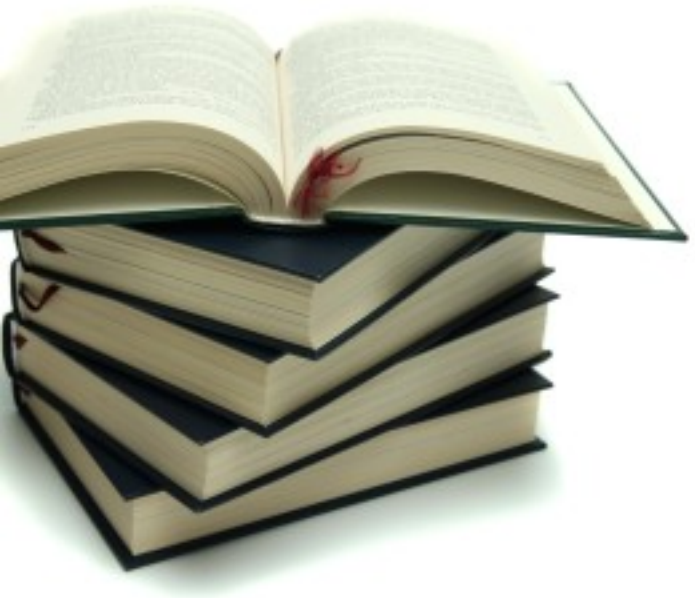
Test Curs 9

- Ce operație se face în cadrul rutinei de tratare pentru o întrerupere de pachet trimis?
- Care este numărul minim de paşi necesar pentru a ruta o adresă IPv4 prin ruta default, dacă adresa nu este în cache-ul de rute şi avem cel puțin o rută pentru fiecare din măştile de reațea posibile ?
- Ce operație skb se foloseşte în procesul de încapsulare al unui pachet? Dar pentru decapsulare?

