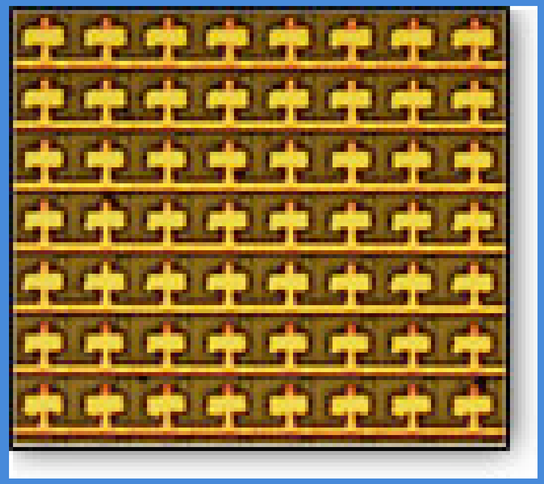
Pervasive Computing with Inferno and Limbo

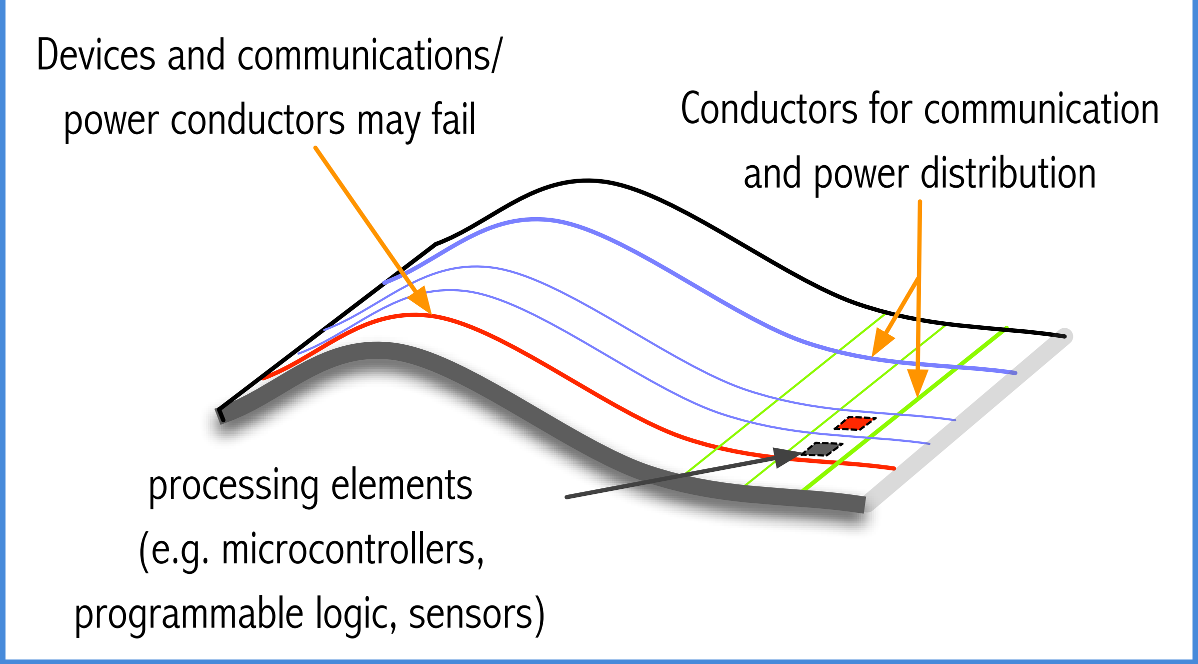
Phillip Stanley-Marbell
Dept. of ECE, Carnegie Mellon
http://www.ece.cmu.edu/~pstanley

Introduction

- What I do
  - Systems, programming languages, and analysis techniques for regular substrates with 1000's of failure-prone, energy-constrained devices per m^2



[Image courtesy Xerox PARC Large-Area Electronics / Large-Area MEMS]



  - This is work done under the direction of my research advisor, Diana Marculescu
  - Energy-Aware Computing Research Group `http://www.ece.cmu.edu/~enyac`
- This talk is about not about that (unfortunately...)

Talk Outline

- Inferno Overview
- Abstraction and *resources as files* in Inferno
- The Limbo programming language
- Pervasive computing with Inferno and Limbo
- Summary

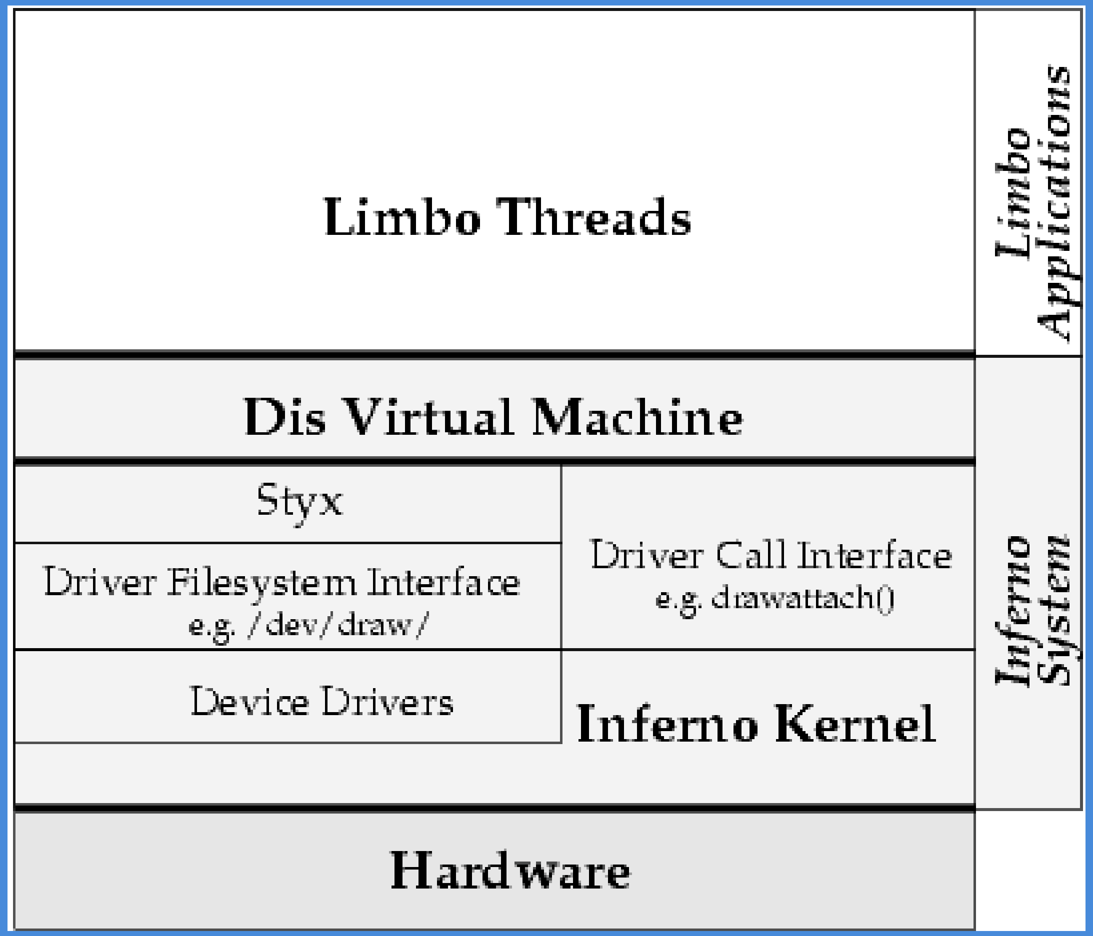
Overview

- Inferno
  - An operating system for networked devices
- Limbo
  - A programming language for developing applications for Inferno
  - There is (was) also support for running Java programs
- Dis
  - Inferno abstracts away the hardware with a virtual machine, the Dis VM
  - The VM and programming language cooperate to provide safety

