# 98-023A : Concurrent and Distributed Programming w/ Inferno and Limbo

Phillip Stanley-Marbell
pstanley@ece.cmu.edu

# Lecture Outline

- Introduction to the Limbo Programming language
- Limbo language genealogy
- (next week: Limbo data types and the Dis VM)

# Course Outline : Syllabus

- Week 1: Introduction to Inferno
- Week 2: Overview of the Limbo programming language
- Week 3: Types in Limbo
- Week 4: Inferno Kernel Overview
- Week 5: Inferno Kernel Device Drivers
- Week 6: NO CLASS
- Week 7: C applications as resource servers: Built-in modules and device drivers
- Week 8: Case study I — building a distributed multi-processor simulator

Spring Break

- Week 9: Platform independent Interfaces: Limbo GUIs; Project Update
- Week 10: Programing with threads, CSP
- Week 11: Debugging concurrent programs; Promela and SPIN
- Week 12: Factotum, Secstore and Inferno's security architecture
- Week 13: Case study II — Edisong, a distributed audio synthesis and sequencing engine

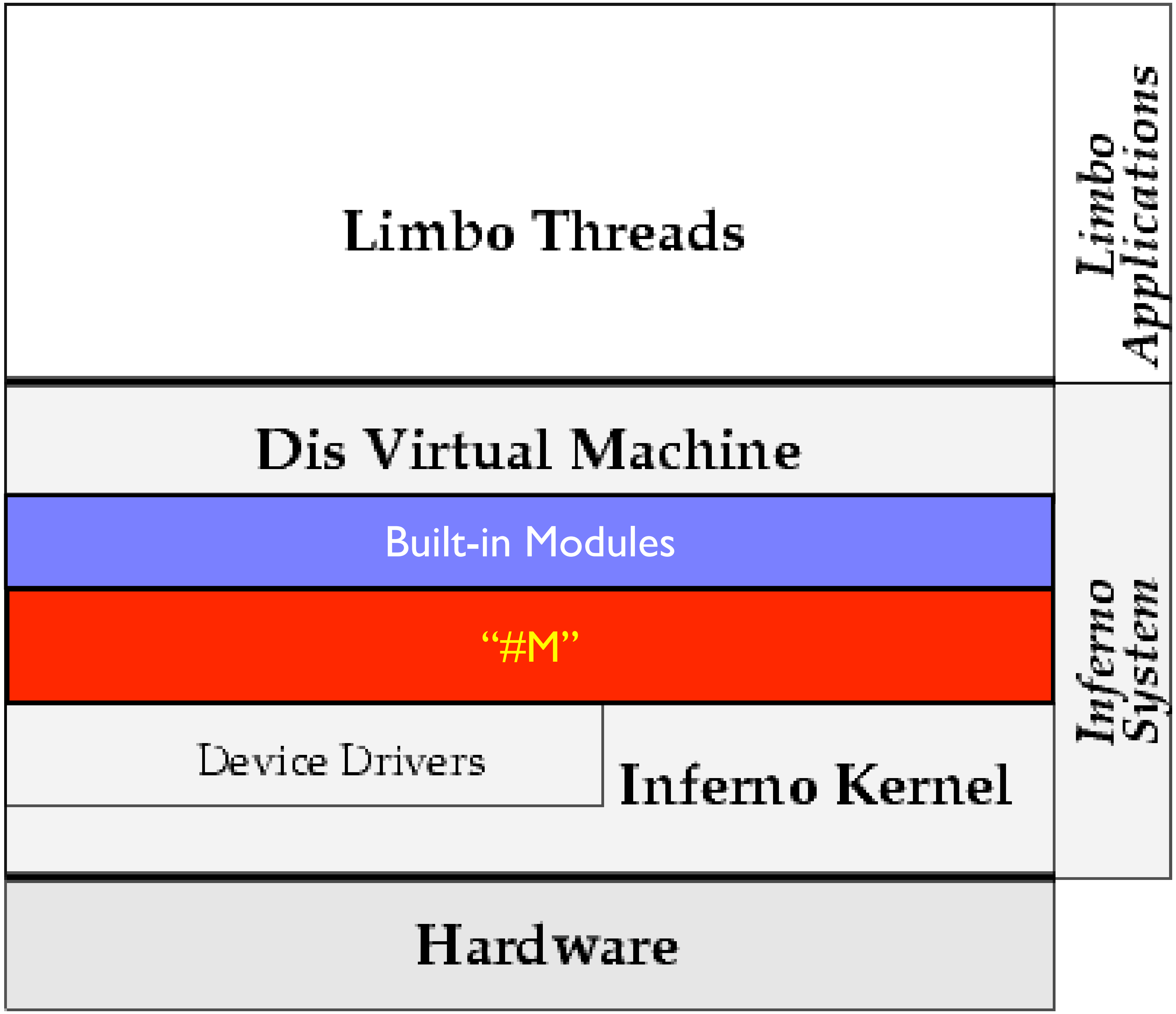
# Schedule Update

- Changed topic for weeks 4 and 5
  - Will be covering the Inferno kernel and device drivers, due to popular demand
- I will be out of town (out of the country) during week 6
  - No class February 16 and February 18th

# The Limbo Programming Language

- Limbo is a concurrent programming language
  - Language level support for thread creation, inter-thread communication over typed channels
- Language-level communication channels
  - Based on ideas from Hoare's Communicating Sequential Processes (CSP)
- Features
  - Safe : compiler and VM cooperate to ensure this
  - Garbage collected
  - Not O-O, but *rather*, employs a powerful module system
  - Strongly typed (compile- and run-time type checking)