Cursul 10



10

Linux kernel library

6 mai 2010

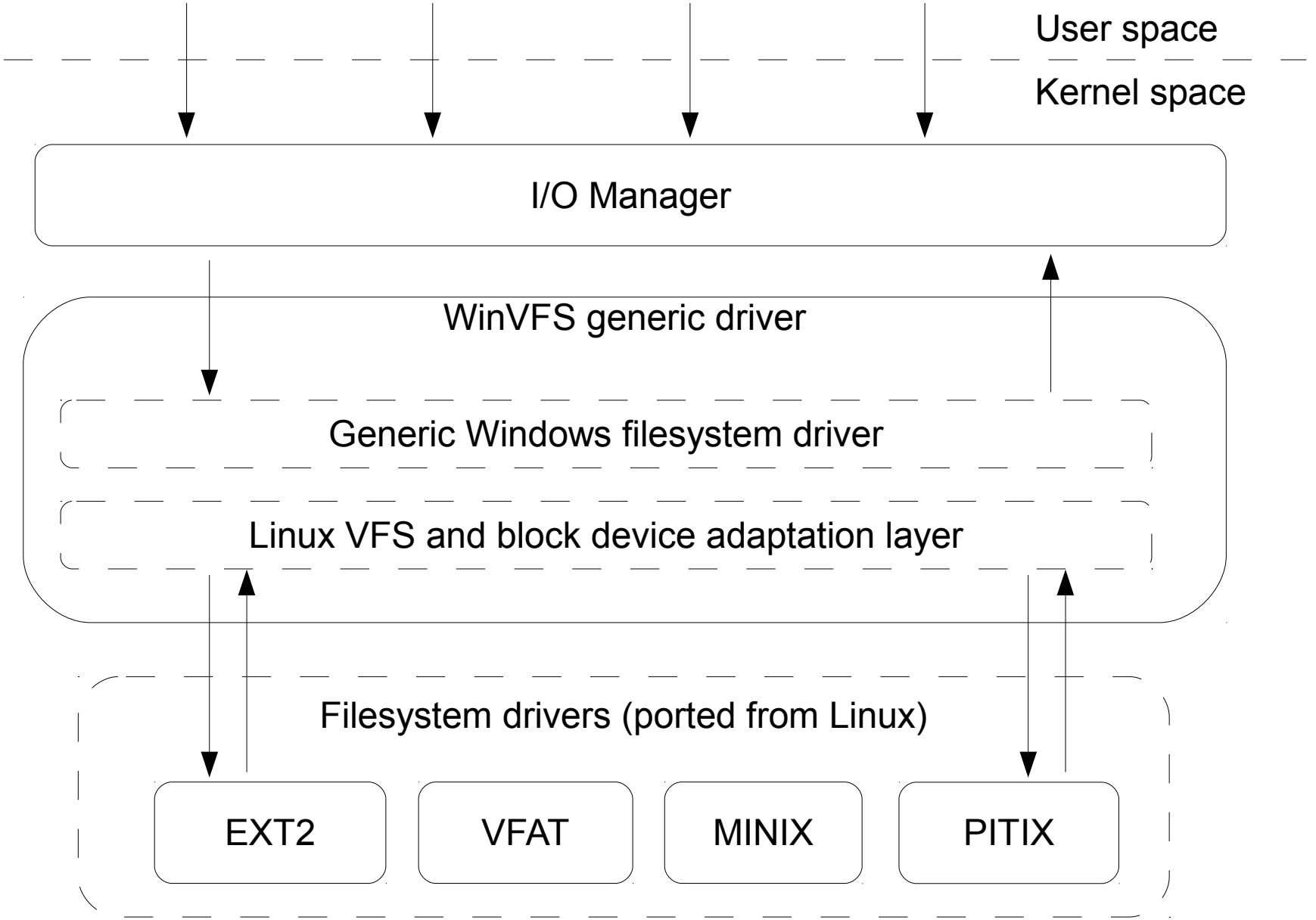
Linux kernel library

- Initially started
  - As a revival of the WinVFS project but in such a way that we could keep up with the Linux change rate
  - To test a new idea: FTP server as a portable way of offering access to Linux filesystems
- Ended up
  - An infrastructure which allows one to reuse generic Linux kernel code
- Related areas
  - UML
  - CoLinux
  - FuSE, Ndiswrapper
  - Paravirtualization

WinVFS

- Create an Windows ext2 driver as well drivers for other Linux filesystems (reiserfs)
- Completely reuse Linux filesystem drivers: just recompile the driver source code
- WinVFS = the infrastructure needed for complete code reutilization

WinVFS architecture



Proof of concept results

- Generic filesystem driver:
  - Read-only support only
- Adaptation layers:
  - partial porting, partial reimplementation of the VFS primitives needed by drivers
  - A lot of the generic Linux code was pulled in because of VFS dependencies on various subsystems
- Drivers ported: ext2, minix, vfat
  - Trivial source code modification required (compiler related)

s2 kernel crunch

Overview of bits that got pulled in

bitops.h config.h errno.h fd.h init.h minix_fs.h msdos_fs.h quota.h

slab.h time.h blkdev.h ctype.h ext2_fs.h file.h ioctl.h minix_fs_i.h

msdos_fs_i.h quotaops.h smp_lock.h types.h blk.h ext2_fs_i.h

fs.h kdev_t.h minix_fs_sb.h msdos_fs_sb.h rwsem.h spinlock.h

wait.h byteorder dcache.h ext2_fs_sb.h fs_struct.h kernel.h

mm.h nls.h rwsem-spinlock.h stat.h capability.h dirent.h

fat_cvf.h highmem.h list.h module.h pagemap.h sched.h

stddef.h compiler.h dnotify.h fcntl.h highuid.h locks.h mount.h

posix_types.h semaphore.h string.h ./mm/page_alloc.c

./mm/kmem_cache.c ./mm/filemap.c ./fs/inode.c ./fs/file_table.c

./fs/attr.c ./fs/namespace.c ./fs/bad_inode.c ./fs/dcache.c

./fs/namei.c ./fs/buffer.c ./fs/readdir.c ./fs/open.c ./fs/super.c ./fs/block_dev.c

./fs/read_write.c ./fs/devices.c ./lib/vsprintf.c ./lib/string.c ./lib/ctype.c

Later developments & decline

- Switched to mingw
- Switched to 2.6 kernel
- TotalCommander plugins
- Security attributes: Linux – Windows adaptation
- We proved it is possible to completely reuse Linux filesystem drivers code to create Windows drivers
- Switching to 2.6 posed significant challenges
- Keeping track with 2.6 development became impractical