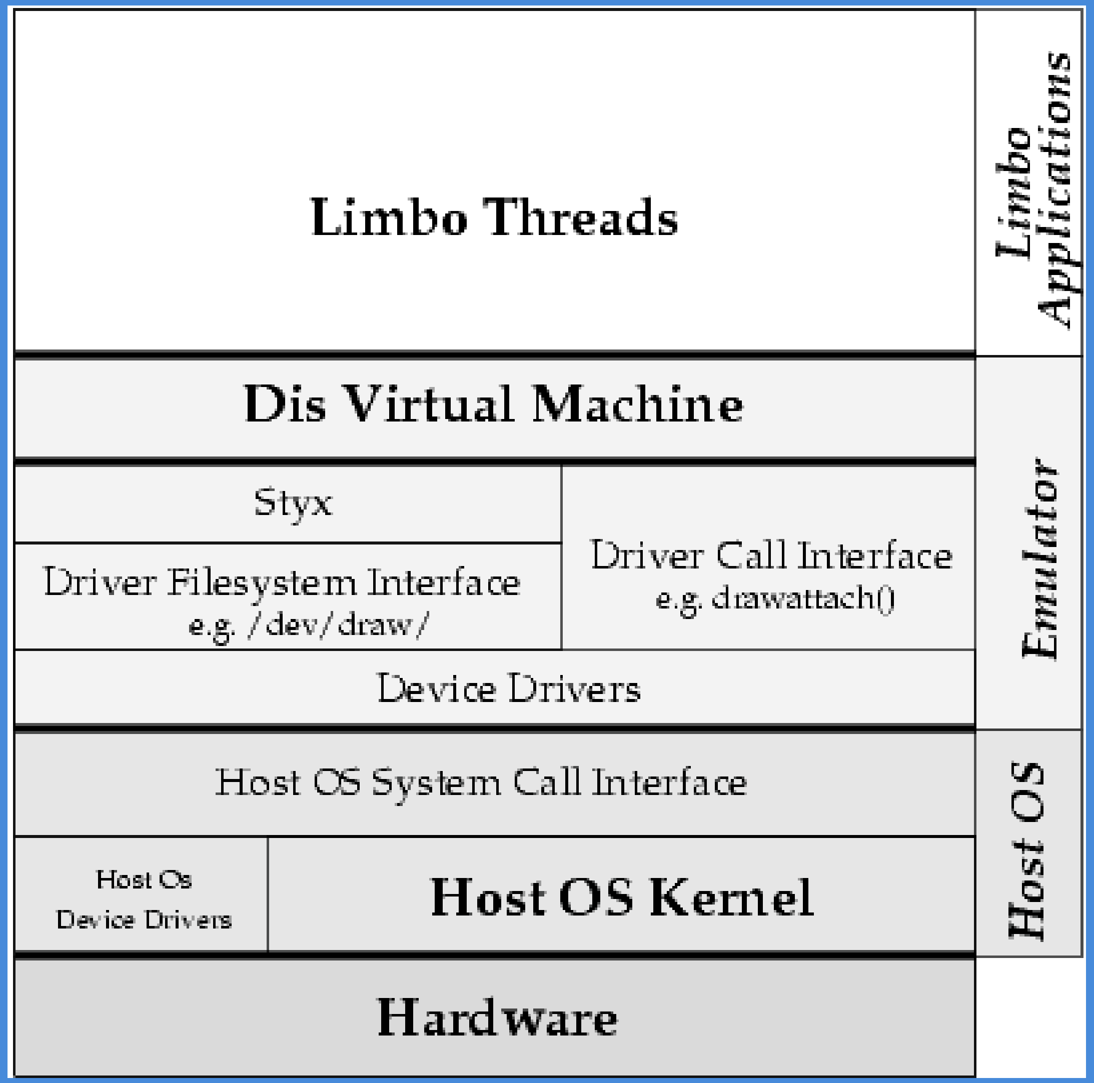
Inferno

- Inferno runs directly over bare hardware (PowerPC, Intel x86, SPARC, MIPS, ARM, more...)
- Also available as an *emulator* which runs over many modern operating systems (Windows, Linux, *BSD, Solaris, IRIX, MacOS X)
- 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, not just hardware

