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

Phillip Stanley-Marbell
pstanley@ece.cmu.edu

Lecture Outline

- Emulator Overview
- Terminology: Limbo Threads versus Emulator Processes
- Emulator data structures

The Inferno Emulator, *emu*

- The Inferno emulator is an application that runs unprivileged over a host OS
- It emulates the whole Inferno OS, from the virtual machine down to device drivers (*inclusive*)
  - Most device drivers available in native Inferno are duplicated in the emulator
  - Device drivers in emulator are not real "device drivers", i.e., they do not drive actual hardware
  - Device drivers in emulator provide (almost) the same interface as native drivers, but call on host OS to do the dirty work

Emulator Components

- Virtual machine
- Built-in modules
- Device drivers
- Facilities
  - Thread / process creation (depends on host OS model)
  - Synchronization primitives
  - Memory management primitives
  - More on emulator threads follows...

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

Emulator Processes

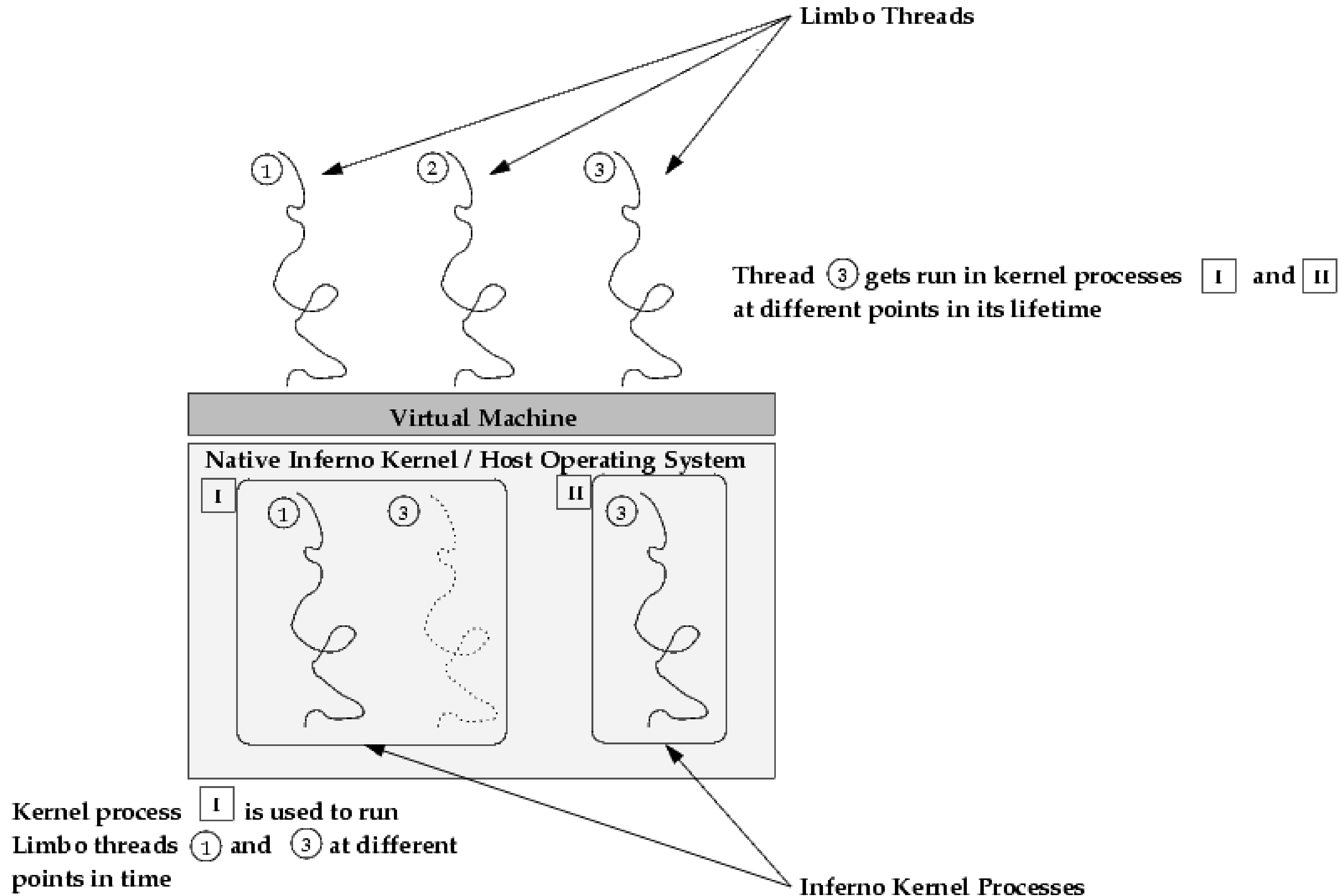
- The core of the emulator (Dis VM) executes as a single thread