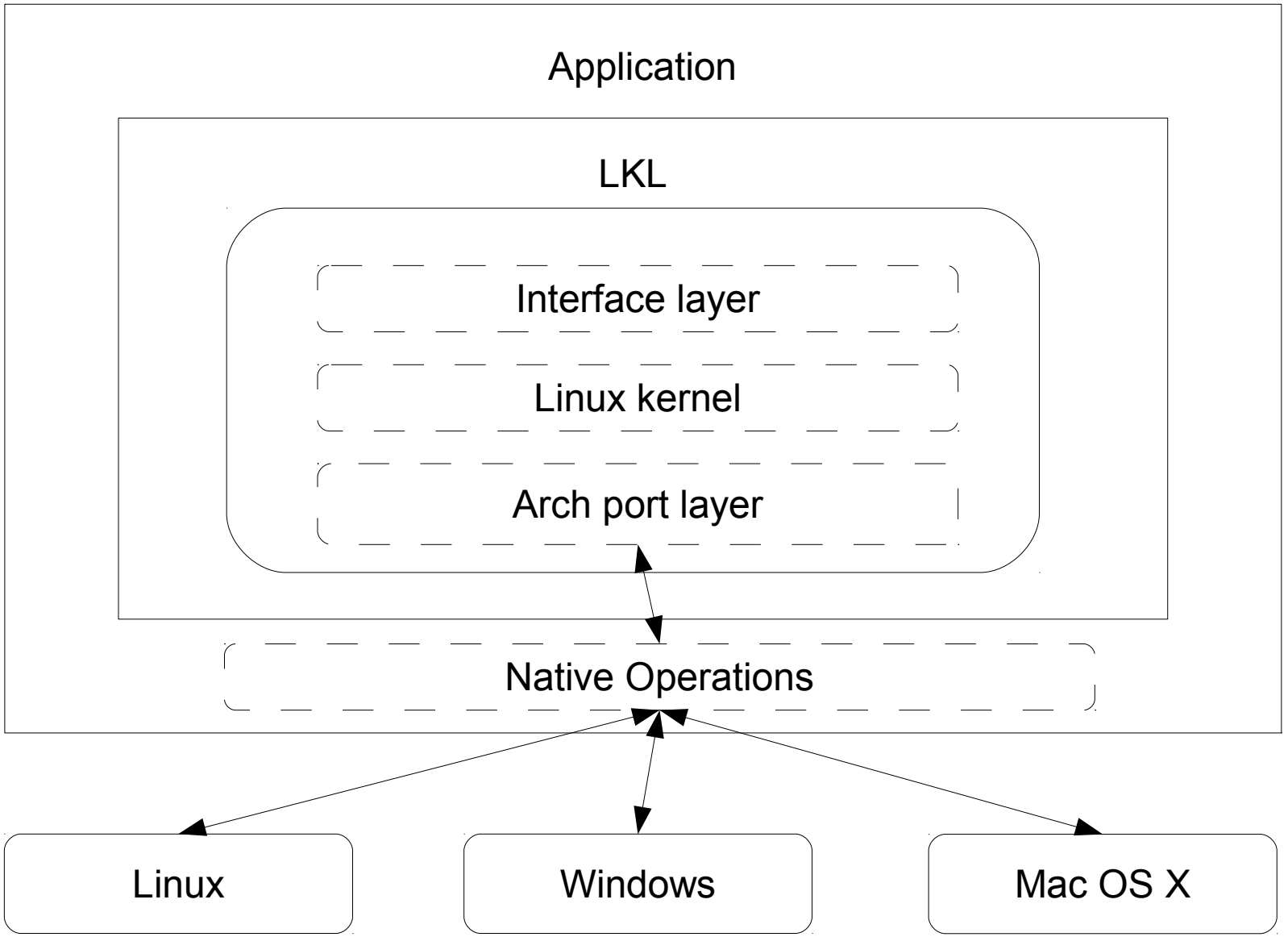
LKL goals

- Allow applications to reuse code from the Linux kernel without needing to separate, isolate and extract it from Linux
- Run in as diverse environments as possible: cross OS, cross platforms, both kernel and user
- Allow full Linux subsystem to be reuse (e.g. filesystem drivers, TCP/IP stack)
- Linux kernel modifications should be isolated (for easy tracking of Linux kernel development)
- Easy to use (from application point of view)

LKL design decisions

- Make it a library
- The library should contain the full Linux kernel
- Highly customizable – make menuconfig
- Implement it as a new arch port layer
- API based on the Linux system call interface
- Offload some operations to application
- No user / kernel separation or abstractions

Architecture



Native operations

- Offers services needed by the Linux kernel (e.g. memory management, thread management, time management, etc.)
- By design, the operations are basic and as generic as possible
- It is the role of the arch port layer to map these operations to the services required by the Linux kernel