Native and Hosted Environments

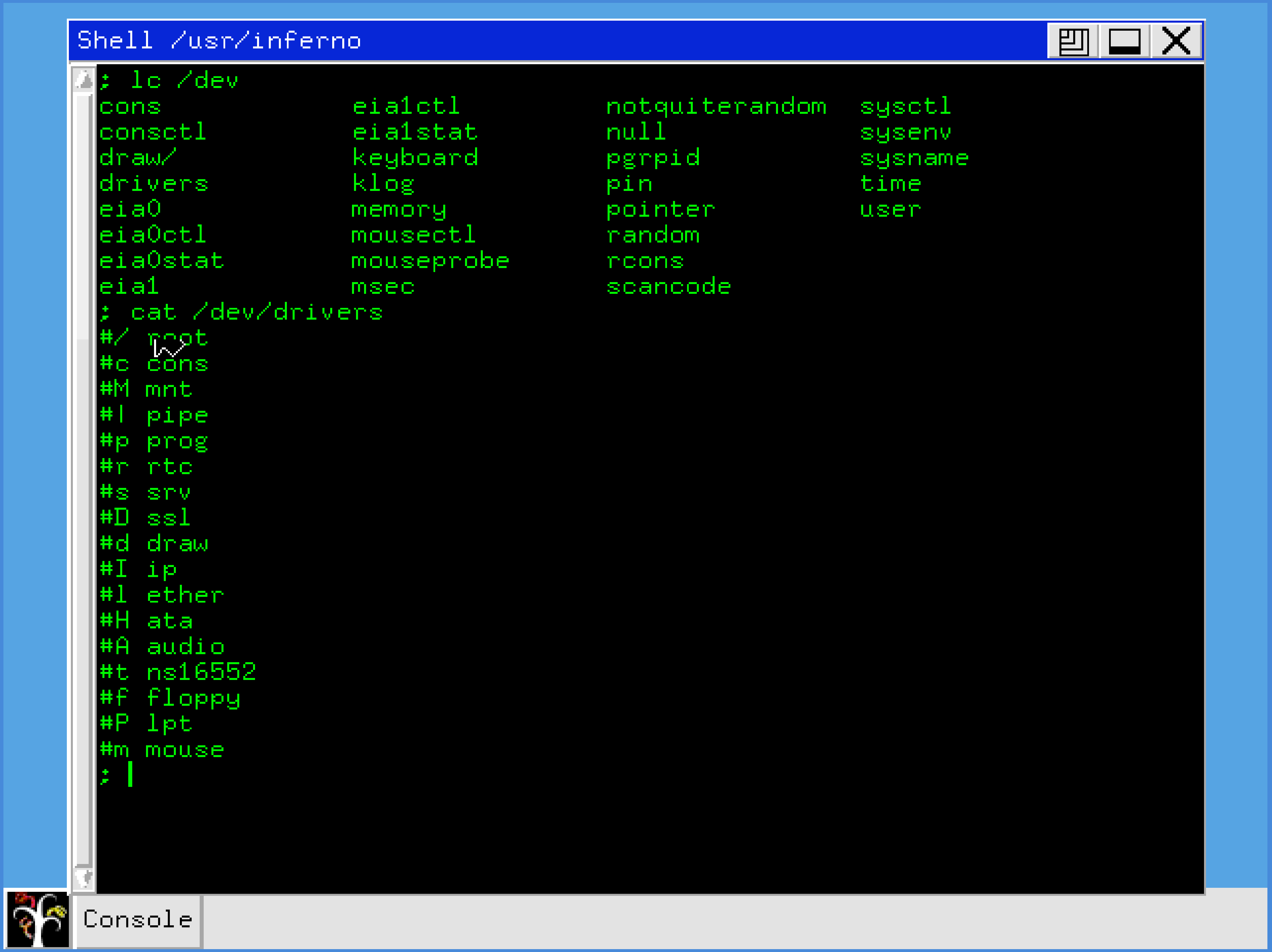


Native

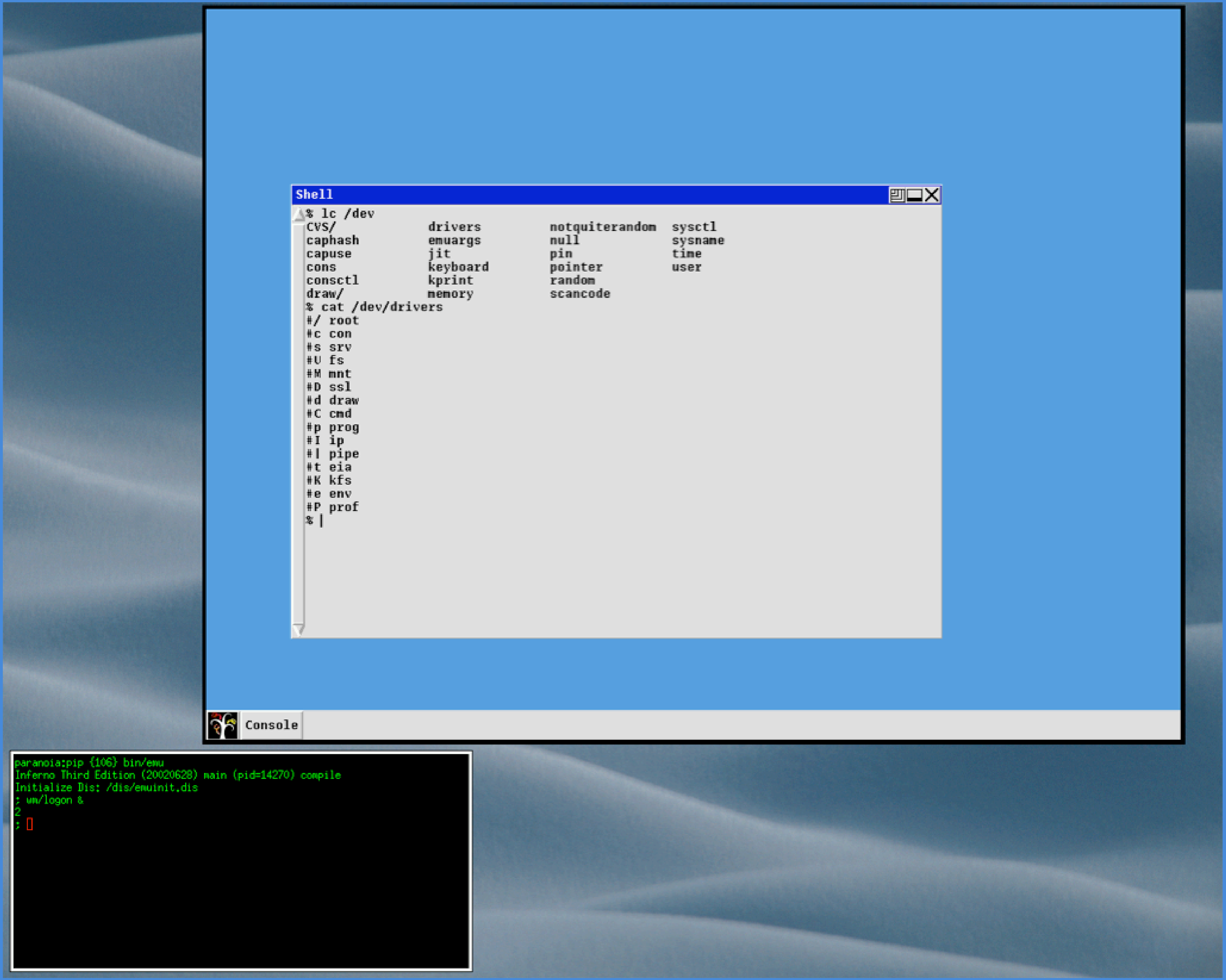


Hosted

Native Inferno Screenshot



Inferno Emulator on OpenBSD



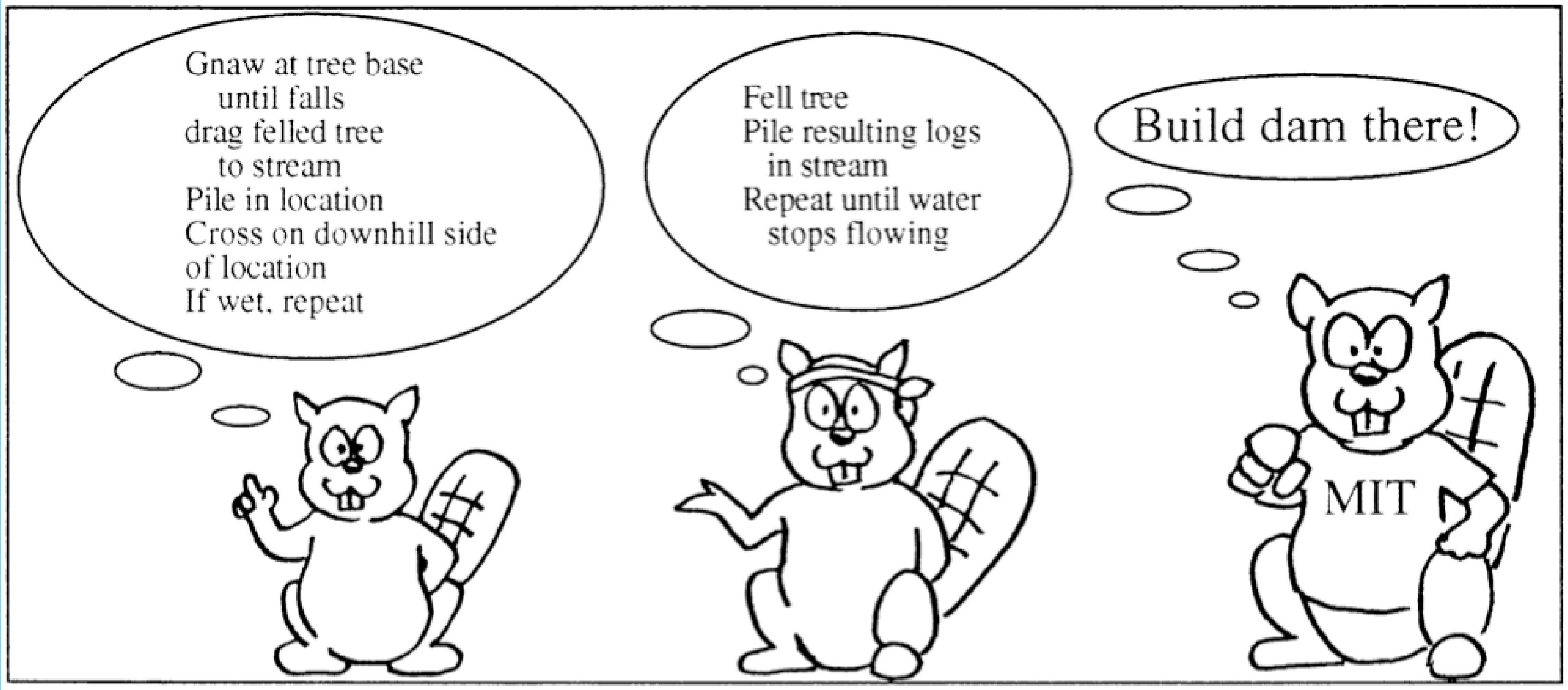
Available Software

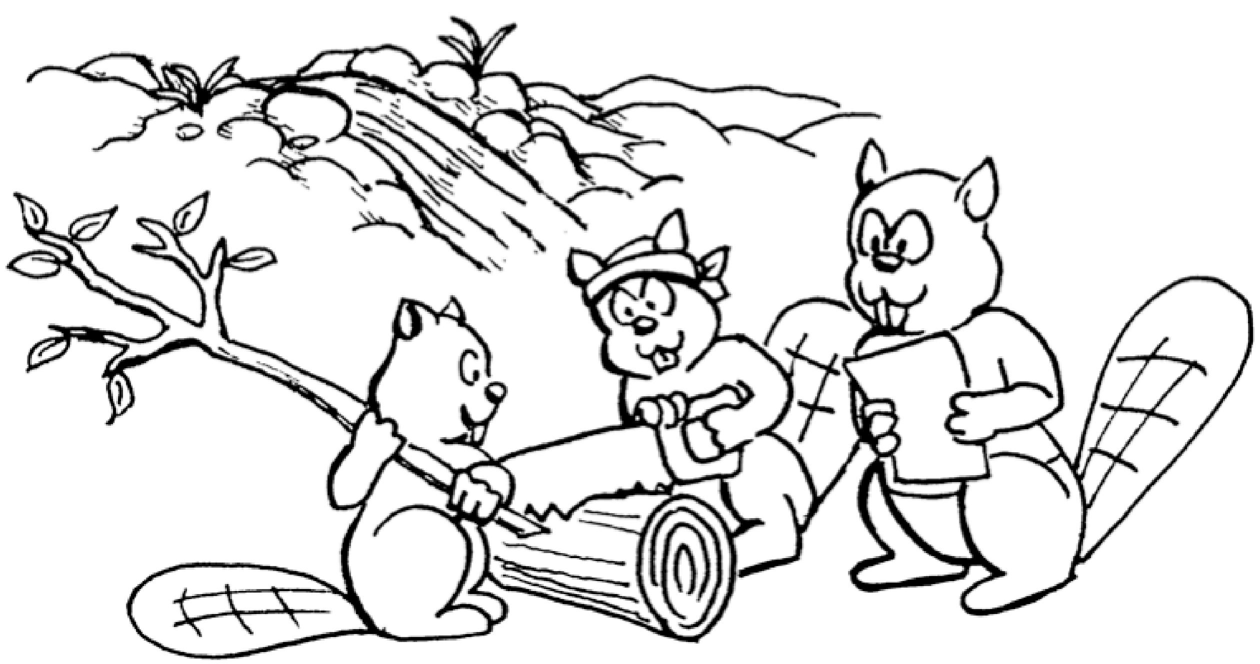
- Text/SGML editors
- Web browser, WML browser, Mail Client
- Graphical debugger
- Games
- Grid computing tools
- Clones of Unix tools (`sed`, `banner`, etc.)
- Other (not part of the distribution)
  - Audio editor / sequencer / synthesis
  - Image manipulation tools

Talk Outline

- Inferno Overview
- Abstraction and *resources as files* in Inferno
- The Limbo programming language
- Pervasive computing with Inferno and Limbo
- Summary

Abstract Up





Compile Down

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 systems
- 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

Files = Names

- Can think of files as names with special properties
  - Size
  - Access permissions
  - State (creation/modification/access time)
  - These properties are largely a historical vestige — we could imagine files with more sophisticated ‘types’

- Files are just an abstraction
  - There’s nothing inherently tying files (names) to bytes on disk
  - Association with disk files just happens to be most common use

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 ?
  - Graphics ?
- This interface/abstraction is not inherently more expensive than, say, a system call interface
- If we had a simple protocol for accessing files (names) over network, we could build interesting distributed/ pervasive applications...

Inferno : Resources as files

- Builds on the ideas developed in the Plan 9 Operating System
  - Most system resources are represented as names in a hierarchical name space
  - Single, simple protocol (Styx) for accessing these names, whether local or over network
  - These names provide abstraction for resources (such as those available in other systems via system calls)
    - Graphics
    - Networking
    - Process control

