Embedded Linux

Mark McDermott

What is Linux

- A fully-networked 32/64-Bit Unix-like Operating System
  - Unix Tools
  - Compilers
  - Network Tools
- Multi-user, Multitasking, Multiprocessor
- X Windows GUI, KDE, etc
- Runs on multiple platforms
  - MIPS, X86, ARM, PPC, BLACKFIN, SPARC, IA64, etc.
- Includes the Source Code
  - /home/ecelrc/faculty/mcdermot/linux/coglinux-2.6.16
  - /home/projects/courses/spring_10/ee382n_16820/tll-mx21-2.6.32.y

History

- UNIX: 1969 Thompson & Ritchie AT&T Bell Labs
- BSD: 1978 Berkeley Software Distribution
- Commercial Vendors: Sun, HP, IBM, SGI, DEC
- GNU: 1984 Richard Stallman, Free Software Foundation (FSF)
- POSIX: 1986 IEEE Portable Operating System unIX
- Minix: 1987 Andy Tannenbaum
- SVR4: 1989 AT&T and Sun
- Linux: 1991 Linus Torvalds Intel 386 (i386)
- Open Source: GPL

Where did Linux come from?

- **Linus Torvalds created it**
  - with assistance from programmers around the world
  - first posted on Internet in 1991
- **Linux 1.0 in 1994; 2.6 in 2003**
- **Open Source:**
  - When programmers on the Internet can read, redistribute, and modify the source for a piece of software, it evolves
  - People improve it, people adapt it, people fix bugs. And this can happen at a speed that, compared to conventional software development, seems astonishing

Linux Features

- UNIX-like operating system.
- Features:
  - Preemptive multitasking.
  - Virtual memory (protected memory, paging).
  - Shared libraries.
  - Demand loading, dynamic kernel modules.
  - Shared copy-on-write executables.
  - TCP/IP networking.
  - SMP support.
  - Open source.

Kernel Design & Architecture

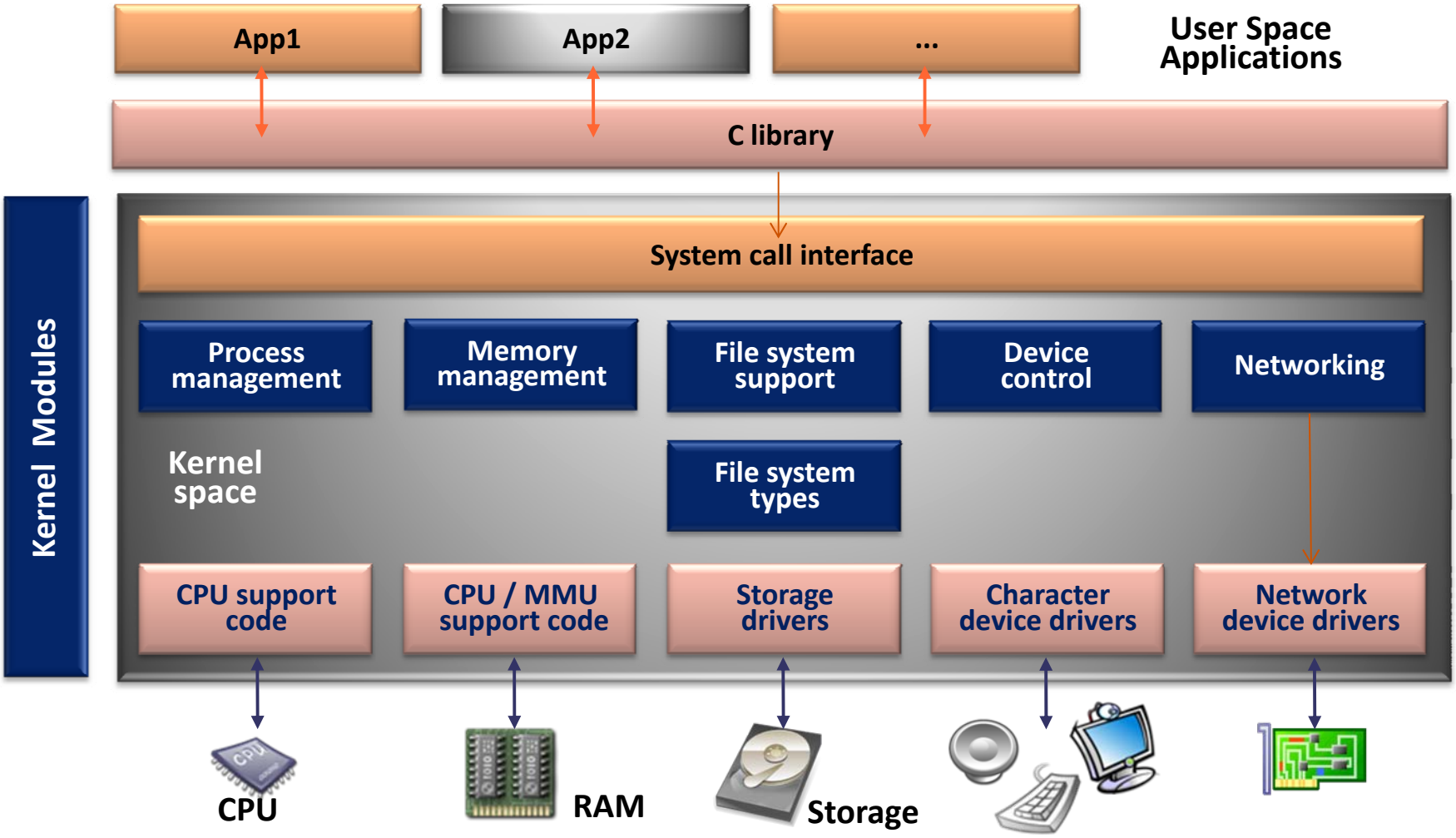
What is a Kernel?

- Executive, system monitor.
  - Controls and mediates access to hardware.
- Implements and supports fundamental abstractions:
  - Processes, files, devices etc.
- Schedules & allocates system resources:
  - Memory, CPU, disk, descriptors, etc.
- Enforces security and protection.
- Responds to user requests for service (system calls).

