

# Building Distributed Applications with Inferno and Limbo

Phillip Stanley-Marbell

[pstanley@ece.cmu.edu](mailto:pstanley@ece.cmu.edu)

<http://www.ece.cmu.edu/~pstanley>

# Talk Outline

- Terminology, Overview and History
- Abstraction and Names: Resources as Names (files) in Inferno
- The Limbo Programming Language
- Multi-platform applications
- Ideas & Summary

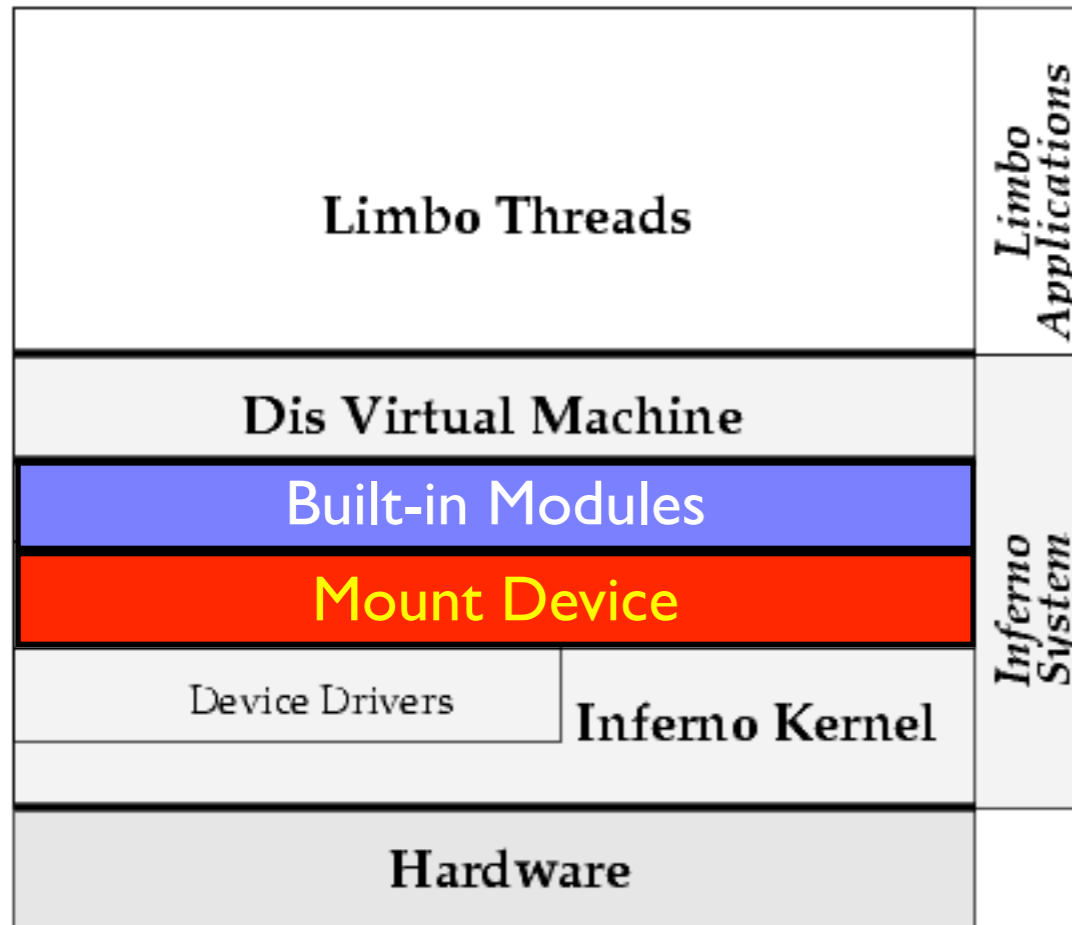
# Terminology

- **Inferno**
  - An operating system
- **Limbo**
  - A programming language for developing applications under Inferno
- **Dis**
  - Inferno abstracts away the hardware with a virtual machine, the Dis VM
  - Limbo programs are compiled to bytecode for execution on the Dis VM
- **Plan 9**
  - A research operating system, being actively developed at Bell Labs and elsewhere
  - A direct ancestor of Inferno

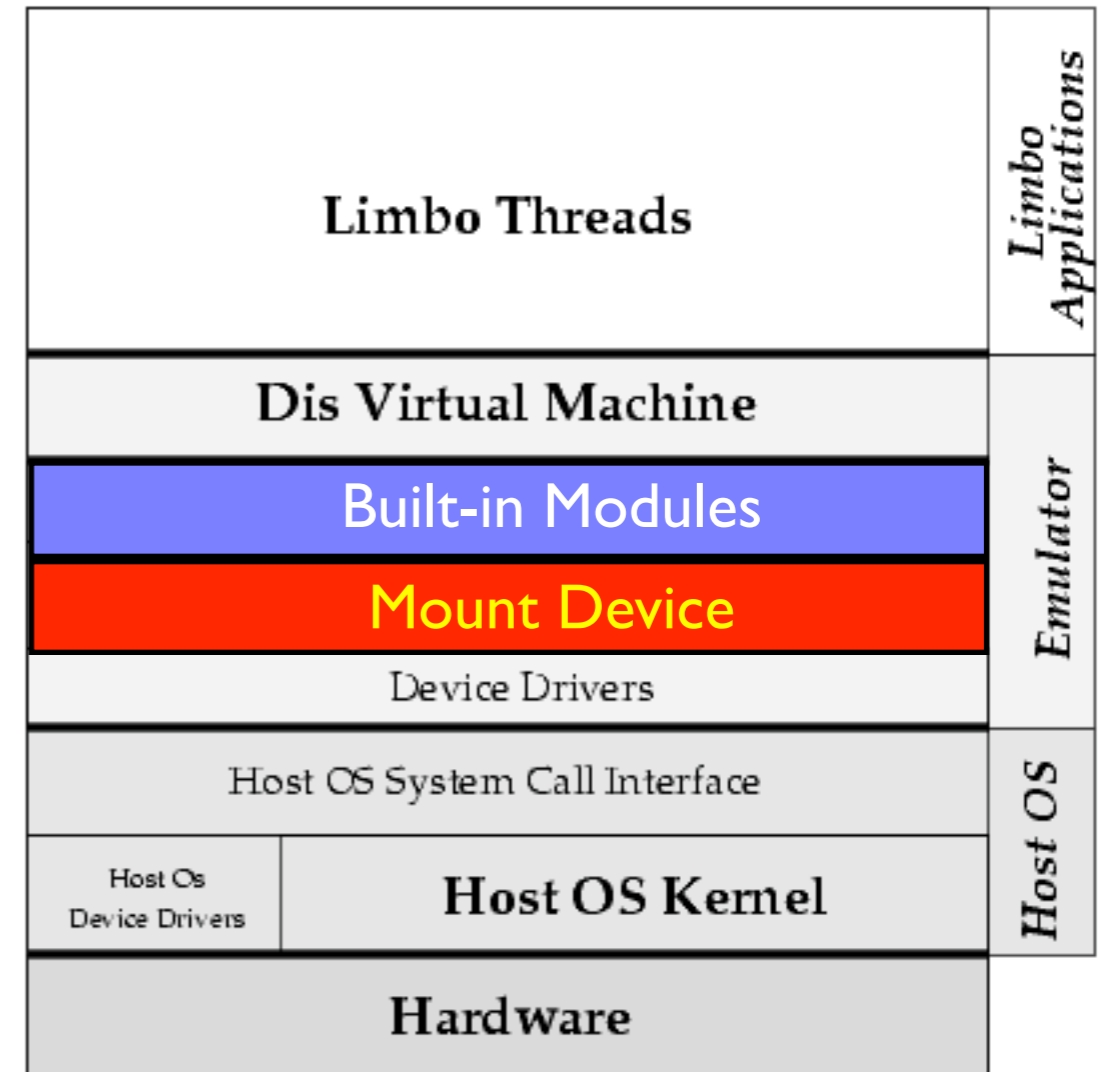
# Inferno

- **Runs directly over bare hardware**
  - Intel x86, PowerPC, SPARC, MIPS, ARM, more...
  - Like any other traditional OS
- **Also available as an emulator**
  - Runs over many modern operating systems (Windows, Linux, \*BSD, Solaris, IRIX, MacOS X) or as a browser plugin in IE under Windows
- **Emulator provides interface identical to native OS, to both users and applications**
  - Filesystem and other system services, applications, etc.
  - The emulator **virtualizes the entire OS, including filesystem, network stack, graphics subsystem — everything — not just code execution** (e.g., in Java Virtual Machine)

