The Linux Kernel: Architecture and Programming

Kernels 2.4 and 2.6

Lecturer:
Ing. Luca Pizzamiglio
Politecnico di Milano, DEI
luca.pizzamiglio@gmail.com

Overview

- Introduction
  - History
  - Linux Features
  - Anatomy of a Kernel
  - Linux Conceptual Architecture
- Details on the Linux Kernel
  - Source Tree
  - Recompiling
- Booting and Kernel Initialization
- System Calls

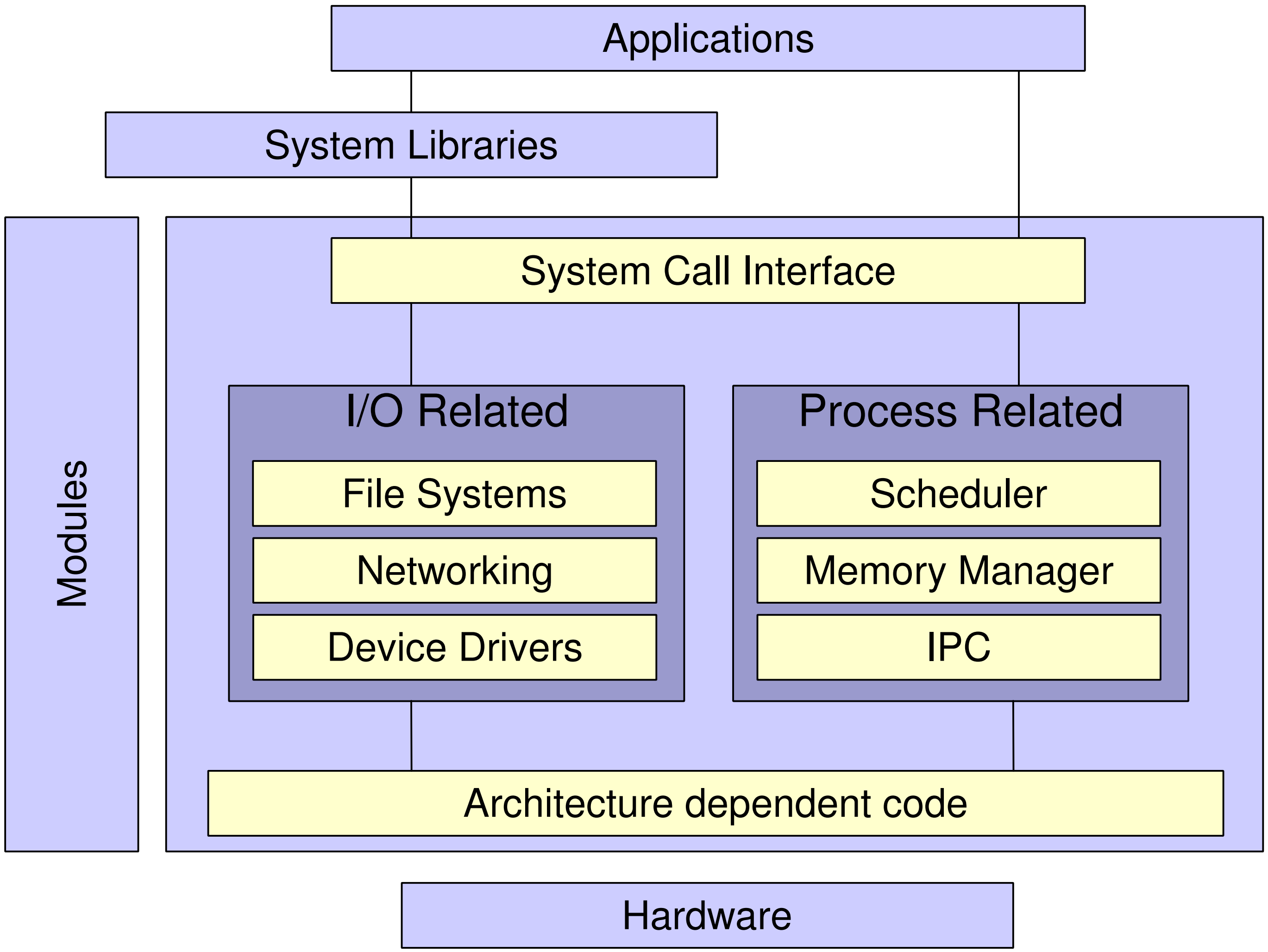
What is a Kernel?