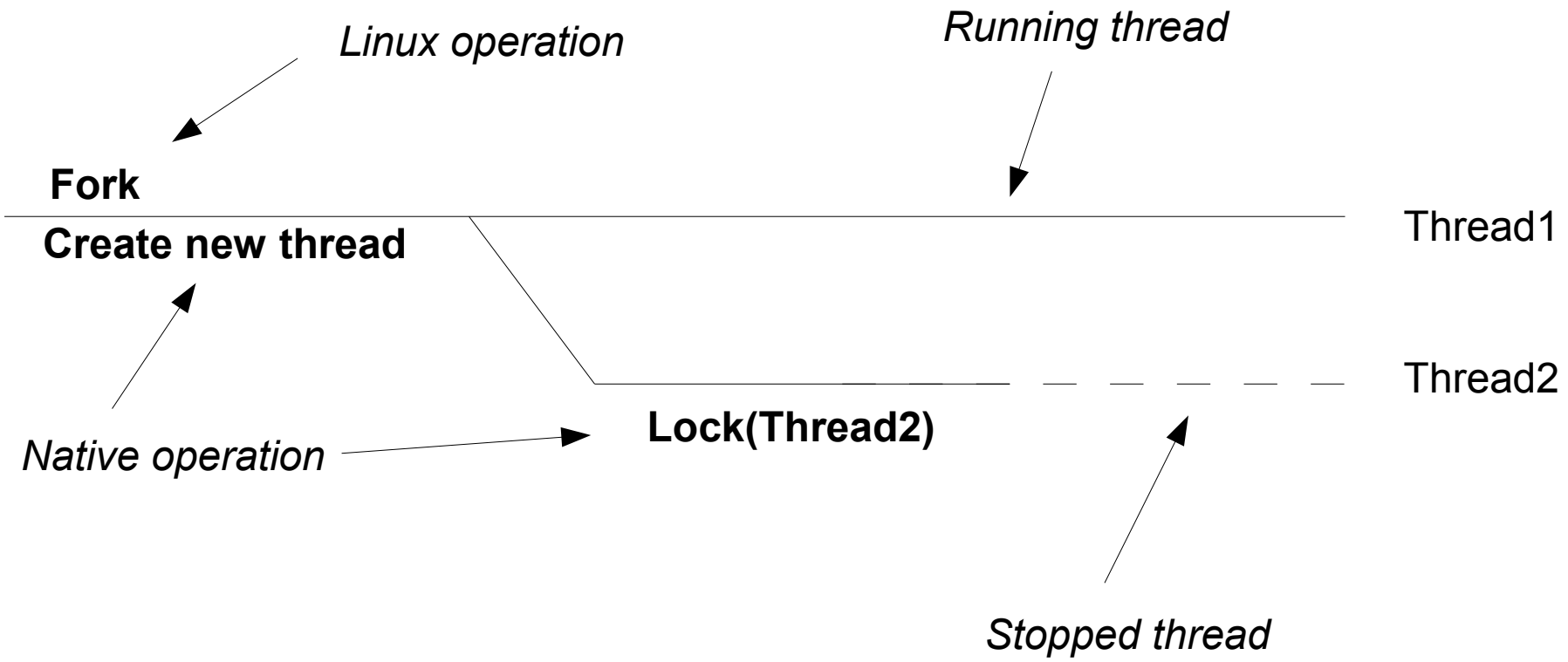
Memory management

- lkl arch is a “non-MMU” arch
- “Physical memory” allocated by the native environment
- Initially: allocate the whole physical memory during initialization
- Later: use native operations to allocate memory
  - Hot plug memory

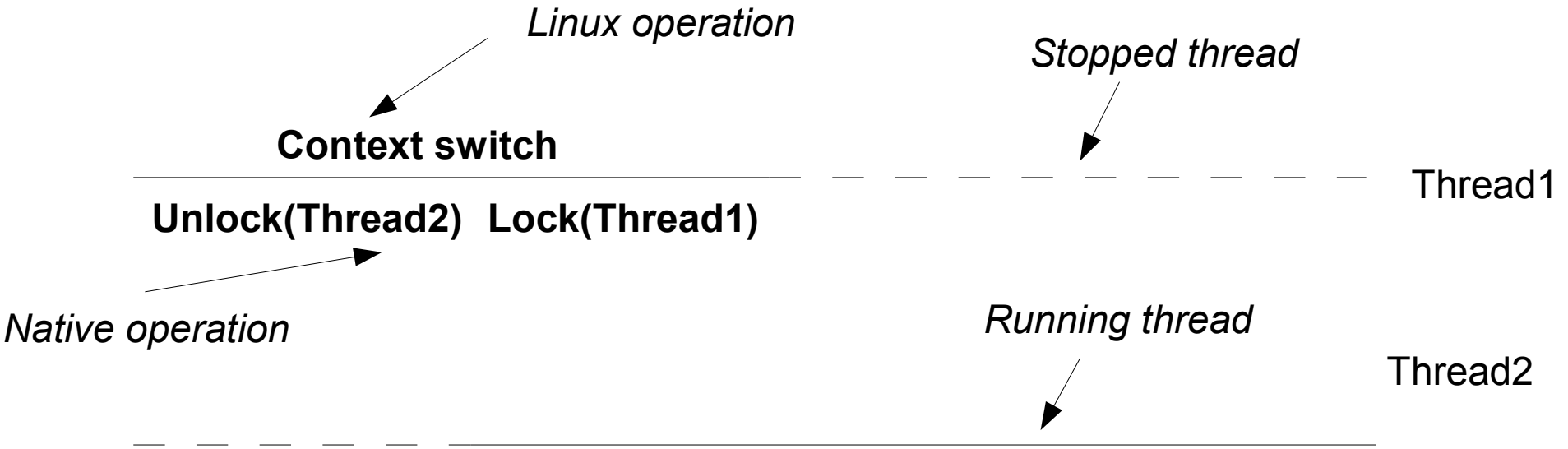
Thread support

- No need for user processes, but...
- We need to support kernel threads
- Micro/internal LKL threading – discarded
- Support from the execution environment
- Put the Linux scheduler in control:
  - Each thread has a control semaphore
  - Native operations for semaphore control