# Inferno System Structure



# Inferno's VM: Dis

- Applications compiled for execution on the Dis VM
- Dis has a *memory-to-memory architecture*, optimized for on-the-fly compilation (contrast to the Java Virtual Machine's stack architecture)
- Many Dis VM opcodes map directly to Limbo language constructs, but can support other languages
- We'll see more of this correspondence between Limbo data types and Dis VM internals next week

# Hello World

```
implement HelloWorld;

include “sys.m”;
include “draw.m”;

sys: Sys;

HelloWorld: module
{
      init: fn(ctxt: ref Draw->Context, args: list of string);
}

init(ctxt: ref Draw->Context, args: list of string)
{
      sys = load Sys Sys->PATH;

      #   This is a comment
      sys->print(”Hello World!\n”);
}
```

- Limbo module implementations (like above) usually placed in a file with “`.b`” suffix
- Compiled modules placed in “`.dis`” (contain bytecode for execution on Dis VM)

# Demo: Compiling and running HelloWorld

# Hello World Module

```
implement HelloWorld;

include “sys.m”;
include “draw.m”;

sys: Sys;

HelloWorld: module
{
      init: fn(ctxt: ref Draw->Context, args: list of string);
}

init(ctxt: ref Draw->Context, args: list of string)
{
      sys = load Sys Sys->PATH;

      #   This is a comment
      sys->print(”Hello!\n”);
}
```

Module Name

Various Includes

Module Type (interface) Definition

Module Implementation

- Module interface definitions often placed in separate “.m” files by convention
  - Module definitions define a new “type”
  - Compiled modules in “.dis” file contains this type information
  - *lvalue* of a `load` statement must match this type

# Interface vs. Implementation

- Module interface defines a new type
- Module interface (type) can either be included inline, or stored in a separate "`.m`" file and included
  - Other modules which want to use eventual implementation will include this "`.m`" header file
- Module implementation is an instance of this new type
  - Implementation source is in "`.b`", and eventual compiled binary in a "`.dis`"

# Modules

- Applications are structured as a collection of modules

- Component modules of an application are loaded dynamically and type-checked at runtime
  - Each compiled program is a single module

  - Any module can be loaded dynamically and used by another module
    - Shell loads “`helloworld.dis`” when instructed to, and “runs” it

  - There is no static linking
    - Compiled “Hello World” does not contain code  for print etc.

```
init(ctxt: ref Draw->Context, args: list of string)
{
      sys = load Sys Sys->PATH;

      #   This is a comment
      sys->print(”Hello!\n”);
}
```

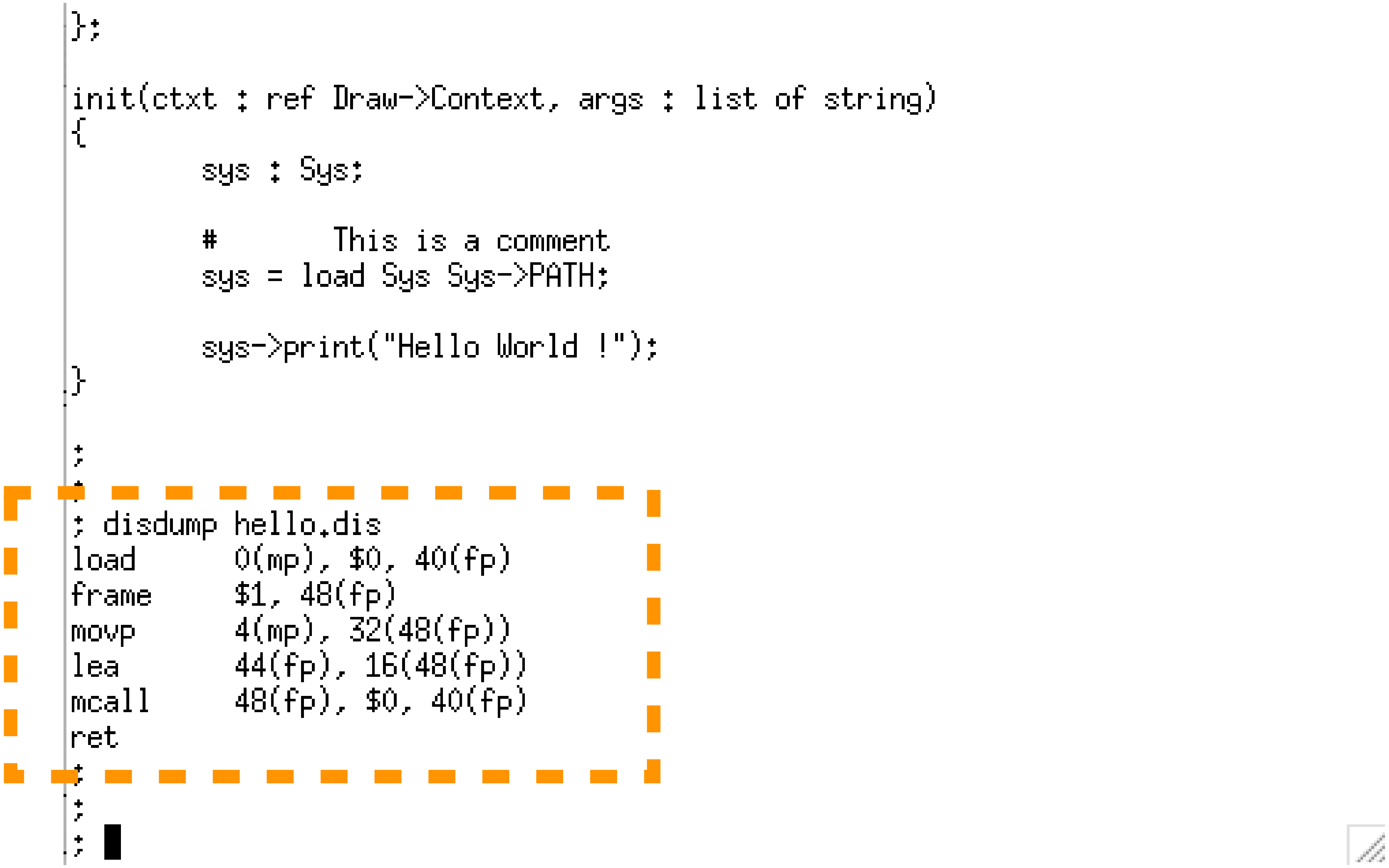
# Compiled module (”`.dis`”) contents

- Compiled modules contain only the code as defined in source file

- Other modules used (e.g., for print) are not “compiled in”, but are *ALL* loaded dynamically, at runtime, from a specified file