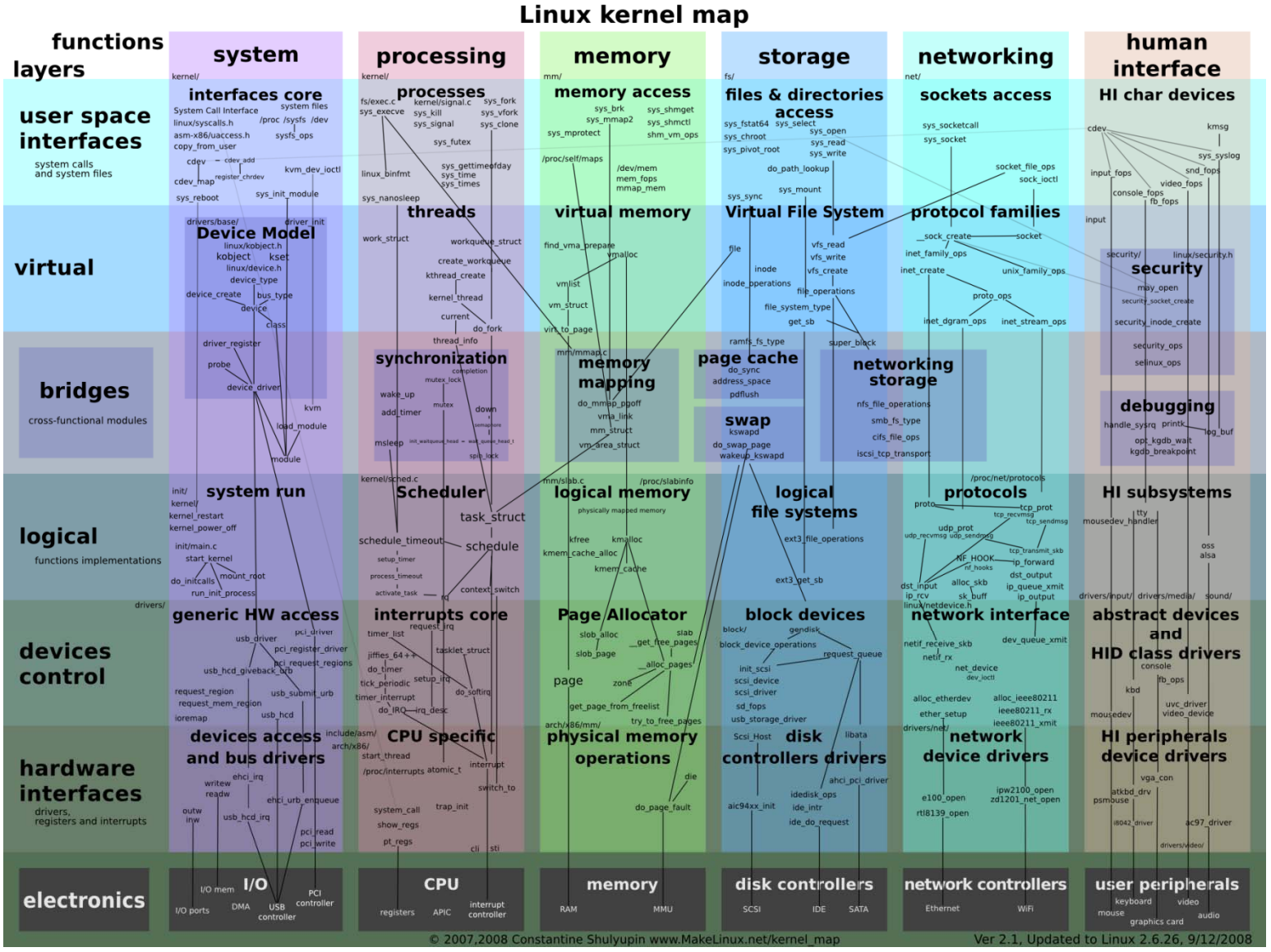
Hardware Dependencies

- Architecture (cpu)
  - dependent (/arch)
  - independent (everything else)
- Abstract dependencies hidden behind functions and macros
- Link in the appropriate version at compile-time
- Device-dependencies isolated in device drivers
- Provide general abstractions that map to reality
  - e.g. three-level page tables

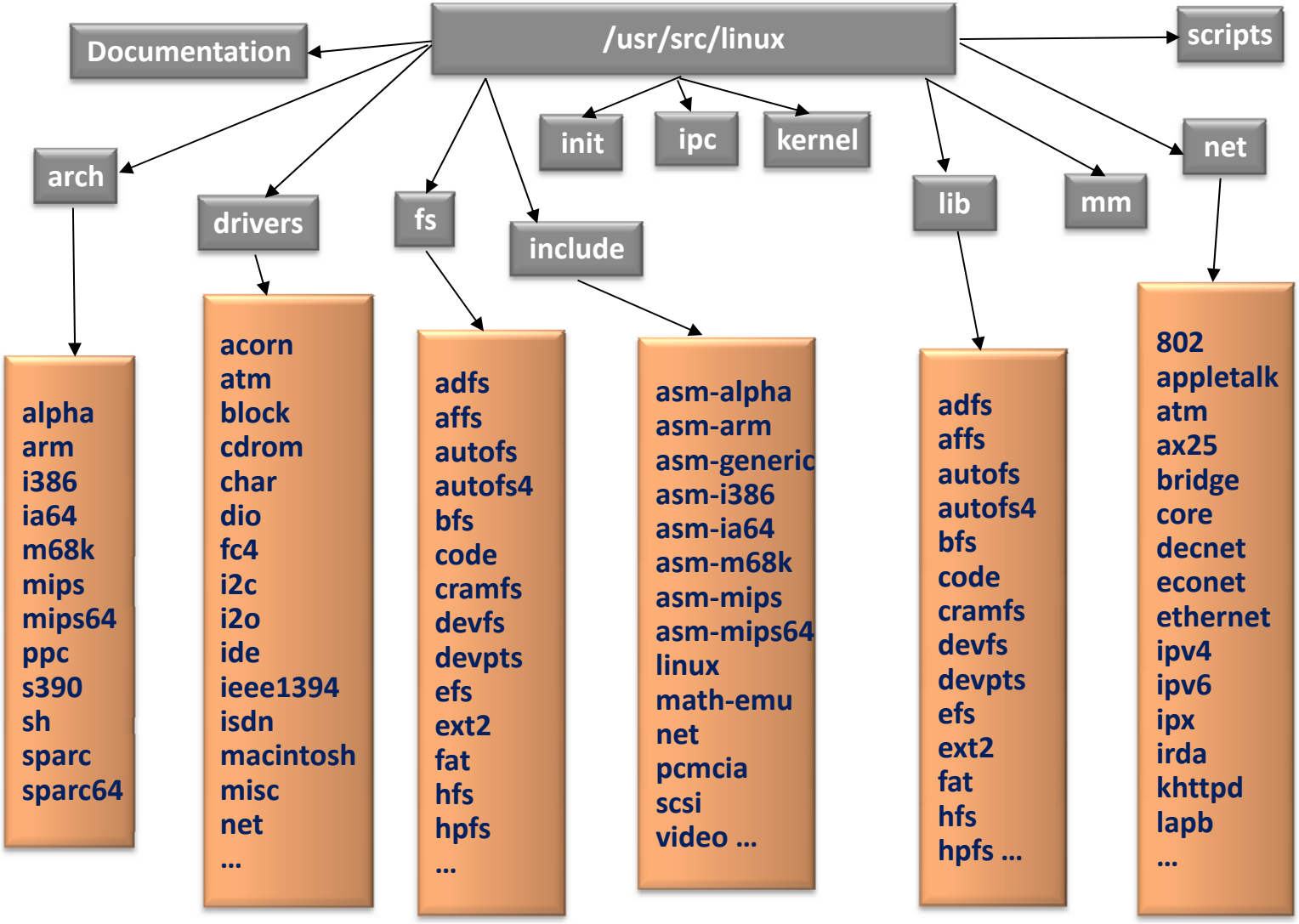
Linux Kernel Architecture



http://www.makelinux.net/kernel_map



Linux Source Tree Layout



linux/arch

- Subdirectories for each current port.
- Each contains kernel, lib, mm, boot and other directories whose contents override code stubs in architecture independent code.
- lib contains highly-optimized common utility routines such as memcpy, checksums, etc.
- "arch" as of 2.6.32 includes the following processors:
  - alpha, arm, avr32, blackfin, cris, frv, h8300, ia64, m32r, m68k, m68knommu, microblaze, mips, mn10300, parisc, powerpc, s390, sparc, x86, xtensa