# Inferno System Architecture



**Native** (i.e., running directly over hardware)

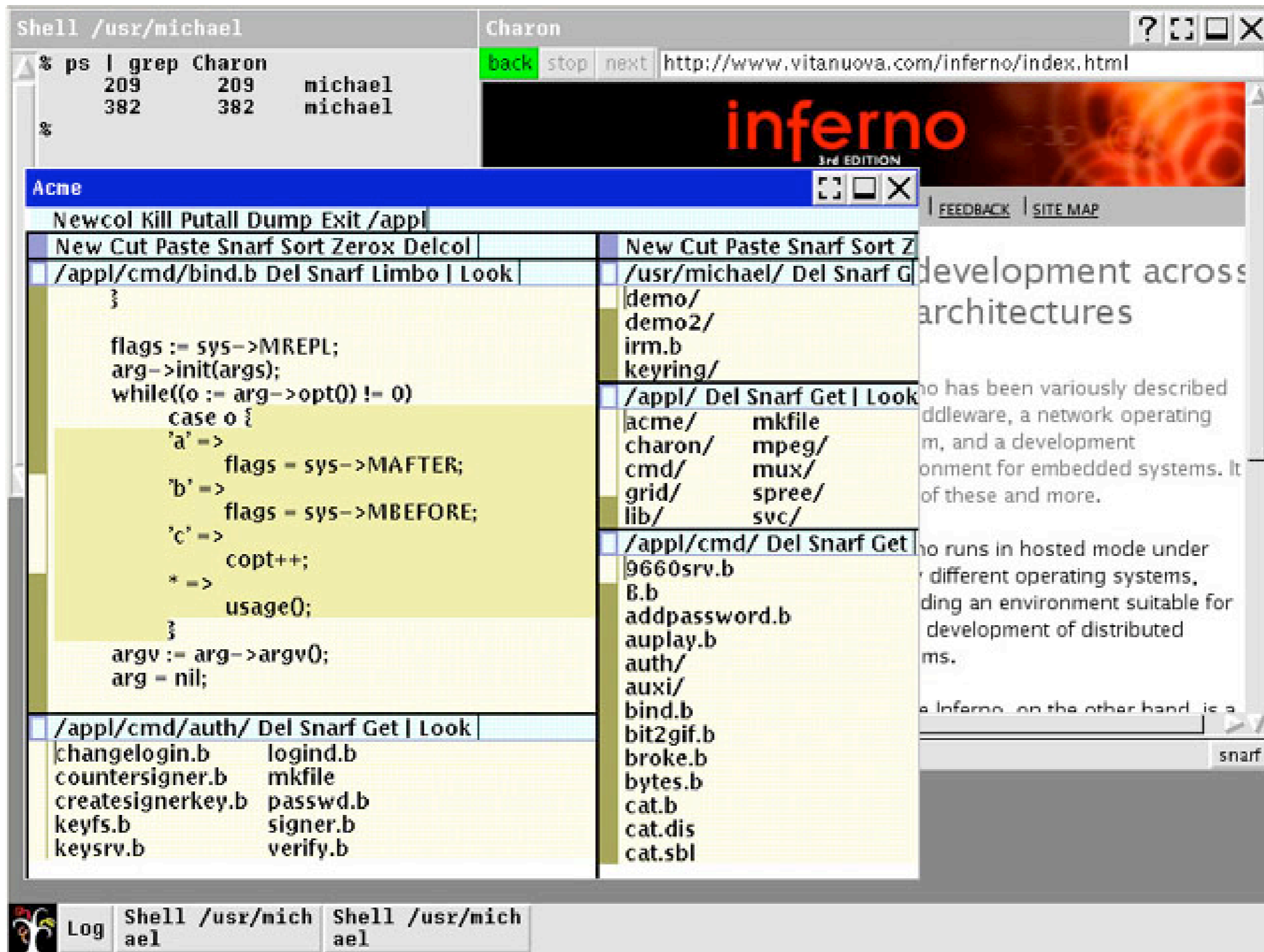


**Hosted** (i.e., emulator)

# Available Software

- Text/SGML editors
- Web browser, WML browser, Mail Client
- Graphical debugger
- Games
- “Grid computing” tools
- Clones of Unix tools ([sed](#), [banner](#), etc.)
- Other (not part of the distribution)
  - Audio editor / sequencer / synthesis ([edison](#))
  - Image manipulation tools

# Applications



- Shell, Acme editor, Charon web browser

# History

- Developed, circa 1995, by the Computing Science Research Center at Bell Labs

Sean Dorward, Rob Pike, David Presotto, Dennis Ritchie, Howard Trickey, Phil Winterbottom



- Drew upon lessons learned building Plan 9 research OS
- Developed as a decidedly commercial product
  - Developed by Bell Labs research, transferred to the “Inferno Business Unit” a semi-autonomous spinoff, housed within Lucent (Murray Hill, NJ)



# History

- Inferno Business Unit attempted to market Inferno as the ideal solution for developing “Screen Phones”



Philips Consumer Communications IS2630 Screen Phone  
Developed by Philips facility in Eatontown NJ,  
Manufactured in Guadalajara Mexico

- Using Inferno for “screen phones” was not a particularly inspired idea.
- No significant improvement of Inferno (*as delivered by research group*) is performed before attempting to sell it
  - Circa 1999, Inferno Business Unit folds.

# History

- 2000: Rights to Inferno sold to Vita Nuova Holdings, York England



- VN was co-founded by Charles Forsyth, who had been actively involved in Plan 9 and Inferno work, and wrote the Plan 9 PowerPC compiler
- Source code license reduced from \$1,000,000/\$300,000 to \$100, more liberal license
- 2003: Inferno, including source code to entire system, made available under a dual licensing scheme, for free

# History

- Some commercial products that used inferno

- Lucent Managed Firewall



- Lucent PathStar Access Server

- PathStar was a carrier-grade voice-over-IP platform
- Served multiple POTS phone lines (bottom of cabinet), converging both voice traffic and data traffic, to be sent over a WAN, possibly to another PathStar
- Inferno used for control / management / user interface
- Custom “microkernel”, LCOS, run on the Line Cards
- Original prototype designed and built primarily by Phil Winterbottom and Ken Thompson

- Philips Consumer Communications IS2630 Screen phone



# Talk Outline

- Terminology, Overview and History
- Abstraction and Names: Resources as Names (files) in Inferno
- The Limbo Programming Language
- Multi-platform applications
- Ideas & Summary

# Resource abstraction

- **Resource abstraction is a good thing**
  - Operating systems abstract away CPU, disk, network as *system calls*
  - System call abstraction is unfortunately not easily scalable across, e.g., network (well, there's RPCs)
- **Files are one abstraction**
  - Abstraction for bytes on disk (or elsewhere)
  - Nothing inherently tying the concept of files to bytes on disk
    - Except of course, the operating system / file server's implementation

# Names

- Can think of **files as names with special properties**
  - Size
  - Access permissions
  - Other state (creation/modification/access time)
  - These properties (e.g., Unix **struct stat**) are largely a historical vestige — *we could imagine files with more sophisticated ‘types’*
- **Files are just an abstraction**
  - There’s nothing inherently tying files (*i.e., names*) to bytes on disk
  - Association with disk files just happens to be most common use
  - **This abstraction is however very nice to deal with** : many of us regularly access files on remote hosts, e.g., via AFS

# Resources as files

- Since files are so easy to deal with
  - Can we represent all resources as names (files) in a name space ?
    - Process control ?*
    - Network connections / access to network protocol stack ?*
    - Graphics facilities ?*
  - Get rid of all system calls except those for acting on files (*open, close, read, write, stat*, etc.) ?
  - This file/name abstraction is **not much more expensive** than system call interface
- **Resources as files + remote access to files**
  - We could build interesting distributed systems, with resources (files, i.e., names) spread across networks

# Inferno : Resources as files

- Builds on ideas developed in Plan 9
  - *Most system resources represented as names* (files) in a hierarchical *name space*
  - *Names provide abstraction for resources*
    - Graphics
    - Networking
    - Process control
  - Resources accessed by simple file operations (*open, close, read, write, stat*, etc.)
  - System transforms file operations into messages in a simple protocol (“*Styx*”) for accessing remote names
- Implications
  - *Access local and remote resources with the same ease as local/remote files*
  - Name space is “per process”, so different programs can have different views of available resources
  - Restrict access to resources by restricting access to portions of name space



# Resources as files (names)

- Networking
  - Network protocol stack represented by a hierarchy of names

```
; du -a /net
0  /net/tcp/0/ctl
0  /net/tcp/0/data
0  /net/tcp/0/listen
0  /net/tcp/0/local
0  /net/tcp/0/remote
0  /net/tcp/0/status
0  /net/tcp/0
0  /net/tcp/clone
0  /net/tcp/
0  /net/arp
0  /net/iproute
...
```

- Graphics
  - Access to drawing and image compositing primitives through a hierarchy of names

```
; cd /dev/draw
; lc
new
; tail -f new &
1 0 3 0 0 640 480
; lc
1/      new
; cd 1
; lc
ctl  data  refresh
```

# Example `/prog` : process control

- Connect to a remote machine and attach its name space to the local one at `/n/remote`:

```
; mount net!haus.gemusehaken.org /n/remote
```

- Union remote machine's `/prog` into local `/prog`:

```
; bind -a /n/remote/prog /prog
```

- `ps` will now list processes running on both machines, because *it works entirely through the `/prog` name space*:

```
; ps
```

1	1	pip	release	74K	Sh[\$Sys]
7	7	pip	release	9K	Server[\$Sys]
8	1	pip	alt	9K	Cs
10	7	pip	release	9K	Server[\$Sys]
11	7	pip	release	9K	Server[\$Sys]
15	1	pip	ready	73K	Ps[\$Sys]
1	1	abby	release	74K	Sh[\$Sys]
8	1	abby	release	73K	SimpleHTTPD[\$Sys]

- *Can now simultaneously debug/control processes running on both machines*

# Questions to mull on

- Contrast the behavior of `/prog` in Inferno to `/proc` in Unix
  - The `ps` utility does not work exclusively through `/proc`
  - Debuggers like GDB do not debug processes exclusively through `/proc`
  - `ps` and `gdb` cannot be directed to list processes on a remote machine or debug a process on a remote machine, even if they (somehow) have access to the `/proc` filesystem remotely
  - Can you mount and see the `/proc` of a remote system, by, say, AFS ? NFS ?

