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

Phillip Stanley-Marbell
pstanley@ece.cmu.edu

Lecture Outline

- Native Kernel Overview

- Kernel Compilation

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 and device drivers
- 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

Kernel Components

- Virtual machine

- Built-in modules

- Device drivers
  - Virtual devices like `devprog`
  - Hardware device drivers like `devns16552` (Natl. semi UART), `dev8139` (Realtek Ethernet)

- Facilities
  - Process creation, process scheduling
  - Synchronization primitives
  - Memory management primitives

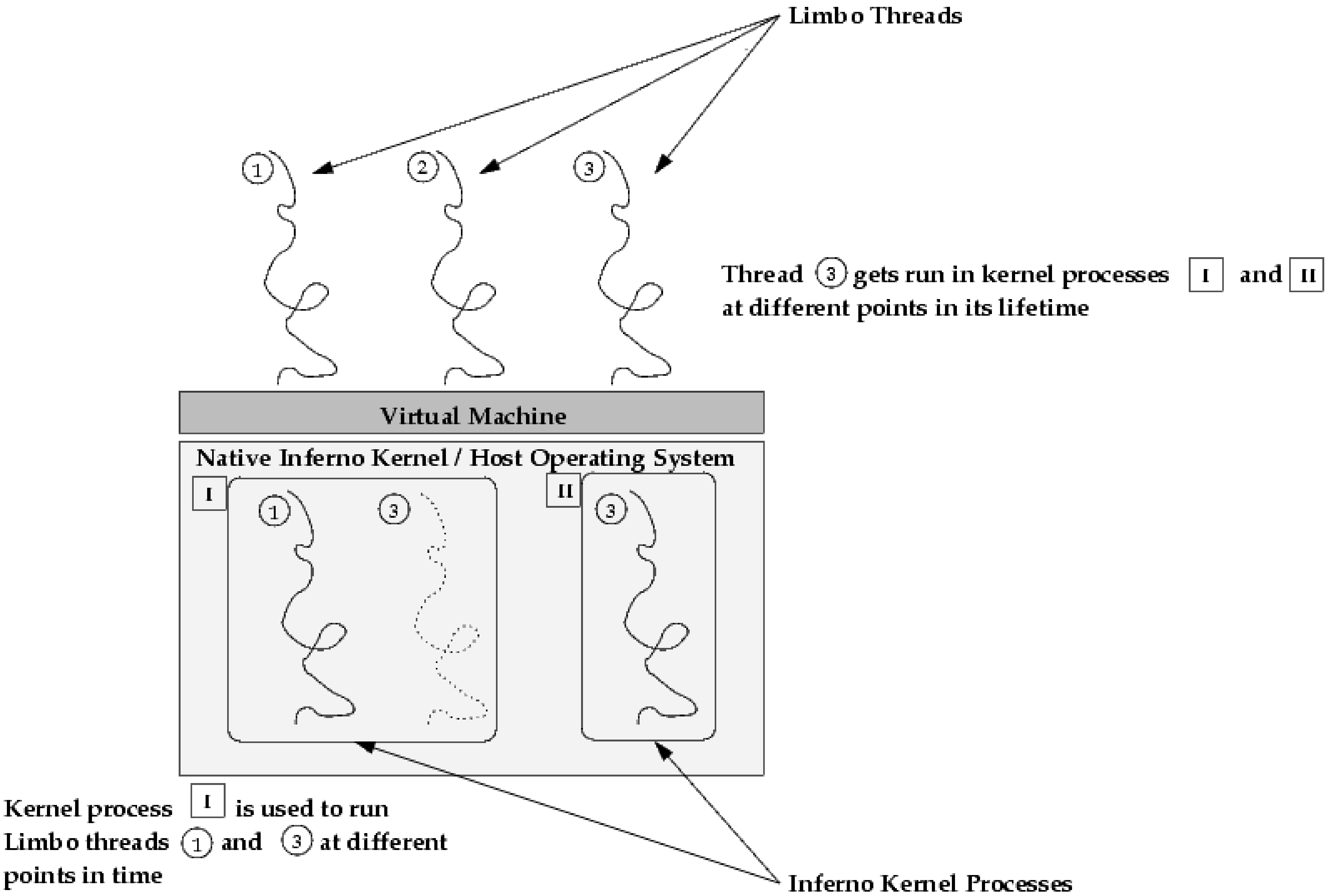
Threads *versus* Processes

- To make the following discussion easier, some terminology:
  - We will use *thread* henceforth to refer to a Limbo thread, executing over the Dis VM
  - We'll use the term *process* to refer to a host OS or native Inferno kernel thread/process, regardless of whether it is implemented as a real process, or using e.g., pthreads

Kernel Processes

- The core of the emulator (Dis VM) executes as a single thread
- New threads may be created in response to actions of device drivers or built-in modules
  - In general, a device drivers will call upon emulator facilities to create a new process if it needs to perform some task offline
  - Example: `sys->export()` with the flag `Sys->EXPASYNC` does this