# Compiled module (”.dis”) contents

- HelloWorld module only contains code to load Sys module then do a module function call



```
};

init(ctxt : ref Draw->Context, args : list of string)
{
        sys : Sys;

        #       This is a comment
        sys = load Sys Sys->PATH;

        sys->print("Hello World !");
}

;
;
; disdump hello.dis
load     0(mp), $0, 40(fp)
frame    $1, 48(fp)
movp     4(mp), 32(48(fp))
lea      44(fp), 16(48(fp))
mcall    48(fp), $0, 40(fp)
ret
;
;
; █
```

# Demo: `disdump` and Module manager — Examining an executable's contents

# Demo (`objdump -d` on a compiled C program)

# Language Data Types

- Basic types
    - `int` — 32-bit, signed 2's complement notation
    - `big` — 64-bit, signed
    - `byte` — 8-bit, unsigned
    - `real` — 64-bit IEEE 754 long float
    - `string` — Sequence of 16-bit Unicode characters

- Structured Types
    - `array` — Array of basic or structured types
    - `adt`, `ref adt` — Grouping of data and functions
    - `list` — List of basic or structured data types, list of list, etc.
    - `chan`
    - Tuples

# Limbo Modules

- How do you know where to load module implementation from ?
  - By convention, location of implementation is stored in the constant "PATH" of the module's interface declaration

- Example: `/module/smtp.m`

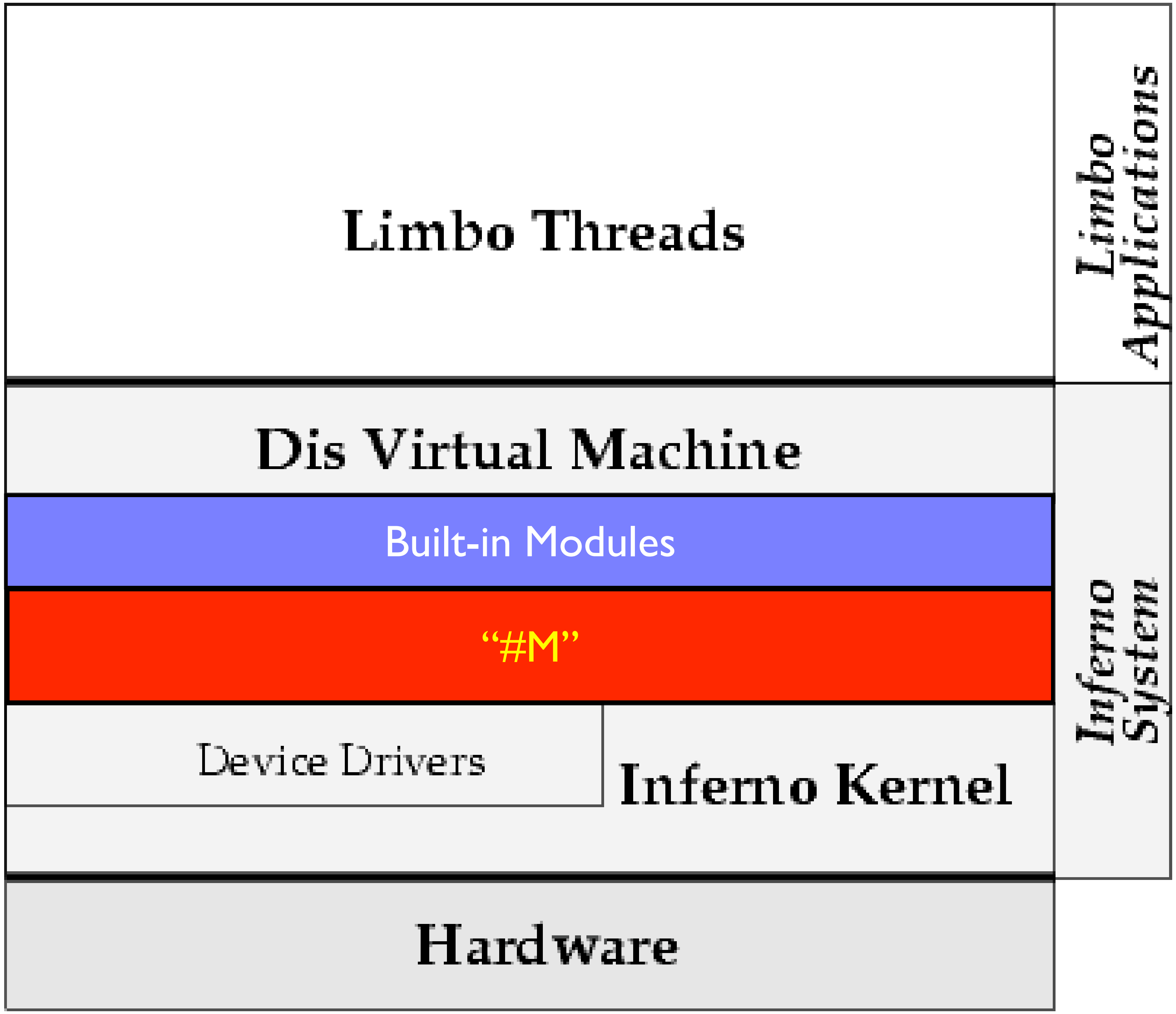
```
# smtp protocol independent access to an email server.

Smtp : module
{
        PATH : con "/dis/lib/smtp.dis";

        open: fn(server : string) : (int, string);
        sendmail: fn(fromwho: string,
                             towho: list of string,
                             cc : list of string,
                             msg: list of string) : (int, string);

         # close the connection - returns (status, error string)
         close: fn() : (int, string);
};
```