- AKA: executive, system monitor
- Controls and mediates access to hardware
- Implements and supports fundamental abstractions
  - Processes, files, devices, ...
- Schedules and allocates system resources
  - Memory, CPU, disk, descriptors, ...
- Enforces security and protection
- Responds to user requests for services

Kernel Design Goals

- Performance: efficiency, speed
  - The best use of resources with the lowest overhead
- Stability: robustness, resilience
  - Uptime, graceful degradation
- Capability: features, flexibility, compatibility
- Security and protections
  - To protect users from each other
  - To protect system from malicious users
- Portability
- Extensibility

Example "Core" Kernel



History

- **1969 :** UNIX, Thompson & Ritchie (AT&T Bell Labs)
- **1978** : BSD, Berkeley Software Distribution
  - Commercial vendors: Sun, HP, IBM, SGI, DEC
- **1984** : GNU, Richard Stallman (FSF)
- **1986 :** POSIX (Portable Operating System Interface)
- **1987** : Minix, Andy Tanenbaum
- **1989** : SVR4, AT&T and Sun
- **1991** : birth of Linux

Linux history

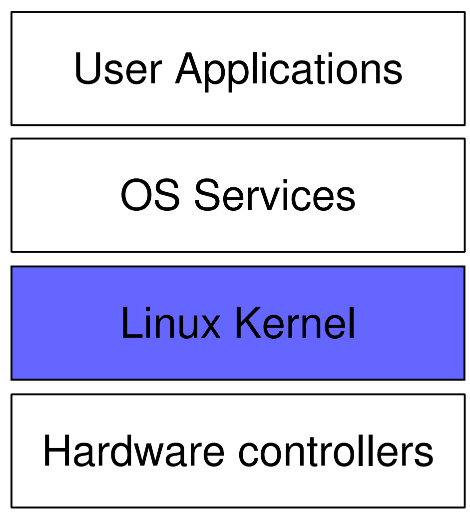
- **1991** : Minix-like OS by Linus Torvalds on his i386
- **1994 :** Linux 1.0
  - Only single-processor i386 machines
- **1995** : Linux 1.2
  - Support for different architectures (Alpha, Sparc, MIPS)
- **1996** : Linux 2.0
  - Support for new architectures
  - SMP
- **1999** : Linux 2.2
- **2001** : Linux 2.4
  - ISA PnP, USB, …
- **17/12/2003** : Linux 2.6

Linux Main Features

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

Linux Conceptual Architecture

- Linux OS is composed by four major subsystems
  - User Applications
  - OS Services
  - Linux Kernel
  - Hardware Controllers
- This decomposition shows a layered style