- New threads may be created in response to actions of device drivers or built-in modules

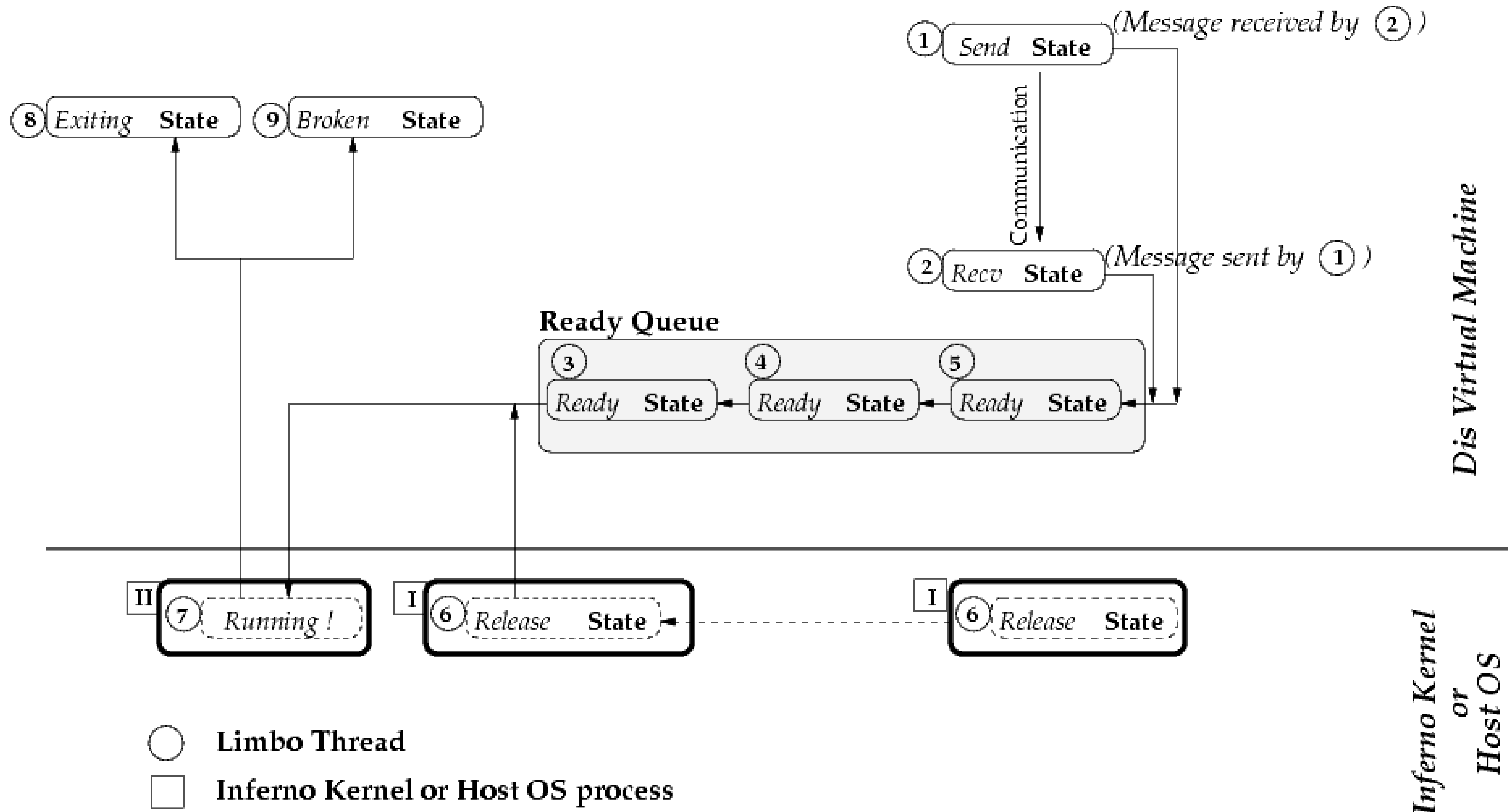
  - In general, a device drivers will call upon emulator facilities to create a new process if it needs to perform some task offline

