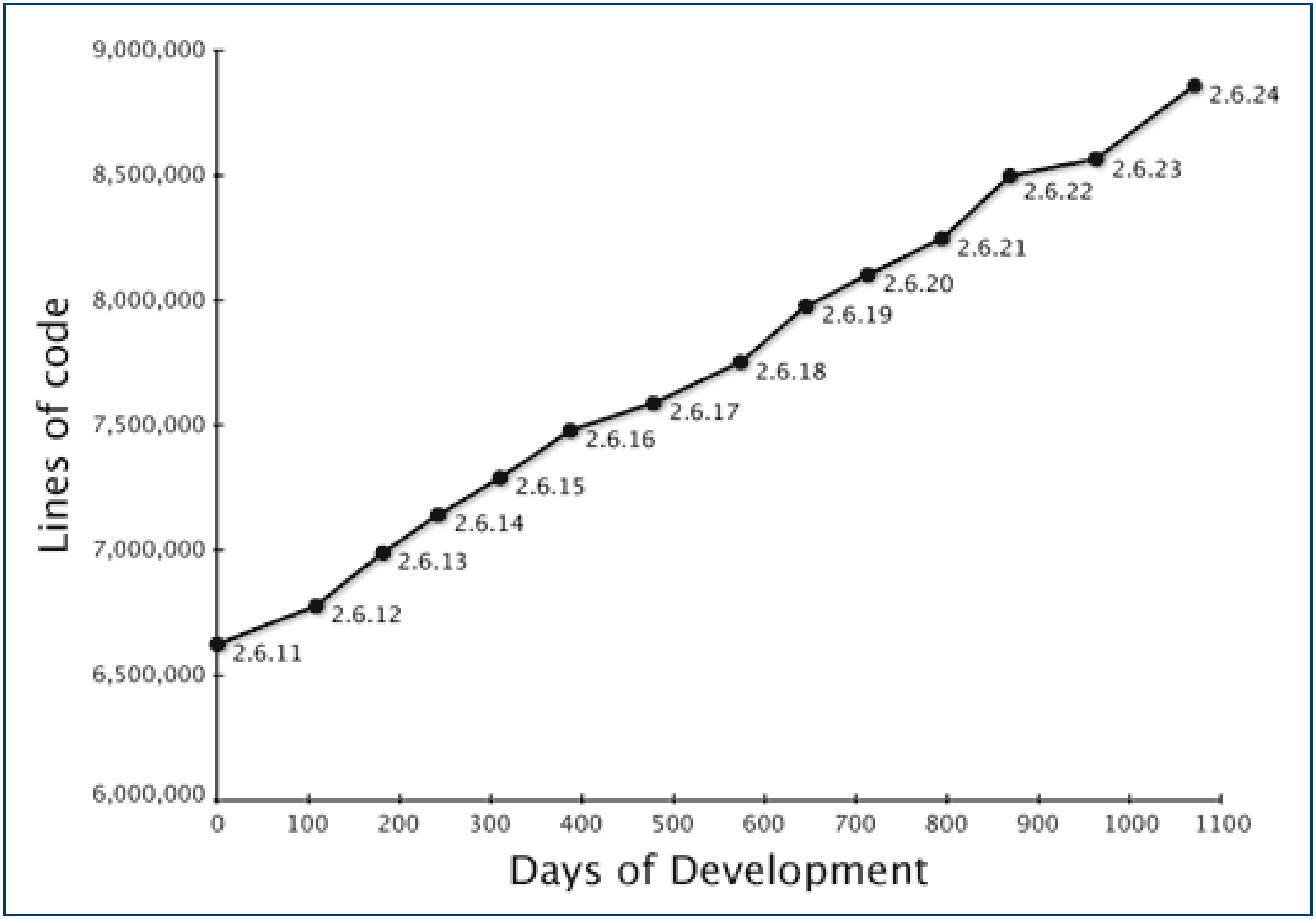
Linux Kernel Overview

Manolis Marazakis
Michail Flouris
FORTH-ICS (CARV Lab)