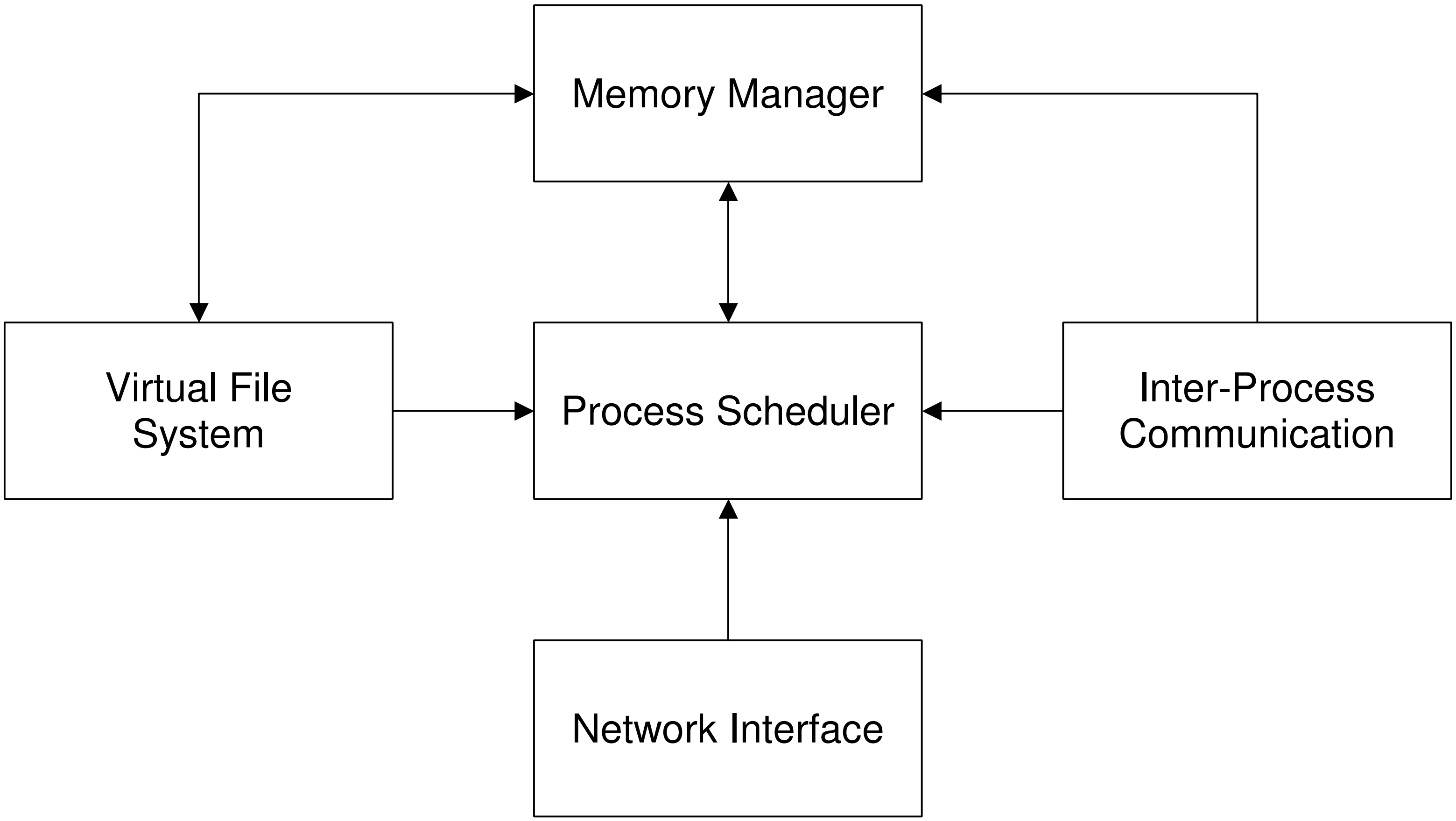
The Linux Kernel

- Virtual machine interface to user processes
  - Processes are written without knowledge of the physical hardware installed
- Each process can act as though it is the only process running on the machine
  - The kernel mediates access to hardware resources so that each process gains fair access
  - The kernel enforces inter-process security too

Kernel Structure: subsystems

- The Linux kernel is composed by five subsystems
  - The Process Scheduler (SCHED) controls access to the CPU, giving fair access to all processes
  - The Memory Manager (MM) permits multiple processes to share memory, implementing virtual memory mechanisms
  - The Virtual File System (VFS) abstracts the details of hardware devices by presenting a common file interface
  - The Network Interface (NET) provides access to networking systems
  - The Inter-Process Communication (IPC) provides mechanisms for process communication

Overview of the Kernel Structure



Dependencies (1/2)

- All subsystems depend on the SCHED because they need to suspend and resume processes
  - E.g. process that attempts to send a message on the network: hardware suspends it until message is sent
- The SCHED uses the MM to adjust the hardware memory map of a specific process when that process is resumed
- The IPC depends on MM to support shared-memory communication mechanisms

Dependencies (2/2)

- The VFS depends on NET and MM to support networked file systems and ramdisks
- The MM uses the VFS to swap data (implementing virtual memory mechanisms)
  - When a process accesses memory that is currently swapped out, the MM takes a request to the VFS to fetch the memory from persistent storage, and suspends the process

Additional dependencies

- All subsystems rely on some common resources that are not shown in any subsystem
  - Procedures used to allocate and free memory for the kernel's use
  - Procedures to print warning or error messages
  - System debugging routines
- Each subsystem contains state information that are accessed using a procedural interface
- Subsystems are responsible for maintaining the integrity of their managed resources

System Data Structures (1/2)

- Task List
  - The scheduler maintains a block of data for each process that is active
  - These blocks are stored in a linked list
  - The scheduler always maintains a *current* pointer that indicates the active process
- Memory Map
  - The MM stores a mapping of virtual to physical addresses on a per-process basis
    - It knows how to fetch and replace particular pages
  - The memory-map data structure is stored in the task list

