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

Phillip Stanley-Marbell
pstanley@ece.cmu.edu

Lecture Outline

- Lecture 1 review
- Abstraction and Names
- *Resources as files* in Inferno
- (next 2 lectures: Introduction to Limbo, Limbo data types and the Dis VM)

Lecture 1 Review: Terminology

- Inferno
  - An operating system
- Limbo
  - A programming language for developing applications for 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

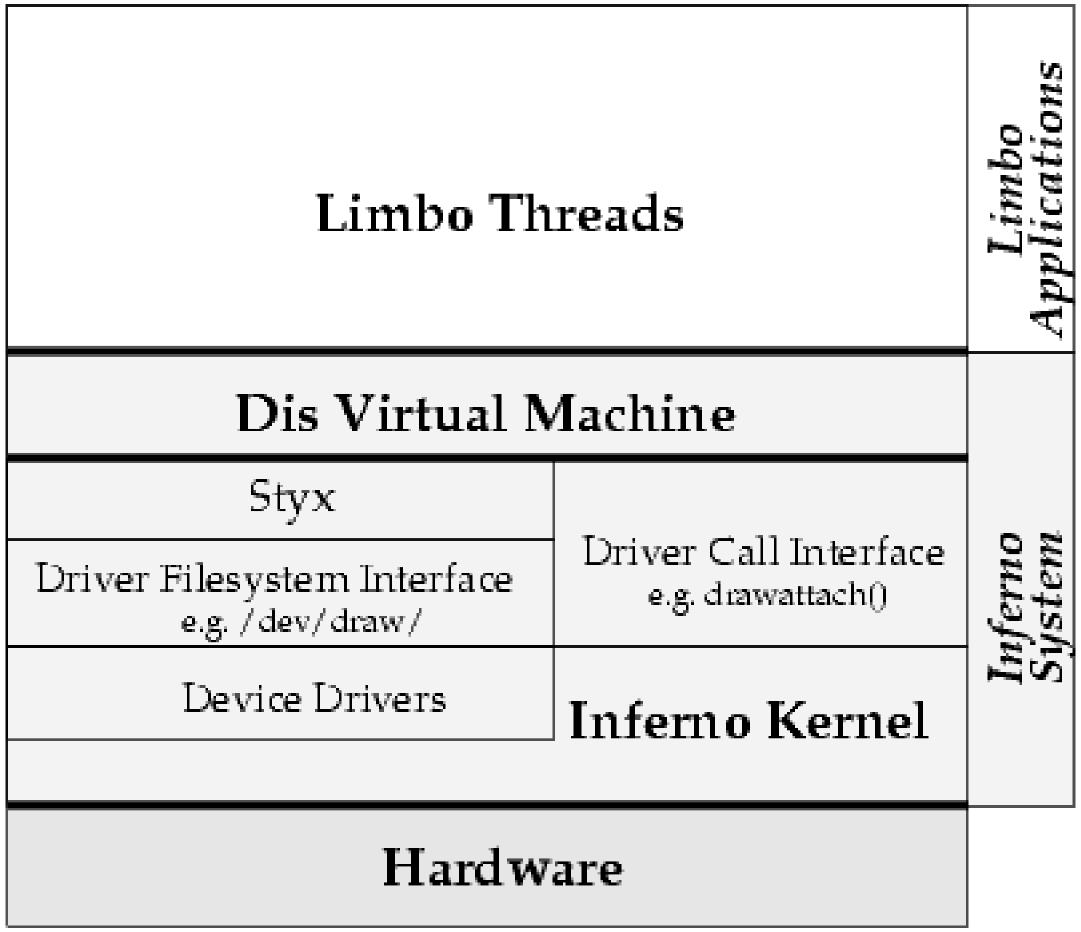
Lecture 1 Review: Inferno

- Like any other traditional OS, 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, including filesyste, network stack, graphics subsystem — ***everything*** — not just code execution (e.g., in Java Virtual Machine)

Lecture 1 Review: Inferno System Architecture



Limbo Threads

Dis Virtual Machine

Styx

Driver Filesystem Interface
e.g. /dev/draw/

Driver Call Interface
e.g. drawattach()