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

```
; mount net!www.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
       9        9        pip    release    13K Virgild[$Sys]
      10        7        pip    release     9K Server[$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

Access *and* Control via Name Space

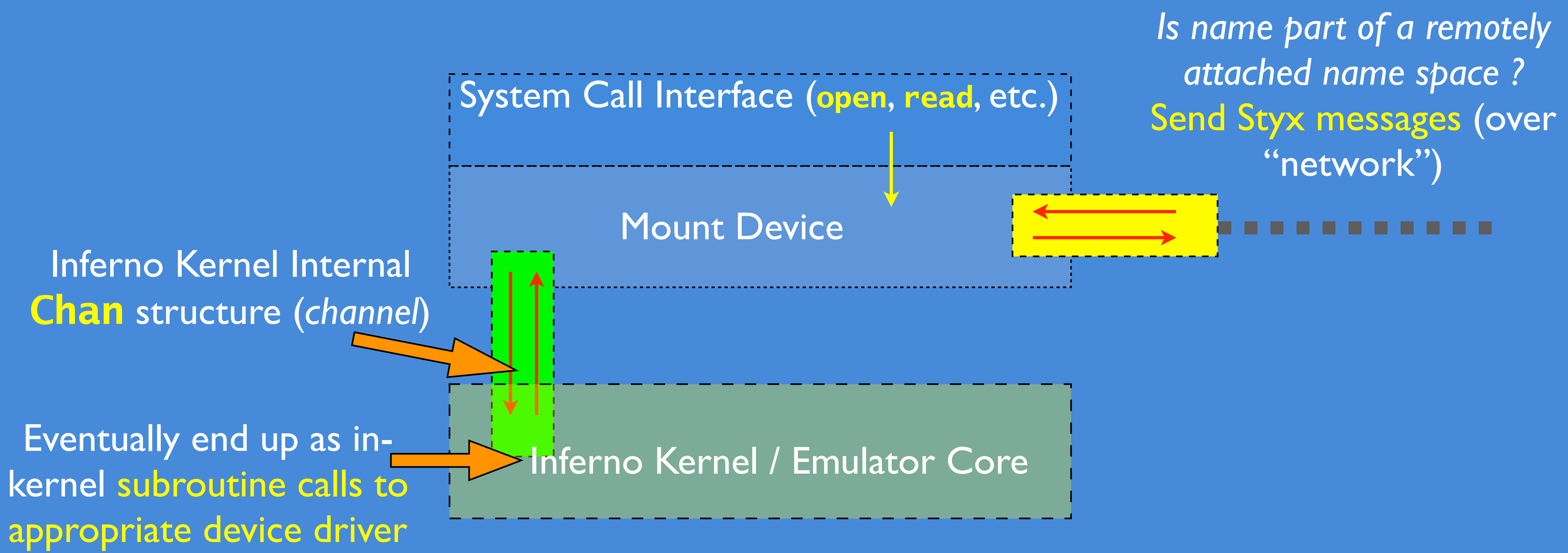
- Files used for both resource access *and* control

- Contrast this to Unix `/dev/`
  - Do entries in `/dev/` have the same semantics as ordinary files ?
  - Why can't you access `/dev/` over, say, NFS ?
  - What about `ioctl()` for controlling devices ? Why is device *access* via filesystem but device *control* via system call ?!

Accessing Names

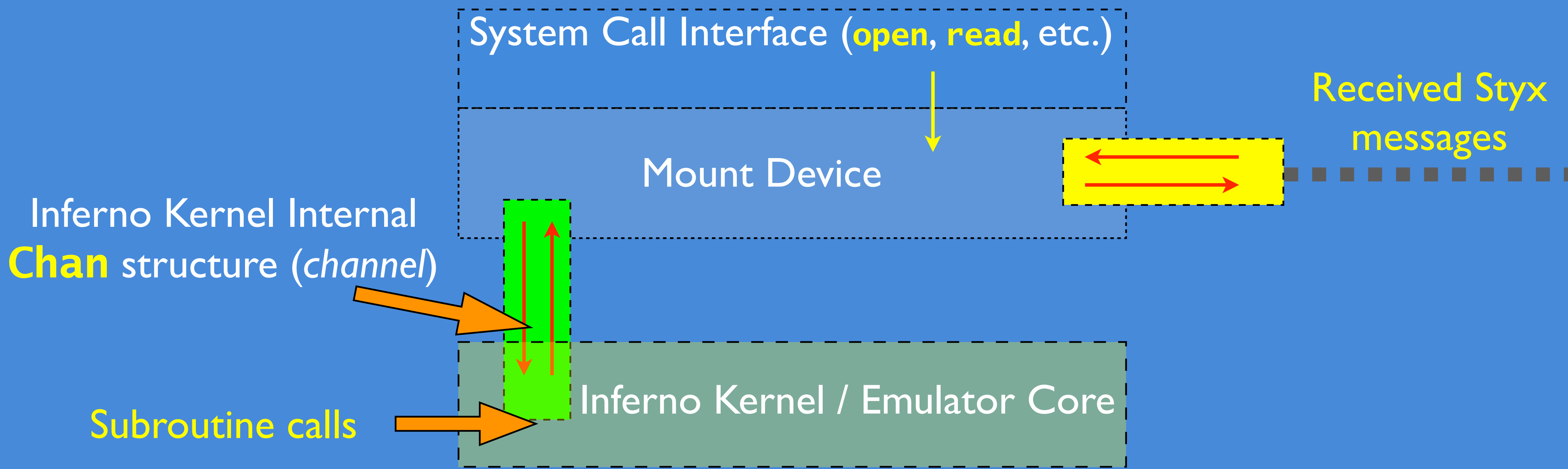
- What happens when names are accessed ?
  - Operations on a single name: open, read, write
  - Traversing hierarchies of names
- Styx Protocol
  - A simple protocol used as the underlying method for accessing names
  - Seen as subroutine calls when accessing local resources
  - Programmers usually do not deal with Styx directly

Accessing Name Space Entries: The *Mount Device*



- Mount device 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

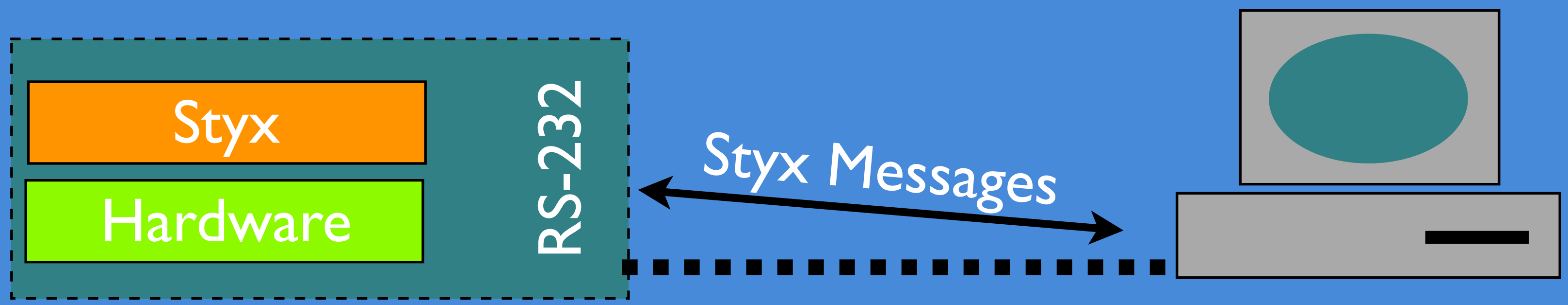


- Mount driver also converts Styx messages coming in over the network into calls to local device drivers
- Any entity that can speak Styx protocol can take advantage of system resources and hardware
  - *This is a good thing for building distributed systems*

Styx in a Nutshell

- 14 message types
  - Initiate connection (`Attach`)
  - Traversing hierarchy (`Clone`, `Walk`)
  - Access, creation, read, write, close, delete (`Open`, `Create`, `Read`, `Write`, `Close`, `Remove`)
  - Retrieve/set properties (`Stat`, `Wstat`)
  - Error (`Error`)
  - End connection (`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, 32K RAM, 16K ROM)

Example : Snooping on Styx

- *Interloper* 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/#/
;
```

Talk Outline

- Inferno Overview
- Abstraction and *resources as files* in Inferno
- The Limbo programming language
- Pervasive computing with Inferno and Limbo
- Summary

Programming in Limbo

- 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)

Language Data Types

- Basic types
  - `int` — 32-bit, signed 2's complement notation
  - `big` — 64-bit, signed 2's complement notation
  - `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` — channel (inter-thread communication path) of basic or structured type
  - `Tuples` — Unnamed collections of basic / structured types

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)

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");
}
```

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!\n”);
}
```