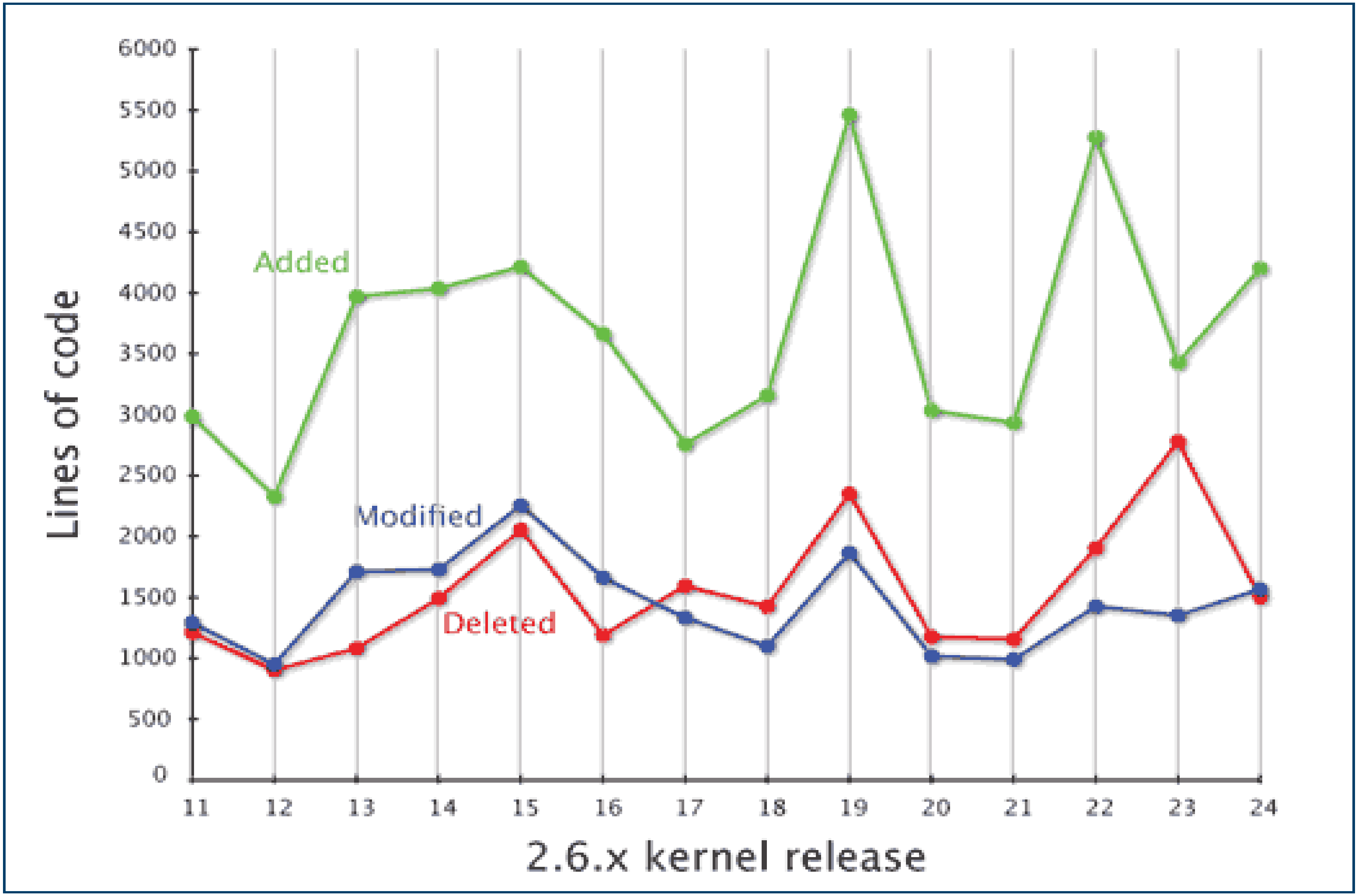
References

- J. Corbet, A. Rubini, G. Kroah-Hartman: Linux Device Drivers (O’Reilly, 3rd edition)
- R. Love: Linux Kernel Development (Novell, 2nd edition)
- C. Hallinan: Embedded Linux Primer: A Practical Real-World Approach (Prentice Hall)
- S. Venkateswaran: Essential Linux Device Drivers (Prentice Hall)
- LWN.net
- LXR – Linux Cross Reference: http://lxr.linux.no/
- http://www.linuxfoundation.org/
- LKML.org