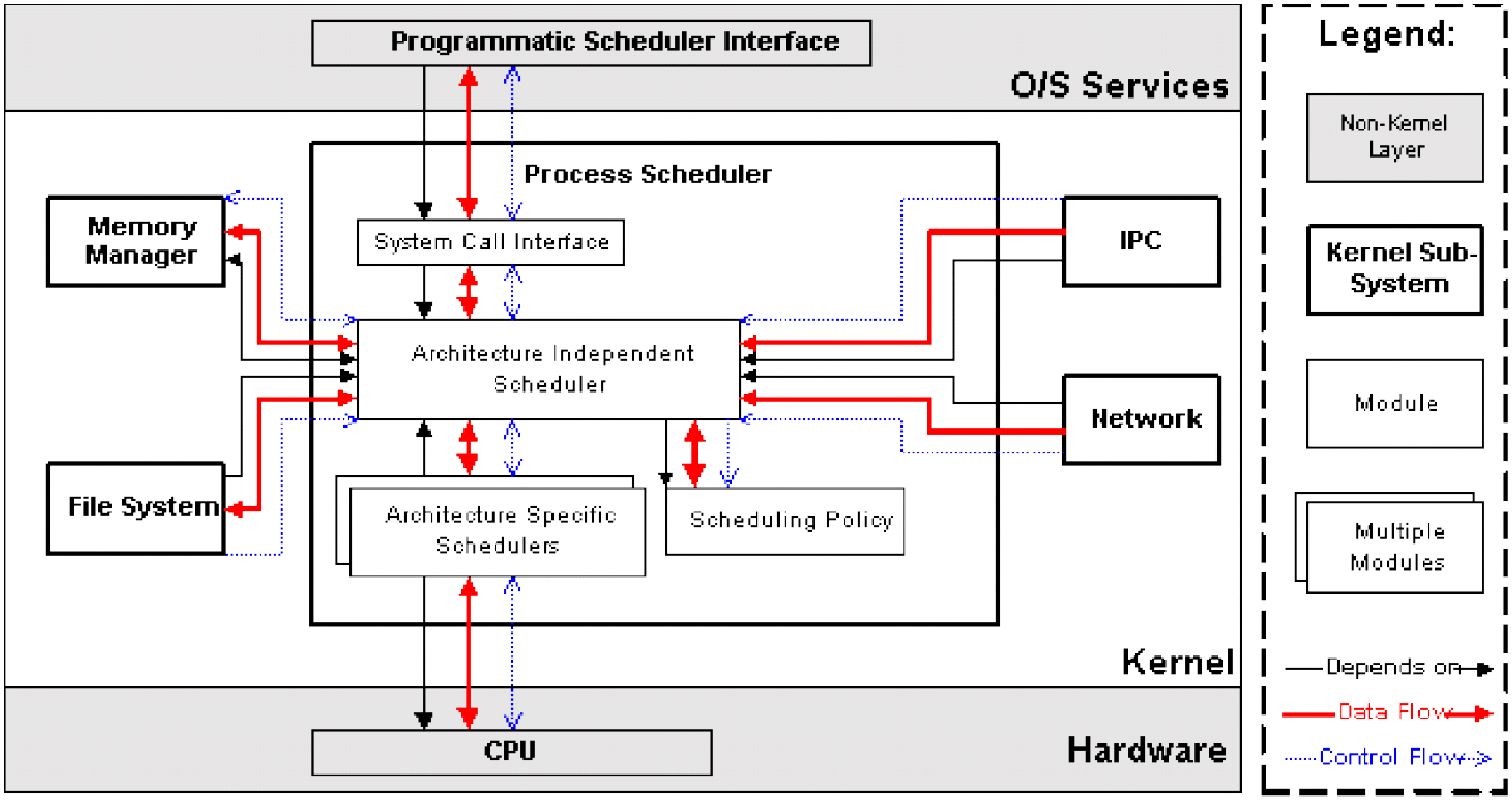
System Data Structures (2/2)

- I-Nodes
  - The VFS uses index-nodes to represent files on a logical file system
  - I-nodes data structure stores the mapping of file block numbers to physical device addresses
  - Can be shared across processes
- Data Connection
  - Pointers to
    - Memory mapping information
    - I-nodes representing all of the opened files
    - Data structures representing all of the opened network connections

Process Scheduler Architecture

- The scheduler is divided into four modules
  - The scheduling policy module judges which process will have access to the CPU
  - Architecture-specific modules abstract the details of any particular computer architecture; they communicate with a CPU to suspend and resume a process
  - The architecture-independent module communicates with the policy module to determine the next process, then calls the architecture-specific module to resume it. In addition, it calls the MM to ensure that the memory is restored properly for the resumed process
  - The system call interface module allows user to access only those resources that are explicitly exported by the kernel, limiting the dependency to a well-defined interface