Threads control – forking



Threads control – context switch



Drivers

- LKL needs drivers to interact with the exterior
  - Native part - “the hardware”
  - Linux part – a Linux device driver
- How do we communicate between the two parts?
  - Linux -> Native: direct function calls
  - Native -> Linux: “interrupts”
- Why interrupts?
  - The simplest way of running Linux code in the proper context

Examples of drivers

- Disc driver
  - “Hardware” = file, partition
  - “Hardware” = device object
- Network driver
  - “Hardware” = interface
  - “Hardware” = socket
- Timer driver
- Console driver

Interrupts

- The application can trigger IRQs
- The Linux kernel will pick it up and run the associated interrupt handler
- LKL does not support SMP
- We need to serialize the interrupt handler routines with the rest of the kernel
- Run them from the idle thread
  - Whenever the Linux has nothing to do it runs the idle thread
  - Waits on a semaphore until an interrupt is generated

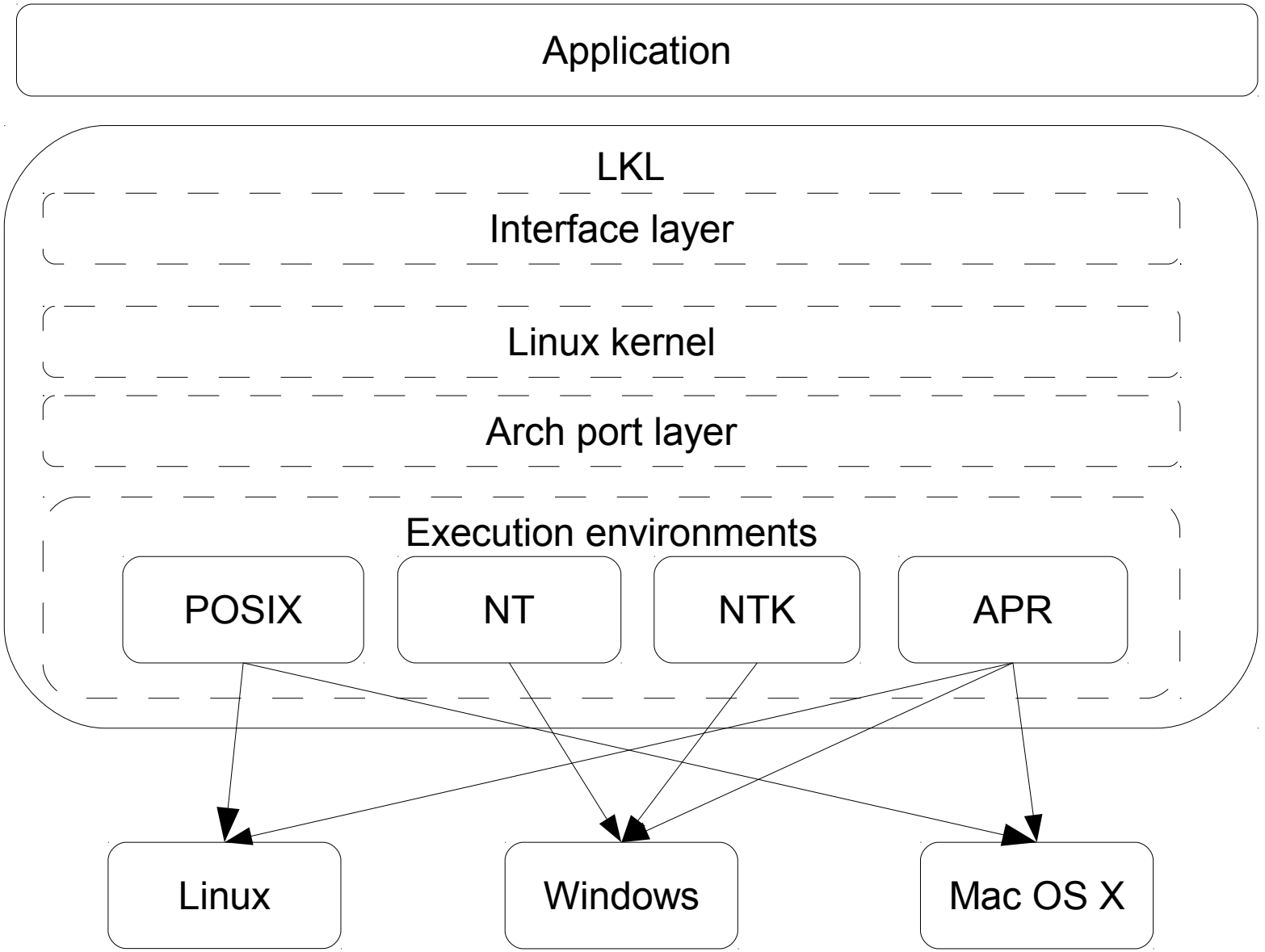
Time management

- Essential for proper kernel functioning
  - TCP/IP timers
  - RCU synchronization
- Supported with two native operations: time and timer
- time() – returns the current time
- timer() – setups a native timer which triggers IRQ_TIMER
- LKL uses NO_HZ