Module Name

Various Includes

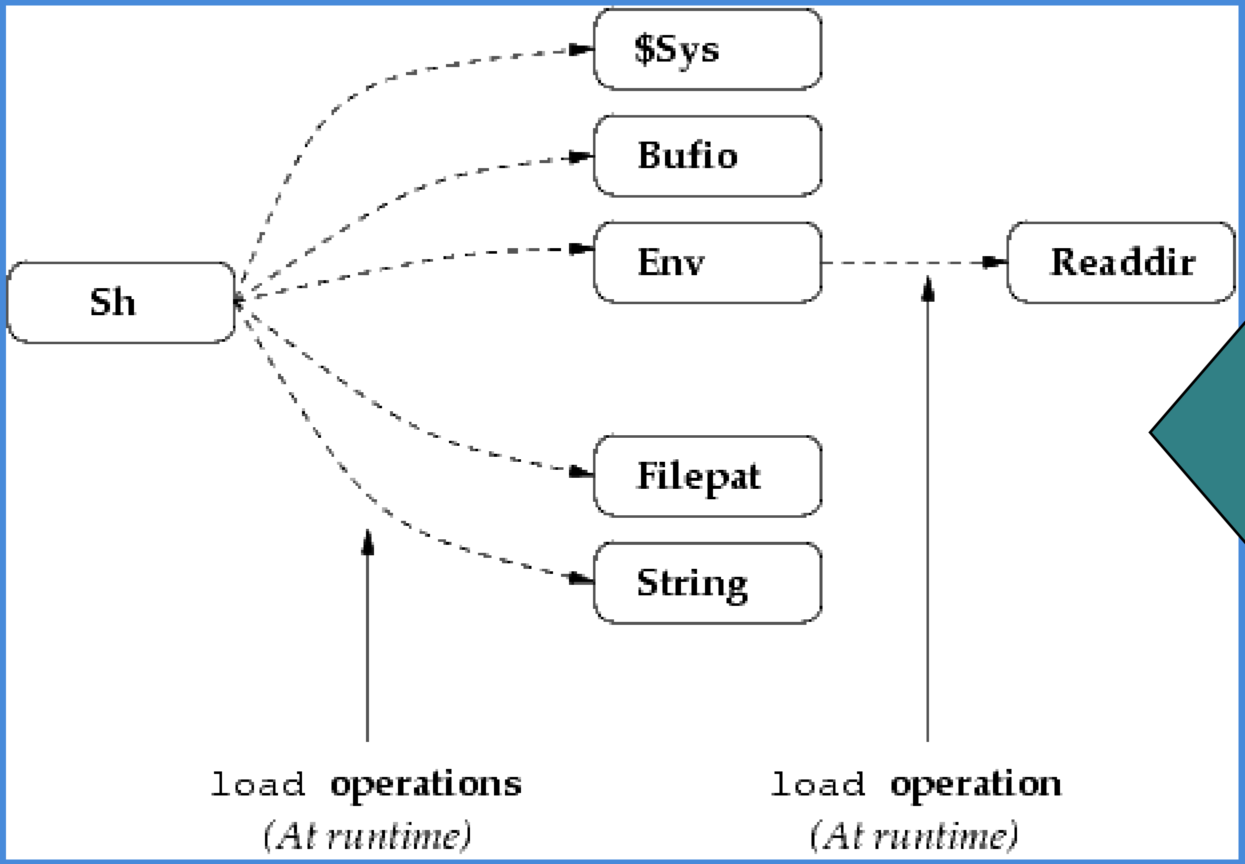
Module Type (interface) Definition

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

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!

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)

Open data interface for Ethernet driver

Open control interface for Ethernet driver

spawn statement creates new thread from function

```
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);
}
```

Xsniff (3)

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;
}
```

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)

Channels : Eratosthenes Sieve

```
implement Eratosthenes;
...
init(nil : ref Draw->Context, nil : list of string)
{
	sys = load Sys Sys->PATH;

	i := 2;
	sourcechan := chan of int;
	spawn sieve(i, sourcechan);
	while () sourcechan <-= i++;
}

sieve(ourprime : int, inchan : chan of int)
{
	n : int;
	sys->print("%d ", ourprime);
	newchan := chan of int;

	while (!((n = <-inchan) % ourprime)) ;