Incidentally, `/proc` in Unix was done by T. J. Killian, who was affiliated with the Plan 9 development group. See [T. J. Killian, "Processes as Files". In \*Proceedings of the 1984 Usenix Summer Conference\*, pp. 203 - 207. Salt Lake City, UT.](#)

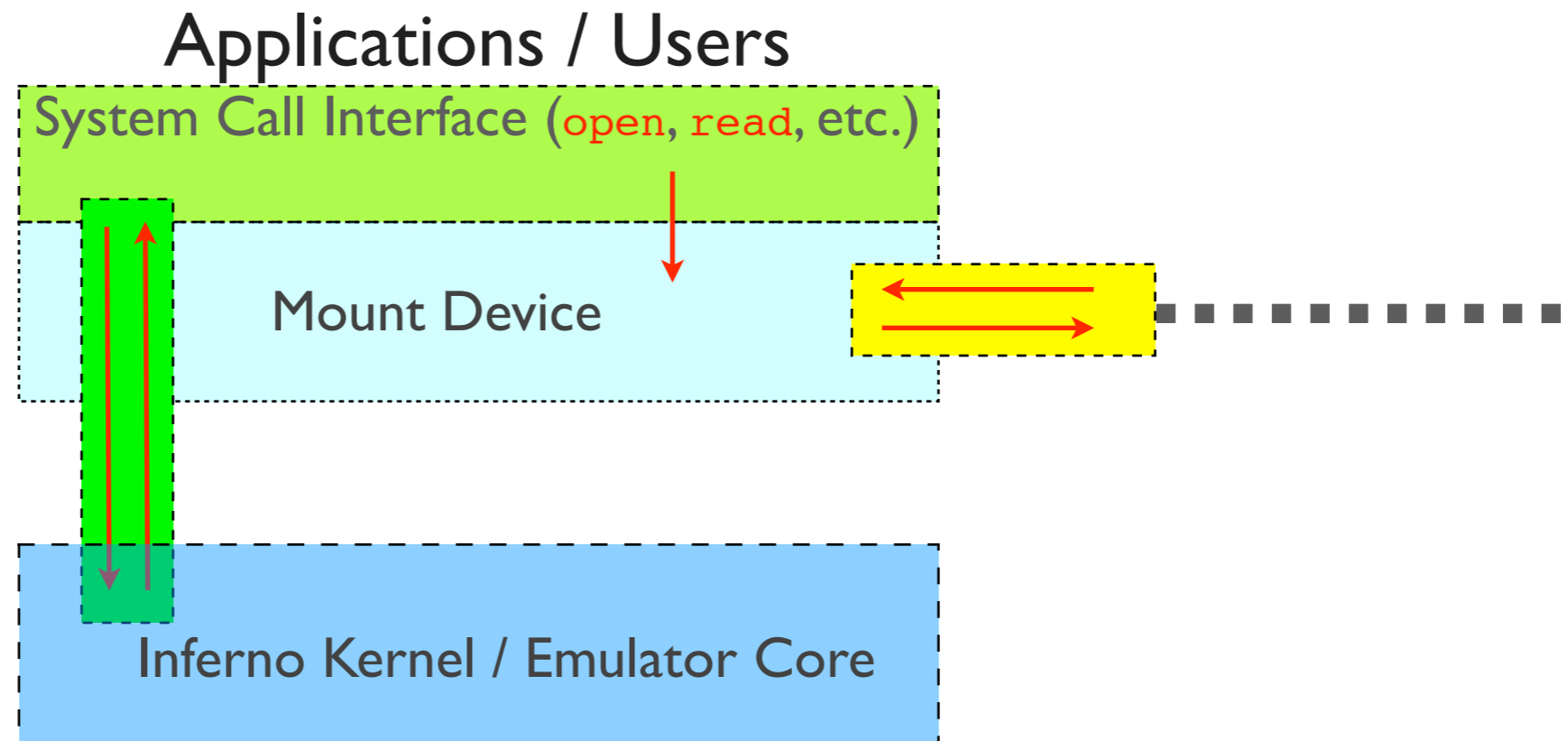
# Access *and* Control via Name Space

- Unix `/dev/` : Accessing device drivers via filesystem
  - *Device special files* created by `mknod()` system call, *linked to in-kernel device drivers*
  - *Properties* of driver serving device special file *manipulated by `ioctl()` syscall*
    - **Example:** Can write an archive to a tape drive by writing to `/dev/rst0`, but need to perform an `ioctl()` system call to write the *end-of-tape* mark
    - **Example:** Can play audio by writing PCM encoded audio data directly to `/dev/audio`, but can only change sample rate via `ioctl()`
- Inferno: No special syscalls for fiddling with devices
  - E.g., `/dev/audio` for audio data, `/dev/audioc1` for parameter control
  - `/net/tcp/clone` to allocate resources for a new TCP connection, `/net/tcp/n/` (*an entire per-connection directory of “synthetic files”, allocated when `/net/tcp/clone` is read*) for controlling connection and sending data

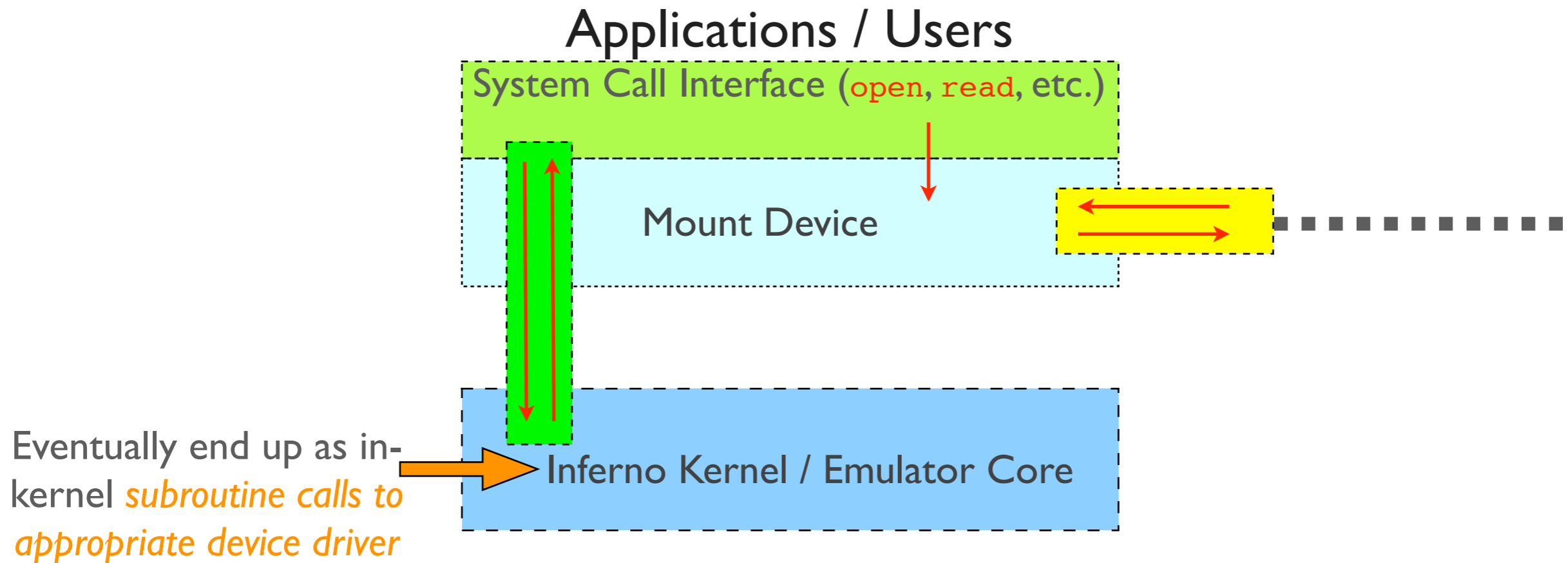
# Accessing Files (Names)

- What happens when files (names) are accessed ?
  - Operations on a single name: **open**, **read**, **write**
  - Traversing hierarchies of names

