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

Phillip Stanley-Marbell
pstanley@ece.cmu.edu

Lecture Outline

- A bit more about data types : ADTs and ref ADTs
- Dis VM architecture and internal data types

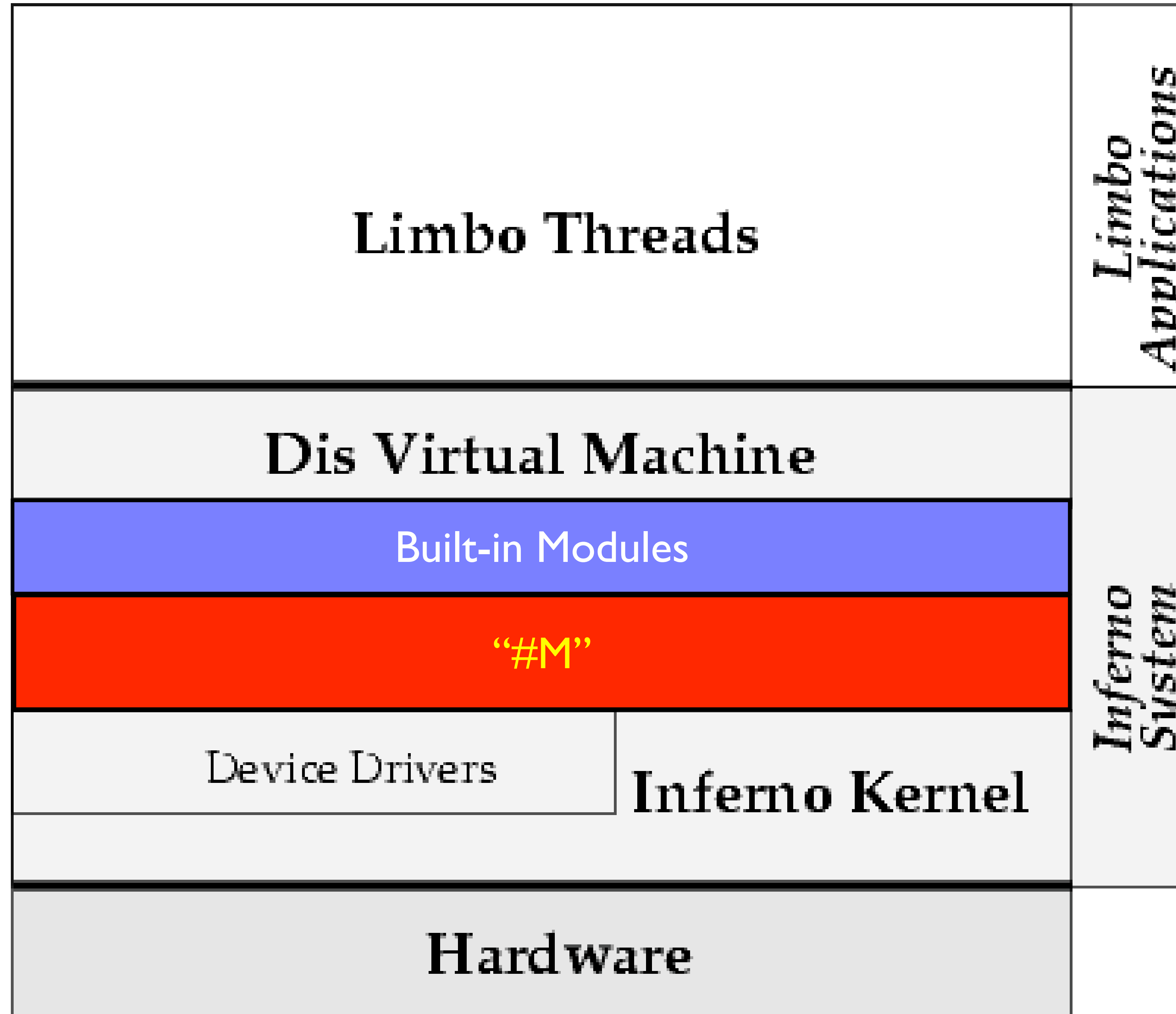
Course Outline : Syllabus

- 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 7: Case study I — building a distributed multi-processor simulator

Spring Break

- Week 8: Platform independent Interfaces: Limbo GUIs; Project Update
- 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

Inferno System Structure



ADTs

- ADTs — Abstract Data Types or Aggregate Data types
  - Collection of functions and data

```
Machine : adt
{
   vdd    : real;
   freq   : real;
   ID     : string
   fmt    : fn(mach : Machine); # Note: takes a value of an ADT
}
```

  - They are essentially like tuples, except that they can contain functions, and datums have names

  - Can cast from tuples to ADTs

```
m := Machine (3.3, 60.0, “none”);
```

  - ADTs are a value type
    - In above example of `Machine` ADT, any changes to the received ADT *instance* made by the `fmt` ADT function member will won’t be seen elsewhere. Why ? (hint, what is the function’s return type ?)