linux/drivers

- Largest amount of code in the kernel tree (~1.5M).
- device, bus, platform and general directories.
- drivers/char → n_tty.c is the default line discipline.
- drivers/block →elevator.c, genhd.c, linear.c, ll_rw_blk.c, raidN.c.
- drivers/net → specific drivers and general routines Space.c and net_init.c.
- drivers/scsi → scsi_*.c files are generic; sd.c (disk), sr.c (CD-ROM), st.c (tape), sg.c (generic).
- General:
  - cdrom, ide, isdn, parport, pcmcia, pnp, sound, telephony, video.
- Buses – fc4, i2c, nubus, pci, sbus, tc, usb.
- Platforms – acorn, macintosh, s390, sgi.

linux/fs

- **Contains:**
  - virtual filesystem (VFS) framework.
  - subdirectories for actual filesystems.
- **VFS-related files:**
  - exec.c, binfmt_*.c - files for mapping new process images.
  - devices.c, blk_dev.c – device registration, block device support.
  - super.c, filesystems.c.
  - inode.c, dcache.c, namei.c, buffer.c, file_table.c.
  - open.c, read_write.c, select.c, pipe.c, fifo.c.
  - fcntl.c, ioctl.c, locks.c, dquot.c, stat.c.

linux/include

- include/asm-*:
  - Architecture-dependent include subdirectories.
- include/linux:
  - Header info needed both by the kernel and user apps.
  - Usually linked to /usr/include/linux.
  - Kernel-only portions guarded by #ifdefs
    - #ifdef __KERNEL__
    - /* kernel stuff */
    - #endif
- Other directories:
  - math-emu, net, pcmcia, scsi, video.

linux/init

- Just two files: version.c, main.c.
- start_kernel is the primary entry point.
- version.c – contains the version banner that prints at boot.
- main.c – architecture-independent boot code.

```
asmlinkage void __init start_kernel(void)
/*
 * Interrupts are still disabled. Do necessary setups, then
 * enable them
 */
    lock_kernel();
    page_address_init();
    printk(KERN_NOTICE);
    printk(linux_banner);
    setup_arch(&command_line);
    setup_per_cpu_areas();

    /*
     * Mark the boot cpu "online" so that it can call console drivers in printk() and can access its
      per-cpu storage.
     */
    smp_prepare_boot_cpu();

    /*
     * Set up the scheduler prior starting any interrupts (such as the timer interrupt). Full
      * topology setup happens at smp_init() time - but meanwhile we still have a
      * functioning scheduler.
     */
    sched_init();
    /*
     * Disable preemption - early bootup scheduling is extremely
     * fragile until we cpu_idle() for the first time.
     */
    preempt_disable();
    build_all_zonelists();
    page_alloc_init();
    printk(KERN_NOTICE "Kernel command line: %s\n", saved_command_line);
    parse_early_param();
    parse_args("Booting kernel", command_line, __start___param,
            __stop___param - __start___param,
            &unknown_bootoption);
    sort_main_extable();
    trap_init();
    rcu_init();
    init_IRQ();
    pidhash_init();
    init_timers();
    hrtimers_init();
    softirq_init();
    time_init();
```

linux/ipc

- System V IPC facilities.
- If disabled at compile-time, util.c exports stubs that simply return –ENOSYS (function not implemented)
- One file for each facility:
  - sem.c – semaphores.
  - shm.c – shared memory.
  - msg.c – message queues.

linux/kernel

- The core kernel code.
- sched.c – "the main kernel file":
  - scheduler, wait queues, timers, alarms, task queues.
- Process control:
  - fork.c, exec.c, signal.c, exit.c etc…
- Kernel module support:
  - kmod.c, ksyms.c, module.c.
- Other operations:
  - time.c, resource.c, dma.c, softirq.c, itimer.c.
  - printk.c, info.c, panic.c, sysctl.c, sys.c.

linux/lib

- kernel code cannot call standard C library routines.
- Files:
  - brlock.c – “Big Reader” spinlocks.
  - cmdline.c – kernel command line parsing routines.
  - errno.c – global definition of errno.
  - inflate.c – “gunzip” part of gzip.c used during boot.
  - string.c – portable string code.
    - Usually replaced by optimized, architecture-dependent routines.
  - vsprintf.c – libc replacement.

linux/mm

- **Paging and swapping:**
  - swap.c, swapfile.c (paging devices), swap_state.c (cache).
  - vmscan.c – paging policies, kswapd.
  - page_io.c – low-level page transfer.
- **Allocation and deallocation:**
  - slab.c – slab allocator.
  - page_alloc.c – page-based allocator.
  - vmalloc.c – kernel virtual-memory allocator.
- **Memory mapping:**
  - memory.c – paging, fault-handling, page table code.
  - filemap.c – file mapping.
  - mmap.c, mremap.c, mlock.c, mprotect.c.

linux/scripts

- **Scripts for:**
  - Menu-based kernel configuration.
  - Kernel patching.
  - Generating kernel documentation.

Kernel Data Types and Sizes

Kernel Data Types

- For portability
  - Should compile with `-Wall` `-Wstrict-prototypes` flags

- Three main classes
  - Standard C types (e.g., int)
  - Explicitly sized types (e.g., u32)
  - Types for specific kernel objects (e.g., pid_t)

Standard C Types

- The following table shows the various data types and their corresponding size. Note the inconsistencies for long and ptr

arch Size:	char	short	int	long	ptr	long-long	u8	u16	u32	u64
i386	1	2	4	4	4	8	1	2	4	8
alpha	1	2	4	8	8	8	1	2	4	8
armv4l	1	2	4	4	4	8	1	2	4	8
ia64	1	2	4	8	8	8	1	2	4	8
m68k	1	2	4	4	4	8	1	2	4	8
mips	1	2	4	4	4	8	1	2	4	8
ppc	1	2	4	4	4	8	1	2	4	8
sparc	1	2	4	4	4	8	1	2	4	8
sparc64	1	2	4	4	4	8	1	2	4	8
x86_64	1	2	4	8	8	8	1	2	4	8

- Knowing that pointers and long integers have the same size
  - Using `unsigned long` for kernel addresses prevents unintended pointer dereferencing

Explicit Sized Data Items

- See <asm/types.h>

```
u8; /* unsigned byte (8-bits) */
u16; /* unsigned word (16-bits) */
u32; /* unsigned 32-bit value */
u64; /* unsigned 64-bit value */
```

- If a user-space program needs to use these types, use __ prefix (e.g., __u8)

- Kernel also uses conventional types, such as `unsigned int`
  - Usually done for backward compatibility

- *Interface-specific type*: defined by a library to provide an interface to specific data structure (e.g., `pid_t`)

