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

Phillip Stanley-Marbell
pstanley@ece.cmu.edu

Lecture Outline

- C applications as Inferno Resource Servers

- Styx Servers vs Builtin-Modules and Device Drivers : *Tradeoffs*

No Class Next Week

- 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, device drivers, external Styx servers
- 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

Combining C code w/ Limbo

- Reasons for integrating C code w/ Limbo applications
  - You already have a substantial application / algorithm implemented in C, and you don't want to re-implement it in Limbo
  - You need to integrate a performance-critical facility
  - **Note**: Should only consider integrating C code when the facility is going to be used often
- Implementation options
  1. Built-in modules
  2. Device Drivers
  3. External Styx Servers
- Extant examples: Sys, Draw, Math, Tk, Prefab, Keyring
  - Implemented as built-in modules, to provide high performance for math operations etc.

Built-in modules

- Seen by Limbo programs as other Limbo modules
  - Gain access to module as usual by doing a `load`
- Not loaded from a Dis bytecode file (obviously)
  - Loaded form the special name "$*modulename*"
- The built-in modules (C code) provide module interface definitions just as though they were Limbo modules
  - E.g., `/module/math.m`
  - Limbo modules can call functions defined in built in modules, access constants defined in module interface, all as usual

Built-in Module Example: Math

- Module interface (recall, a *type*) defined in `/module/math.m`

- Example use (*nothing peculiar*):

```
init(ctxt: ref Draw->Context, args : list of string)
{
   math := load Math Math->PATH;
   cosπover2 = math->cos(Math->Pi/2);
   ...
}
```

  - **Advantage**: looks identical to a module implemented in Limbo, but when you call, e.g., `math->cos()`, the code is not running over the VM, but is a compiled C routine running as part of the emulator / kernel

- Downsides
  - Facilities are easily accessed by other hosts over network (relative to device drivers)
  - Built-in module is linked directly into emulator / native kernel
  - Bugs in your C code will crash the emulator / native kernel

Implementing Built-in Modules

- Limbo compiler provides some help
- Steps:

1. Define module interface. Following examples will use /module/math.m and will step through process of re-implementing the Math built-in module
2. Use Limbo compiler to generate a skeletal C implementation. This will define functions which system will expect to exist at runtime, based on module interface
3. Flesh out skeletal C implementation

Abridged `math.m`

```
Math: module
{
        PATH:   con     "$Math";

        ...

        sinh:           fn(x: real): real;
        sort:           fn(x: array of real, pi: array of int);
        sqrt:           fn(x: real): real;
        tan:            fn(x: real): real;
        tanh:           fn(x: real): real;
        y0:             fn(x: real): real;
        y1:             fn(x: real): real;
        yn:             fn(n: int, x: real): real;


        import_int:     fn(b: array of byte, x: array of int);
        import_real32:  fn(b: array of byte, x: array of real);
        import_real:    fn(b: array of byte, x: array of real);
        export_int:     fn(b: array of byte, x: array of int);
        export_real32:  fn(b: array of byte, x: array of real);
        export_real:    fn(b: array of byte, x: array of real);
};
```

- As usual for a Limbo module interface, defines module functions, constants and data structures/types

Generating C Implementation Stubs

- Steps
  - Generate C stubs with
    - `limbo -T` *Modulename* *file_containing_module_interface_defn*
    - e.g., `limbo -T Math math.m > mathmod.c`
  - Generate structure definitions and function prototypes needed by above C stub
    - `limbo -a` *file_containing_module_interface_defn*
    - e.g., `limbo -a math.m > math.h`
  - Generate Linkage Table which contains function signatures for built-in functions
    - e.g., `limbo -t Math math. > mathmod.h`

Generated files...

So how do they get linked in / initialized ?

- Once again, the emu/kernel config file

```
mod
        sys
        draw
        tk
        math
        srv     srv
        keyring
        loader
        freetype
```

- The generated emu.c (analogously for kernel) will contain calls to `sysmodinit()`, `drawmodinit()` etc, based above entries
- Actual module implementation usually linked into `libinterp`

Should you go Built-in ?

- The answer is usually *NO*.
- Device drivers provide same performance advantage, and resources can be made visible over network
- You can also implement a Styx server outside the emulator / kernel
  - This is often the way to go

Project

- Email me a semi-formal description of what you want to do for the final project
- Format
  - 1 - 2 pages, describing:
    1. Motivation (why you want to do it)
    2. Approach (How you think you're going to implement it)
    3. Goal / Delivery (What you will be able to show when you're done)
    4. Timeline (when you're going to finish what parts)

3. Due Next Monday (will count as mini-project grade)

Next

- Standalone Styx servers

Fin.