	spawn sieve(n, newchan);
	while ()
	{
		if ((n = <-inchan) % ourprime)
		{
			newchan <-= n;
		}
	}
}
```

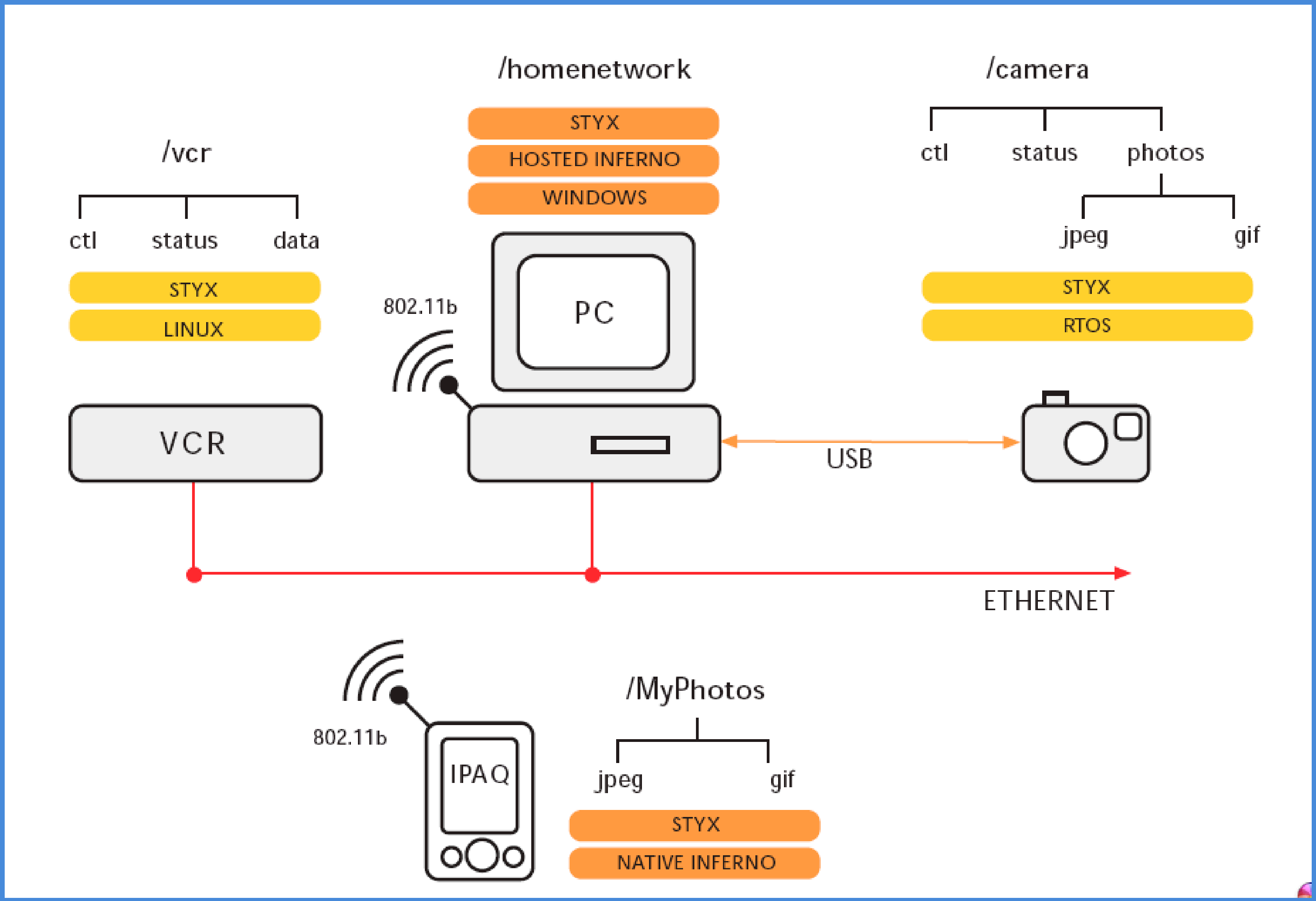
Talk Outline

- Inferno Overview
- Abstraction and *resources as files* in Inferno
- The Limbo programming language
- Pervasive computing with Inferno and Limbo
- Summary

Inferno and Limbo for Pervasive Computing

- Build distributed applications
  - Limbo module system, language level-channels, ease of writing user-level resource servers (resources as files)
- Cross platform
  - Portions of single application can run on a heterogeneous set of hardware and OS platforms, with a combination of native Inferno and emulator or Styx implementation
  - Easily integrate special purpose hardware (e.g., a networked sensor) using Styx
- Cross protocol
  - Uniformly deploy networked applications, taking advantage of network protocol, authentication and encryption support

Example

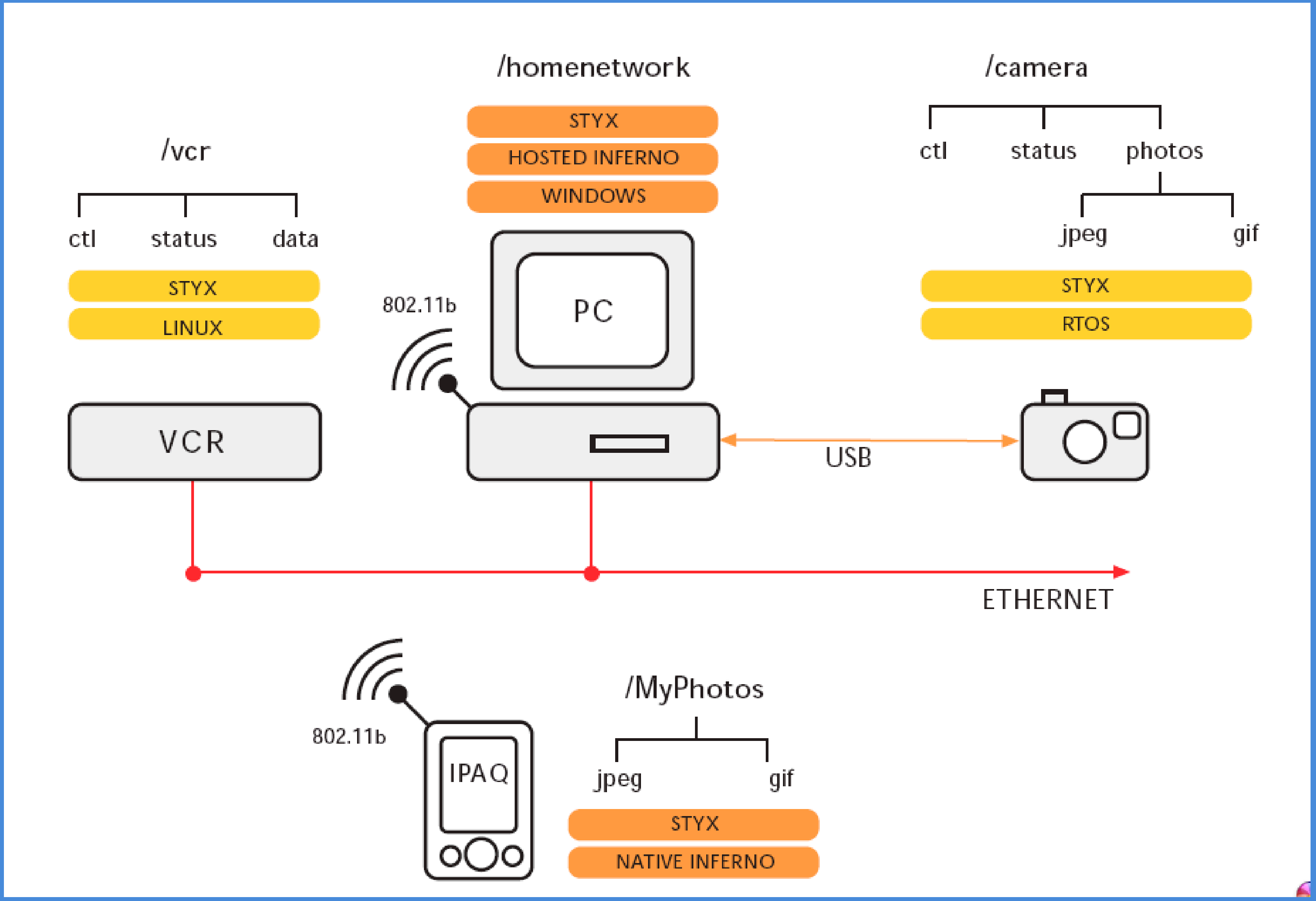


(Example from Vita Nuova Inferno Overview Document)

- PC
  - Running emulator over Windows
- VCR/DVR
  - Running Linux, and a Styx server implemented in C
- Digital Camera
  - Running some RTOS (e.g., DigitaOS) and a Styx implementation (C ? ASM ?)
- PDA
  - Ipaq running native Inferno for the StrongARM processor

Goal : Take pictures on camera, store time-lapse images on DVR, control from either PC, camera or PDA

Example

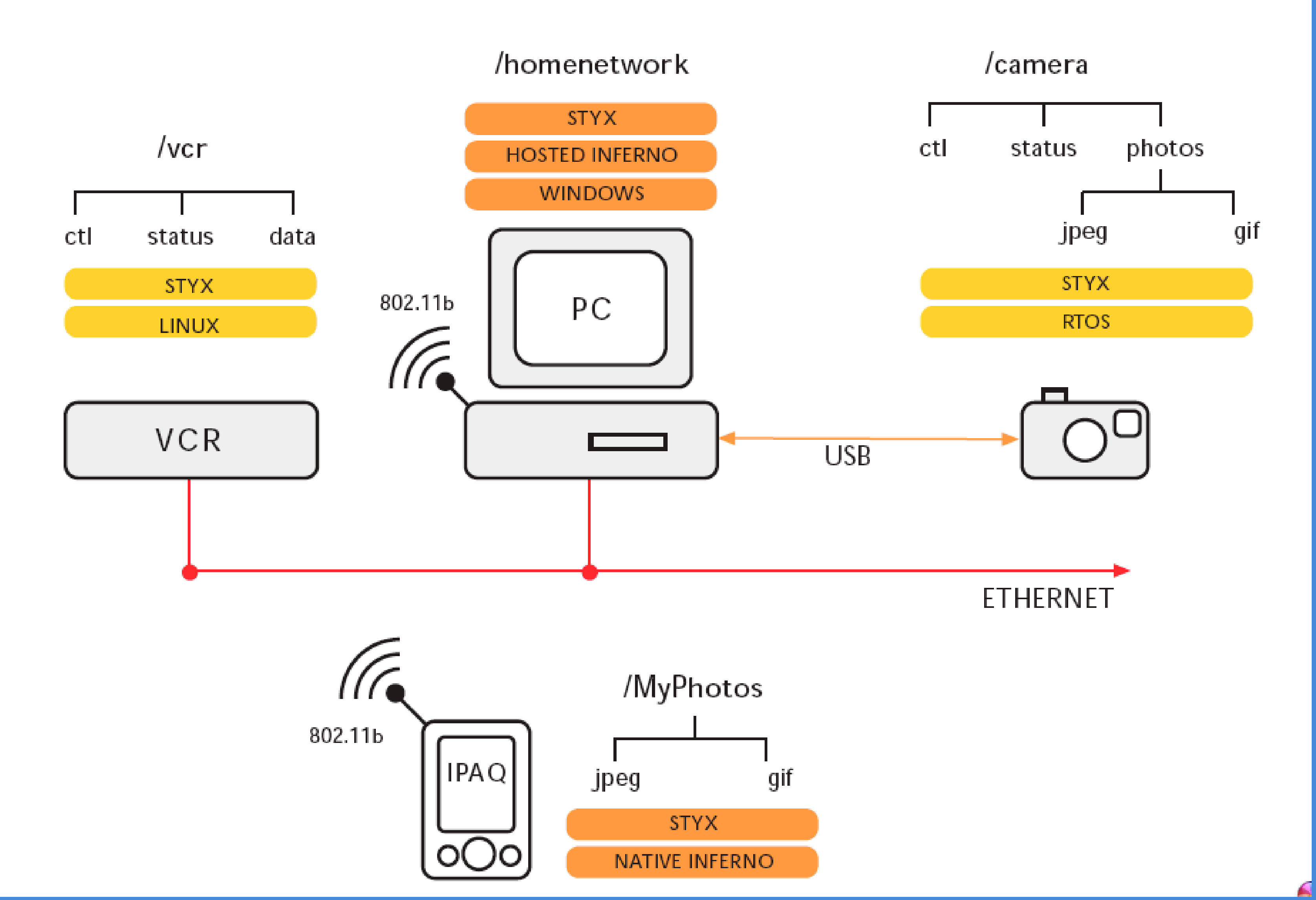


(Example from Vita Nuova Inferno Overview Document)

- Attach remote name space via `mount` (recall discussion of mount driver, and Styx)