Interface-Specific Types

- Many _t types are defined in `<linux/types.h>`
  - Problematic in printk statements
  - One solution is to cast the value to the biggest possible type (e.g., unsigned long)
    - Avoids warning messages
    - Will not lose data bits

Other kernel specific data sizes

- **Timer Intervals:**
  - Do not assume 1000 jiffies per second
    - Scale times using HZ (number of interrupts per second)
      - For example, check against a timeout of half a second, compare the elapsed time against HZ/2
      - Number of jiffies corresponding to msec second is always msec*HZ/1000
- **Page Size**
  - Memory page is PAGE_SIZE bytes, not 4KB
    - Can vary from 4KB to 64KB
    - `PAGE_SHIFT` contains the number of bits to shift an address to get its page number
    - See `<asm/page.h>`
    - User-space program can use getpagesize() library function

Other kernel specific data sizes (cont)

- **Byte order**
  - PC stores multibyte values low-byte first (little-endian)
  - Some platforms use big-endian
  - Use predefined macros
    - `<linux/byteorder/big_endian.h>`
    - `<linux/byteorder/little_endian.h>`
  - Examples
    - `u32 cpu_to_le32(u32);`
      - `cpu` = internal CPU representation
      - `le` = little endian
    - `u64 be64_to_cpu(u64);`
      - `be` = big endian
    - `U16 cpu_to_le16p(u16);`
      - `p` = pointer

Data Alignment

- How to read a 4-byte value stored at an address that is not a multiple of 4 bytes?
  - i386 permits this kind of access
  - Not all architectures permit it
    - Can raise exceptions
- Use the following typeless macros

```
#include <asm/unaligned.h>
get_unaligned(ptr);
put_unaligned(val, ptr);
```

- Another issue is the portability of data structures
  - Compiler rearranges structure fields to be aligned according to platform-specific conventions
  - Automatically add padding to make things aligned
    - May no longer match the intended format

System Calls

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 (Master Boot Record) which is easy to do on WIN-XP
- Interface between user-level processes and hardware devices.
  - CPU, memory, disks etc.
- Make programming easier:
  - Let kernel take care of hardware-specific issues.
- Increase system security:
  - Let kernel check requested service via syscall.
- Provide portability:
  - Maintain interface but change functional implementation.

POSIX* APIs

- API = Application Programmer Interface.
  - Function definition specifying how to obtain service.
  - By contrast, a system call is an explicit request to kernel made via a software interrupt.
- Standard C library (libc) contains wrapper routines that make system calls.
  - e.g., malloc, free are libc routines that use the brk system call.
  - The brk() and sbrk() functions are used to change dynamically the amount of space allocated for the calling process's data segment
- POSIX-compliant = having a standard set of APIs.
- Non-UNIX systems can be POSIX-compliant if they offer the required set of APIs.

* Portable Operating System Interface

Linux System Calls

- **Invoked by executing SWI.**
  - Programmed exception vector number 0x000008.
  - CPU switches to kernel mode & executes a kernel function.
- **Calling process passes syscall number identifying system call in a register.**
- **Syscall handler responsible for:**
  - Saving registers on kernel mode stack.
  - Invoking syscall service routine.
  - Exiting by calling ret_from_sys_call().
- **System call dispatch table:**
  - Associates syscall number with corresponding service routine.
  - Stored in `sys_call_table` array having up to `NR_syscall` entries (usually 256 maximum).
  - nth entry contains service routine address of syscall n.

Kernel Modules

Kernel Modules

- Kernel Modules can be compiled and dynamically linked into kernel address space after the kernel has been compiled
  - Useful for device drivers that need not always be resident until needed.
    - Keeps core kernel "footprint" small.
  - Can be used to "extend" functionality of kernel
- Kernel modules run in kernel space
  - Execute in the supervisor mode
  - Everything is allowed
  - Share the same address space

```
/* hello.c */
    #include <linux/init.h>
    #include <linux/module.h>
    #include <linux/kernel.h>

    static int __init hello_init(void)
    {
       printk(KERN_ALERT "Good morrow")
      printk(KERN_ALERT "to this fair assembly.\n");
       return 0;
    }

    static void __exit hello_exit(void)
    {
       printk(KERN_ALERT "Alas, poor world, what treasure");
       printk(KERN_ALERT "hast thou lost!\n");
    }

    module_init(hello_init);
    module_exit(hello_exit);
   MODULE_LICENSE("GPL");
    MODULE_DESCRIPTION("Greeting module");
    MODULE_AUTHOR("William Shakespeare");
```

Installing Modules into the Kernel

- Module object file is installed in running kernel using

  `insmod module_name.ko`

  - Loads module into kernel address space and links unresolved symbols in module to symbol table of running kernel.
- To observe the kernel modules loaded use '`lsmod`' command
  - On a server you would see the following:

```
Module                  Size  Used by
ip_conntrack           53025  2 ip_conntrack_netbios_ns,xt_state
nfnetlink              10713  1 ip_conntrack
iptable_filter          7105  1
ip_tables              17029  1 iptable_filter
nfs                   227905  4
lockd                  59081  2 nfs
fscache                20449  1 nfs
nfs_acl                 7617  1 nfs
autofs4                24517  2
rfcomm                 42457  0
l2cap                  29633  8 hidp,rfcomm
bluetooth              53925  5 hidp,rfcomm,l2cap
sunrpc                144253  4 nfs,lockd,nfs_acl
```

The Kernel Symbol Table

- **Symbols accessible to kernel-loadable modules appear in /proc/kallsyms.**
  - ***register_symtab*** **registers a symbol table in the kernel's main table.**
  - **Real hackers export symbols from the kernel by modifying kernel/ksyms.c**
  - **Typical symbol table would look like:**

```
c04011f0 T _stext
c04011f0 t run_init_process
c04011f0 T stext
c040122c t init_post
c04012e7 t rest_init
c0401308 t try_name
c0401485 T name_to_dev_t
c04016cc T calibrate_delay
c04019b0 T hard_smp_processor_id
c04019c0 t target_cpus
c04019c7 t check_apicid_used
c04019ca t check_apicid_present
c04019d0 t multi_timer_check
c04019d3 t apic_id_registered
c04019d9 t apicid_to_node
c04019dd t cpu_to_logical_apicid
c04019f1 t cpu_present_to_apicid
c0401a05 t ioapic_phys_id_map
```

Process Management

Process Management

- UNIX process management separates the creation of processes and the running of a new program into two distinct operations.
  - The fork system call creates a new process
  - A new program is run after a call to 'execve'
    - execve() executes the program pointed to by *filename*. *filename* must be either a binary executable.
    - A thread is a process that happens to share the same address spaces as its parent(uses clone system call instead of execve)
- Under UNIX, a process encompasses all the information that the operating system must maintain to track the context of a single execution of a single program
- Under Linux, process properties fall into three groups: the process's identity, environment, and context

Process Identity

- Process ID (PID). The unique identifier for the process; used to specify processes to the operating system when an application makes a system call to signal, modify, or wait for another process
- Credentials. Each process must have an associated user ID and one or more group IDs that determine the process's rights to access system resources and files
- Personality. Not traditionally found on UNIX systems, but under Linux each process has an associated personality identifier that can slightly modify the semantics of certain system calls
  - Used primarily by emulation libraries to request that system calls be compatible with certain specific flavors of UNIX

Process Environment

- The process's environment is inherited from its parent, and is composed of two null-terminated vectors:
  - The argument vector lists the command-line arguments used to invoke the running program; conventionally starts with the name of the program itself
  - The environment vector is a list of "NAME=VALUE" pairs that associates named environment variables with arbitrary textual values
- Passing environment variables among processes and inheriting variables by a process's children are flexible means of passing information to components of the user-mode system software
- The environment-variable mechanism provides a customization of the operating system that can be set on a per-process basis, rather than being configured for the system as a whole

Process Context

- The (constantly changing) state of a running program at any point in time
- The scheduling context is the most important part of the process context; it is the information that the scheduler needs to suspend and restart the process
- The kernel maintains accounting information about the resources currently being consumed by each process, and the total resources consumed by the process in its lifetime so far
- The file table is an array of pointers to kernel file structures
  - When making file I/O system calls, processes refer to files by their index into this table

Process Context (Cont.)