# Accessing Name Space Entries: The *Mount Device*, #M

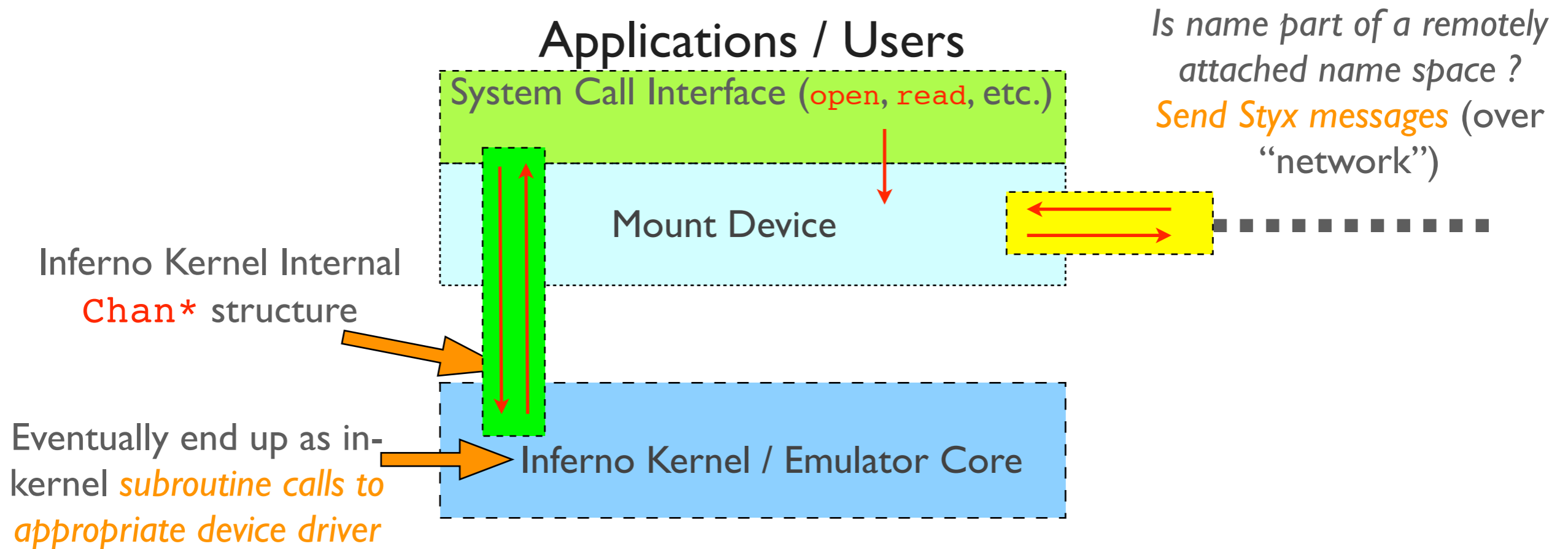


# Accessing Name Space Entries: The *Mount Device*, #M



- System *delivers file operations to appropriate local device driver via subroutine calls*

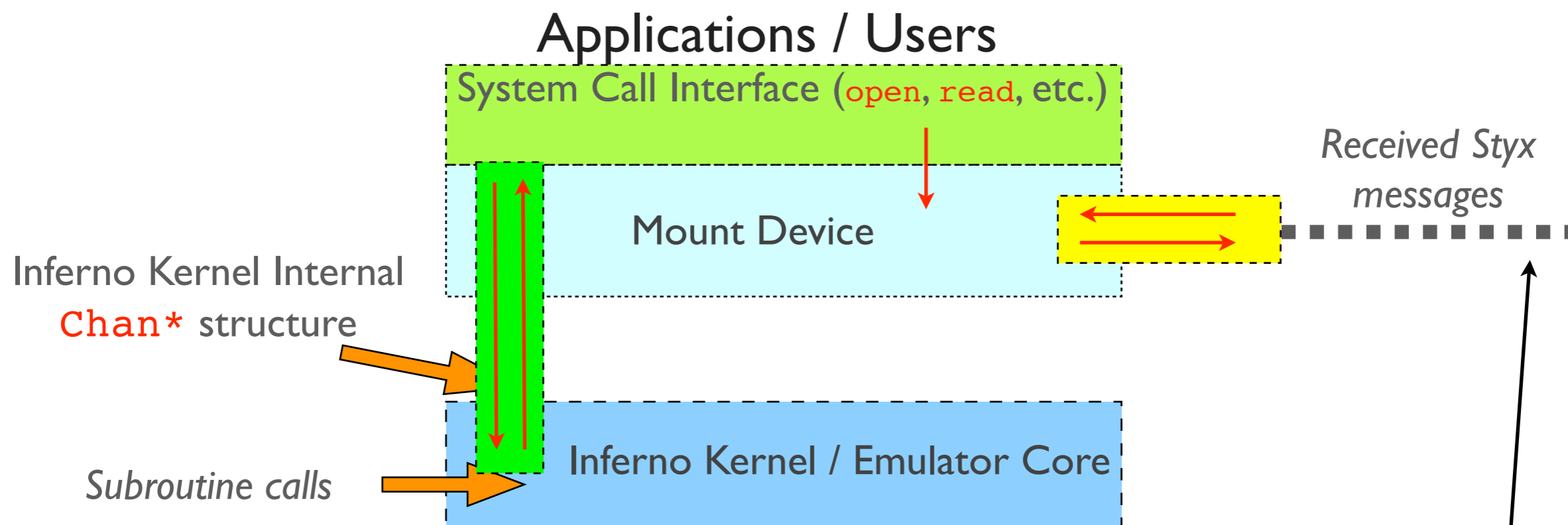
# Accessing Name Space Entries: The *Mount Device*, #M



- System *delivers file operations to appropriate local device driver via subroutine calls*
- If file being accessed is from an attached namespace, *deliver styx messages to remote machine's mount driver*



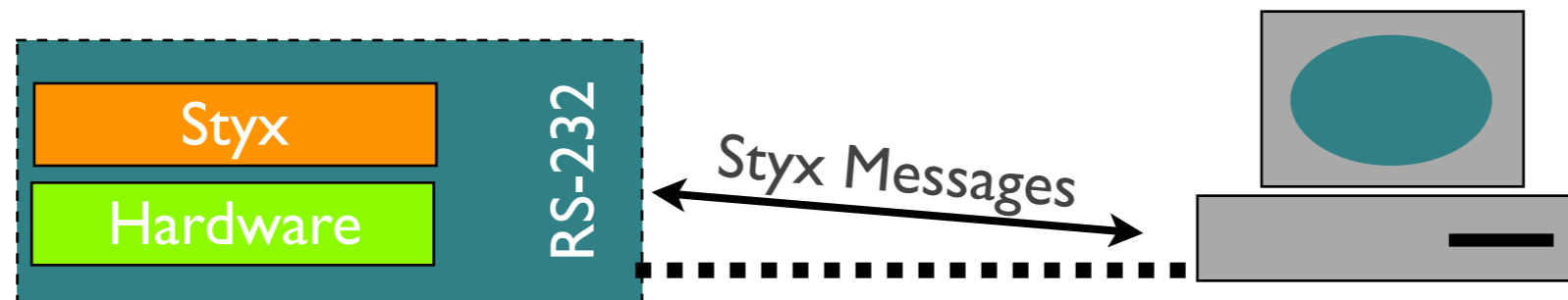
# Converting Styx messages to local subroutine calls



- Mount driver converts Styx messages coming in over the network into calls to local device drivers
- Any entity that can speak the Styx protocol can take advantage of system resources and hardware (subject to permissions / auth)
  - Makes distribution of resources in a network simple : one protocol, Styx

# Styx in a Nutshell

- 14 message types
  - Initiate connection (**Attach**)
  - Traversing hierarchy (**Clone**, **Walk**)
  - Access, creation, read, write, close, delete (**Open**, **Create**, **Read**, **Write**, **Clunk**, **Remove**)
  - Retrieve/set properties (**Stat**, **Wstat**)
  - Error (**Error**)
  - Interrupt operation (**Flush**)
  - No-op (**Nop**)
- Easy to implement on, say, an 8-bit microcontroller



This device can now access network protocol stack, process control, display device etc. of the connected workstation

*Real world example: Styx on Lego Rcx Brick (Hitachi H8 microcontroller, 32K RAM, 16K ROM)*

# Talk Outline

- Terminology, Overview and History
- Abstraction and Names: Resources as Names (files) in Inferno
- The Limbo Programming Language
- Multi-platform applications
- Ideas & Summary

# Programming in Limbo

- Limbo is a concurrent programming language
  - Language level support for **thread creation** and inter-thread **communication over typed channels**
  - Channels based on ideas from Hoare's *Communicating Sequential Processes* (**CSP**)

```
# Declare a variable that is a channel of integers
sync := chan of int;
# Create a new thread with a reference to this channel
spawn worker(sync);
# Read from the channel. Will block until a
# corresponding write is performed by the worker thread
v =<- sync;
```

- Some Limbo language 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)

# 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 definition)
  - 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

# Xsniff (I)

```
implement Xsniff;  
  
include "sys.m";  
include "draw.m";  
include "arg.m";  
include "xsniff.m";
```

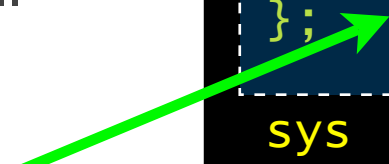
```
Xsniff : module  
{  
    DUMPBYTES : con 32;  
  
    init : fn(nil : ref Draw->Context, args : list of string);  
};
```

```
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 Module Definition

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



# Xsniff (2)

Read `/net/ether0/clone`  
Get back, e.g., the string "7"

Open `/net/ether0/7/data`

Open `/net/ether0/7/ctl`

Write config. commands into  
`/net/ether0/7/ctl`

Spawn new thread with ref  
to descriptor open on  
`/net/ether0/7/data`

```
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");
```

```
line := int (hd dirstr);
```

```
infd := sys->open(dev+sys->sprint("/%d/data", line),  
                sys->ORDWR);
```

```
sys->print("Sniffing on %s/%d...\n", dev, line);
```

```
tmpfd = sys->open(dev+sys->sprint("/%d/ctl", line),  
                sys->ORDWR);
```