```
mount tcp!182.1.1.2 /n/remote/vcr
mount tcp!182.1.1.3 /n/remote/camera
```

Example

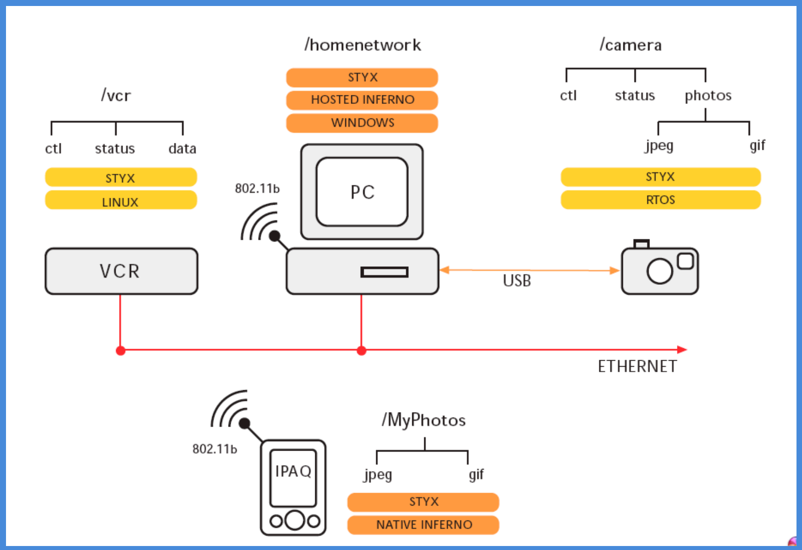


(Example from Vita Nuova Inferno Overview Document)

- Re-align the placement of remote name space in current name space by `bind`

```
bind -a /n/remote/vcr /homenetwork/vcr
bind -a /n/remote/camera /homenetwork/camera
bind -a '#Uc:/MyPhotos'/homenetwork/MyPhotos
```

Example

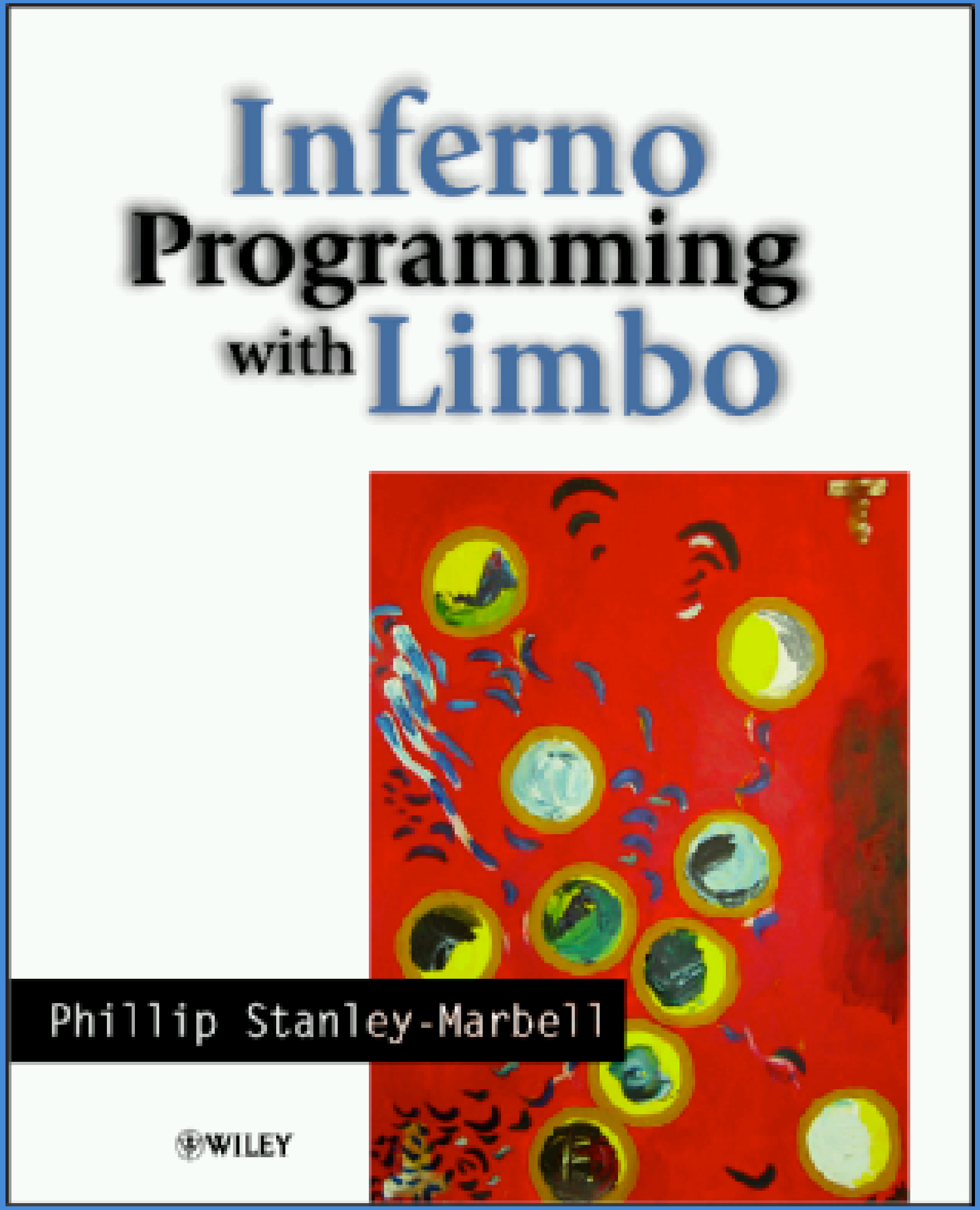



```
echo ‘record single frame’> /homenetwork/vcr/ctl
echo ‘picture type jpg’> /homenetwork/camera/ctl
while : ; do
echo ‘snap’> /homenetwork/camera/ctl
photo=‘cat /homenetwork/status’
cp /homenetwork/camera/photos/$photo.jpg /homenetwork/MyPhotos
cp /homenetwork/camera/photos/$photo.jpg /homenetwork/vcr/data
echo ‘next frame’> /homenetwork/vcr/ctl
echo ‘delete $photo’> /homenetwork/camera/ctl
sleep 10
done
echo ‘record off’> /homenetwork/vcr/ctl
echo ‘rewind’> /homenetwork/vcr/ctl
```

- Controlling entire heterogeneous system is easy because all resources can be controlled by simple commands from the command line (or in a simple application)
- Can easily add or remove resources, change which device controls or stores, simply by rearranging name space

Summary

- Resource abstraction is good
- Files are just an abstraction, not inherently tied to disk
- Represent resources as files
- Access resources with a simple protocol (Styx)
- Limbo language is good clean fun!
- Inferno
  - Runs natively on many processor architectures
  - Emulator runs on a wide variety of host platforms
- It's easy to distribute resources in a heterogeneous network when all resources are represented as files



Book's web page

http://www.gemusehaken.org/ipwl/

Complete source for all examples from book, and more

Free Review Copy

http://dsonline.computer.org/books/list.htm

Great opportunity to see your review in print, and get a free copy to boot!

Thank you