Device Drivers

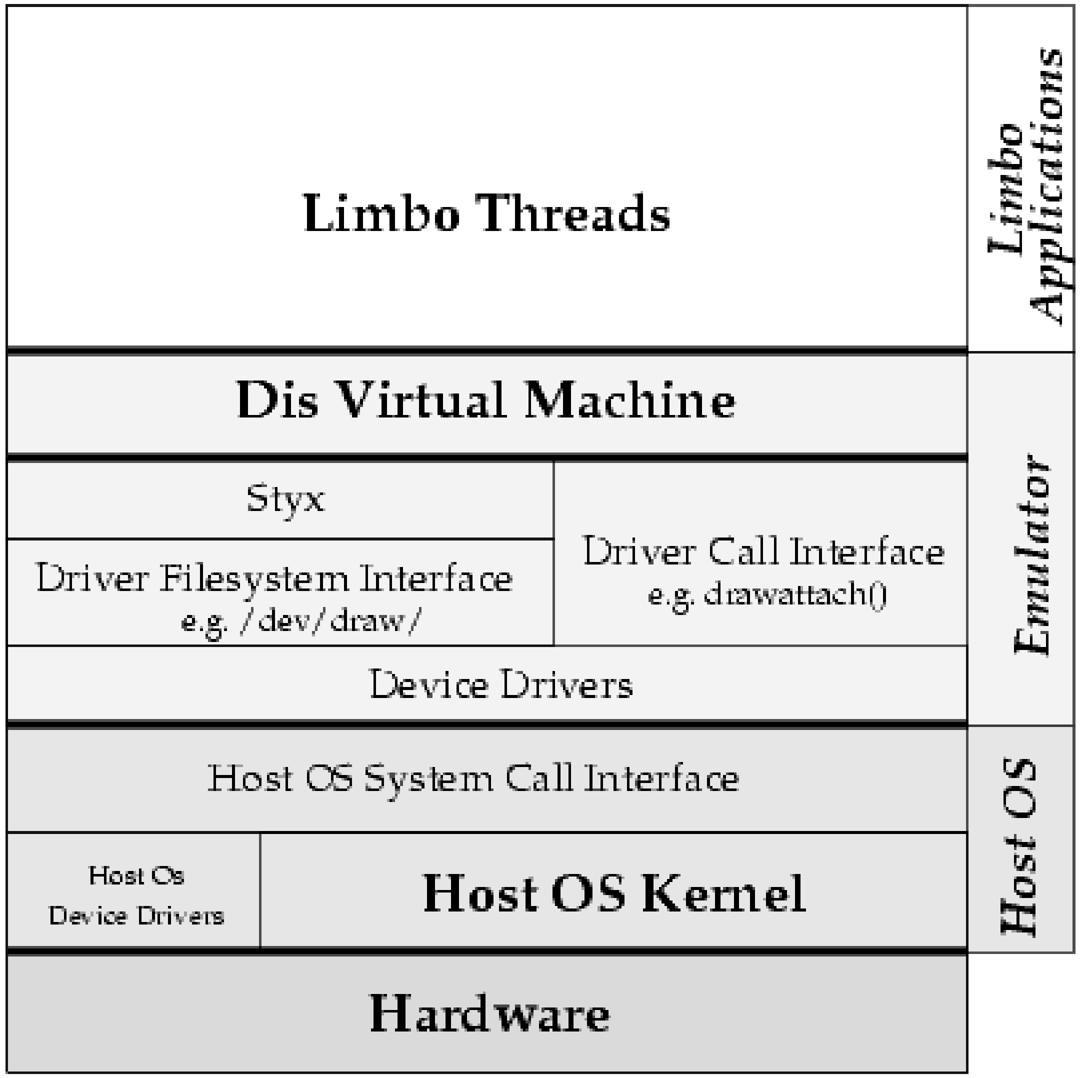
Inferno Kernel

Hardware

Limbo Applications

Inferno System

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



Limbo Threads

Dis Virtual Machine

Styx

Driver Filesystem Interface
e.g. /dev/draw/

Driver Call Interface
e.g. drawattach()