- Whereas the file table lists the existing open files, the file-system context applies to requests to open new files
  - The current root and default directories to be used for new file searches are stored here
- The signal-handler table defines the routine in the process's address space to be called when specific signals arrive
- The virtual-memory context of a process describes the full contents of the its private address space

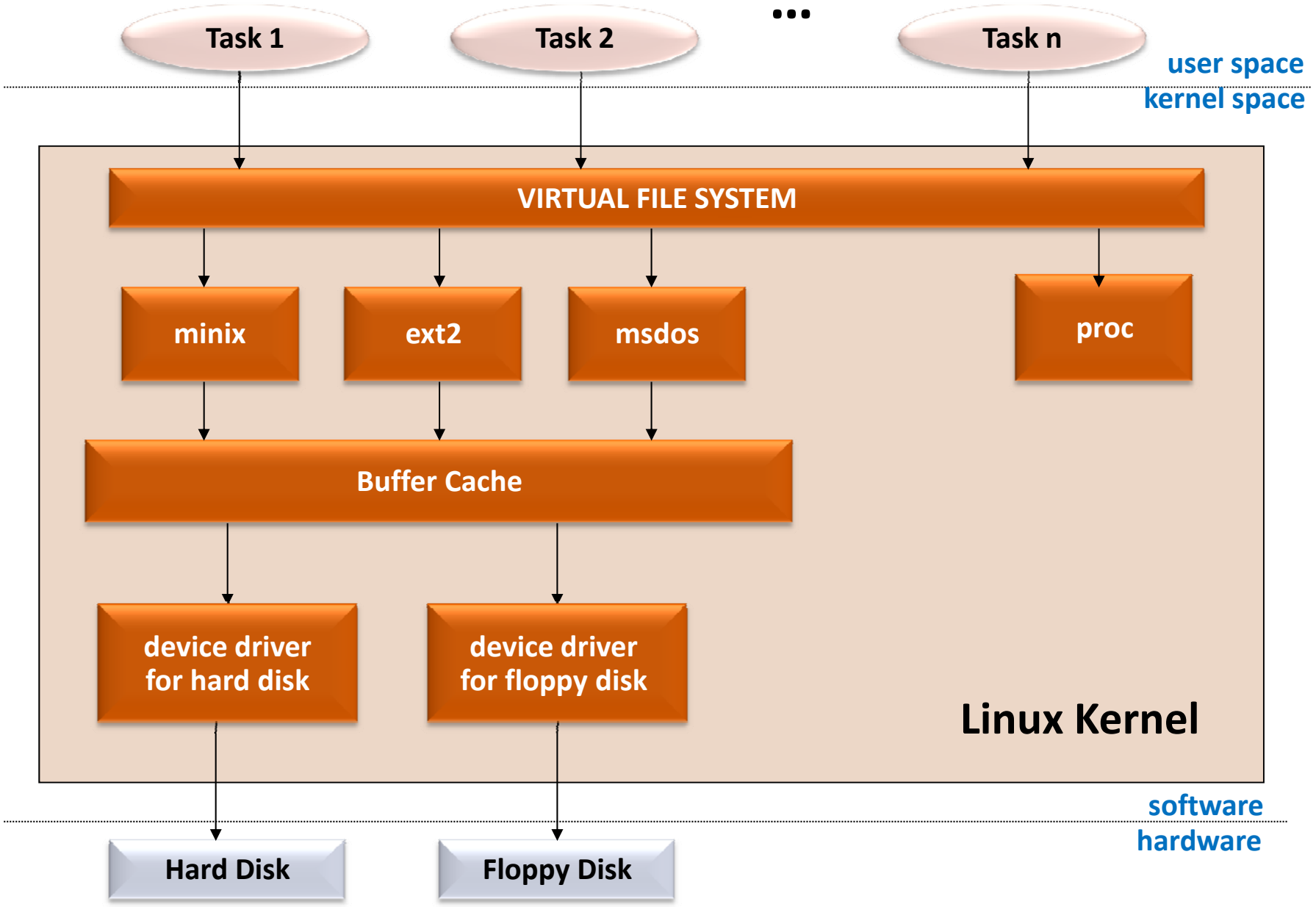
Processes and Threads

- Linux uses the same internal representation for processes and threads; a thread is simply a new process that happens to share the same address space as its parent
- A distinction is only made when a new thread is created by the clone system call
  - fork creates a new process with its own entirely new process context
  - clone creates a new process with its own identity, but that is allowed to share the data structures of its parent
- Using clone gives an application fine-grained control over exactly what is shared between two threads