Linux Kernel: size

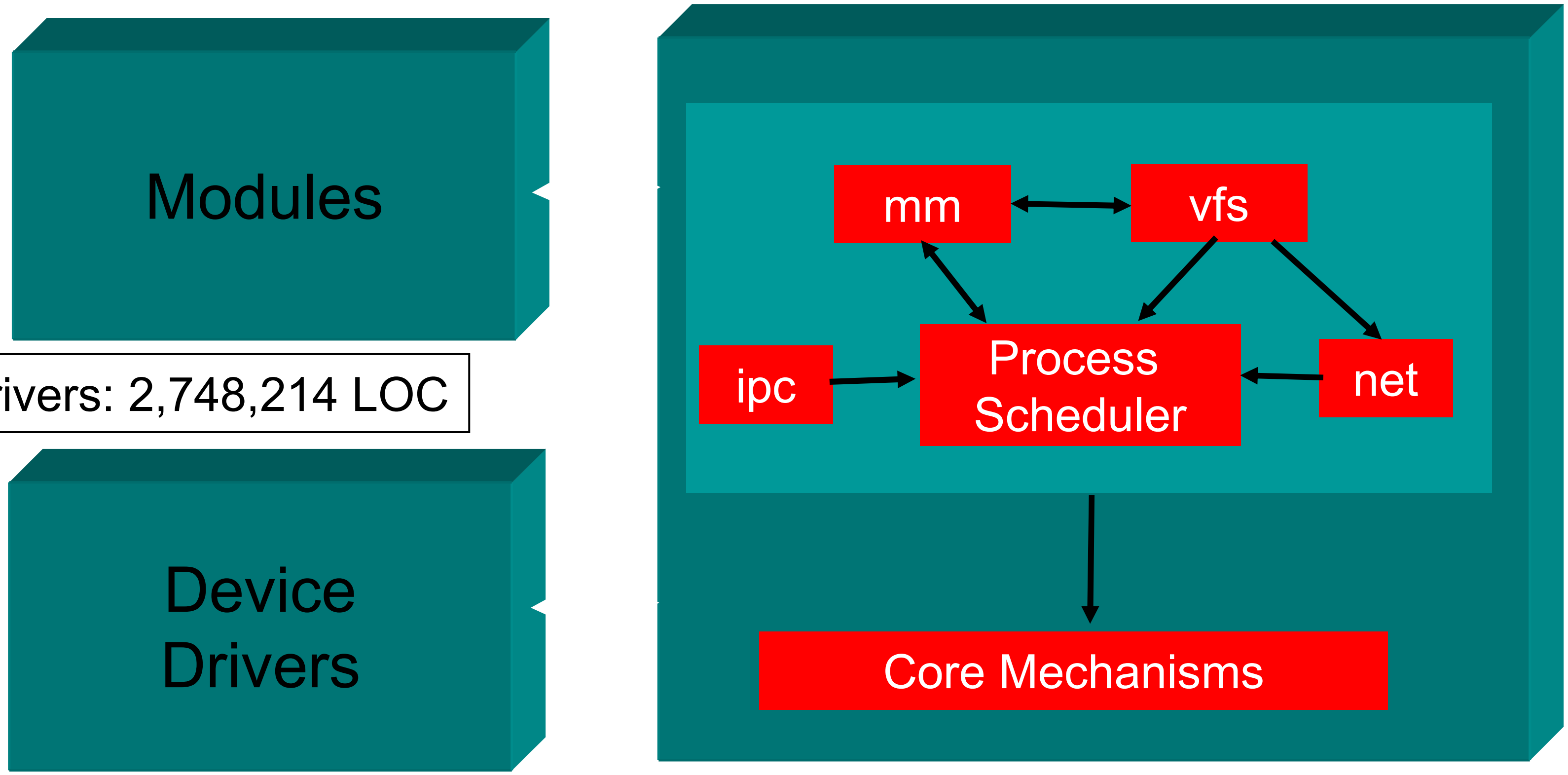


Linux Kernel: rate of change



Linux Internals

Total LOC for 2.6.18 kernel: 5,499,231



drivers: 2,748,214 LOC

arch: 920,662 (24 variations: powerpc, arm, x86_64, i386, ia64, sparc, sparc64, mips, sh, sh64, m68k, s390, …)

core kernel: 47,115
init: 2,215
ipc: 4,681 | mm: 24,618 | net: 336,302

Device Drivers

- Black boxes to hide details of hardware devices
- Use standardized calls
  - Independent of the specific driver
- Main role
  - Map standard calls to device-specific operations
- Can be developed separately from the rest of the kernel
  - Plugged in at runtime when needed

Role of a Device Driver

- Implements *mechanisms* to access the hardware
  - E.g., show a disk as an array of data blocks
- Does not enforce particular *policies* on users
  - Examples:
    - Who may access the drive?
    - Is the drive accessed via a file system?
    - May users mount file systems on the drive?