# Built-in Modules



- These are modules built into the system, such as Sys
- Built-in modules are implemented in C
- How are they loaded since there is no .dis file ?
  - `handle = load “$Name”;`

# The Sys Built-in Module

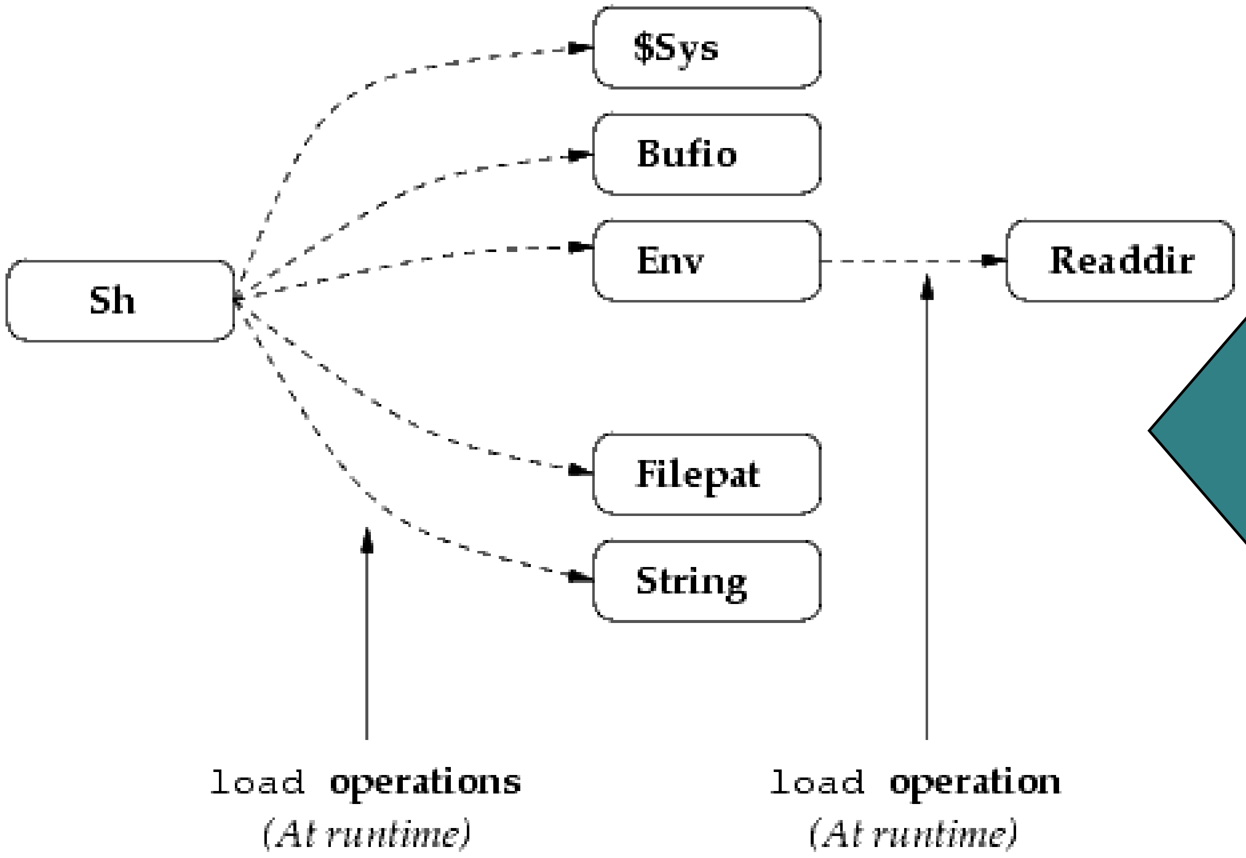
- This provides the link between Limbo application and Inferno kernel / emulator facilities
- Provides facilities for I/O etc.

# More details on modules

- The "$Loader" built-in module
- Module signatures
- Module structure
- Generating C stubs from Limbo module definitions
  - ...All of the above will be covered when we talk about Built-in Modules later in the semester