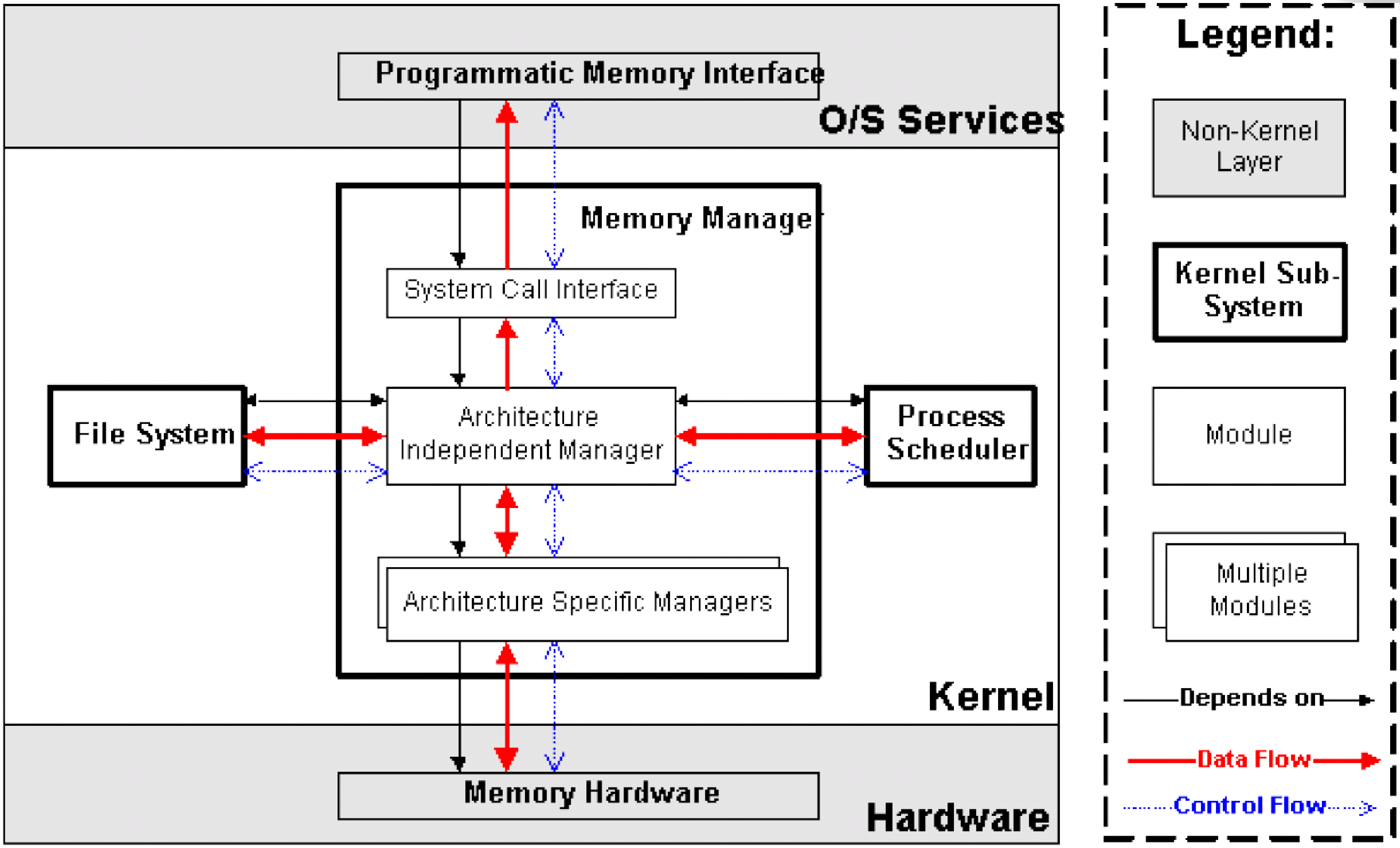
Process Scheduler Architecture



Memory Manager Architecture

- The MM is composed from three modules
  - The architecture specific module is a virtual interface to the memory management hardware
  - The architecture independent manager performs per-process mapping and virtual memory swapping; it determines which memory pages will be retrieved when there is a page fault (there is no separate policy module since it is not expected that this policy will need to change)
  - The system call interface provides restricted access to user processes; it allows them to allocate and free storage and to perform memory mapped file I/O