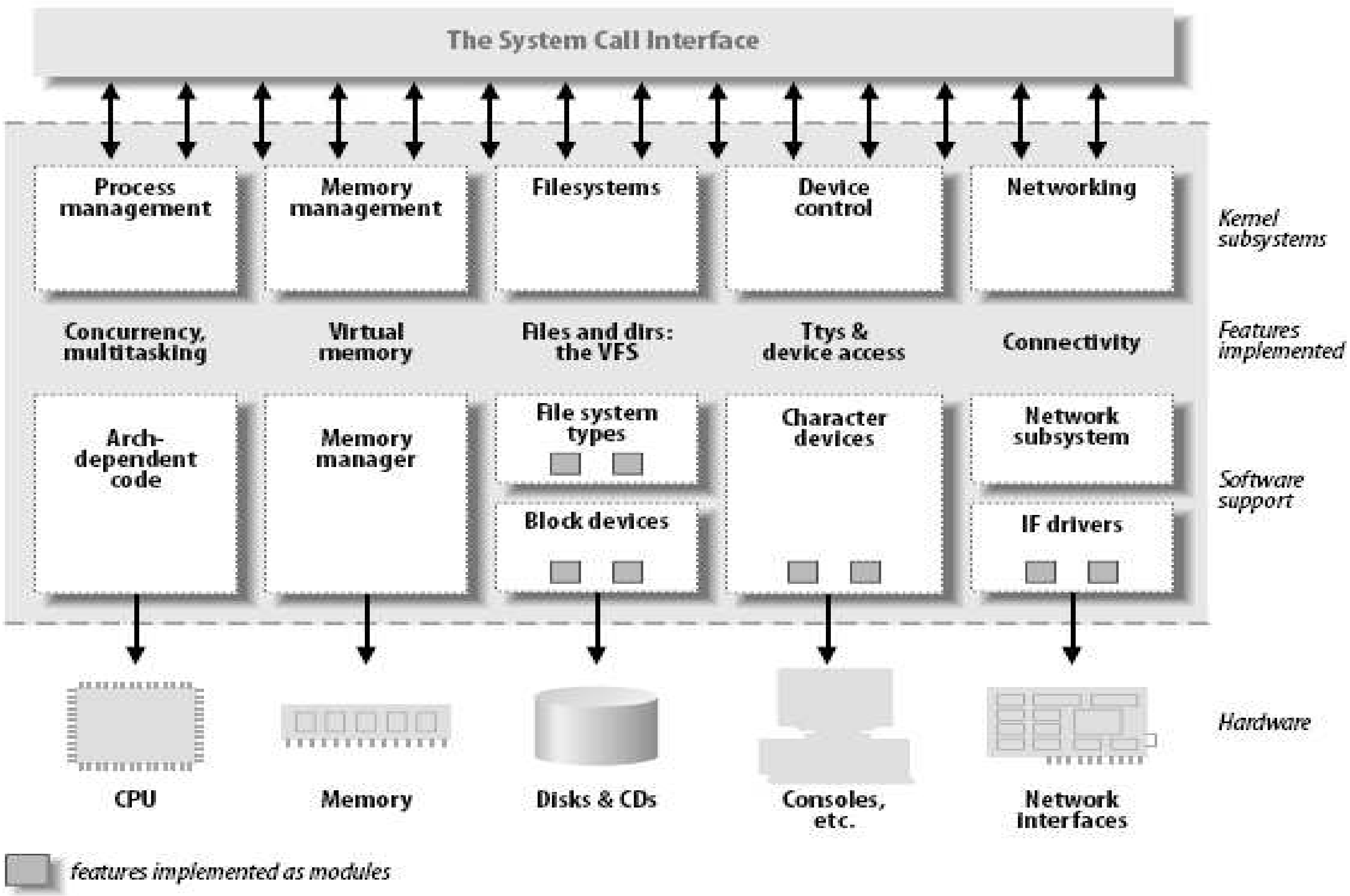
Kernel Subsystems

- Process management
  - Creates, destroys processes
  - Supports communication among processes
    - Signals, pipes, etc.
  - Schedules how processes share the CPU
- Memory management
  - Virtual addressing

Kernel Subsystems

- File systems
  - Everything in UNIX can be treated as a file
  - Linux supports multiple file systems
- Device control
  - Every system operation maps to a physical device
    - Few exceptions: CPU, memory, etc.
- Networking
  - Handles packets
  - Handles routing and network address resolution issues

Kernel Subsystems



Loadable Kernel Modules

- The ability to add and remove kernel features at runtime
- Each unit of extension is called a *module*
- Use **`insmod`** program to add a kernel module
- Use **`rmmod`** program to remove a kernel module

Role of a Module

- Extend kernel functionality
  - Modularized code running in kernel space
- Example: filesystems (xfs, jffs2, fuse, …)

Classes of Devices and Modules

- Character devices
- Block devices
- Network devices
- Others: USB, SCSI, FireWire, I2C, MTD

Character Devices

- Abstraction: a stream of bytes
  - Examples
    - Text console (**`/dev/console`**)
    - Serial ports (**`/dev/ttyS0`**)
  - Usually supports **`open`**, **`close`**, **`read`**, **`write`**
  - Accessed sequentially (in most cases)
  - Might not support file seeks
  - Exception: frame grabbers
    - Can access acquired image using **`mmap`** or **`lseek`**

Block Devices

- Abstraction: array of storage blocks
- However, applications can access a block device in bytes
  - Block and char devices differ only at the kernel level
  - A block device can host a file system

Network Devices

- Abstraction: data packets
- Send and receive packets
  - Do not know about individual connections
- Have unique names (e.g., **`eth0`**)
  - Not in the file system
  - Support protocols and streams related to packet transmission (i.e., no **`read`** and **`write`**)

Filesystem Modules

- Software drivers, not device drivers
- Serve as a layer between user API and block devices
- Intended to be device-independent

HelloWorld module

```
#include <linux/init.h>
#include <linux/module.h>
MODULE_LICENSE("GPL");

static __init int hello_init(void) {
   printk(KERN_ALERT "Hello, world\n");
   return 0;
}

static __exit void hello_exit(void) {
   printk(KERN_ALERT "Goodbye, cruel world\n");
}

module_init(hello_init);
module_exit(hello_exit);
```

```
% cat Makefile
obj-m := module.o
module-objs := file1.o file2.o

% make –C /usr/src/linux M=`pwd` modules
```

Kernel Modules vs. Applications

- Applications
  - Can access various functions in user-level libraries (e.g., **`printf`** in C library)
- Kernel modules
  - No user-level libraries
  - **`printk`** is defined within the kernel
    - Exported to modules
  - Should include only header files defined within the kernel source tree

Threads/Processes

- Thread: A sequential execution stream
- Address space: Chunks of memory and everything needed to run a program
- Process: An address space + thread(s)

User Space vs Kernel Space

- Kernel modules run in *kernel space*
  - Execute in the *supervisor mode*
  - Everything is allowed
  - Share the same address space
- Applications run in *user space*
  - Execute in the *user mode*
  - Restricted access to hardware
  - Each has its own address space

System Calls

- *System calls* allow processes running at the *user mode* to access kernel functions that run under the *kernel mode*
- Prevent processes from doing bad things, such as
  - Halting the entire operating system
  - Modifying the MBR

User Level Drivers

- \+ Fast development
- \+ C library support
- \+ Conventional debugger
- \+ Fault isolation
- \+ Portability

- Interrupts not available
- Privileged access required for direct memory access
- Poor performance

Hardware Interrupts

- Can suspend user-level processes
  - Transfers execution from user space to kernel space
  - Interrupts are handled by separate threads
    - Not related to any user-level processes
    - Asynchronous