Limbo Threads and Kernel Processes



Limbo Threads

Thread 3 gets run in kernel processes I and II at different points in its lifetime

Virtual Machine

Native Inferno Kernel / Host Operating System

Kernel process I is used to run Limbo threads 1 and 3 at different points in time

Inferno Kernel Processes

Kernel Source

- Emulator source resides in /os/:

```
/os/
    ipaq1110/
        archipaq.c
        dat.h
        deveia.c
        defont.c
        devaudio.c
        ...
        main.c
```

- Each system architecture directory contains platform specific code for kernel on that host platform
  - Most of the data structures defined in emulators /emu/port/dat.h are in /os/port/portdat.h
  - Each architecture usually defines its dat.h with arch-specific data structures

Supported system architectures

- cerf1110
- cerf405
- fads
- ipaq1110
- ipengine
- js
- ks32
- mpc
- omap
- pc
- rpcg
- sa1110

Kernel source

- The bulk of the kernel source is architecture independent, and is in /os/port/

```
/emu/
    port/
        alarm.c
        alloc.c
        chan.c
        ...
        devaudio.c
        devprog.c
        devssl.c
        taslock.c
```

- Kernel source relies on many routines implemented in the libraries (e.g., libdraw, libinterp, etc), which are shared with emulator

Important Header Files: /os/*archname*/`dat.h`

- Each specific system architecture has its own dat.h, containing architecture specific data structures

Usually contains structures accessed by `l.s`, assembler startup code

- Lock data structures: `struct Lock`
- Machine configuration: `struct Conf`
- Machine state (e.g., CPU speed, time since boot, etc): `struct Mach`

Important Header Files: /os/port/portdat.h

- Important data structures and constants are defined in

  /os/port/portdat.h

  - Defines Chan, Proc, Osenv, Dev, Dirtab (discussed in previous lecture) and other data structures

Important Header Files: dat.h

```
struct Chan
{
        Lock    l;
        Ref     r;
        Chan*   next;                   /* allocation */
        Chan*   link;
        vlong   offset;                 /* in file */
        ushort  type;
        ulong   dev;
        ushort  mode;                   /* read/write */
        ushort  flag;
        Qid     qid;
        int     fid;                    /* for devmnt */
        ulong   iounit;                 /* chunk size for i/o; 0==default */
        Mhead*  umh;          /* mount point that derived Chan; used in unionread */
        Chan*   umc;                    /* channel in union; held for union read */
        QLock   umqlock;                /* serialize unionreads */
        int     uri;                    /* union read index */
        int     dri;                    /* devdirread index */
        ulong   mountid;
        Mntcache *mcp;                  /* Mount cache pointer */
        Mnt     *mux;                   /* Mnt for clients using me for messages */
        void*   aux;                    /* device specific data */
        Chan*   mchan;                  /* channel to mounted server */
        Qid     mqid;                   /* qid of root of mount point */
        Cname   *name;
};
```

Chan structure : used to manage communication between *Mount Driver* (recall, #M) and device drivers