Native operations summary

- void (*print)(const char *str, int len);
- long (*panic_blink)(long time);
- void* (*sem_alloc)(int count);
- void (*sem_free)(void *sem);
- void (*sem_up)(void *sem);
- void (*sem_down)(void *sem);
- void* (*thread_create)(void (*f)(void*), void *arg);
- void (*thread_exit)(void *thread);
- void* (*thread_id)(void);
- void* (*mem_alloc)(unsigned int);
- void (*mem_free)(void *);
- void (*timer)(unsigned long delta);
- unsigned long long (*time)(void);
- int (*init)(void);
- void (*halt)(void);

Execution environments



NTK: semaphore operations

```
static void* sem_alloc(int count)
{
    KSEMAPHORE *sem=ExAllocatePool(PagedPool, sizeof(*sem));
    if (!sem) return NULL;
    KeInitializeSemaphore(sem, count, 100);
    return sem;
}

static void sem_up(void *sem)
{
    KeReleaseSemaphore((KSEMAPHORE*)sem, 0, 1, 0);
}

static void sem_down(void *sem)
{
   KeWaitForSingleObject((KSEMAPHORE*)sem, Executive, KernelMode,
                         FALSE, NULL);
}

static void sem_free(void *sem)
{
    ExFreePool(sem);
}
```

NTK: threads & mem operations

```
static void* thread_create(void (*fn)(void*), void *arg)
{
    void *thread;
    if (PsCreateSystemThread(&thread, THREAD_ALL_ACCESS, NULL,
                             NULL, NULL, (void DDKAPI (*)(void*))fn,
                             arg) != STATUS_SUCCESS)
        return NULL;
    return thread;
}

static void thread_exit(void *arg)
{
    PsTerminateSystemThread(0);
}

static void* mem_alloc(unsigned int size)
{
    return ExAllocatePool(NonPagedPool, size);
}

static void mem_free(void *data)
{
    ExFreePool(data);
}
```