Concurrency in the Kernel

- Sources of concurrency
  - Hardware interrupts
  - Kernel timers
  - Multiple CPUs
  - Preemption

Handling Concurrency

- Kernel code needs to be *reentrant*
  - Capable of running in more than one thread execution context at the time
  - Prevent corruption of shared data
  - Avoid race conditions
    - Results depend on the timing of their executions

DRAM-peek module (I)

```
#include <linux/module.h>
#include <linux/highmem.h>
#include <asm/uaccess.h>

char modname[] = "dram";
int my_major = 253;
unsigned long dram_size;

static loff_t my_llseek( struct file *file, loff_t offset, int whence );
static ssize_t my_read( struct file *file, char *buf, size_t count, loff_t *pos );

struct file_operations my_fops =   {
                owner:          THIS_MODULE,
                llseek:         my_llseek,
                read:           my_read,
};
```

DRAM-peek module (II)

```
static __init int init_module(void) {
        printk( "<1>\nInstalling \'%s\' module ", modname );
        printk( "(major=%d)\n", my_major );
        dram_size = num_physpages * PAGE_SIZE;
        printk( "<1>  ramtop=%08lX (%lu MB)\n",
                dram_size, dram_size >> 20 );
        return  register_chrdev( my_major, modname, &my_fops );
}

static __exit void cleanup_module(void) {
        unregister_chrdev( my_major, modname );
        printk( "<1>Removing \'%s\' module\n", modname );
}
```

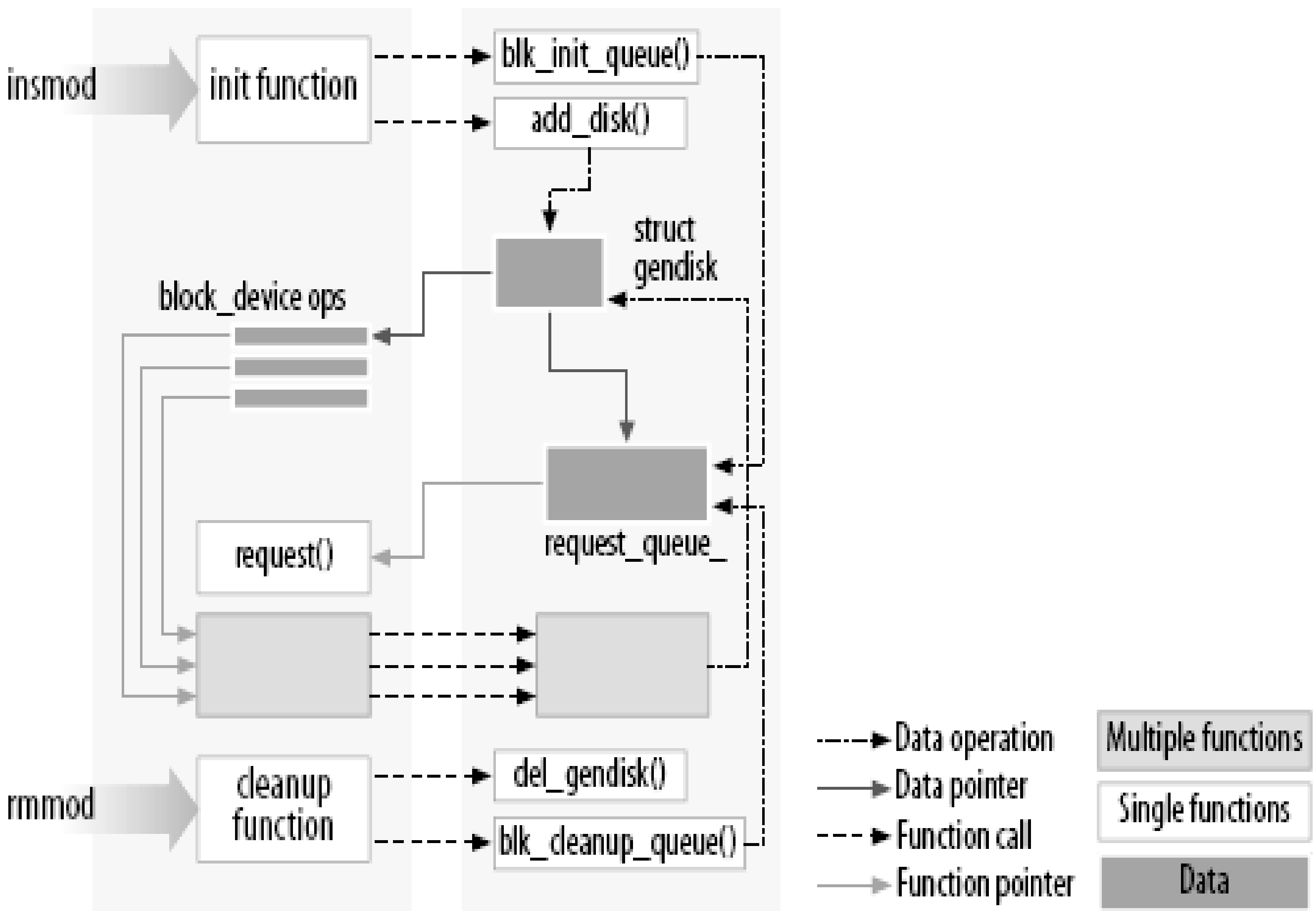
DRAM-peek module (III)

```
static ssize_t my_read( struct file *file, char *buf, size_t count, loff_t *pos ) {
        struct page         *pp;
        void                *from;
        int                 page_number, page_indent, more;

        if ( *pos >= dram_size ) return 0;
        page_number = *pos / PAGE_SIZE;
        page_indent = *pos % PAGE_SIZE;

        pp = &mem_map[ page_number ];
        from = kmap( pp ) + page_indent;
        if ( page_indent + count > PAGE_SIZE ) {
                count = PAGE_SIZE - page_indent;
        }
        more = copy_to_user( buf, from, count );
        kunmap( pp );
        if ( more ) { return -EFAULT; }
        *pos += count;
        return count;
}
```