Device Drivers

Host OS System Call Interface

Host Os
Device Drivers

Host OS Kernel

Hardware

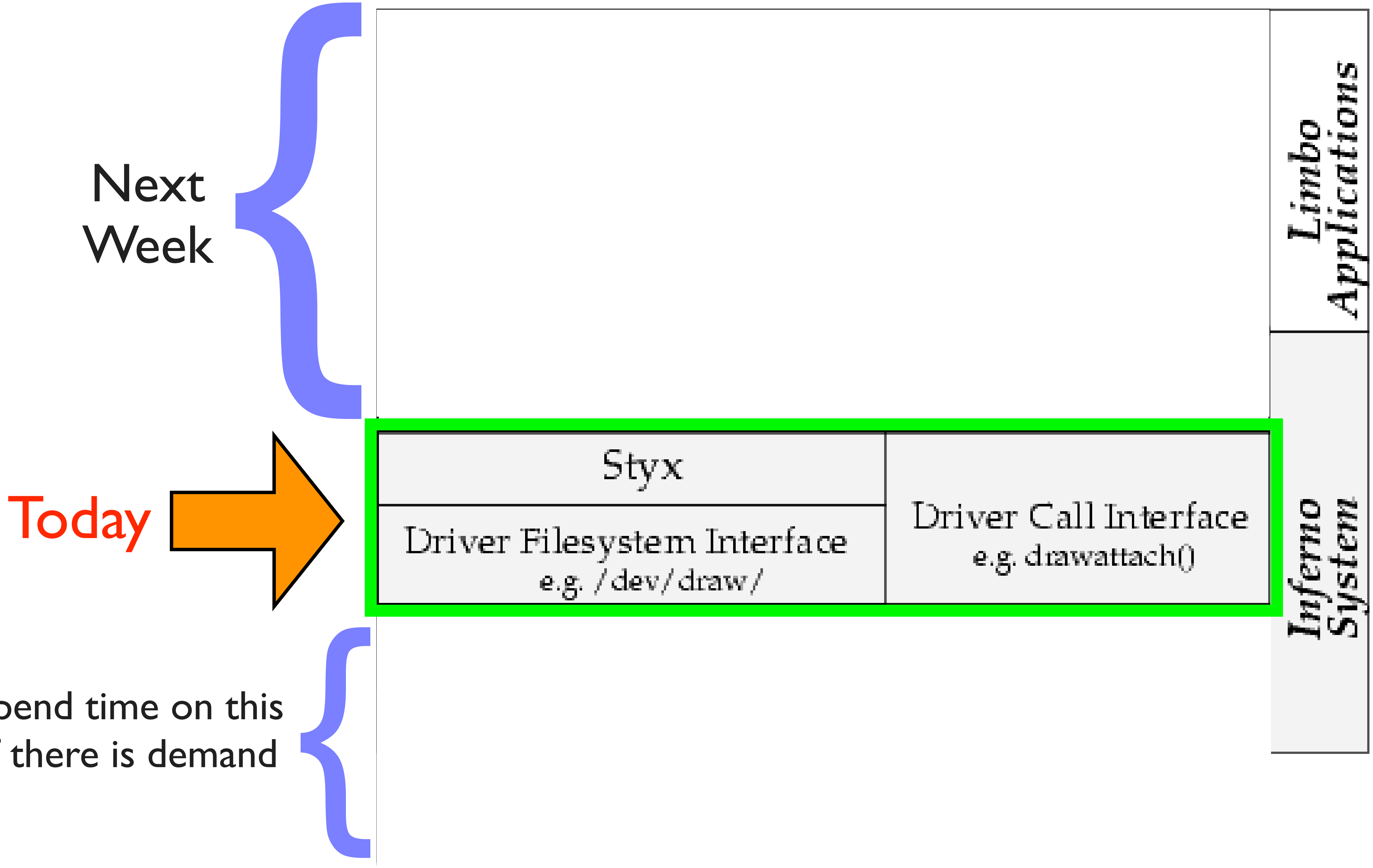
Limbo Applications

Emulator

Host OS

Hosted (i.e., emulator)