File System

Linux File System

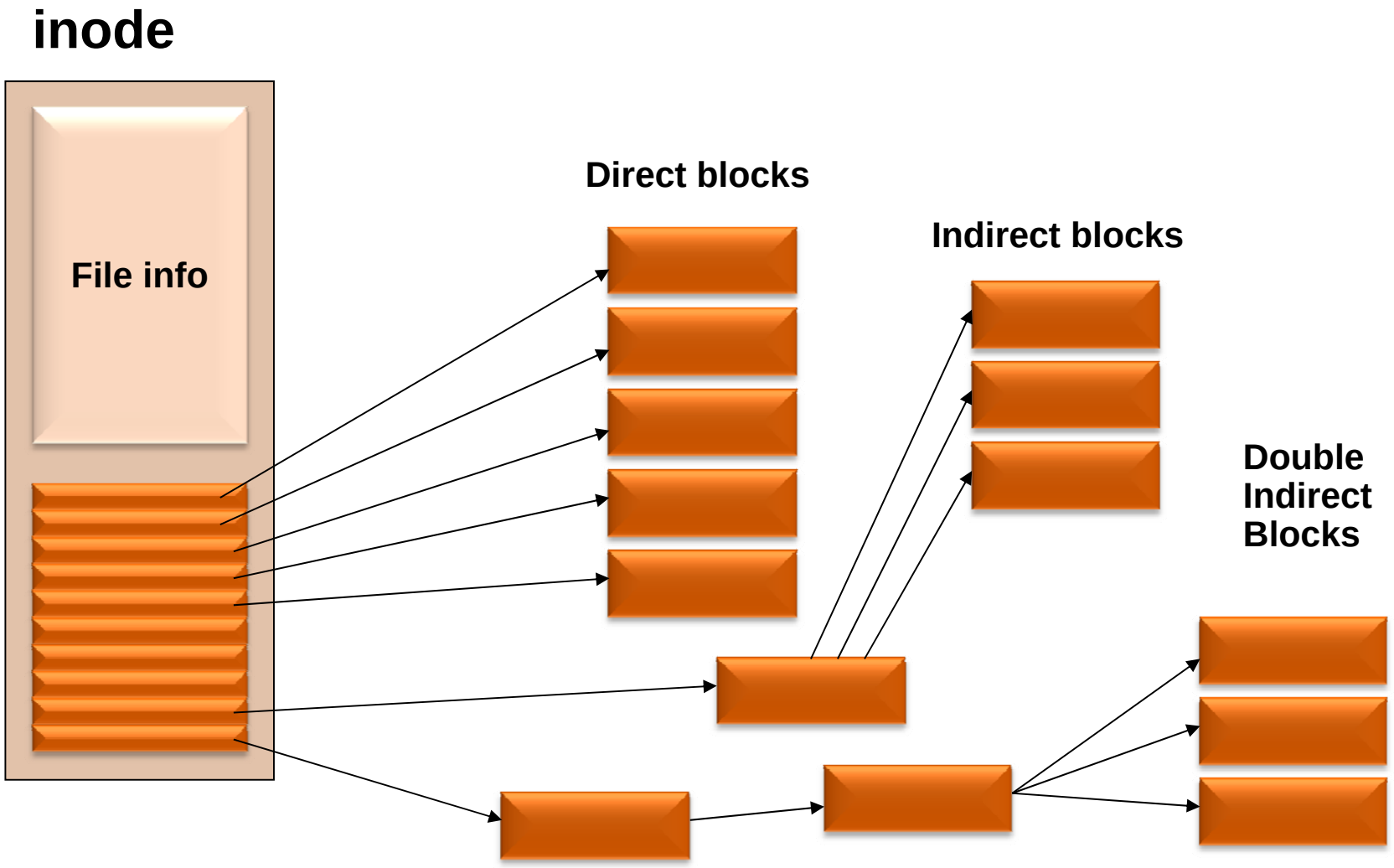
- To the user, Linux's file system appears as a hierarchical directory tree obeying UNIX semantics
- Internally, the kernel hides implementation details and manages the multiple different file systems via an abstraction layer, that is, the virtual file system (VFS)
- Files are represented by inodes
- Directories are special files (dentries)
- Devices accessed by I/O on special files
- UNIX files systems can implement 'links'


Task 1
Task 2
...
Task n
user space
kernel space
VIRTUAL FILE SYSTEM
minix
ext2
msdos
proc
Buffer Cache
device driver for hard disk
device driver for floppy disk
Linux Kernel
software
hardware
Hard Disk
Floppy Disk

Inodes

- A structure that contains file's description:
  - Type
  - Access rights
  - Owners
  - Timestamps
  - Size
  - Pointers to data blocks
- Kernel keeps the inode in memory (open)

inode diagram



File Systems

- **Traditional file systems**
  - ext2
  - MINIX
  - MS-DOS/VFAT
  - HPFS (High Performance File System)
- **Journaling file systems**
  - ext3
  - ReiserFS
  - NTFS (New Technology File System)
  - JFS
  - JFFS2
  - XFS
  - Verita's VxFS

File System Journaling

- Journaling and file system transactions
  - Metadata
    - File information stored by the file system
  - File system transactions
    - Update the file (the data)
    - Update the file metadata
  - Corruption occurs when file data and metadata differ
  - You do not need to check and repair journal-based file systems
  - Journal-based file system keeps a record of all current transactions
    - And updates the journal as transactions are completed
- ext2 and ext3 comparison
  - ext2fs or ext2
    - Linux standard file system
    - After a crash, all files need to be checked
    - Used for small partitions
  - ext3fs
    - Latest version of Linux standard file system
    - Provides journaling functionality
    - After a crash, only open files need to be checked
    - You can journal file data and metadata
      - Or simply the metadata
    - Disadvantage: administrative information overhead

Filesystem Comparison

	Minix	ext	Xia	Ext2/3
Maximal FS size	64MB	2GB	2GB	4TB
Maximal filesize	64MB	2GB	64MB	2GB
Maximal filename	14/30 chars	255 chars	248 chars	255 chars
3 timestamps	no	no	yes	yes
Extensible?	no	no	no	yes
Can vary block size?	no	no	no	yes
Code is maintained?	yes	no	?	yes

The Linux /proc File System

- The `/proc` file system does not store data, rather, its contents are computed on demand according to user file I/O requests
- `/proc` must implement a directory structure, and the file contents within; it must then define a unique and persistent inode number for each directory and files it contains
  - It uses this inode number to identify just what operation is required when a user tries to read from a particular file inode or perform a lookup in a particular directory inode
  - When data is read from one of these files, proc collects the appropriate information, formats it into text form and places it into the requesting process's read buffer

Input Output

Input and Output

- **The Linux device-oriented file system accesses disk storage through two caches:**
  - Data is cached in the page cache, which is unified with the virtual memory system
  - Metadata is cached in the buffer cache, a separate cache indexed by the physical disk block
- **Linux splits all devices into three classes:**
  - block devices allow random access to completely independent, fixed size blocks of data
  - character devices include most other devices; they don't need to support the functionality of regular files
  - network devices are interfaced via the kernel's networking subsystem

Block Devices

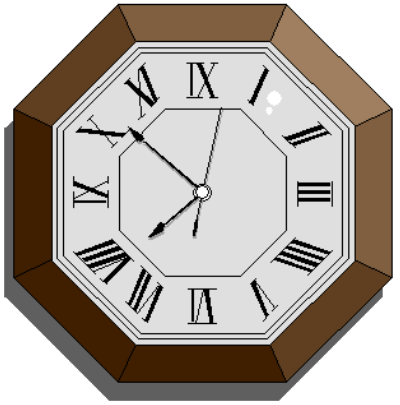
- Provide the main interface to all disk devices in a system
- The *block buffer* cache serves two main purposes:
  - it acts as a pool of buffers for active I/O
  - it serves as a cache for completed I/O
- The *request manager* manages the reading and writing of buffer contents to and from a block device driver

Character Devices

- A “character device” driver does not offer random access to fixed blocks of data
- A character device driver must register a set of functions which implement the driver’s various file I/O operations
- The kernel performs almost no preprocessing of a file read or write request to a character device, but simply passes on the request to the device
- The main exception to this rule is the special subset of character device drivers which implement terminal devices, for which the kernel maintains a standard interface

The Linux Kernel: The Flow of Time

"What time is it?"



- **Need timing measurements to:**
  - Keep track of current time and date for use by e.g. `gettimeofday()`

```
(void)gettimeofday(&start_timestamp, NULL);
```

  - Maintain timers that notify the kernel or a user program that an interval of time has elapsed.