DRAM-peek module (IV)

```
static loff_t my_llseek( struct file *file, loff_t offset, int whence ) {
        loff_t    newpos = -1;
        switch( whence )  {
                case 0: newpos = offset; break;                    // SEEK_SET
                case 1: newpos = file->f_pos + offset; break;  // SEEK_CUR
                case 2: newpos = dram_size + offset; break;    // SEEK_END
        }
        if (( newpos < 0 )||( newpos > dram_size )) return -EINVAL;
        file->f_pos = newpos;
        return    newpos;
}

MODULE(“GPL”);
```

Linking a Module to the Kernel



Loading and Unloading Modules

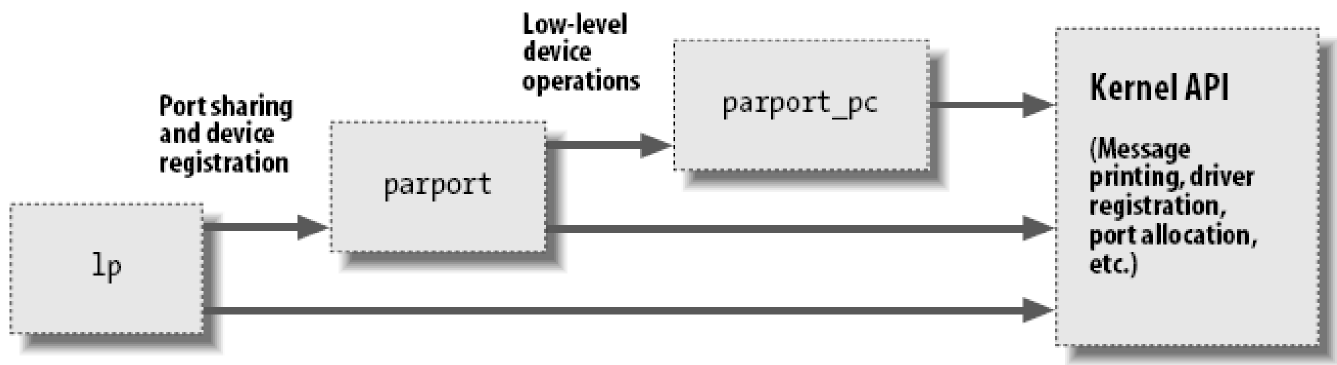
- **`insmod`**
  - Links unresolved symbol in the module to the symbol table of the kernel
  - **`cat /proc/modules`** to see list of currently loaded modules
- **`rmmod`**
  - Removes a kernel module
  - Fails when the kernel believes that it is still in use
    - Or, something has gone wrong
    - Might need to reboot to remove the module

Kernel Symbol Table

- Addresses of global functions and variables
- A module can export its symbols for other modules to use
- *Module stacking*
  - E.g., MSDOS file system relies on symbols exported by the FAT module