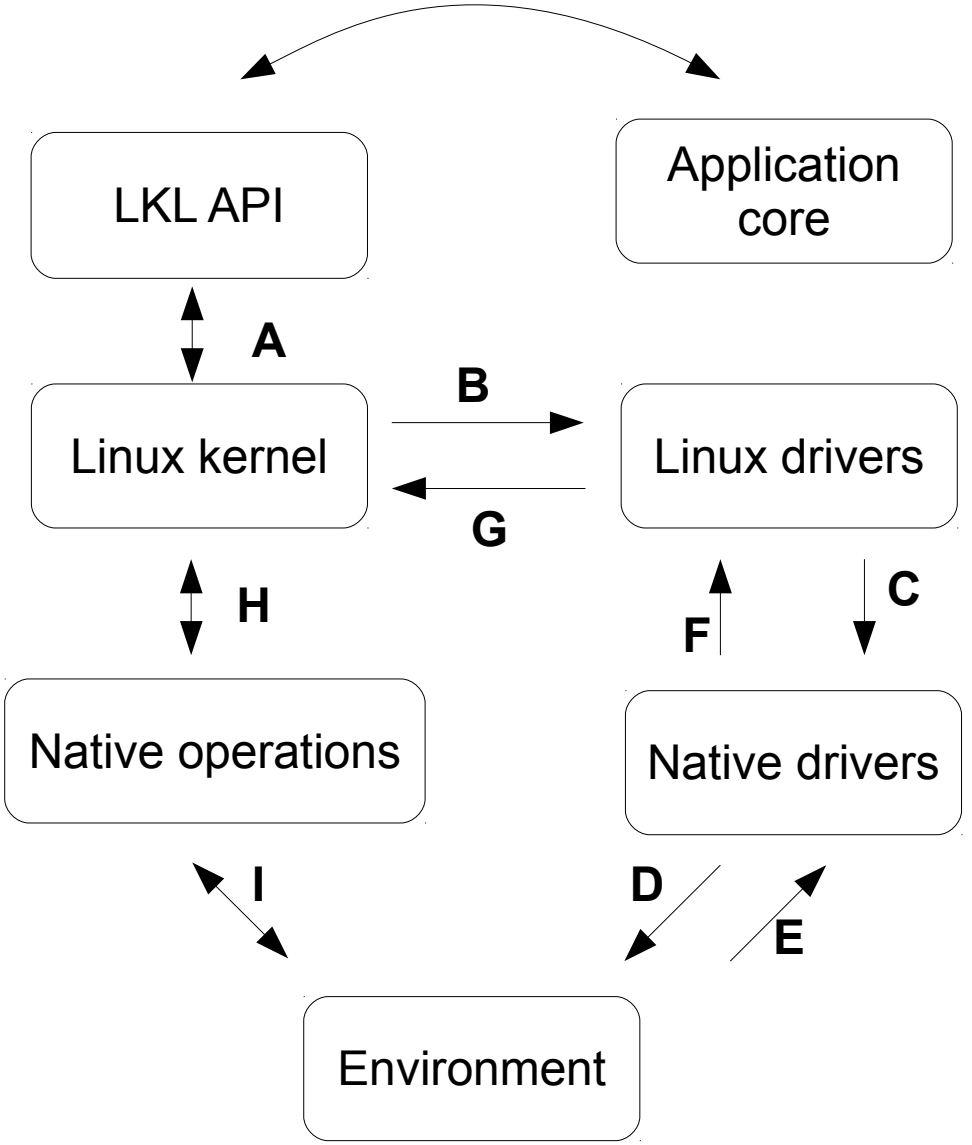
Timer complications

- NT does not have an async notification mechanism
- POSIX does but we can't trigger IRQs from signal handlers
- Timer thread
  - POSIX/APR: Poll on pipe
  - NT/NTK: wait on an event object

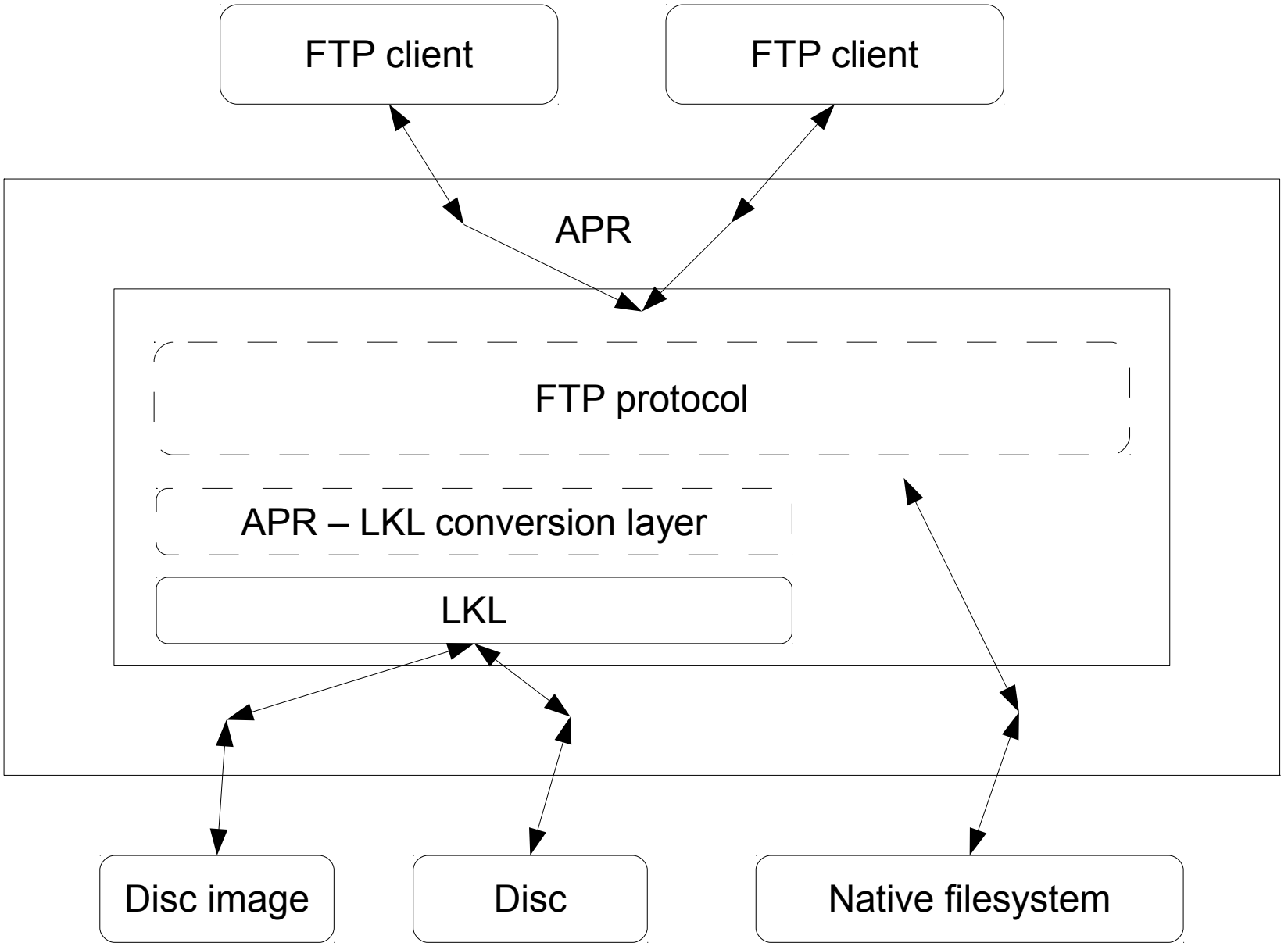
Interface layer

- The application can't call directly the API – Linux functions needs to run in Linux context
- System calls
  - Application triggers IRQ_SYSCALL
  - The interrupt handler schedules the system call in a special kernel thread (ksyscalld)
  - Waits on a semaphore for the system call to be finished
- In multi-threaded application only one system call can be sleeping at a time
- API to create additional syscall kernel threads