Lecture 1 Review: Inferno System Architecture



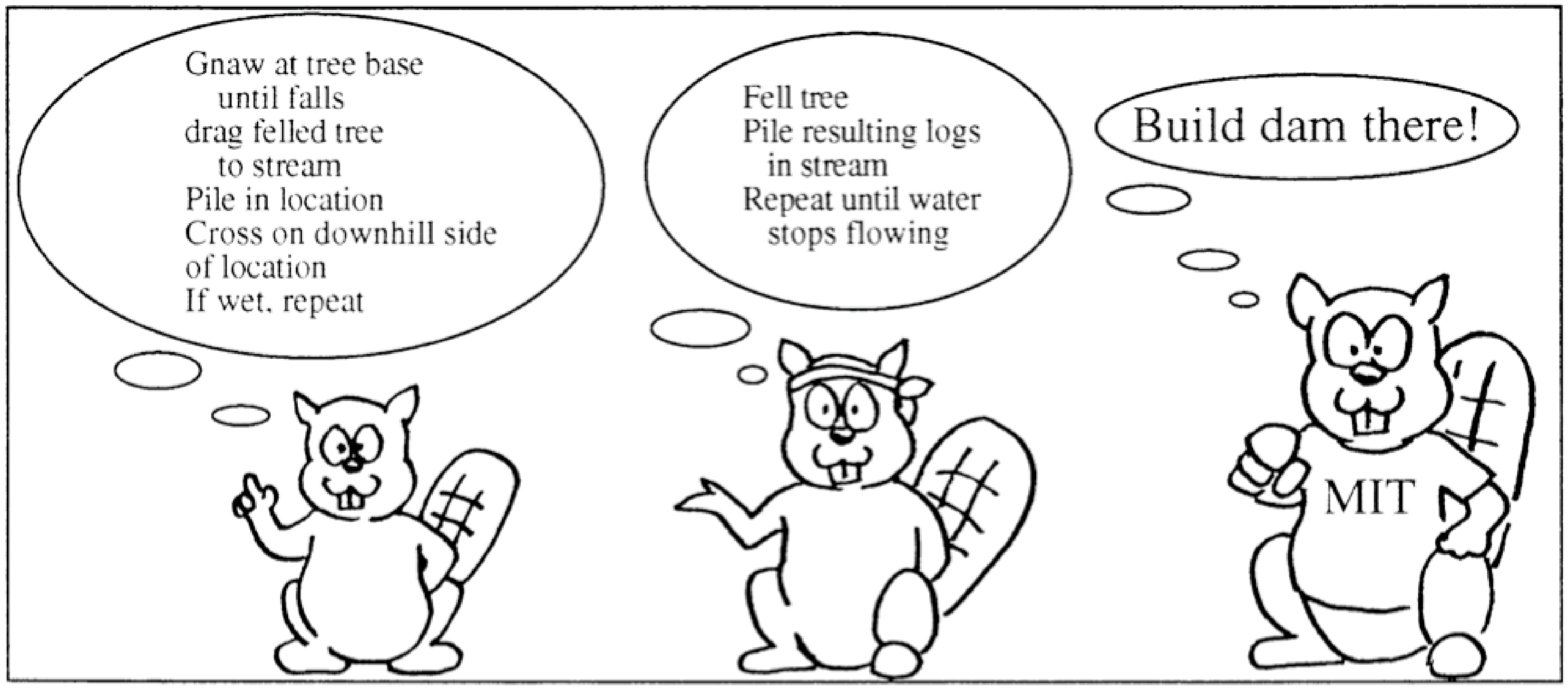
Course Outline : Syllabus

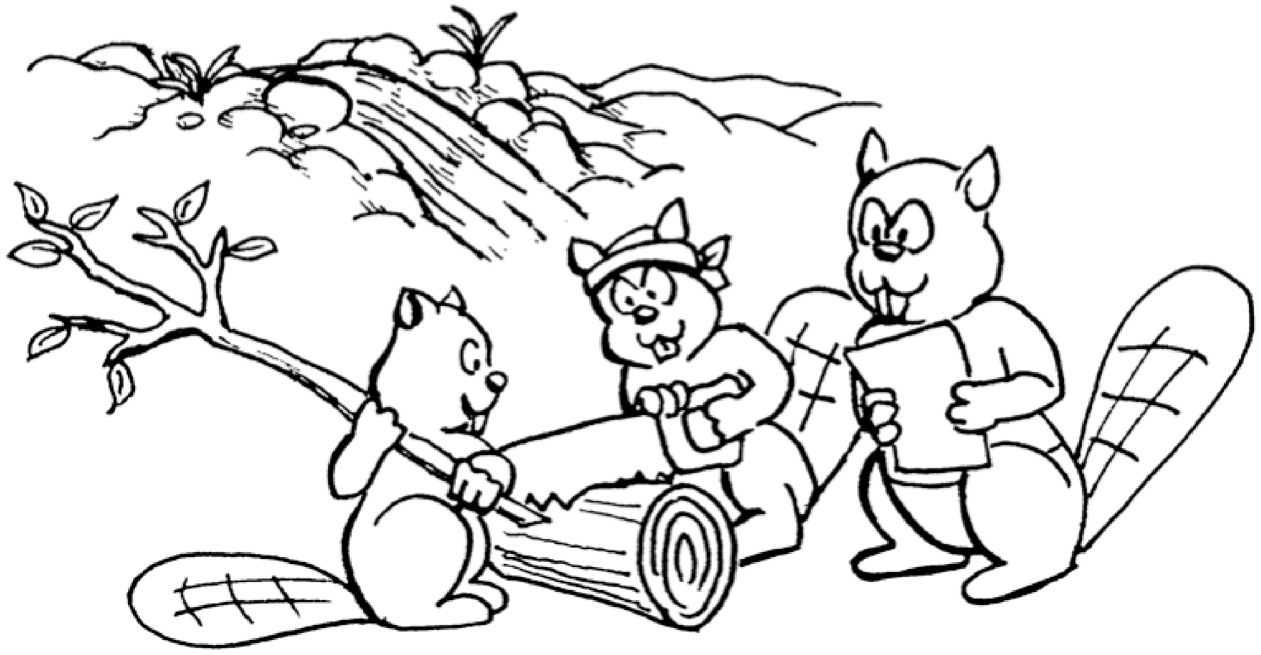
- Week 1: Introduction to Inferno
- Week 2: Overview of the Limbo programming language
- Week 3: Types in Limbo
- Week 4: ~~Abstraction as a design tool, Names, Resources as files~~ Inferno Kernel Overview ?
- Week 5: ~~9P and Styx, Resources as files and Limbo threads~~ Inferno Kernel Device Drivers ?
- Week 6: C applications as resource servers: Built-in modules and device drivers
- Week 7: Case study I — building a distributed multi-processor simulator
- Week 8: Platform independent Interfaces: Limbo GUIs; Project Update Spring Break