Reminder: ".", "->" and "<-"

- The "->" separator is used to access module members

```
sys = load Sys Sys->PATH;
sys->print("yikes!");
```

- The "<-" operator is used to send or receive to a channel

```
mychan : chan of int;
mychan <- = 5;
```

- The "." separator is used to access an ADT member function or datum

```
m.fmt(m);
```

ADT function definitions

- After defining ADT *type*, its *function implementations* must also be provided, if it contains functions

```
Machine : adt
{
   vdd    : real;
   freq   : real;
   ID     : string
   fmt    : fn(mach : Machine); # Note: takes a value of an ADT
}
Machine.fmt(mach : Machine)
{
   sys->print(”%f\n”, mach.vdd);
   sys->print(”%f\n”, mach.freq);
   sys->print(”%s\n”, mach.ID);
   mach.vdd = -99.9;

   return;
}
...
m := Machine (3.3, 60.0, “none”);
m.fmt(m);
m.fmt(m);
```

What is printed ?

Reference ADTs

- These are a variant of ADTs that are passed by reference rather than by value
  - Syntax example

```
m : ref Machine;
```

- Creates a *reference to a copy* of an ADT instance

```
m0 := Machine (3.3, 60.0, “none”);

m0.fmt(m0);
mp = ref m0;
mp.vdd = 1.8;
m0.fmt(m0);
```

what is printed out ?

Reference ADTs

- These are a variant of ADTs that are passed by reference rather than by value
  - Syntax example

```
m : ref Machine;
```

- Creates a *reference to a copy* of an ADT instance, not a reference to the instance named in `ref ...`

```
mp = m0 = ref Machine (3.3, 60.0, “none”);

m0.fmt(m0);
mp.vdd = 1.8;
m0.fmt(m0);
```

In this case, both mp and m0 are references to the same copy

(They’re references to copies of the ADT instance created from tuple `(3.3,60.0,“none”)`

Reference ADTs and `self`

- In examples seen thus far, cumbersome method for having an ADT instance work on its own data:
  ```
  m0.fmt(m0);
  ```

- Functions defined in ref ADTs (and those only!) can specify their first argument is a reference to their own instance
  ```
  Machine.fmt(mach : self ref Machine)
  {
    sys->print(”%f\n”, mach.vdd);
    sys->print(”%f\n”, mach.freq);
    sys->print(”%s\n”, mach.ID);
    return;
  }
  ...
  m := Machine (3.3, 60.0, “none”);
  m.fmt(); # Note: no args at call site, but function defn has
  ```

More

- ADTs and import
  - Imagine:

```
include “mach.m”; # defines the Machine ADT shown earlier

#   Need to load code that implements ADT functions!
machmod  = load Machmod Machmod->PATH;
m : Machine;

#   Will not work!
m.fmt();
```

- Pick ADTs
  - ADTs with union substructures
  - Also permit limited form of “pattern matching on type” (sort of)

- Read the book if you’re interested

The Dis Virtual Machine

- The execution layer in Inferno
- Limbo applications are compiled to an binaries (*bytecode*), that the virtual machine executes
- Abstracts away the machine architecture, so compiled Limbo programs are not tied to the host machine architecture (e.g., x86, MIPS, SPARC etc.)
- The virtual machine is part of the kernel / emulator, and is implemented in C

Dis VM architecture

- Architecture versus Microarchitecture
  - Architecture represents interface seen by programs, i.e., Instruction Set Architecture (ISA)
  - Microarchitecture represents *how* things are implemented inside, e.g., the Intel Pentium versus the Intel 386 : same architecture (ISA) but different microarchitectures
- Like a real machine, it has an architecture (but not microarchitecture)
  - A memory-to-memory machine (think of it as having as many registers as there are words in memory)
  - 3 address instructions: `op src1 src2 dest`
  - Operands have types: `word`, `big`, `byte`, `real` (Do these look familiar ?)

Dis VM types

- `word`: 32-bit, signed
- `byte`: 8-bit unsigned
- `big`: 64 bit, signed
- `real`: 64-bit IEEE 764 float
- `short word`: 16-bit, signed
- `short float`: 32-bit IEEE 764 float
- Instructions operate on data items of these types:
  - E.g., `addb`, `addw`, `addf`, `addl`

Recall: 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
;
;
;
```

Dis

- Handles execution of application code
- Garbage collection
- Channel Communication
- Module signing and verification
- Module load-time type checking of loaded code versus signatures (which are MD5 hashes)

Demo : Looking at Dis VM Spec

Next Lecture

- Next week : Inferno kernel and emulator source structure, kernel and emulator implementation

Fin.