- **Timing measurements are performed by several hardware circuits, based on fixed frequency oscillators and counters.**

Hardware Clocks

- **Real-Time Clock (RTC):**
  - Often integrated with CMOS RAM and Battery on separate chip from CPU: e.g., Motorola MC146818.
  - Issues periodic interrupts on IRQ line (IRQ 8) at programmed frequency (e.g., 2-8192 Hz).
  - In Linux, used to derive time and date.
  - Kernel accesses RTC through 0x70 and 0x71 I/O ports (X86 only)

Timestamp Counter (TSC)

- Intel Pentium (and up), AMD K6 etc incorporate a TSC.
- Processor's CLK pin receives a signal from an external oscillator e.g., 400 MHz crystal.
- TSC register is incremented at each clock signal.
- Using `rdtsc` assembly instruction can obtain 64-bit timing value.
- Most accurate timing method on above platforms.

The "PIT"s

- Programmable Interrupt Timers (PITs):
  - e.g., 8254 chip.
- PIT issues *timer interrupts* at programmed frequency.
- In Linux, PC-based 8254 is programmed to interrupt Hz (=100) times per second on IRQ 0.
  - Hz defined in `<linux/param.h>`
  - PIT is accessed on ports `0x40-0x43`.
- Provides the system "heartbeat" or "clock tick".

```
/proc # cat interrupts
           CPU0
 20:   18233450   IMX-uart
 26:   14566234   i.MX Timer Tick
 55:          0   imx21-hc:usb1
224:     914163   smsc911x
Err:          0
```

"This'll only take a jiffy"

- `jiffies` is incremented every timer interrupt.
  - Number of clock ticks since OS was booted.
- Scheduling and preemption done at granularities of time-slices calculated in units of jiffies.

Timer Interrupt Handler

- **Every timer interrupt:**
  - Update jiffies.
  - Update time and date (in secs & μsecs since 1970).
    - There has been 1.262 billion seconds since 1970
  - Determine how long a process has been executing and preempt it, if it finishes its allocated timeslice.
  - Update resource usage statistics.
  - Invoke functions for elapsed interval timers.

PIT Interrupt Service Routine

- Signal on IRQ 0 is generated:
- `timer_interrupt()` is invoked w/ interrupts disabled (`SA_INTERRUPT` flag is set to denote this).
- `do_timer()` is ultimately executed:
  - Simply increments `jiffies` & allocates other tasks to "bottom half handlers".
  - Bottom half (bh) handlers update time and date, statistics, execute after specific elapsed intervals and invoke `schedule()` if necessary, for rescheduling processes.

Updating Time and Date

- `lost_ticks` (`lost_ticks_system`) store total (system) "ticks" since update to `xtime`, which stores *approximate* current time. This is needed since bh handlers run at convenient time and we need to keep track of when exactly they run to accurately update date & time.
- `TIMER_BH` refers to the queue of bottom halves invoked as a consequence of `do_timer()`.

Scheduling

Process Scheduling

- The job of allocating CPU time to different tasks within an operating system
- While scheduling is normally thought of as the running and interrupting of processes, in Linux, scheduling also includes the running of the various kernel tasks
- Running kernel tasks encompasses both tasks that are requested by a running process and tasks that execute internally on behalf of a device driver
- As of 2.5, new scheduling algorithm – preemptive, priority-based
  - Real-time range
  - 'nice' value

Process Scheduling

- **Linux uses two process-scheduling algorithms:**
  - A time-sharing algorithm for fair preemptive scheduling between multiple processes
  - A real-time algorithm for tasks where absolute priorities are more important than fairness
- **A process's scheduling class defines which algorithm to apply**
- **For time-sharing processes, Linux uses a prioritized, credit based algorithm**
  - The crediting rule

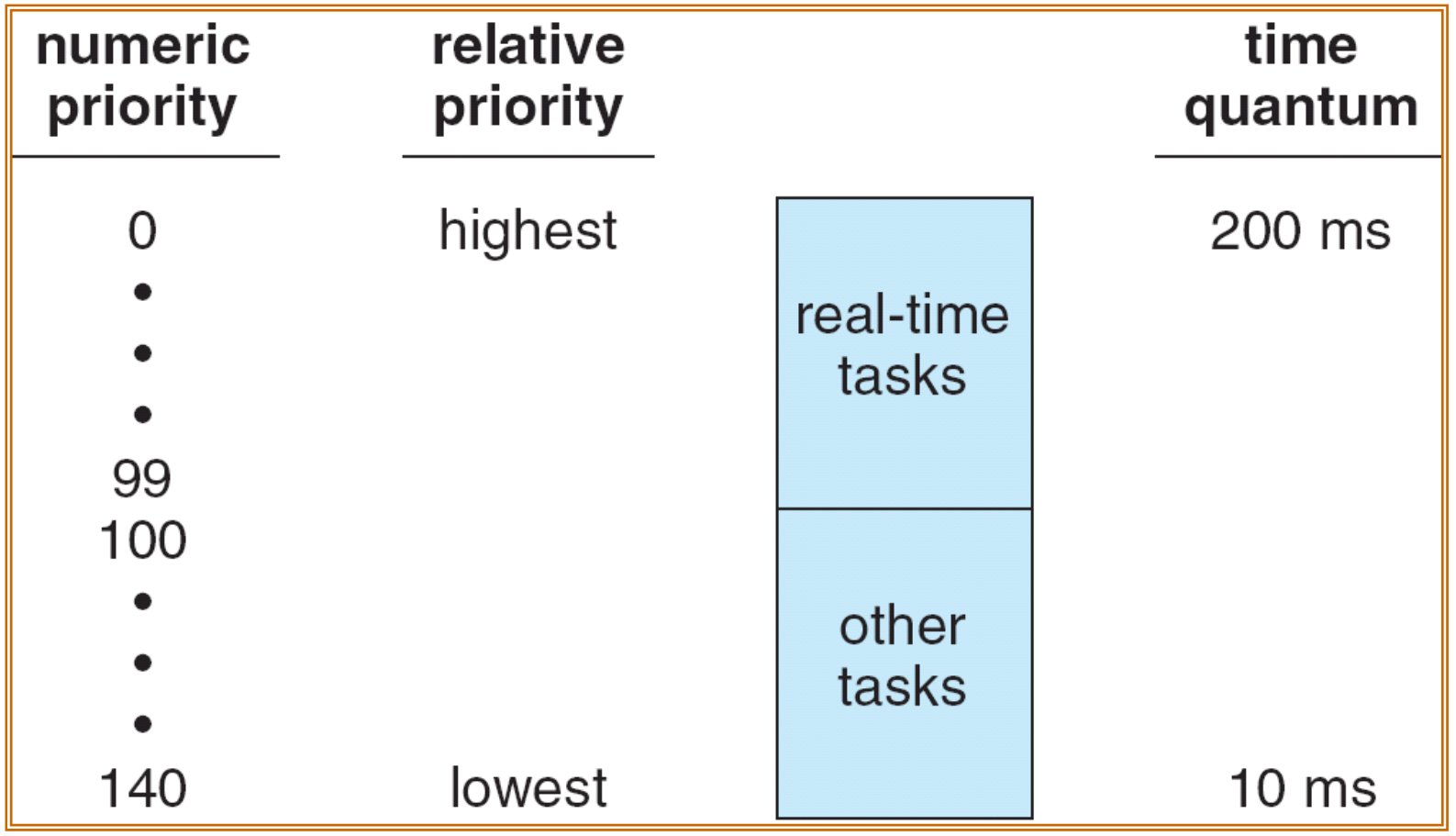
$$\text{credits} := \frac{\text{credits}}{2} + \text{priority}$$

    factors in both the process's history and its priority
  - This crediting system automatically prioritizes interactive or I/O-bound processes