---

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

Abstract Up





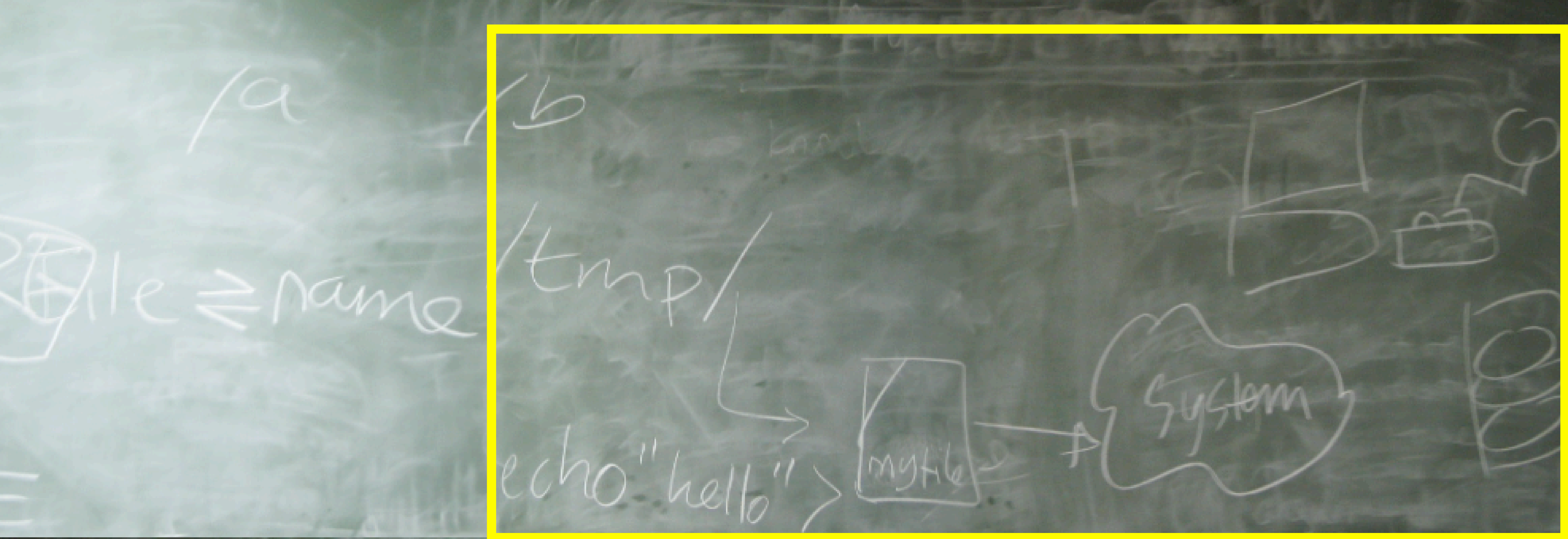
Compile Down

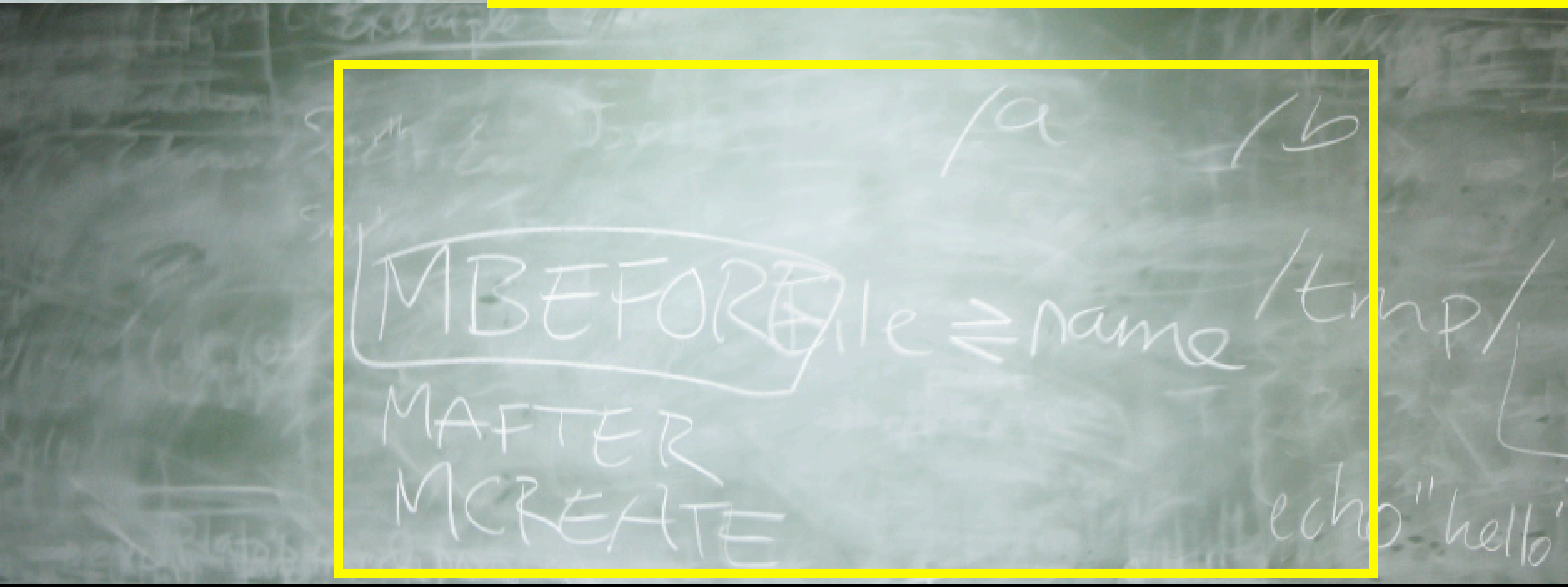
Resource abstraction

- 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, e.g., network (well, there's RPCs, but these are seldom uniform)
- 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

Question to mull on

- What happens when a user at a terminal echos the string "hello" into the file /tmp/myfile
  - At some point file is opened via an open syscall ?
  - At some point a write syscall happens ?
  - Strings goes into the OS buffer cache ?
  - String gets flushed to magnetic disk ?


/a
/b
file ≥ name
/tmp/
echo "hello" > myfile
System


/a
/b
MBEFORE file ≥ name
MAFTER
MCREATE
/tmp/
echo "hello"

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 (*i.e., 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 file/name 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 systems, with resources (files, i.e., names) spread across network

Inferno : Resources as files

- Builds on the ideas developed in the Plan 9 Operating System
  - *Most system resources represented as names* (files) in a hierarchical *name space*
  - Simple protocol ("*Styx*") for accessing names, whether local or over network
  - These *names provide abstraction for resources* (such as those available in, e.g., UNIX, via system calls)
    - Graphics
    - Networking
    - Process control
- Implications
  - Access local and remote resources with the same ease as local/remote files
  - Restrict access to resources by restricting access to portions of name space
  - name space is "per process", so different programs can have different views of available resources

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*

Question 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.