# Dynamic Loading of Modules

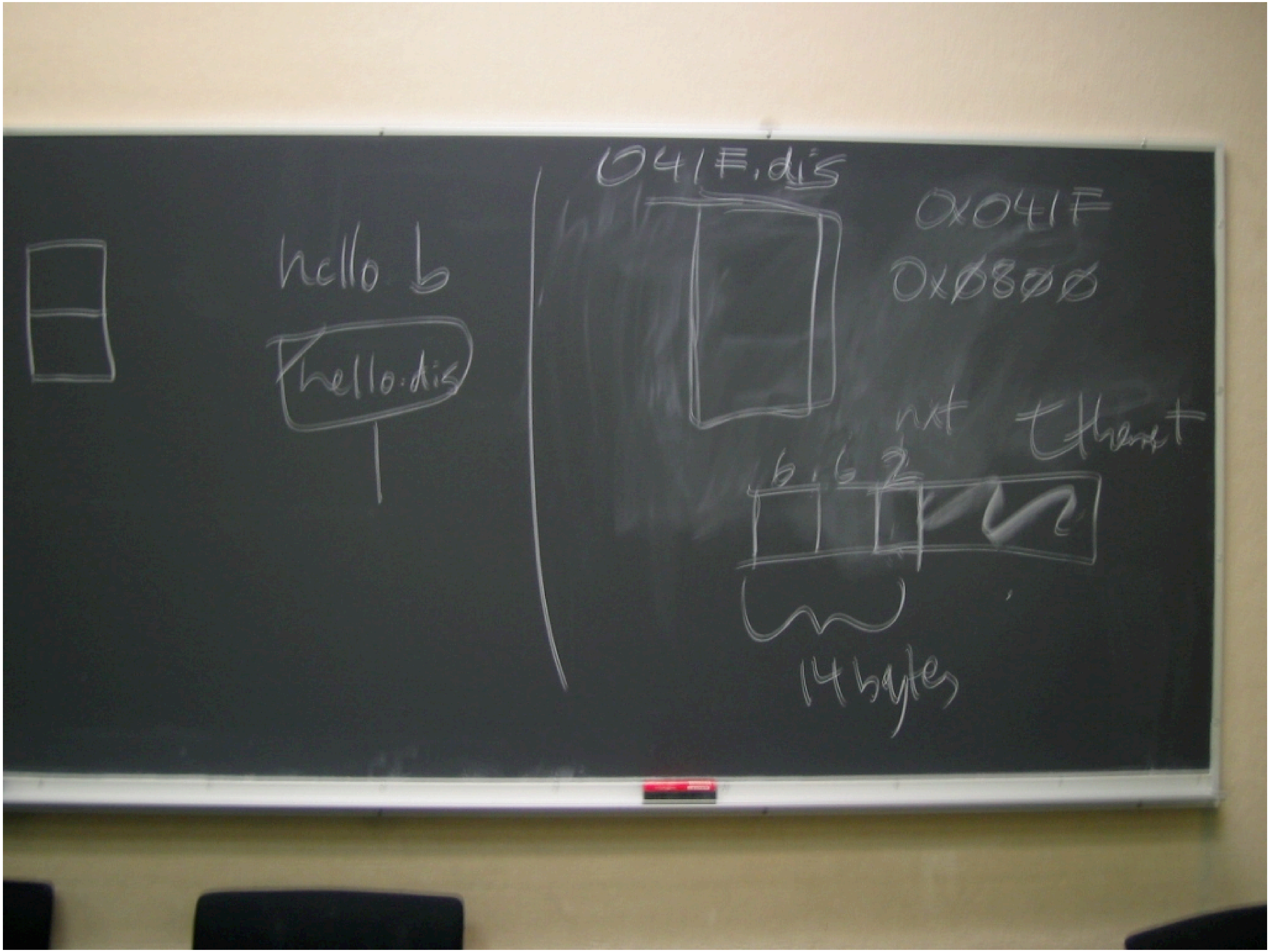
- Module type information is statically fixed in caller module, but the actual implementation loaded at run time is not fixed, as long as it type-checks



`Sh` module (the command shell) loads the `Bufio`, `Env` and other modules at runtime. The `Env` module loads other modules that it may need (e.g., `Readdir`)

# Dynamic loading example: Xsniff

- An extensible packet sniffer architecture
- Dynamically loads and unloads packet decoder modules based on observed packet types
  - All implementations of packet decoders conform to a given module type (module interface difinition)
  - File name containing appropriate decoder module is "computed" dynamically from packet type (e.g., ICMP packet inside Ethernet frame) , and loaded if implementation is present
  - New packet decoders at different layers of protocol stack can be added transparently, even while Xsniff is already running!


hello.b
hello.dis
041F.dis
0x041F
0x0800
nxt
Ethernet
6 6 2
14 bytes

# Xsniff (1)

Xsniff Module Definition

Modules which will be run from shell must define "**init**" with this signature

```
implement Xsniff;

include "sys.m";
include "draw.m";
include "arg.m";
include "xsniff.m";

Xsniff : module
{
      DUMPBYTES : con 32;

      init : fn(nil : ref Draw->Context, args : list of string);
};

sys    : Sys;
arg    : Arg;
verbose    := 0;
etherdump  := 0;
dumpbytes  := DUMPBYTES;

init(nil : ref Draw->Context, args : list of string)
{
      n : int;
      buf := array [Sys->ATOMICIO] of byte;

      sys = load Sys Sys->PATH;
      arg = load Arg Arg->PATH;
```

# Xsniff (2)

```
      dev := "/net/ether0";
      arg->init(args);

      #     Command line argument parsing. Omitted...

      #     Open ethernet device interface
      tmpfd := sys->open(dev+"/clone", sys->OREAD);

      #     Determine which of /net/ether0/nnn
      n = sys->read(tmpfd, buf, len buf);
      (nil, dirstr) := sys->tokenize(string buf[:n], " \t");

      channel := int (hd dirstr);
      infd := sys->open(dev+sys->sprint("/%d/data", channel),
                                  sys->ORDWR);

      sys->print("Sniffing on %s/%d...\n", dev, channel);
      tmpfd = sys->open(dev+sys->sprint("/%d/ctl", channel),
                                 sys->ORDWR);

      # Get all packet types (put interface in promisc. mode)
      sys->fprint(tmpfd, "connect -1");
      sys->fprint(tmpfd, "promiscuous");

      #    Spawn new thread w/ ref to opened ethernet device
      spawn reader(infd, args);
}
```