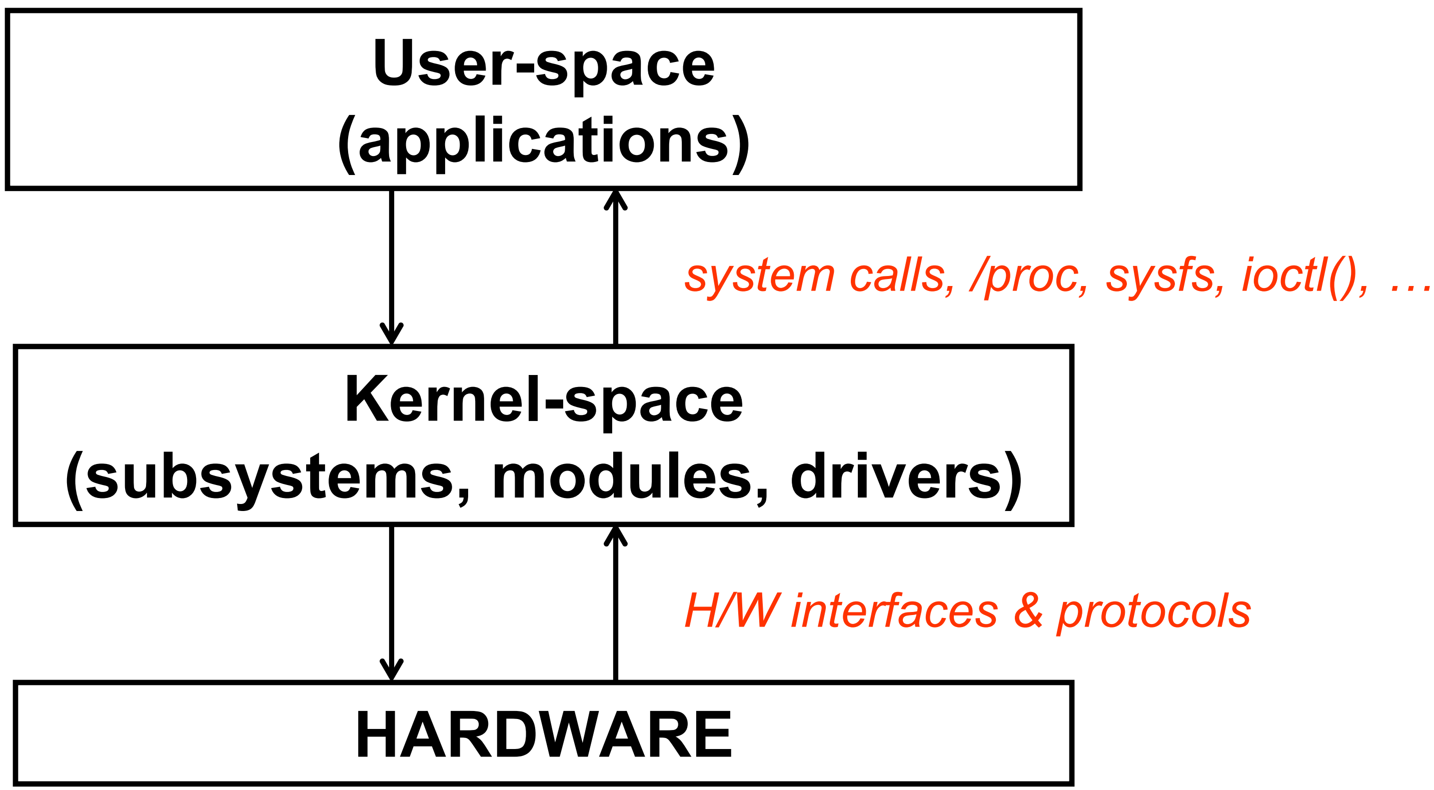
Module Stacking Example

- Stacking of parallel port driver modules



- Can use **modprobe** to load all modules required by a particular module

User-space vs Kernel-space



Device driver events

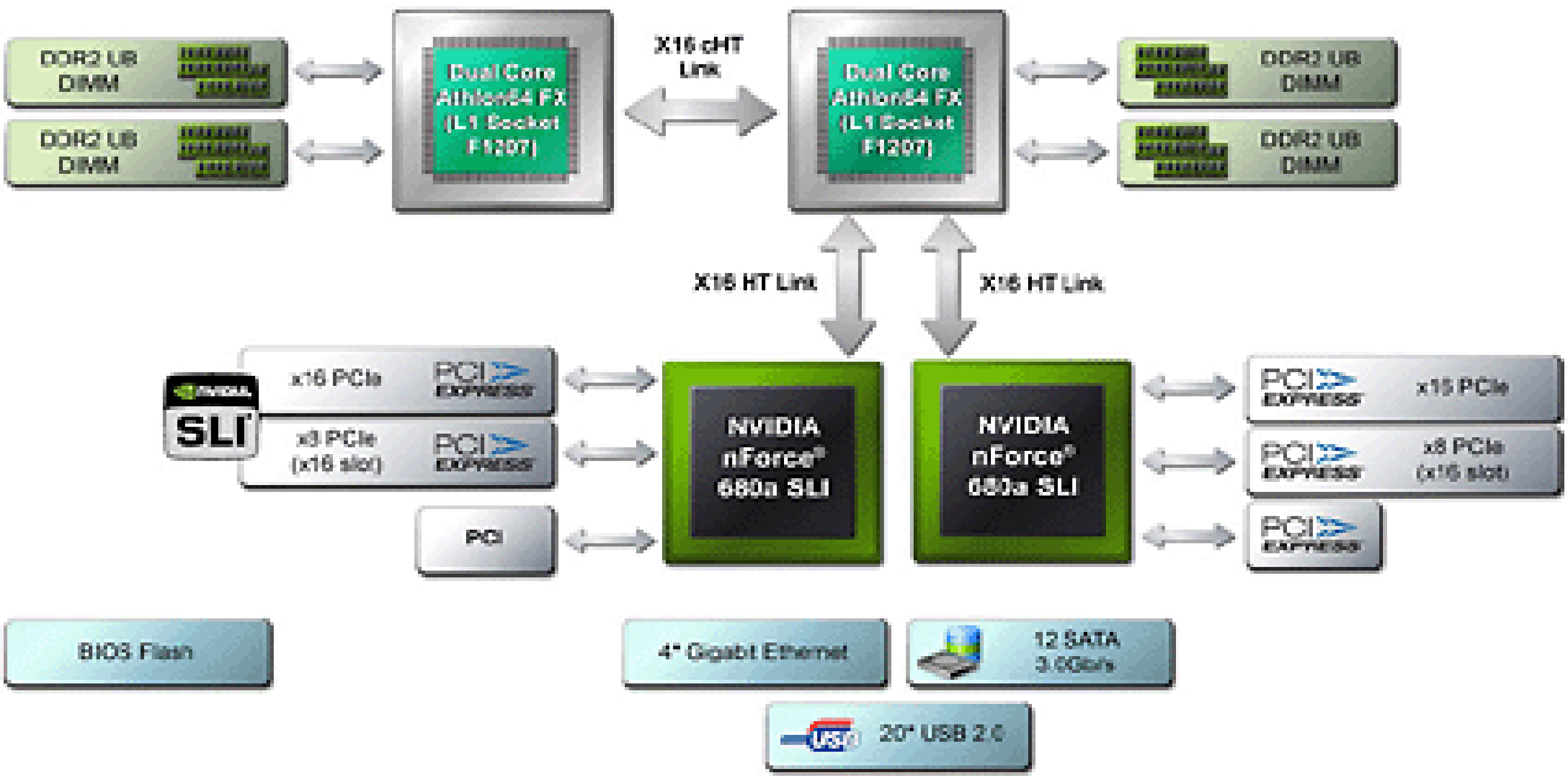
Device driver interfaces

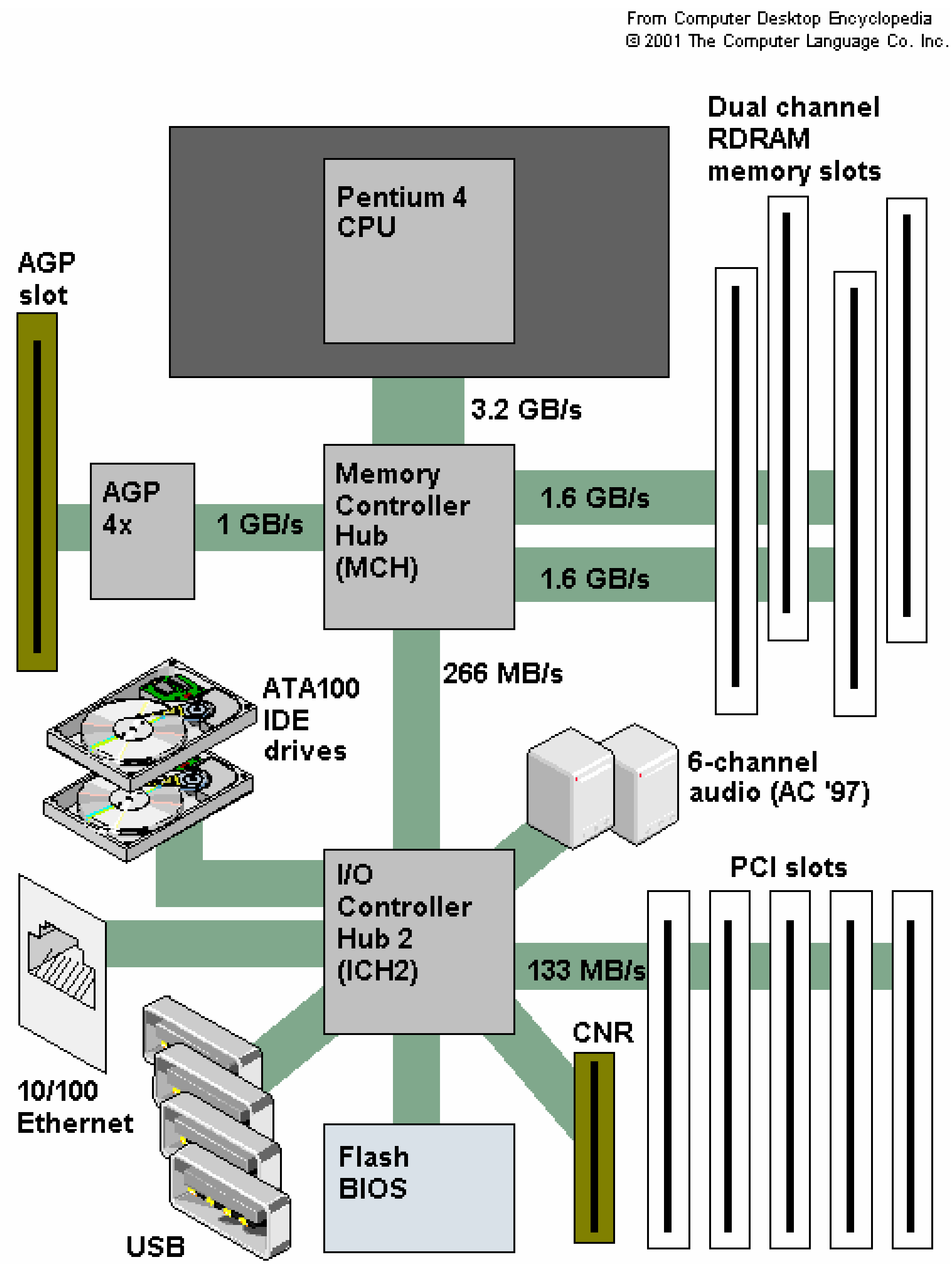
User - Kernel Space Interfaces

Events	User functions	Kernel functions
Load module	insmod	module_init()
Open device	fopen	file_operations: open
Close device	fread	file_operations: read
Write device	fwrite	file_operations: write
Close device	fclose	file_operations: release
Remove module	rmmod	module_exit()

Chipset (AMD/NVIDIA)

NVIDIA nForce® 680a SLI™ System Architecture
Motherboard Works with 1 or 2 CPUs





Chipset (Intel/ICH)

Chipset (Intel/MCH)