```
# Get all packet types (put interface in promisc. mode)
```

```
sys->fprintf(tmpfd, "connect -1");
```

```
sys->fprintf(tmpfd, "promiscuous");
```

```
# Spawn new thread w/ ref to opened ethernet device
```

```
spawn reader(infd, args);
```

```
}
```

# Xsniff (3)

Read Ethernet frame from  
`/net/ether0/7/data`

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

```
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;
}
```



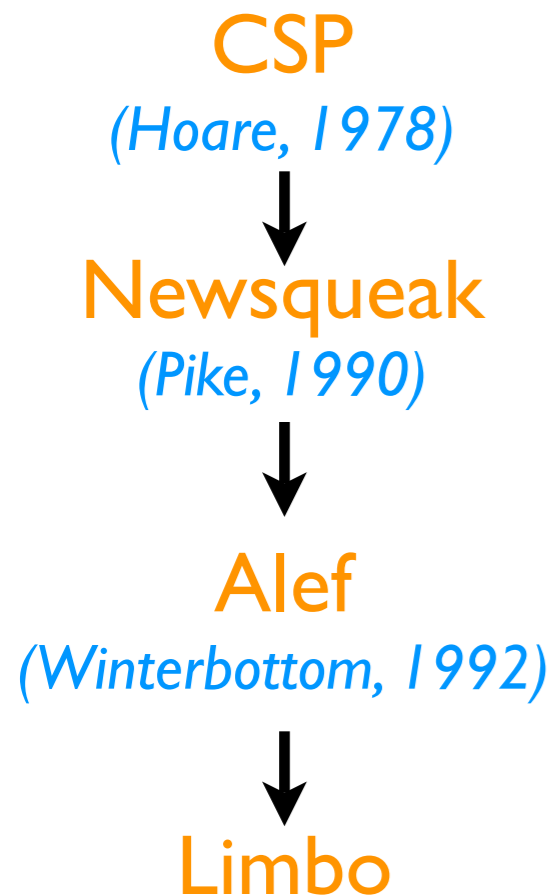
# Limbo Language Genealogy (*abridged*)

Channels

Module System

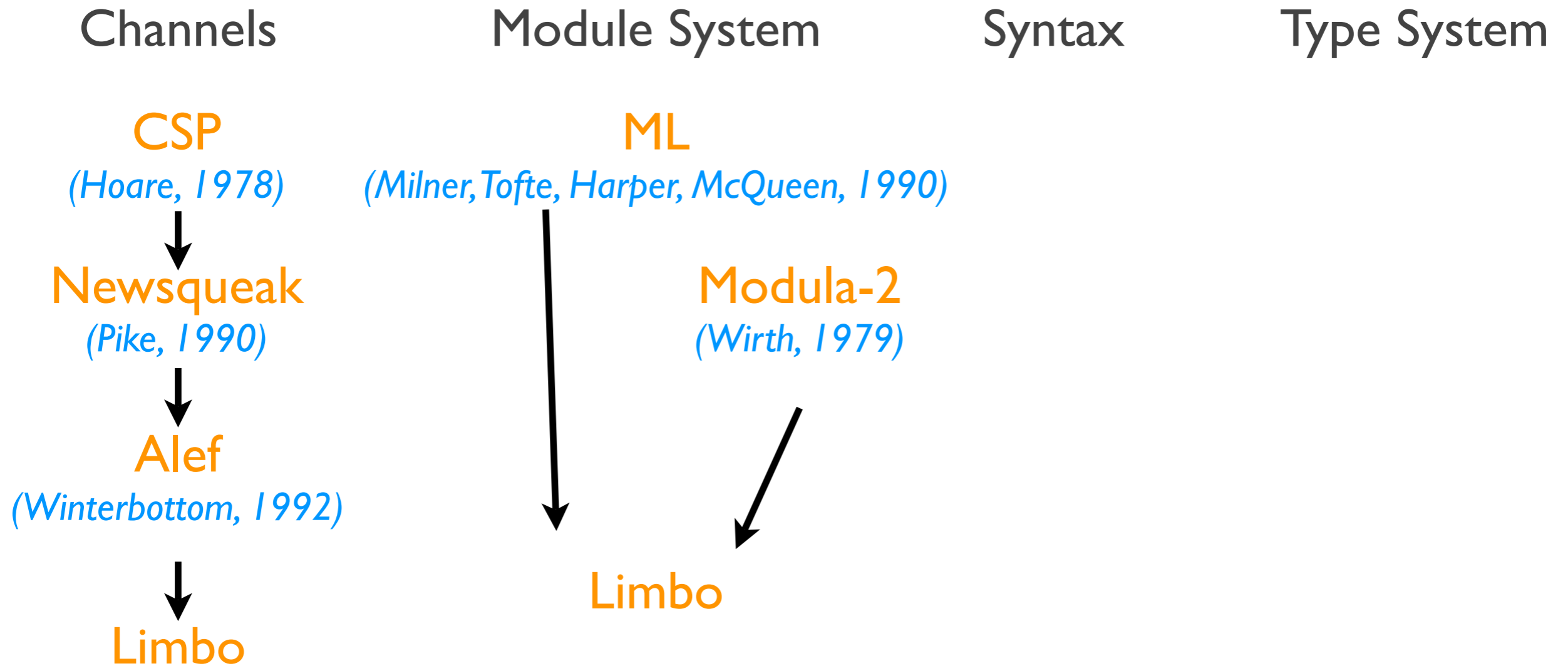
Syntax

Type System



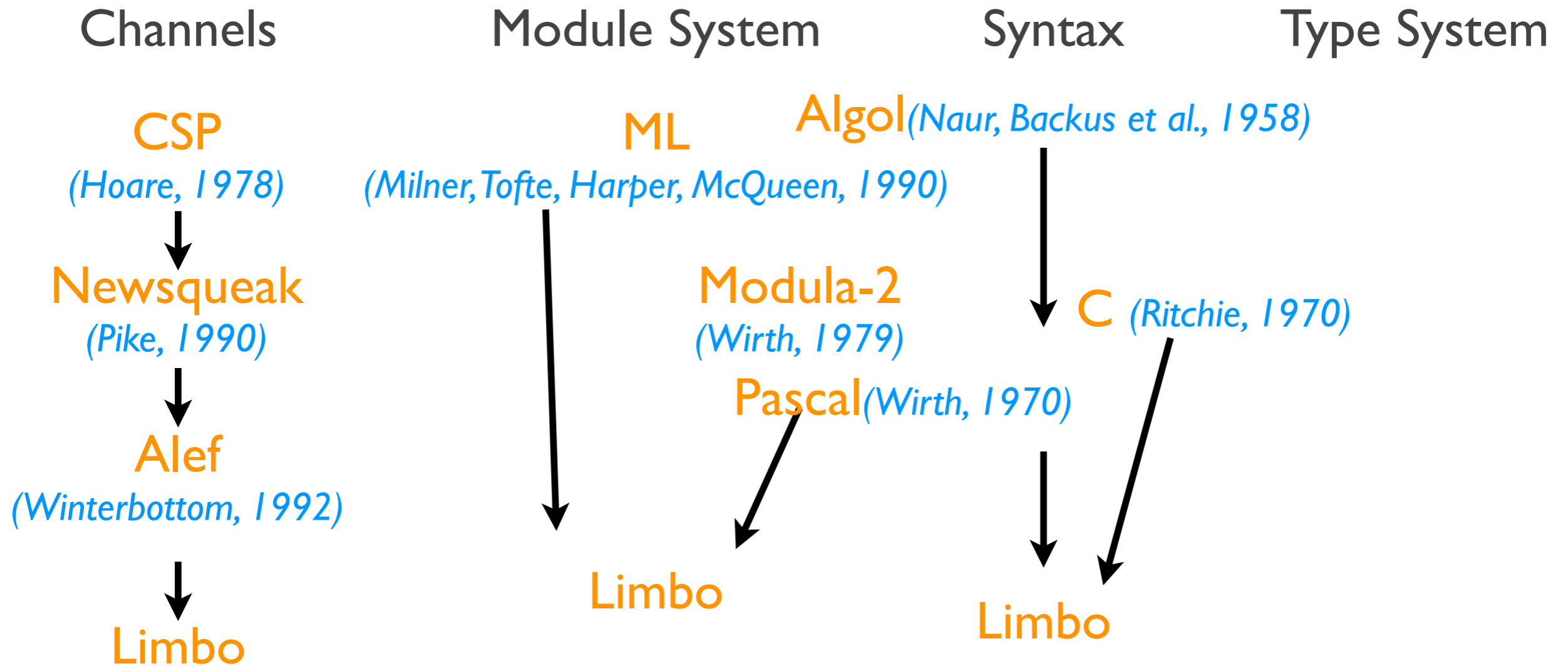
- Language-level “communication variables”, the channel data type, is influenced by *CSP*, via *Alef* and *Newsqueak*

# Limbo Language Genealogy (*abridged*)



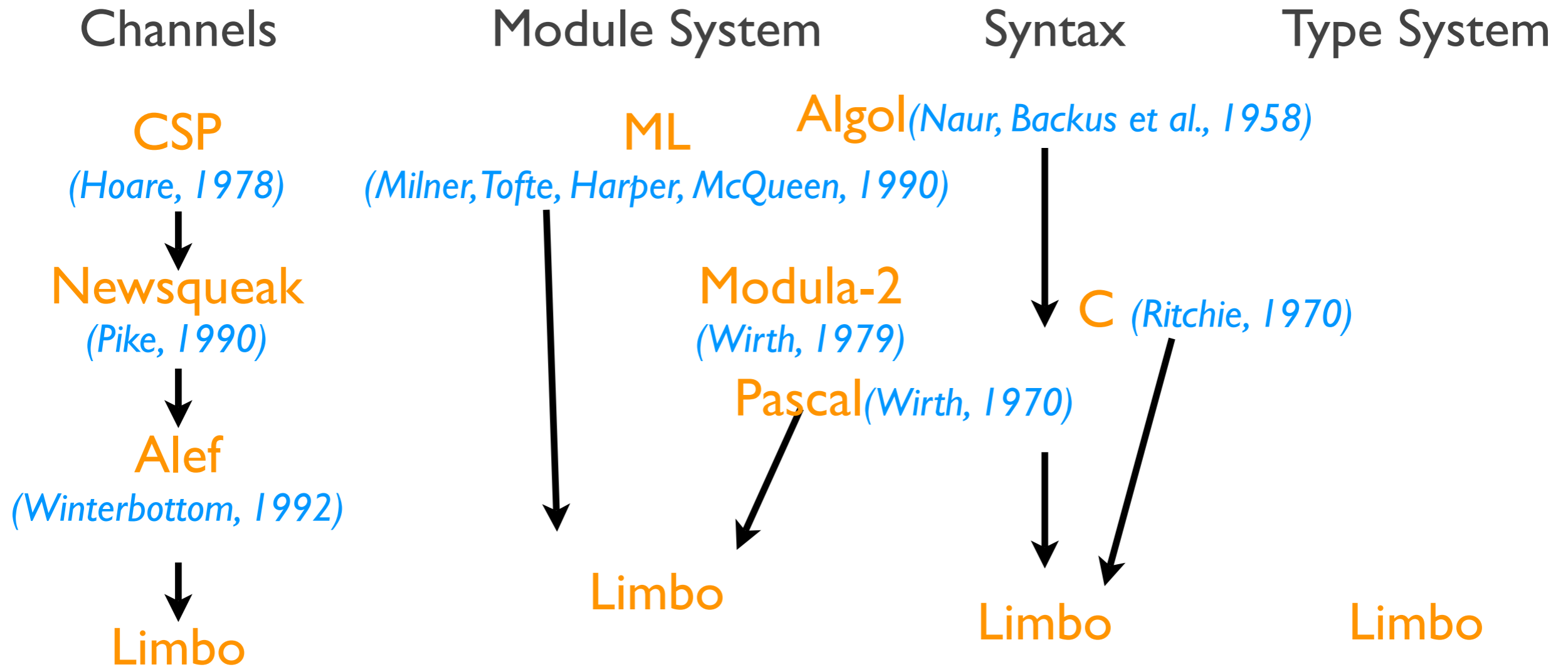
- Limbo's module system is influenced by *ML* and *Modula-2*

# Limbo Language Genealogy (*abridged*)



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

# Limbo Language Genealogy (*abridged*)



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

# Talk Outline

- Terminology, Overview and History
- Abstraction and Names: Resources as Names (files) in Inferno
- The Limbo Programming Language
- Multi-platform applications
- Ideas & Summary

# Distributed Applications

- Representing resources as files + simple access to remote name spaces = easy distribution of resources
  - Abstraction of resources as files, coupled with use of Styx protocol provides a platform-independent interface
- Abstraction of both HW via Dis VM and OS via emulator
  - Complete hardware and system independence of applications. No *if's*, no *but's*
  - **Not just a virtual machine : complete virtualized operating system environment**
- Multi-platform tools and build environment
  - Identical tools for compiling the emulator and native kernel across platforms
  - No need to get bogged down in details of, e.g., MSVC on Windows

# Distributed Applications

- Applications can be readily deployed on any host platform Inferno emulator runs on
- Can combine emulator and Limbo applications into a single binary image per platform
  - No need to make user install Inferno
  - Final binary contains emulator (Dis VM, runtime system), Limbo applications, root filesystem (as a ramdisk), < 1MB compressed for interesting apps
  - Java .jar files and .jnlp — you still have to install Java runtime, > 12MB compressed
- Doing the above is trivial using Inferno and Limbo
  - *The point is not just that you can do it, but rather that you can do it very easily*

# Case Study

- An architectural simulator which models multiple complete embedded systems
  - Regularly used in cycle-accurate simulation of 50+ devices interconnected in arbitrary topologies
  - Desired to simulate 1000+ devices : distribute simulation across multiple hosts
  - Make simulation engines available for multiple platforms
- To be implemented by 1 graduate student, in spare time
  - Simulation framework is “infrastructure”, not the end-goal of student’s research
  - Simulation of 1000’s of nodes makes it possible to model more realistic scenarios



# Case Study

- Simulation platform exploits MIMD<sup>1</sup> parallelism
- Graduate student exploits SSMB<sup>2</sup> parallelism
  - Simulation engine ported as a device driver (# $\mu$ ) in Inferno Emulator
  - User Interface for simulation core implemented in Limbo
  - Generation of stand-alone executables for multiple platforms (Mac OS X, Linux, OpenBSD, Windows, Windows IE plugin) with  $\sim$ 1 man-hour of work

<sup>1</sup> Flynn classification for parallel processors: Multiple Instruction/Multiple Data (e.g. NOW)