Open data interface for Ethernet driver

Open control interface for Ethernet driver

`spawn` statement creates new thread from function

# Xsniff (3)

```
reader(infd : ref Sys->FD, args : list of string)
{
        n : int;
        ethptr : ref Ether;
        fmtmod   : XFmt;

        ethptr = ref Ether(array [6] of byte, array [6] of byte,
                                array [Sys->ATOMICIO] of byte,0);

        while (1)
        {
                n  = sys->read(infd, ethptr.data, len ethptr.data);

                ethptr.pktlen = n - len ethptr.rcvifc;
                ethptr.rcvifc = ethptr.data[0:6];
                ethptr.dstifc = ethptr.data[6:12];

                nextproto := "ether"+sys->sprint("%4.4X",
                          (int ethptr.data[12] << 8) |
                          (int ethptr.data[13]));

                if ((fmtmod == nil) || (fmtmod->ID != nextproto))
                {
                        fmtmod = load XFmt XFmt->BASEPATH +
                        nextproto + ".dis";
                        if (fmtmod == nil) continue;
                }

                (err, nil) := fmtmod->fmt(ethptr.data[14:], args);
        }

        return;
}
```

Compute a module implementation file name, based on Ethernet frame `nextproto` field

Try to load an implementation from the file name computed (e.g., will be `ether0800.dis` if frame contained IP)