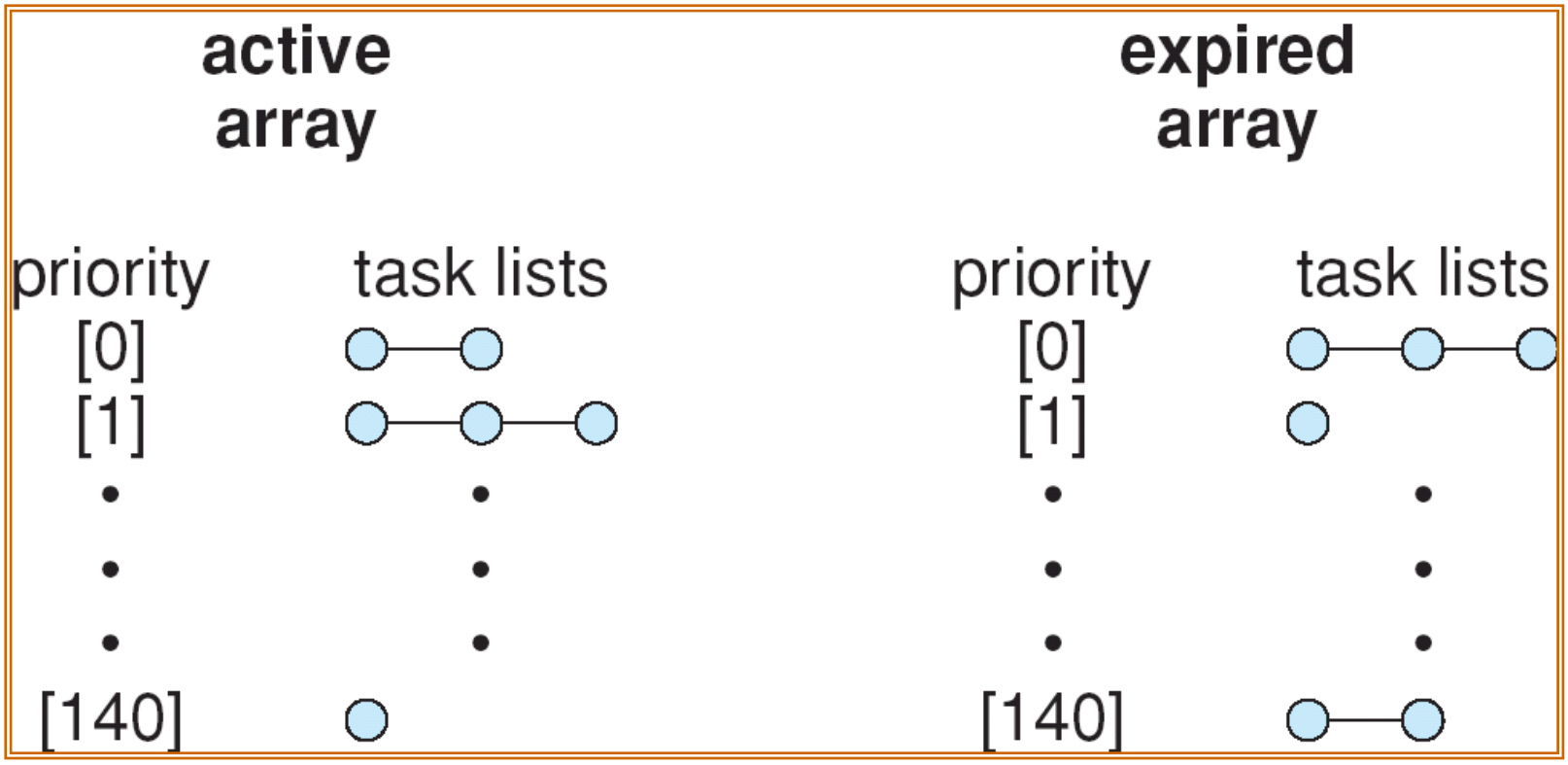
Process Scheduling (Cont.)

- **Linux implements the FIFO and round-robin real-time scheduling classes; in both cases, each process has a priority in addition to its scheduling class**
  - The scheduler runs the process with the highest priority; for equal-priority processes, it runs the process waiting the longest
  - FIFO processes continue to run until they either exit or block
  - A round-robin process will be preempted after a while and moved to the end of the scheduling queue, so that round-robin processes of equal priority automatically time-share between themselves

Relationship Between Priorities and Time-slice Length



List of Tasks Indexed by Priority



Kernel Synchronization

- A request for kernel-mode execution can occur in two ways:
  - A running program may request an operating system service, either explicitly via a system call, or implicitly, for example, when a page fault occurs
  - A device driver may deliver a hardware interrupt that causes the CPU to start executing a kernel-defined handler for that interrupt
- Kernel synchronization requires a framework that will allow the kernel's critical sections to run without interruption by another critical section

Kernel Synchronization (Cont.)

- **Linux uses two techniques to protect critical sections:**
  1. **Normal kernel code is non-preemptible (until 2.4)**
     - **when a time interrupt is received while a process is executing a kernel system service routine, the kernel's need_resched flag is set so that the scheduler will run once the system call has completed and control is about to be returned to user mode**
  2. **The second technique applies to critical sections that occur in an interrupt service routines**
     - **By using the processor's interrupt control hardware to disable interrupts during a critical section, the kernel guarantees that it can proceed without the risk of concurrent access of shared data structures**

Kernel Synchronization (Cont.)

- To avoid performance penalties, Linux's kernel uses a synchronization architecture that allows long critical sections to run without having interrupts disabled for the critical section's entire duration
- Interrupt service routines are separated into a *top half* and a *bottom half.*
  - The top half is a normal interrupt service routine, and runs with recursive interrupts disabled
  - The bottom half is run, with all interrupts enabled, by a miniature scheduler that ensures that bottom halves never interrupt themselves
  - This architecture is completed by a mechanism for disabling selected bottom halves while executing normal, foreground kernel code

Backup

The Linux Ext2fs File System

- Ext2fs uses a mechanism similar to that of BSD Fast File System (ffs) for locating data blocks belonging to a specific file
- The main differences between ext2fs and ffs concern their disk allocation policies
  - In ffs, the disk is allocated to files in blocks of 8Kb, with blocks being subdivided into fragments of 1Kb to store small files or partially filled blocks at the end of a file
  - Ext2fs does not use fragments; it performs its allocations in smaller units
    - The default block size on ext2fs is 1Kb, although 2Kb and 4Kb blocks are also supported
  - Ext2fs uses allocation policies designed to place logically adjacent blocks of a file into physically adjacent blocks on disk, so that it can submit an I/O request for several disk blocks as a single operation

Ext2fs Block-Allocation Policies