<sup>2</sup> Coined for the purpose of this talk: Single Stone Multiple Bird Parallelism

# Implementation

- Simulation engine implemented in ANSI C
  - Models Hitachi SH3 processor, network interface, battery cell's electrochemical characteristics, DC-DC converters, communication network, node/link failures, more... *(details not discussed in this talk)*
  - Compiled as a library that emulator links against
- Device driver interface in Inferno emulator makes simulation engine visible as a dynamic hierarchy of files
  - <500 lines of C code for the dynamic filesystem interface
  - Device driver calls upon facilities of simulation core library
  - Driver is shielded from all arch-specific details by Inferno emulator implementation

# Dynamic Filesystem Interface

- Dynamic hierarchy, one directory per simulated processor
  - Files in each numbered directory provide access to node info / control
  - **ctl**: used to control global simulation parameters, create new nodes, etc.
  - **info**: read to obtain simulation-wide output
  - **netin, netout** : connected across hosts to connect simulated networks
- Complete simulation control via filesystem
  - *Make this interface visible over network...*

/dev/myrmigki/

**/ctl**

**/info**

**/netin**

**/netout**

/0/

/ctl

/stderr

/stdin

/stdout

/info

/1/

/ctl

/stderr

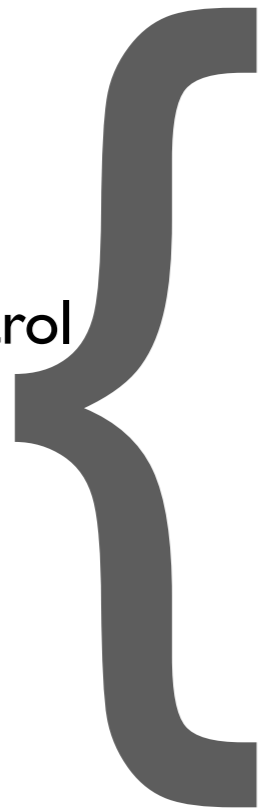
/stdin

/stdout

/info

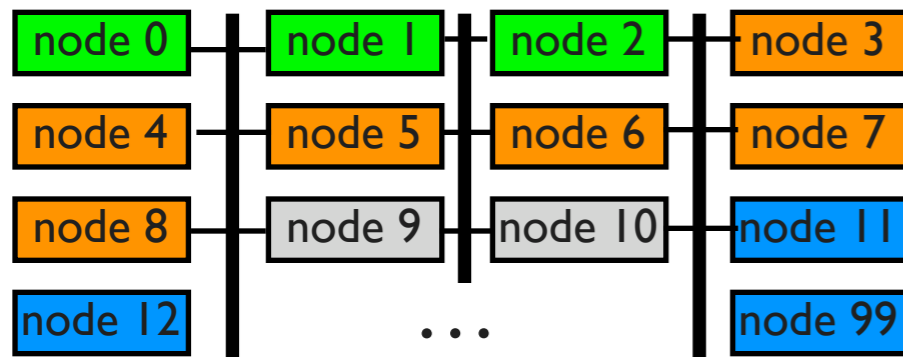
...

Per-node control  
and output

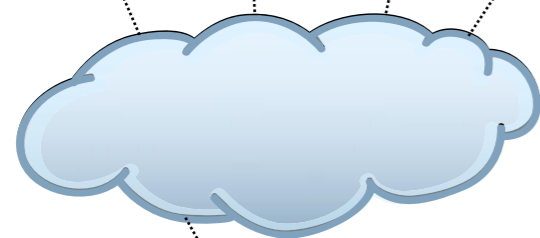
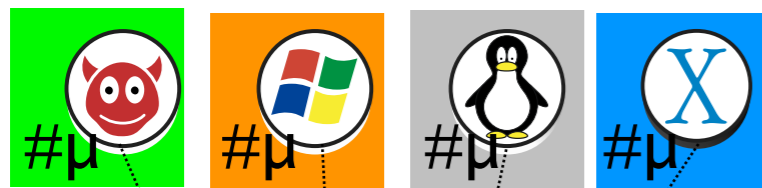


# Gluing Together Multiple Engines

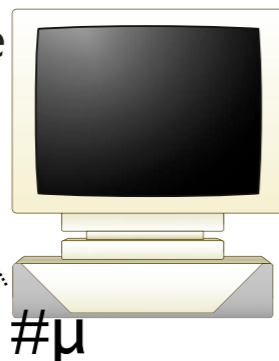
Simulated system w/ 100 nodes  
(processor+battery, NIC) and network



Simulation Hosts



Simulation Host  
/ Control Glue  
and GUI

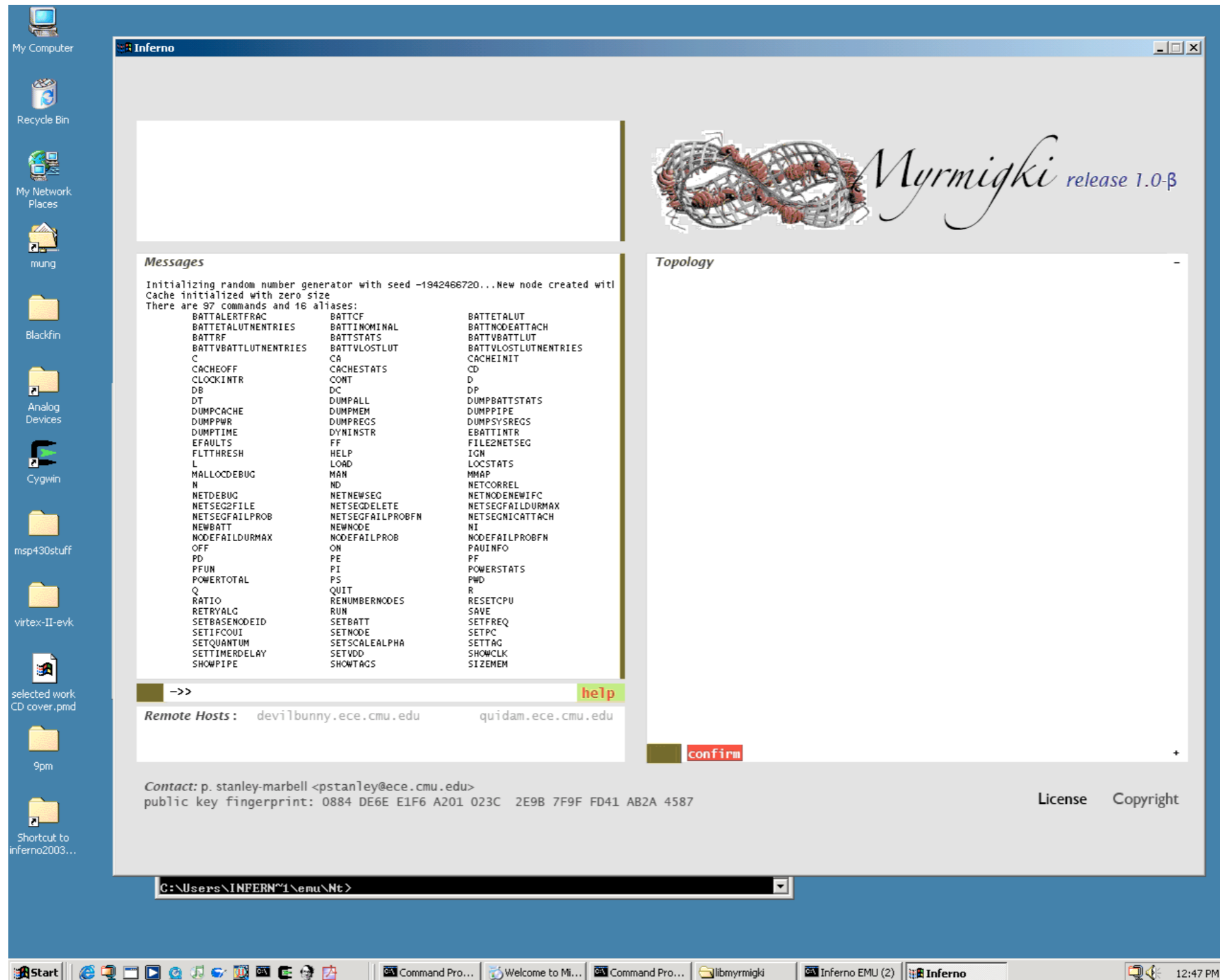


- Join multiple simulation engines together
  - Communication between nodes simulated on different hosts glued together using the **netin** and **netout** interfaces in filesystem
  - A Limbo thread sets up the shuttling of data, **(netin,netout)<->(netout,netin)**
  - User interfaces defines which virtual simulated nodes are mapped to which real simulation hosts
  - Simulation hosts are any platform that Inferno emulator runs on, or a dedicated host running an Inferno kernel

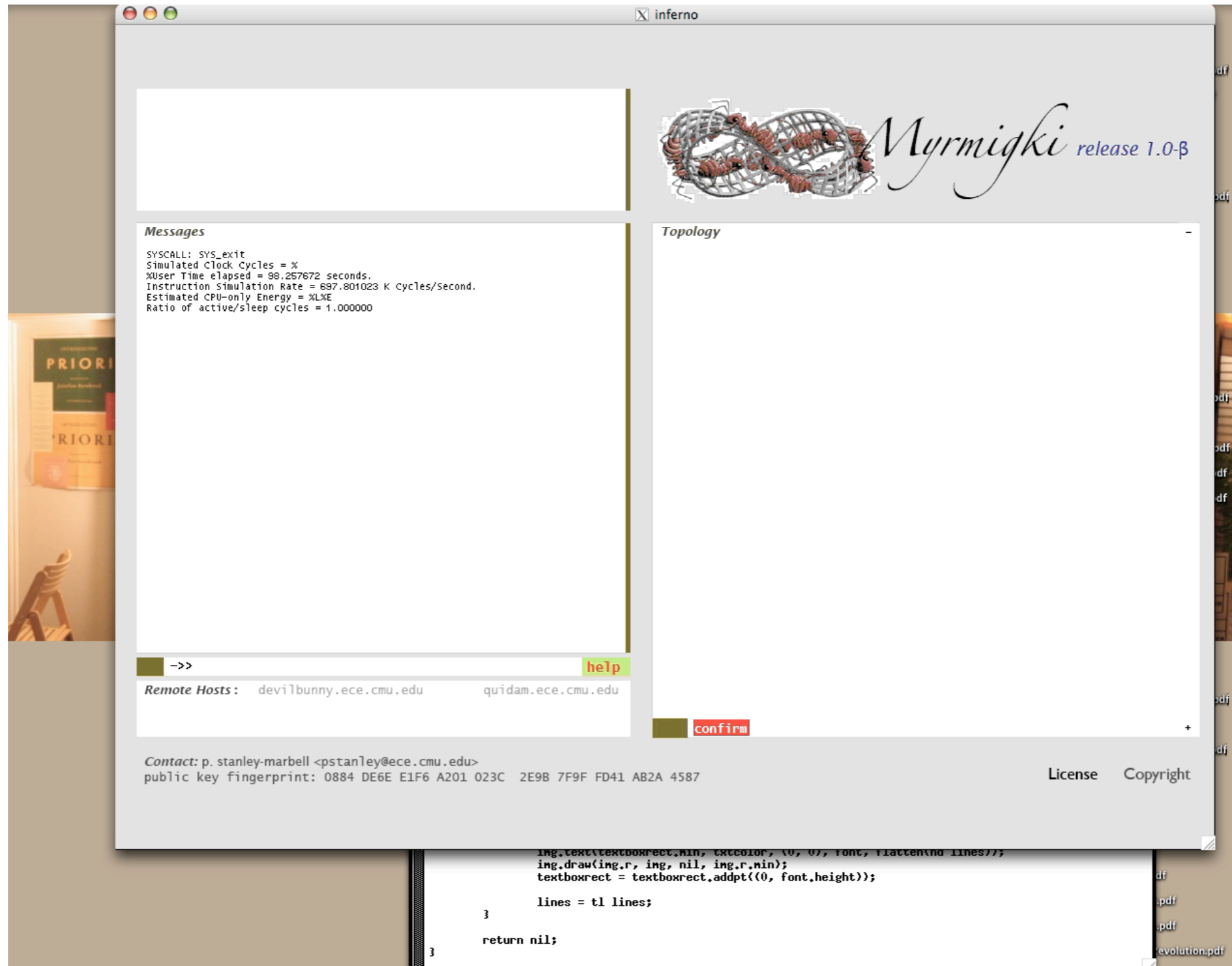
# Implementation: Driver Interface

- Advantages of name (file) interface to simulation engine
  - Uniform platform-independent interface to each simulation engine
  - Any entity that can speak Styx can interact with a simulation engine
  - User interface can easily be attached to a local or remote engine
  - All of this is inherently platform-independent

# Implementation — GUI on Windows



# Implementation — GUI on Max OS X



# Talk Outline

- Terminology, Overview and History
- Abstraction and Names: Resources as Names (files) in Inferno
- The Limbo Programming Language
- Multi-platform applications
- Ideas & Summary



# Ideas

- Representing resources as files makes possible **uniformity in access across platforms**
  - Interfaces to programs as entries in name space
  - Process creation and control via name space, access to network stack, etc.
- Entries in name space still however have **structure dating back to Multics**
  - Entries in name space do not have “types” in the sense of types in programming languages
  - File attributes (name, uid, gid, atime, mtime, ...) are certainly just an aggregate type (`struct stat` in Unix or `Dir` in Inferno)
  - **Could you make data structures within programs visible in name space and tie together programs across the network ?**

# Ideas

- What if you could do:

```
# Channel variable x has type defined by an aggregate type, Complex
x := chan of Complex;
# Make channel x visible in name space. Entry will have type Complex
chan2name x "/tmp/x";
...
# Define channel var with type extracted from "/tmp/x" and connected to it:
a := chan of name2type "/tmp/x";
```

- Writes to channel **a** are now visible on channel **x**
- Channel variable **a** is connected to **x** through name space. **/tmp/x** could be on a remote host...
- This is an underlying idea in the M language [NSC-2, 2003]
  - Also being implemented as an extension to the Limbo runtime for didactic purposes in **98-023** (A CMU StuCo class I'm teaching)

# Summary

- Inferno: Virtualizes both hardware and operating system facilities
- Limbo: A concurrent programming language
  - Language level channels in the spirit of Hoare's CSP
  - Threads are cheap, go well with channels
- Inferno, Limbo and their development tools make it easy to build cross-platform distributed applications
  - *Not so much what they make possible, but rather the ease with which they make it so*
- A particularly attractive code base for systems research
  - Source to entire system available at no cost under a "liberal" source license
  - CMU StuCo 98-023 *Concurrent and Distributed Programming with Inferno and Limbo*

# For more information:

- An Inferno/Limbo Mailing List
  - <http://lists.gemusehaken.org/mailman/listinfo/inferno>
- Public software repository
  - <http://www.gemusehaken.org/ipw1/sourcecode>
- Other resources
  - Public Certificate Authority
  - Public CPU/resource servers (You will need to obtain certificates from above to use these.)

# Inferno Programming with Limbo

Phillip Stanley-Marbell

WILEY



## Inferno Programming with Limbo

Phillip Stanley-Marbell

*Inferno Programming with Limbo* is the first complete developer's guide to programming for the Inferno operating system. Developed at Lucent's Bell Labs, Inferno enables cross-platform, portable, distributed application development that is well suited for networked applications on resource constrained, embedded systems. Limbo is its programming language.

This book will provide you with an introduction to Inferno, and everything you need to know about building applications with Limbo.

The book focuses on the pragmatic aspects of developing Inferno applications with the Limbo language. It includes complete source code for several application examples, ranging from a text editor, file servers and network servers, to graphical applications such as games. Common programming pitfalls are revealed and in-depth analysis of complete sample applications are given.

Also covered in the text are sections on:

- accessing Inferno system facilities from Limbo programs
- building multi-threaded applications with Limbo
- implementing user level file servers in Limbo
- networking in Inferno and constructing networked applications in Limbo
- graphical applications in Inferno
- augmenting Limbo applications with modules written in the C programming language
- cryptographic facilities provided by Inferno
- tools for verification of concurrent multi-threaded programs, such as model checkers
- relevant manual pages and Limbo module definitions

**Phillip Stanley-Marbell** is a Ph.D. student at Carnegie Mellon University, and maintains the Inferno/Limbo FAQ. He has been an Inferno user since its original release, and has worked on two commercial products that used Inferno.

For further information, please visit:  
<http://www.wileyurope.com>

WILEY  
wiley.com



amazon.com.

**Better Together**

Buy this book with [The Art of UNIX Programming](#) by Eric S. Raymond (Author) today!



(Amazon is bundling it with TAOUP. Not shameless self-promotion if I'm promoting someone else's book too ? :)

# Backup slides

# Connecting to remote systems: the *mount(1)* utility

- Connect to remote system, attach (*union*) their filesystem name space to local name space
- Manner in which union happens is determined by flags
  - -b (**MBEFORE** flag in Limbo module version)
  - -a (**MAFTER** flag in Limbo module version)
  - -c (**MCREATE** in Limbo module version)
  - Also, whether or not to authenticate connection, **-A** (*Mount uses a previously saved certificate in authentication, which must have been previously obtained from a certificate authority*)

# 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



# 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.dls** 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");
}
```