Decode frame, possibly passing frame to further filters

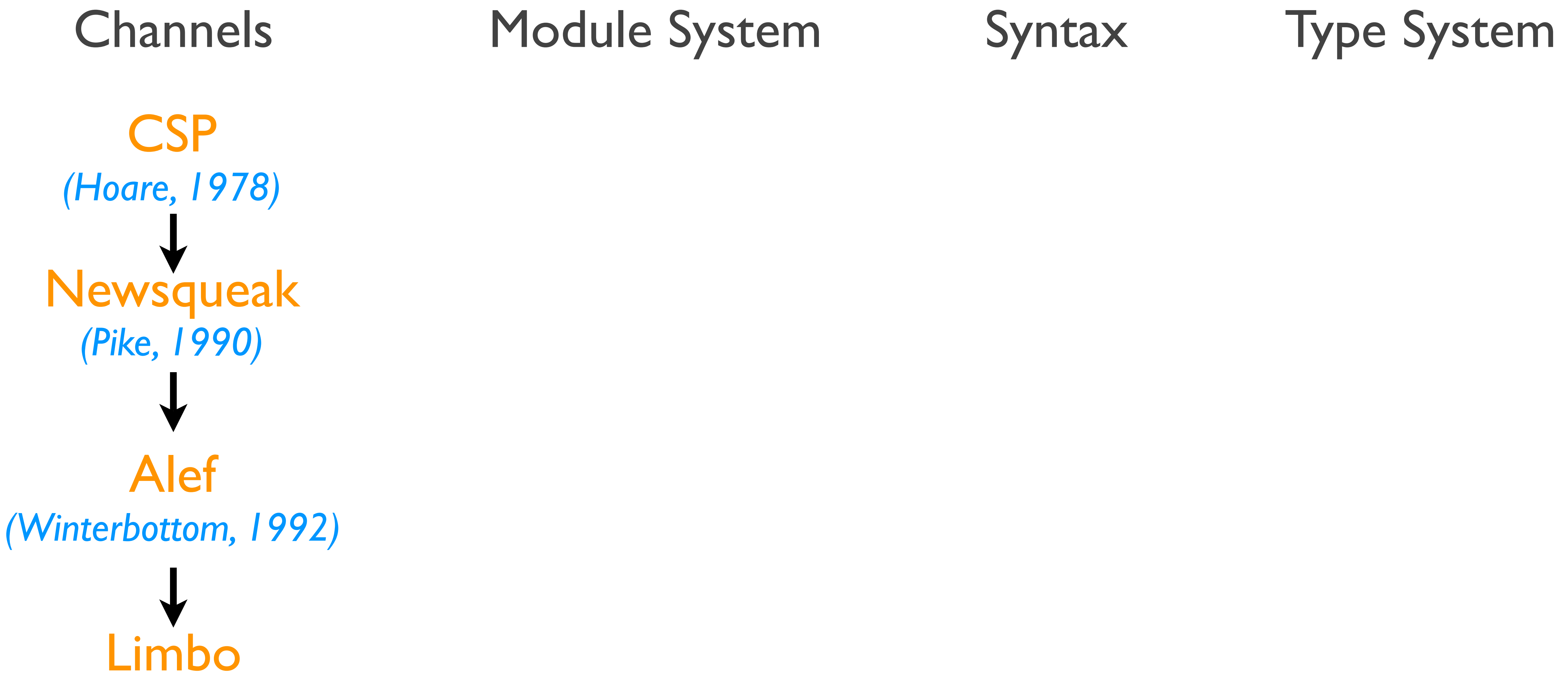
# More Examples

- See book’s web page

http://www.ece.cmu.edu/~pstanley/ipwl/sourcecode/book-examples/

*(remind me to show this to you in a browser, right now)*

# Limbo Language Genealogy (abridged)



Channels

Module System

Syntax

Type System

CSP

*(Hoare, 1978)*

Newsqueak

*(Pike, 1990)*

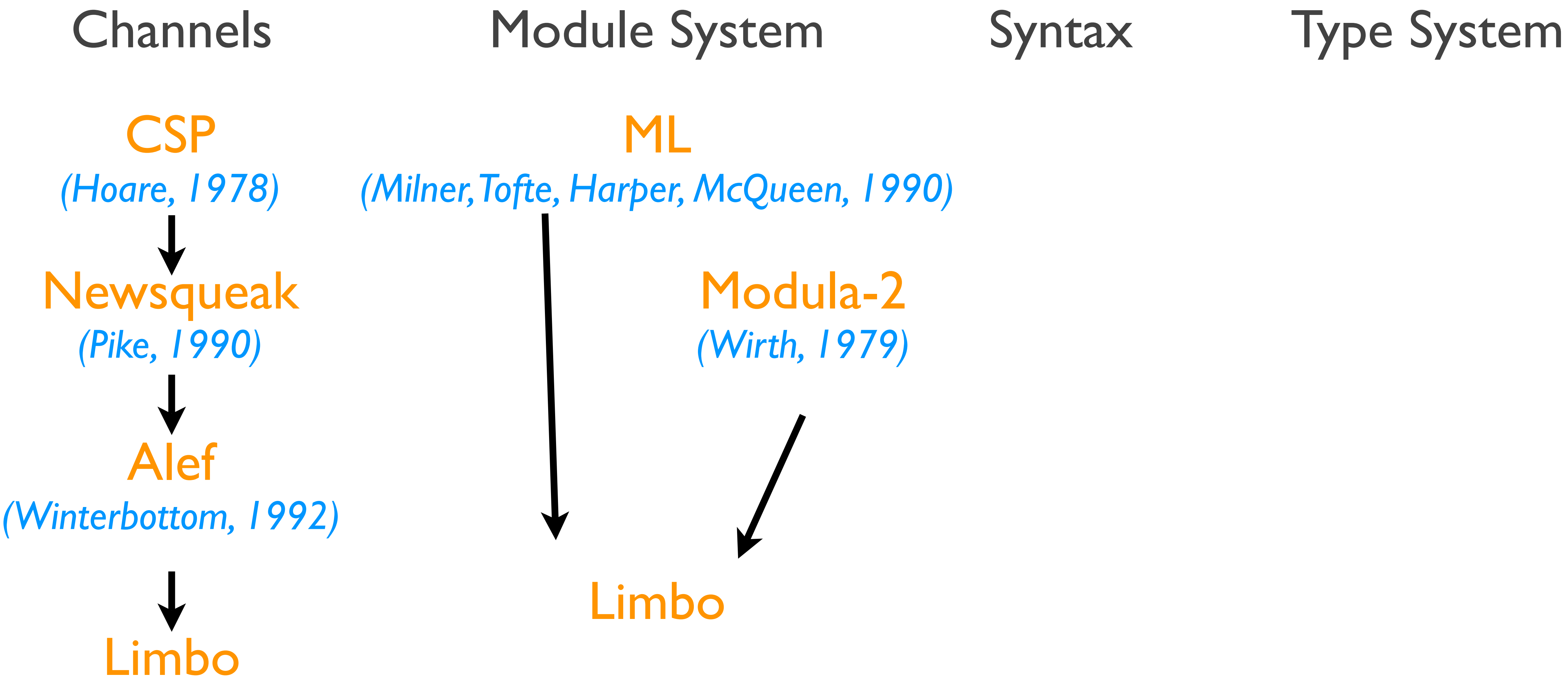
Alef

*(Winterbottom, 1992)*

Limbo

- Language-level "communication variables", the channel data type, is influenced by *CSP*, via *Alef* and *Newsqueak*

# Limbo Language Genealogy (abridged)



Channels

Module System

Syntax

Type System

CSP
*(Hoare, 1978)*

Newsqueak
*(Pike, 1990)*

Alef
*(Winterbottom, 1992)*

Limbo

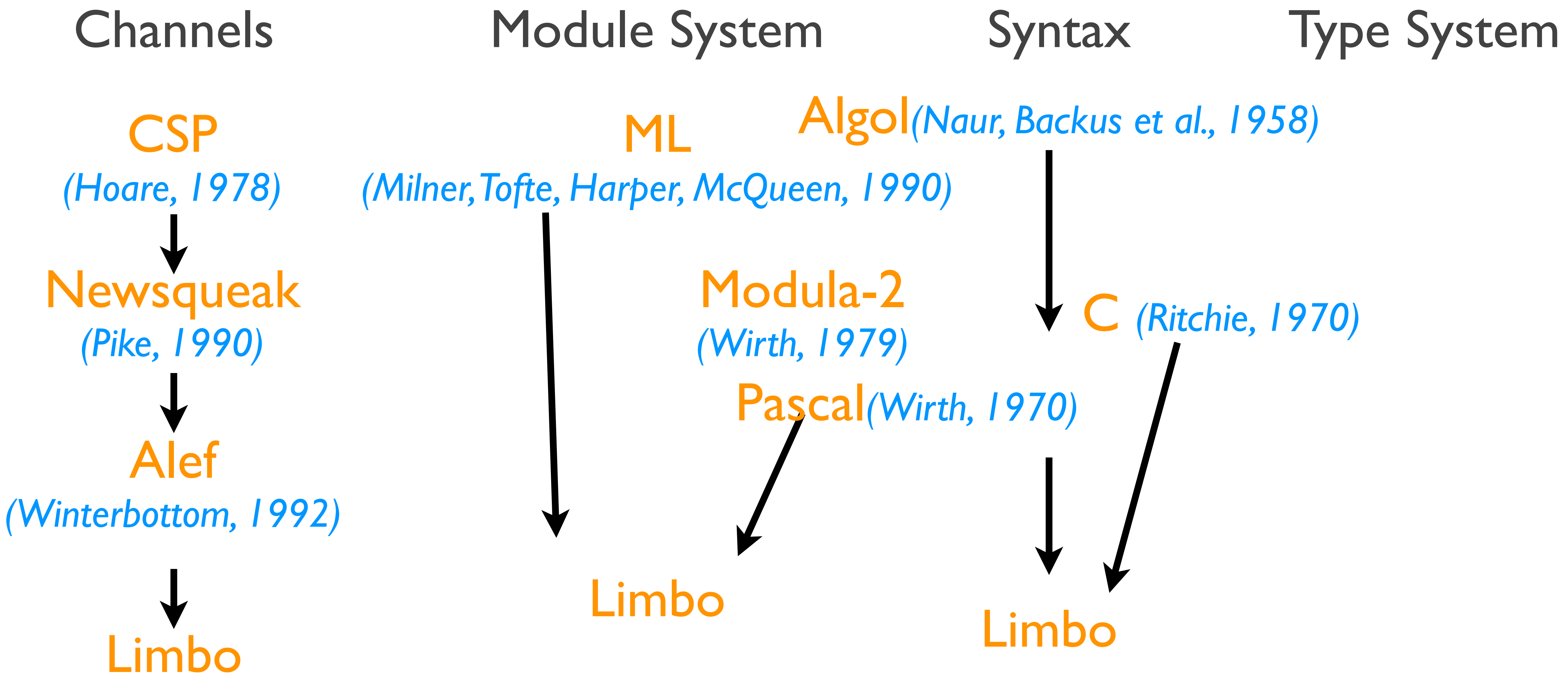
ML
*(Milner, Tofte, Harper, McQueen, 1990)*

