to the leave of the symbol table**

If the breakpoint is set, the nub’s `_Nub_bp` invokes cdb’s breakpoint callback

- **For the lone stopping point on line 24 in `lookup.c`:**

```
if ((__module__V309159f22d5b.coordinates[14].i < 0 &&
    _Nub_bp(14, &L36), next >= sizeof words/sizeof words[0]))
```

Stack Frames

- **`lcc` emits code to build a *<u>shadow stack</u>* embedded in the call stack**

  At entry to `lookup()`: `tos` is a compiler-generated temporary

```
    struct sframe {
        struct sframe *up, *down;
        char *func;
        struct module *module;
        struct ssymbol *tail;
        int ip;
    };
...
    tos.down = _Nub_tos;
    tos.func = "lookup";
    tos.module = &__module__V309159f22d5b;
    _Nub_tos = &tos;
    (symbol structure for p).offset = (char*)&p - (char *)&tos;
    (symbol structure for word).offset = (char*)word - (char *)&tos;
```

  `offset` fields can be set at compile time—with a loss of machine independence

  Shadow frame require *<u>no</u>* allocation/deallocation

  `fp` fields in `_Nub_state_T`s hold pointers to `sframe`s

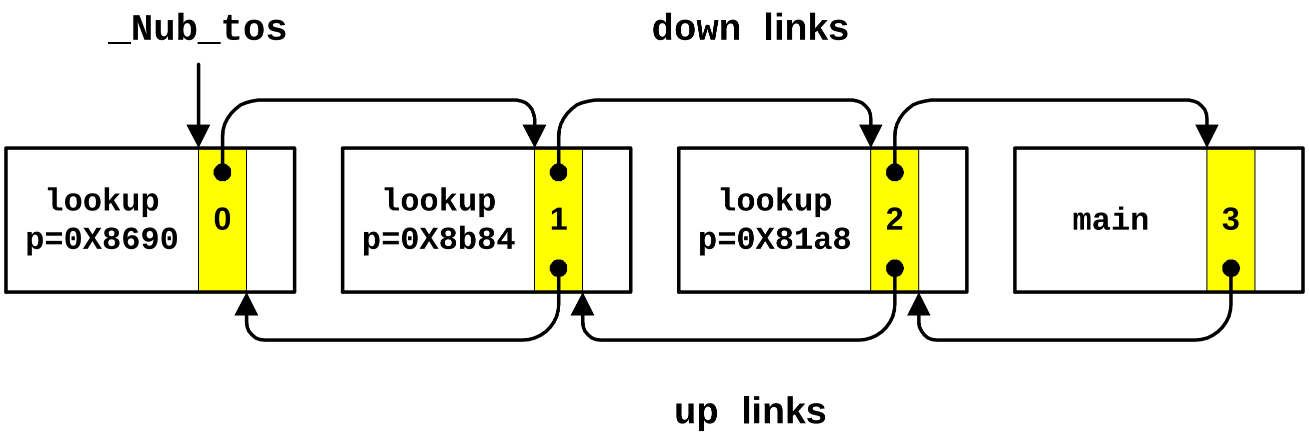
Calls

- **For a call expression *e*, `lcc` emits**

  `(tos.ip =` *n*`,tos.tail =` *tail*`, temp =` *e*`, _Nub_tos = &tos, temp)`

  n **is the index of the source coordinate**
  tail **points to the leave of the symbol table**
  `temp` **is a compiler-generated temporary**

  cdb fills in `up` links when—and if—it moves the focus



Assignment to `_Nub_tos` 'pops' the shadow stack

Popping in return statements doesn't handle `setjmp`/`longjmp` correctly

Overhead

- Space and time overhead are each roughly a factor of 3–4
- Space: building `lcc` (10,000+ lines of C) with 3 variants of itself

text	*data*	*bss*	*variant*
360 KB	20	20	no debugging data
496	32	20	SunOS-specific debugging data
1,584	592	21	cdb debugging data and code

Details:

300,400 bytes	file names and identifiers
291,900	`ssymbol` structures
204,276	`stype` structures
76,292	`scoordinate` structures
348	`module` structures
344	pointers to file names

- Time: compiling `lcc` with each of its three variants

21.9 secs	no debugging data
36.3	SunOS-specific debugging data
93.0	cdb debugging data and code

- Overheads can be easily reduced by sacrificing machine independence

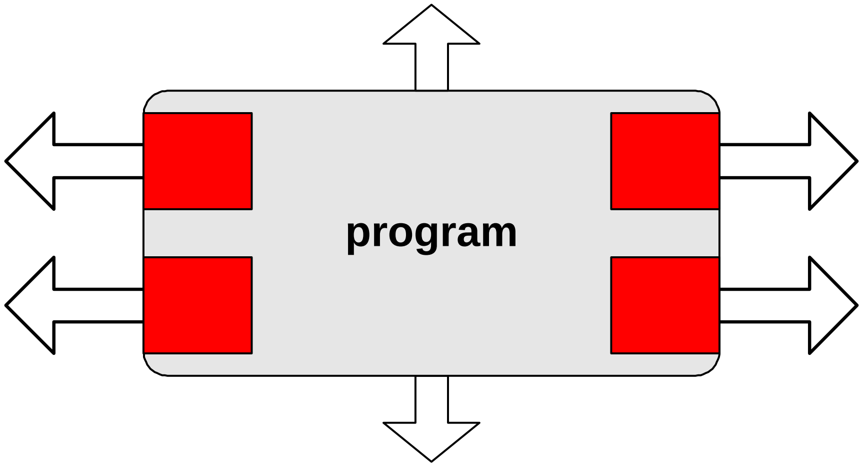
What Happens Next?

- Current projects:

  Single stepping: use existing nub interface or extend it?
  GUI with *Pi*-style point-and-click for exploring structures (Cargill, USENIX’86)
  Duel-style very high-level language for debugging (Golan & Hanson, USENIX’93)
  Nub implementation for UNIX-style symbol tables/executables; ***same*** cdb

- A bigger picture: ‘subterranean’ program interfaces

user interface



operating system interface

Another example: Dynascope (Sosic, PLDI’92)

Inject implementations by
loading with the target, executable editing (Larus & Ball, *SPE*, 2/94),
dynamic loading, page mapping, …