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)*

Demo

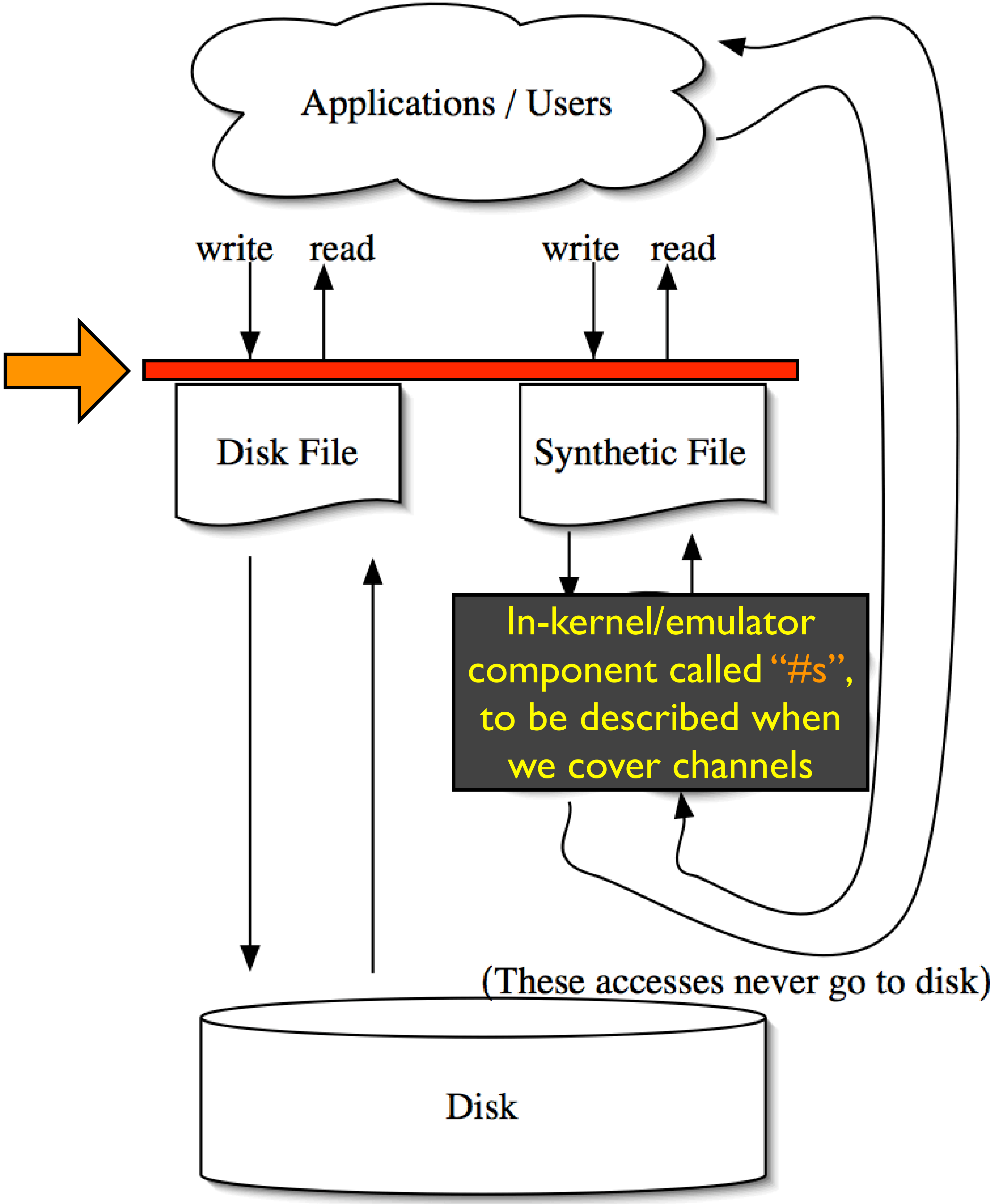
Access *and* Control via Name Space

- 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`** system call
    - **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` or `/dev/sound`, but can only change sample rate via **`ioctl`**
- Inferno: files used for both resource access *and* control
  - `/dev/audio` for audio data, `/dev/audioctl` 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
  - Synthetic files / directories can be created, dynamically, by user-level applications

Accessing Names

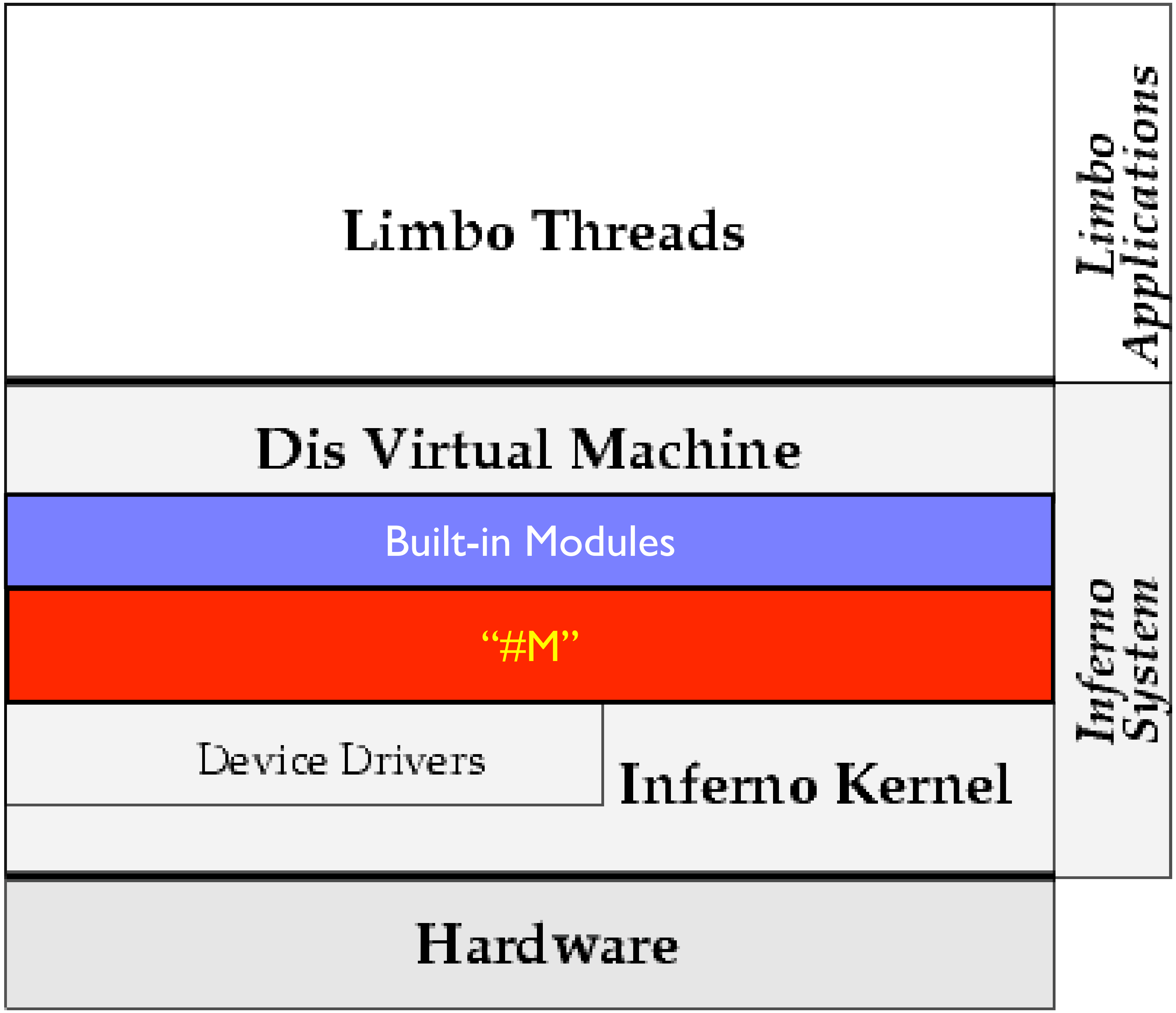
- What happens when names are accessed ?
  - Operations on a single name: open, read, write
  - Traversing hierarchies of names