  - Example: `sys->export()` with the flag `Sys->EXPASYNC` does this

Limbo Threads and Emulator Processes



Limbo Threads and Emulator Processes



Emulator Source

- Emulator source resides in /emu/

```
/emu/
    MacOSX/
        asm-power.s
        cmd.c
        deveia.c
        devfs.c
        ipif.c
        os.c
```

- Each system architecture directory contains platform specific code for emulator on that host platform
  - Code for creating processes etc. (in os.c)
  - Interacting with host's filesystem (devfs.c)
  - Accessing host's network protocol stack (ipif.c), etc.

Emulator source

- The bulk of the emulator source is architecture independent, and is in /emu/port/

```
/emu/
    port/
        audio.h
        ...
        devprog.c
        devssl.c
        win-xll.c
```

  - In general, throughout source tree, architecture independent (or *portable*) code is placed in a directory called port/

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

Important Header Files: `dat.h`, `fns.h`, `error.h`

- Important data structures and constants are defined in `/emu/port/dat.h`
- Function prototype definitions are in `/emu/port/fns.h`
- Error message extern declarations are in `/emu/port/error.h`
- Most device drivers and built-in modules will include all three

Important Header Files: dat.h

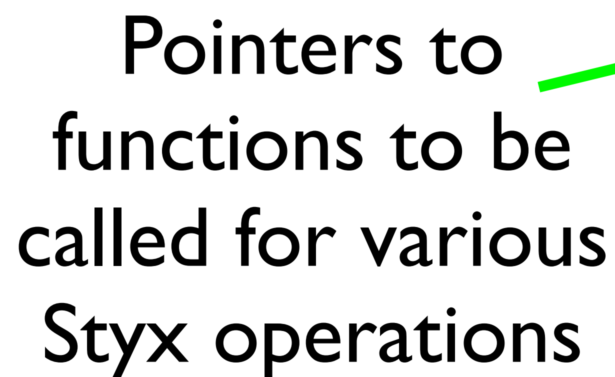
```
struct Chan
{
        Lock     l;
        Ref      r;
        Chan*    next;                 /* allocation */
        Chan*    link;
        vlong    offset;               /* in file */
        ushort   type;
        ulong    dev;
        ushort   mode;                 /* read/write */
        ushort   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

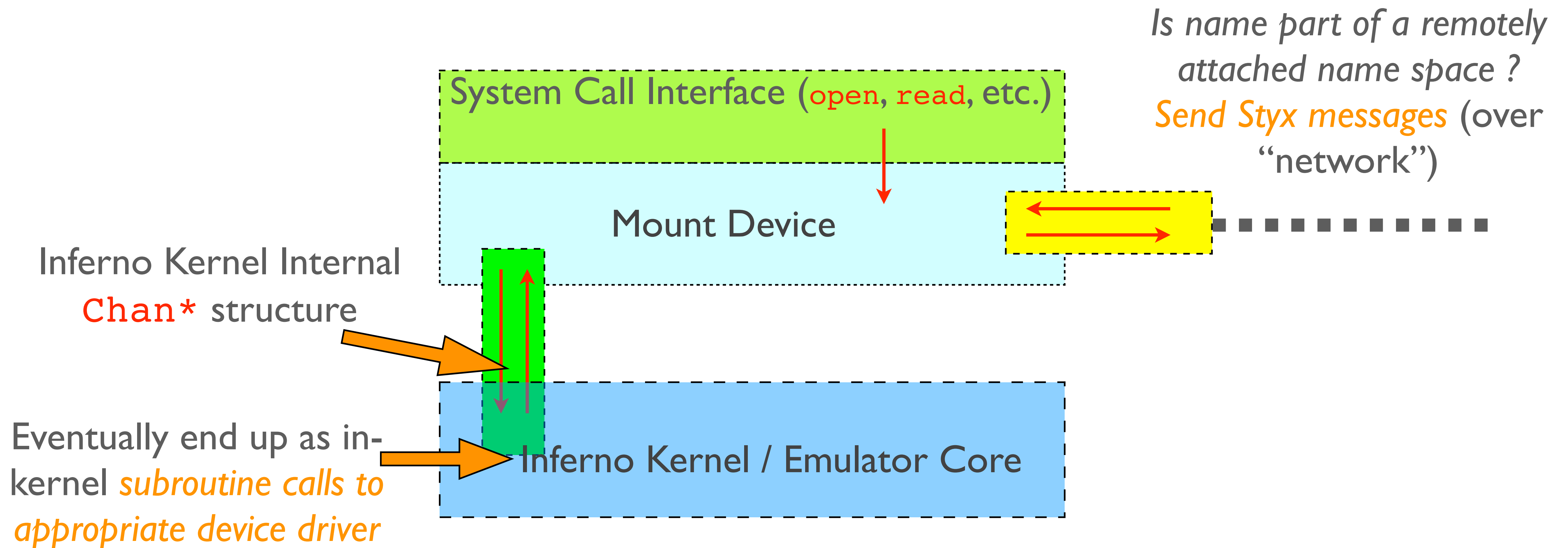
Important Header Files: dat.h