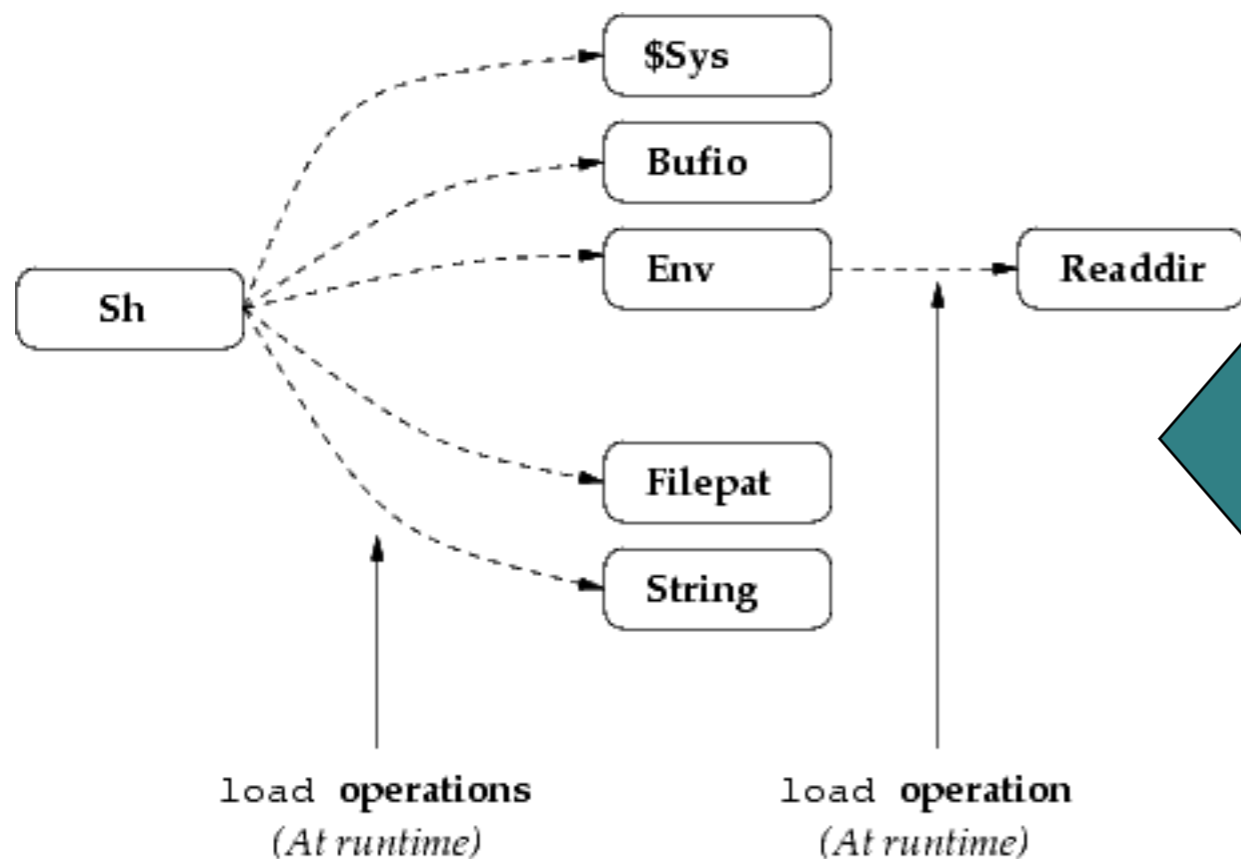
# Hello World

```
implement HelloWorld; ← Module Name
include "sys.m";
include "draw.m"; ← Various Includes
sys: Sys; ← Module Type (interface) Definition
HelloWorld: module ← Module Type (interface) Definition
{
    init: fn(ctxt: ref Draw->Context, args: list of string);
}
init(ctxt: ref Draw->Context, args: list of string) ← Module Implementation
{
    sys = load Sys Sys->PATH;
    # This is a comment
    sys->print("Hello!\n");
}
```

- 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
  - *Ivalue* of a **load** statement must match this type

# Dynamic Loading of Modules

- Module type information is statically fixed in caller module, but the actual implementation loaded at runtime 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**)

# Channels

- Channels are communication paths between threads
- Declared as `chan of <any data type>`
  - `mychan : chan of int;`
  - `somechan : chan of (int, string, chan of MyAdt);`
- Synchronous (blocking/rendezvous) communication between threads
- Channel operations
  - Send : `mychan <-= 5;`
  - Receive : `myadt = <- somechan;`
  - Alternate (monitor multiple channels for the capability to send or receive)

# Example : Snooping on Styx

- *Interloper* (ipwl book, pg. 192) is a simple program that lets you observe Styx messages/local procedure calls generated by name space operations

```
; interloper
Message type [Tattach] length [61] from MOUNT --> EXPORT
Message type [Rattach] length [13] from EXPORT --> MOUNT
; cd /n/remote
; pwd
Message type [Tclone] length [7] from MOUNT --> EXPORT
Message type [Rclone] length [5] from EXPORT --> MOUNT
Message type [Tstat] length [5] from MOUNT --> EXPORT
Message type [Rstat] length [121] from EXPORT --> MOUNT
Message type [Tclunk] length [5] from MOUNT --> EXPORT
Message type [Rclunk] length [5] from EXPORT --> MOUNT
/n/#/
;
```