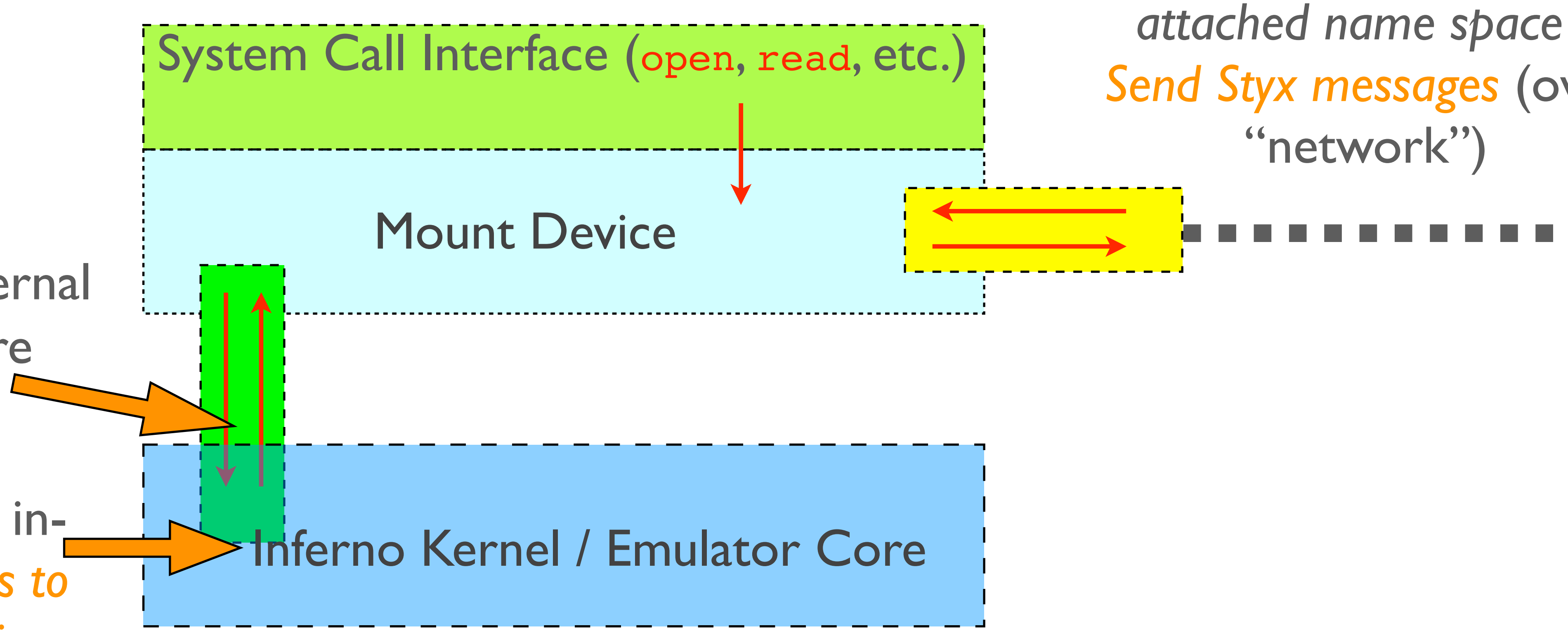
Important Header Files: dat.h

```
struct Dev
{
        int     dc;
        char*   name;

        void    (*init)(void);
        Chan*   (*attach)(char*);
        Walkqid*        (*walk)(Chan*, Chan*, char**, int);
        int     (*stat)(Chan*, uchar*, int);
        Chan*   (*open)(Chan*, int);
        void    (*create)(Chan*, char*, int, ulong);
        void    (*close)(Chan*);
        long    (*read)(Chan*, void*, long, vlong);
        Block*  (*bread)(Chan*, long, ulong);
        long    (*write)(Chan*, void*, long, vlong);
        long    (*bwrite)(Chan*, Block*, ulong);
        void    (*remove)(Chan*);
        int     (*wstat)(Chan*, uchar*, int);
};
```

Pointers to functions to be called for various Styx operations

Remember The *Mount Device*, #M ?


System Call Interface (open, read, etc.)
Mount Device
Inferno Kernel / Emulator Core

Is name part of a remotely attached name space ? *Send Styx messages* (over "network")

Inferno Kernel Internal Chan* structure

Eventually end up as in-kernel *subroutine calls to appropriate device driver*

- 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*

```
struct Proc
{
        int     type;           /* interpreter or not */
        char    text[KNAMELEN];
        Proc*   qnext;          /* list of processes waiting on a Qlock */
        long    pid;
        Proc*   next;           /* list of created processes */
        Proc*   prev;
        Lock    rlock;          /* sync between sleep/swiproc for r */
        Rendez* r;              /* rendezvous point slept on */
        Rendez  sleep;          /* place to sleep */
        int     killed;         /* by swiproc */
        int     swipend;        /* software interrupt pending for Prog */
        int     syscall;        /* set true under sysio for interruptable syscalls */
        int     intwait;        /* spin wait for note to turn up */
        int     sigid;          /* handle used for signal/note/exception */
        Lock    sysio;          /* note handler lock */
        char    genbuf[128];    /* buffer used e.g. for last name element from namec */
        int     nerr;           /* error stack SP */
        osjmpbuf estack[NERR];  /* vector of error jump labels */
        char*   kstack;
        void    (*func)(void*); /* saved trampoline pointer for kproc */
        void*   arg;            /* arg for invoked kproc function */
        void*   iprog;          /* work for Prog after release */
        void*   prog;           /* fake prog for slaves eg. exportfs */
        Osenv*  env;            /* effective operating system environment */
        Osenv   defenv;         /* default env for slaves with no prog */
        osjmpbuf        privstack;      /* private stack for making new kids */
        osjmpbuf        sharestack;
        Proc    *kid;
        void    *kidsp;
        void    *os;            /* host os specific data */
};
```

Important Header Files: dat.h

Compiling a Kernel

- Native Inferno kernels are not compiled with gcc
  - Compiled with the Plan 9 compiler toolchain,e.g., for 386, 8a, 8c, 8l
    - 8a — The assembler (also, 5a (arm), qa (powerpc) etc.)
    - 8c — The C compiler (also 5c, (arm), qc (powerpc) etc.)
    - 8l — The linker/loader, but also does some optimization
- Implementation uses some features outside ANSI C
  - Unnamed union substructures
  - Unnamed function parameters

Kernel Config file

- Kernel config file (format as in emulator config file discussed in previous lecture)
- Parsed by the several shell scripts to fill out the mkfile, create table of device drivers, etc.

Example: Compiling a native kernel

Next

- Kernel initialization/startup sequence

Fin.