In-kernel/emulator component called "#M", to be described next

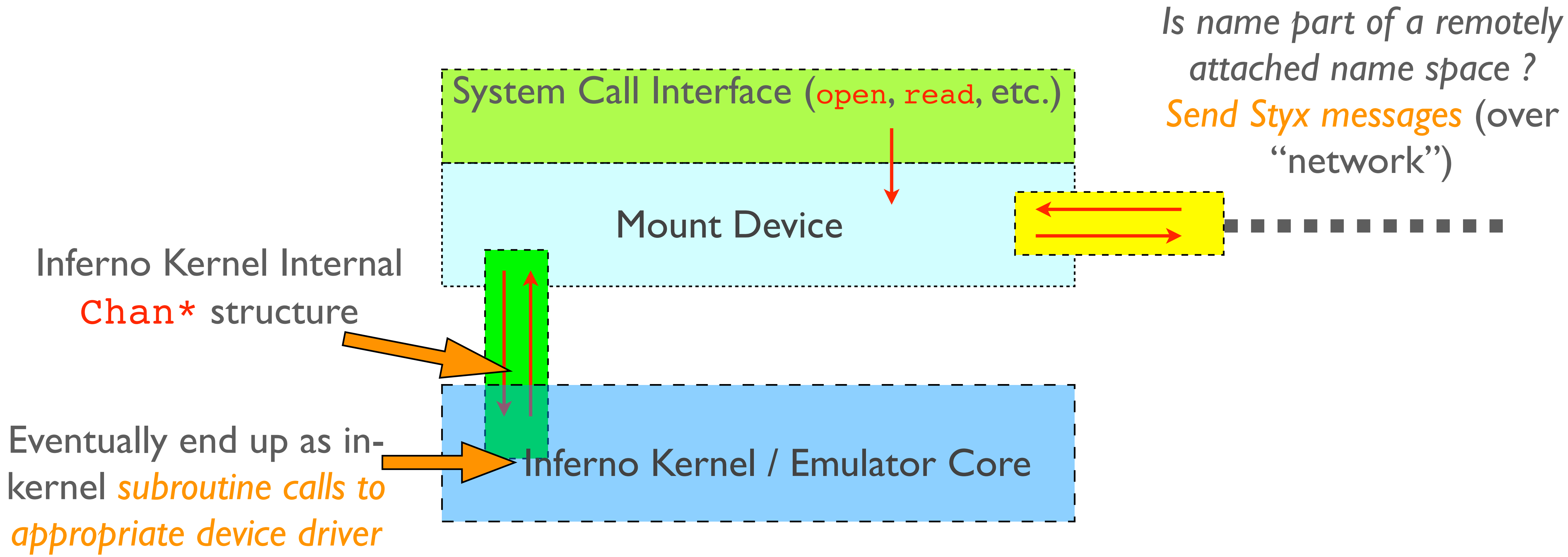


In-kernel/emulator component called "#s", to be described when we cover channels

Inferno System Structure

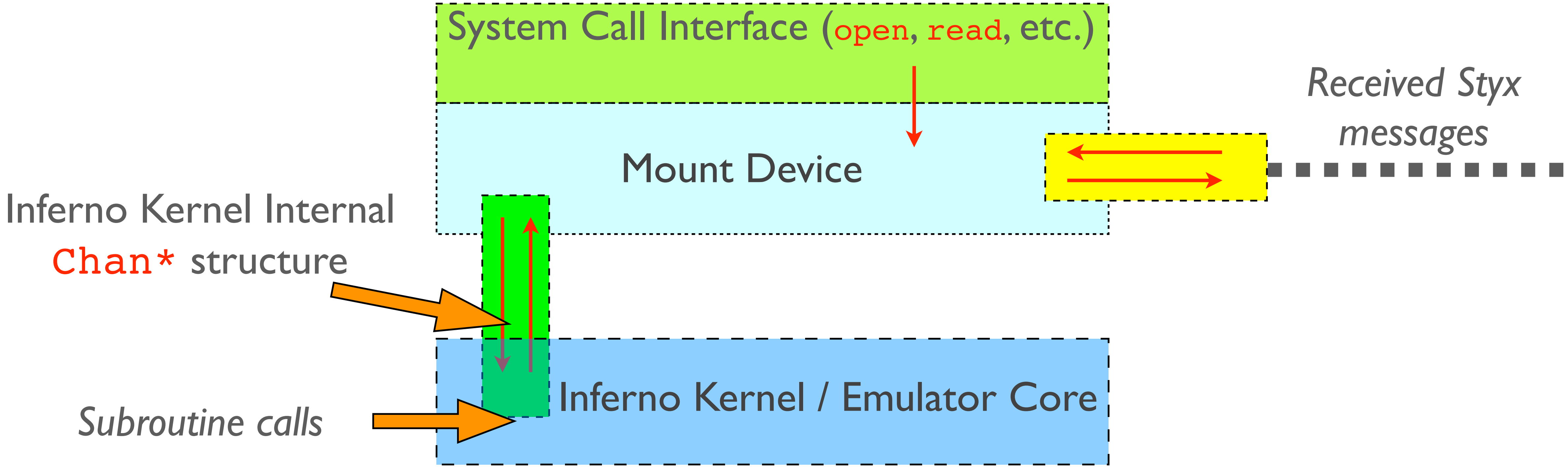


Accessing Name Space Entries: The *Mount Device*, #M


Is name part of a remotely attached name space ?
Send Styx messages (over "network")
System Call Interface (open, read, etc.)
Mount Device
Inferno Kernel Internal Chan* structure
Eventually end up as in-kernel subroutine calls to appropriate device driver
Inferno Kernel / Emulator Core