```
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);
};
```



Remember The *Mount Device*, *#M* ?



- 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*

Important Header Files: dat.h

```
struct Dirtab
{
        char    name[KNAMELEN];
        Qid     qid;
        vlong   length;
        long    perm;
};
```

The `Dirtab` structure is used to represent information about files and directories.

Recall the Styx RSTAT message (also, remember possible project topic)

```
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

Important Header Files: dat.h

```
struct Osenv
{
	char	*syserrstr;	/* last error from a system call, errbuf0 or 1 */
	char	*errstr;	/* reason we're unwinding the error stack, errbuf1 or 0 */
	char	errbuf0[ERRMAX];
	char	errbuf1[ERRMAX];
	Pgrp*	pgrp;		/* Ref to namespace, working dir and root */
	Fgrp*	fgrp;		/* Ref to file descriptors */
	Egrp*	egrp;	/* Environment vars */
	Skeyset*		sigs;		/* Signed module keys */
	Rendez*	rend;		/* Synchro point */
	Queue*	waitq;		/* Info about dead children */
	Queue*	childq;		/* Info about children for debuggers */
	void*	debug;		/* Debugging master */
	char*	user;	/* Inferno user name */
	FPU	fpu;		/* Floating point thread state */
	int	uid;		/* Numeric user id for host system */
	int	gid;		/* Numeric group id for host system */
	void	*ui;		/* User info for NT */
};
```

Important Header Files: fns.h

```
ulong   FPcontrol(ulong,ulong);
ulong   FPstatus(ulong,ulong);
void            FPsave(void*);
void            FPrestore(void*);
void            Sleep(Rendez*, int (*)(void*), void*);
int             Wakeup(Rendez*);
void            FPinit(void);
void            addprog(Proc*);
Block*  adjustblock(Block*, int);
Block*  allocb(int);
Block*  bl2mem(uchar*, Block*, int);
char*   c2name(Chan*);
int             canlock(Lock*);
int             canqlock(QLock*);
...
long            devbwrite(Chan*, Block*, ulong);
void            devcreate(Chan*, char*, int, ulong);
void            devdir(Chan*, Qid, char*, long, char*, long, Dir*);
long            devdirread(Chan*, char*, long, Dirtab*, int, Devgen*);
void            devinit(void);
...
Chan*   devattach(int, char*);
Block*  devbread(Chan*, long, ulong);
Chan*   devclone(Chan*);
Devgen  devgen;
```

Example: Compiling the Emulator

Reading

- Relevant chapters of the book: Chapter 6
- Do homework 2 !
- Read the document *nativeinferno.pdf* on blackboard that describes building the native kernel and making a bootdisk
    - This may help clarify-reinforce lecture 7 as well as homework 2 question 1.

Next

- Emulator / Kernel device driver interface

Fin.