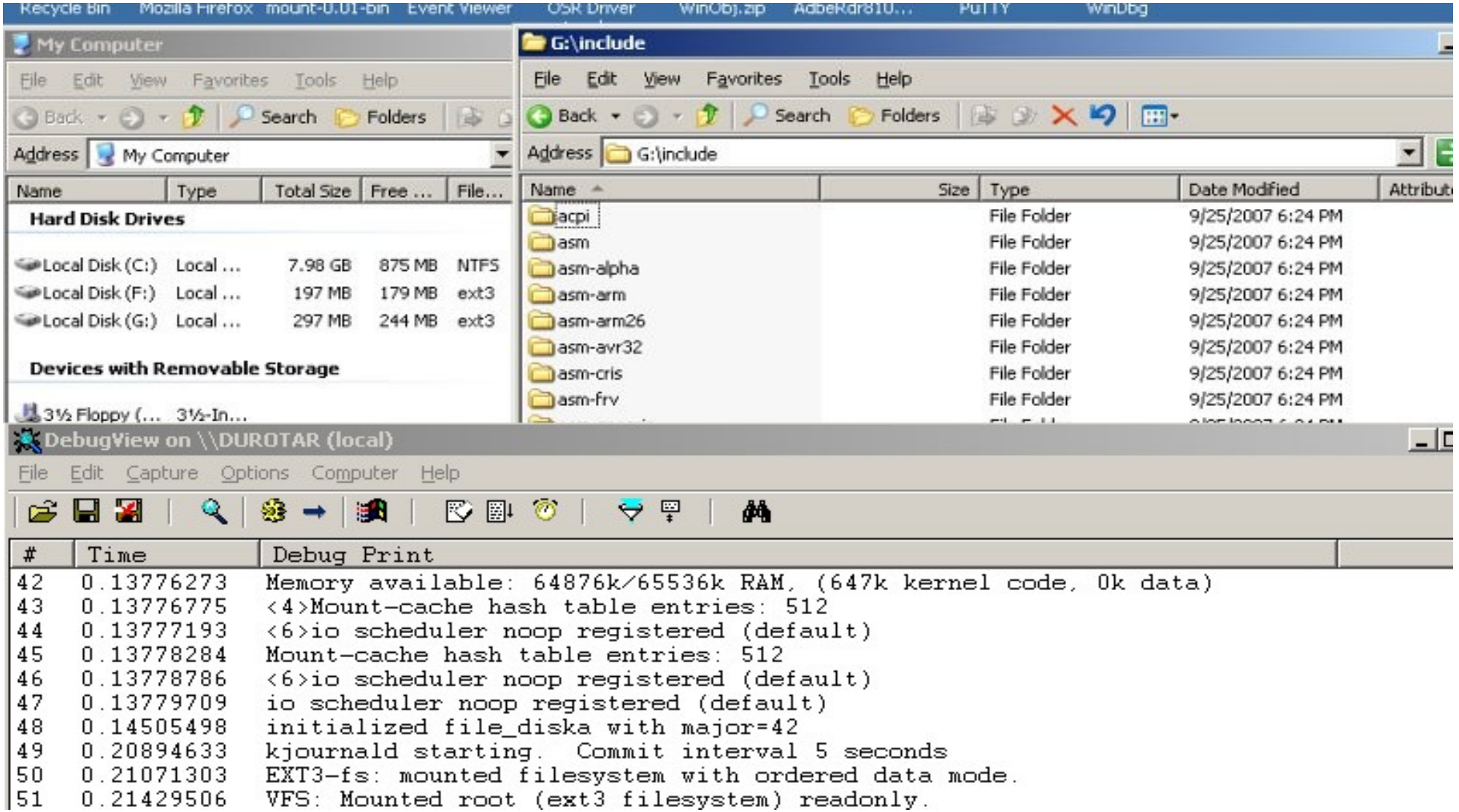
Interactions



FTP server for Linux FS access



Windows driver for Linux FS



Other neat ideas

- Run Valgrind's memcheck on kernel code
  - New SL*B allocator – allows Valgrind to get in the loop
  - TODO: page allocator
- „HTTP” client
  - Reuses the Linux TCP/IP stack
  - Coverage test for Linux's softirq subsystem
  - Tested on PPC
  - Native:LKL performance 4:1
- LUA-LKL
- Network simulator

Conclusions

- The model allows Linux code reutilization across OS, platforms, kernel/user spaces
- It allows us to keep up with the Linux change rate
- Implementing a new execution environment is easy
- Using it to develop applications is easy

http://github.com/lkl