- 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


System Call Interface (open, read, etc.)
Received Styx messages
Mount Device
Inferno Kernel Internal Chan* structure
Subroutine calls
Inferno Kernel / Emulator Core

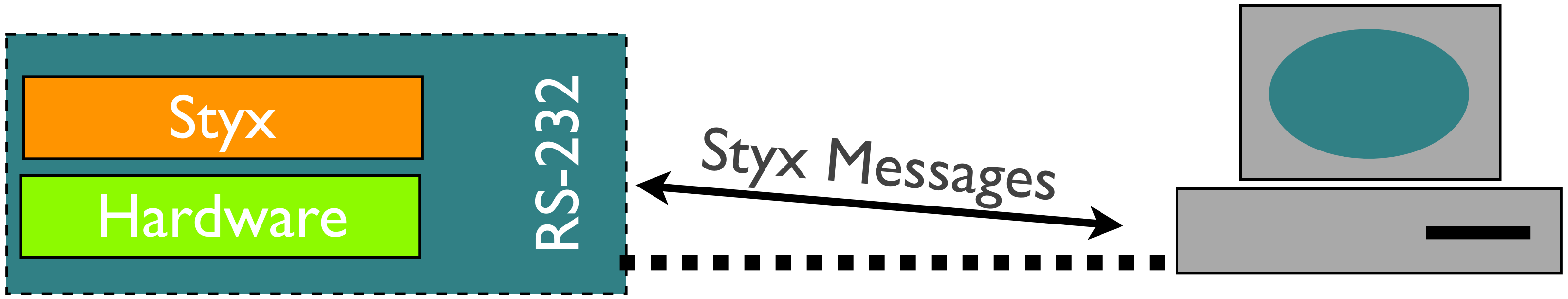
- Mount driver also converts Styx messages coming in over the network into calls to local device drivers

- Any entity that can speak the Styx protocol can take advantage of system resources and hardware (subject to permissions / auth)
    - *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

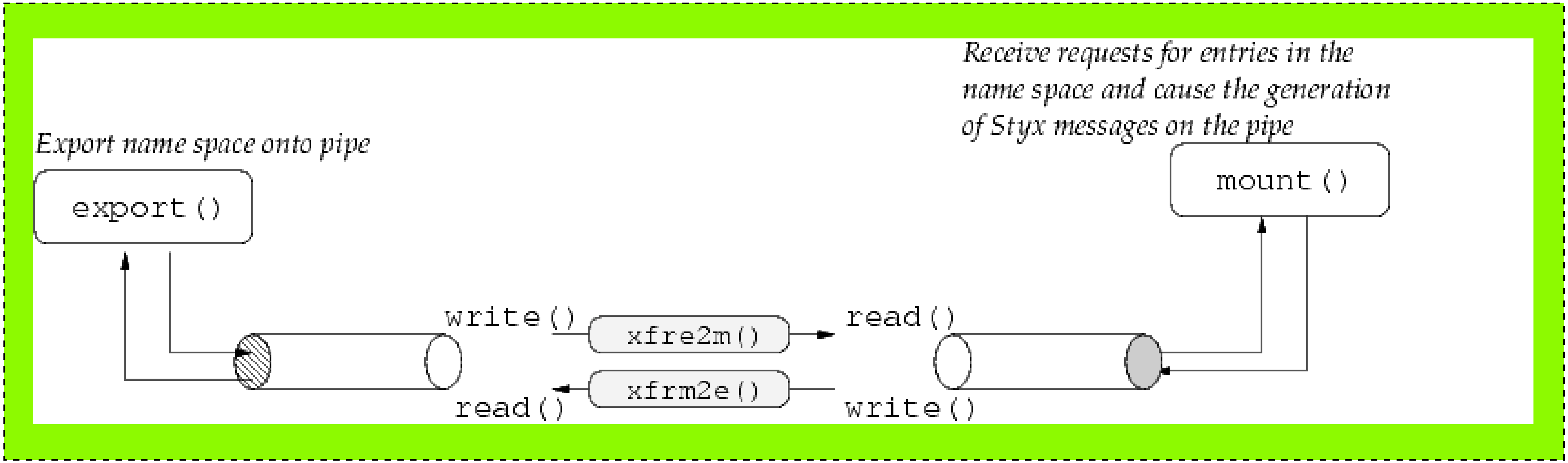
Real world example: Styx on Lego Rcx Brick (Hitachi H8 microcontroller, 32K RAM, 16K ROM)

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/#/
;
```

Intercepting Styx Messages


Receive requests for entries in the name space and cause the generation of Styx messages on the pipe
Export name space onto pipe
export()
mount()
write()
xfre2m()
read()
read()
xfrm2e()
write()

Demo: Interacting with a Styx server written in C (`/tools/styxtest/`)

Reading

- Required Reading
  - “The Styx Architecture for Distributed Systems” (`http://cmu.edu/blackboard`) also available at `http://www.vitanuova.com/inferno/papers/styx.html`)
- Relevant chapter in “Inferno Programming with Limbo”
  - Chapter 8

Next Week

- We'll actually start writing / looking at code
  - Introduction to Limbo (monday)
  - Limbo data types and the Dis Virtual Machine (one week later)

Fin.