Modula-2
*(Wirth, 1979)*

Limbo

- Limbo's module system is influenced by *ML* and *Modula-2*

# Limbo Language Genealogy (abridged)



Channels

CSP
*(Hoare, 1978)*

Newsqueak
*(Pike, 1990)*

Alef
*(Winterbottom, 1992)*

Limbo

Module System

ML
*(Milner, Tofte, Harper, McQueen, 1990)*

Modula-2
*(Wirth, 1979)*

Pascal *(Wirth, 1970)*

Limbo

Syntax

Algol *(Naur, Backus et al., 1958)*

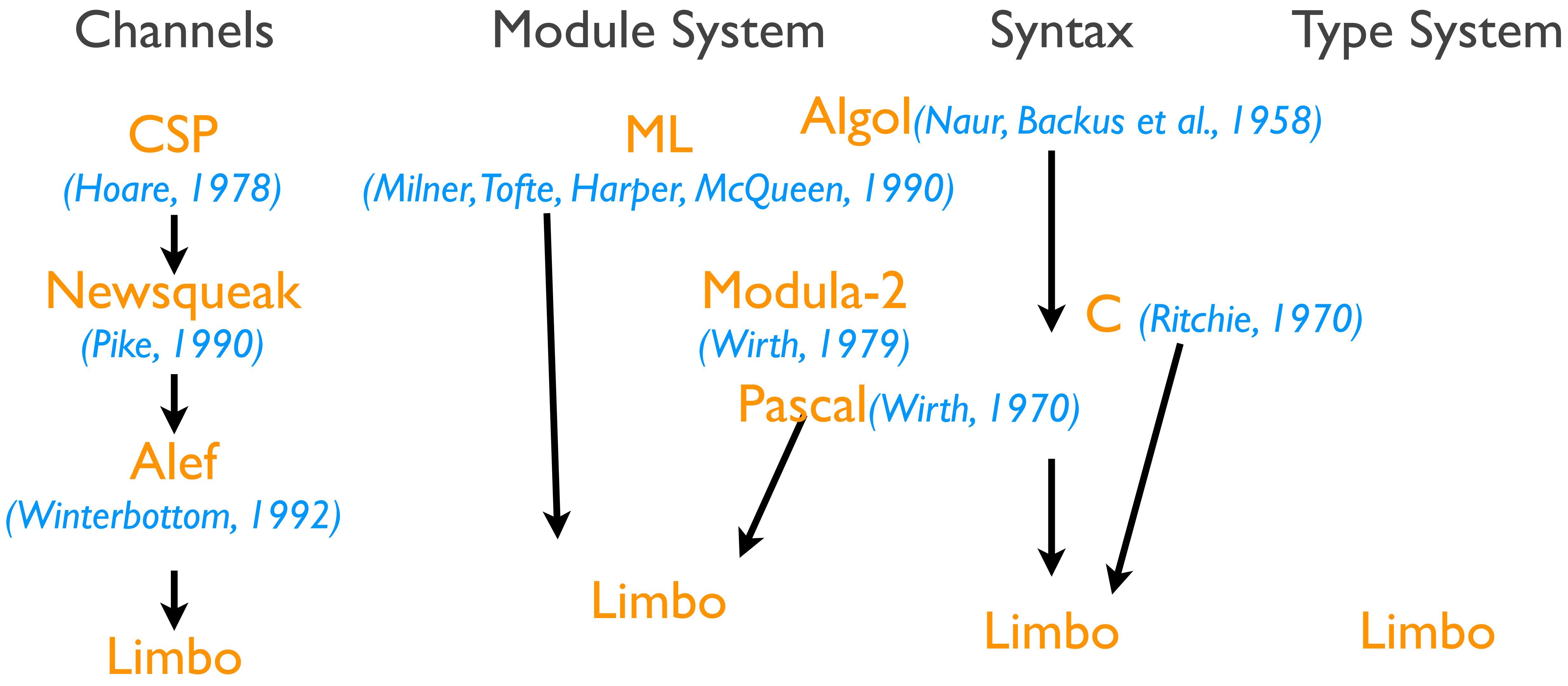
C *(Ritchie, 1970)*

Limbo

Type System

- Syntax is similar to “Algol Family” of languages, most popular of which is probably C

# Limbo Language Genealogy (abridged)



Channels

CSP *(Hoare, 1978)*

Newsqueak *(Pike, 1990)*

Alef *(Winterbottom, 1992)*

Limbo

Module System

ML *(Milner, Tofte, Harper, McQueen, 1990)*

Modula-2 *(Wirth, 1979)*

Pascal *(Wirth, 1970)*

Limbo

Syntax

Algol *(Naur, Backus et al., 1958)*

C *(Ritchie, 1970)*

Limbo

Type System

Limbo

- Shares similarities in data types with *CSP* etc (channels), *ML* (language level lists and operators), module types, *C*

# Next Lecture

- More on Limbo, compilation, debugging, etc.
- Next week ADTs, types and the Dis VM (monday), fixed point arithmetic formats and overview (wednesday)

*Fin.*