Memory Manager Architecture



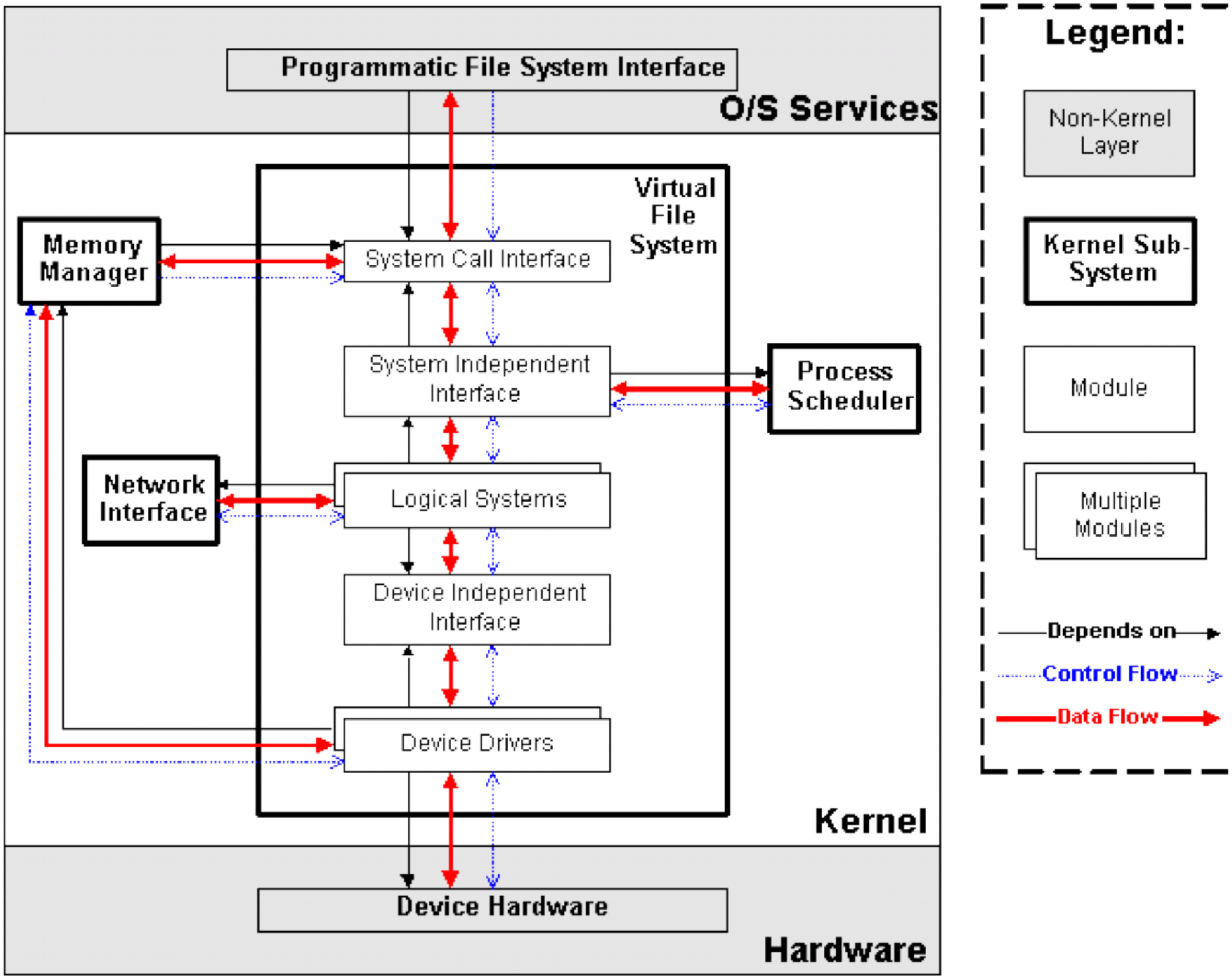
Virtual File System

- Presents a consistent view of data as stored on hardware devices
- Allows the system administrator to mount any of a set of logical file systems on any physical device
- Abstracts the details of both physical device and logical file system
- Is responsible for loading new executable programs, allowing Linux to support different executable formats

Virtual File System: modules

- One device driver module for each supported hardware controller
- The device independent interface module provides a consistent view of all devices
- One logical file system module for each supported file system
- The system independent interface presents all resources using either a block-oriented or character-oriented file interface
- The system call interface provides controlled access to the file system for user processes

Virtual File System



Network Architecture

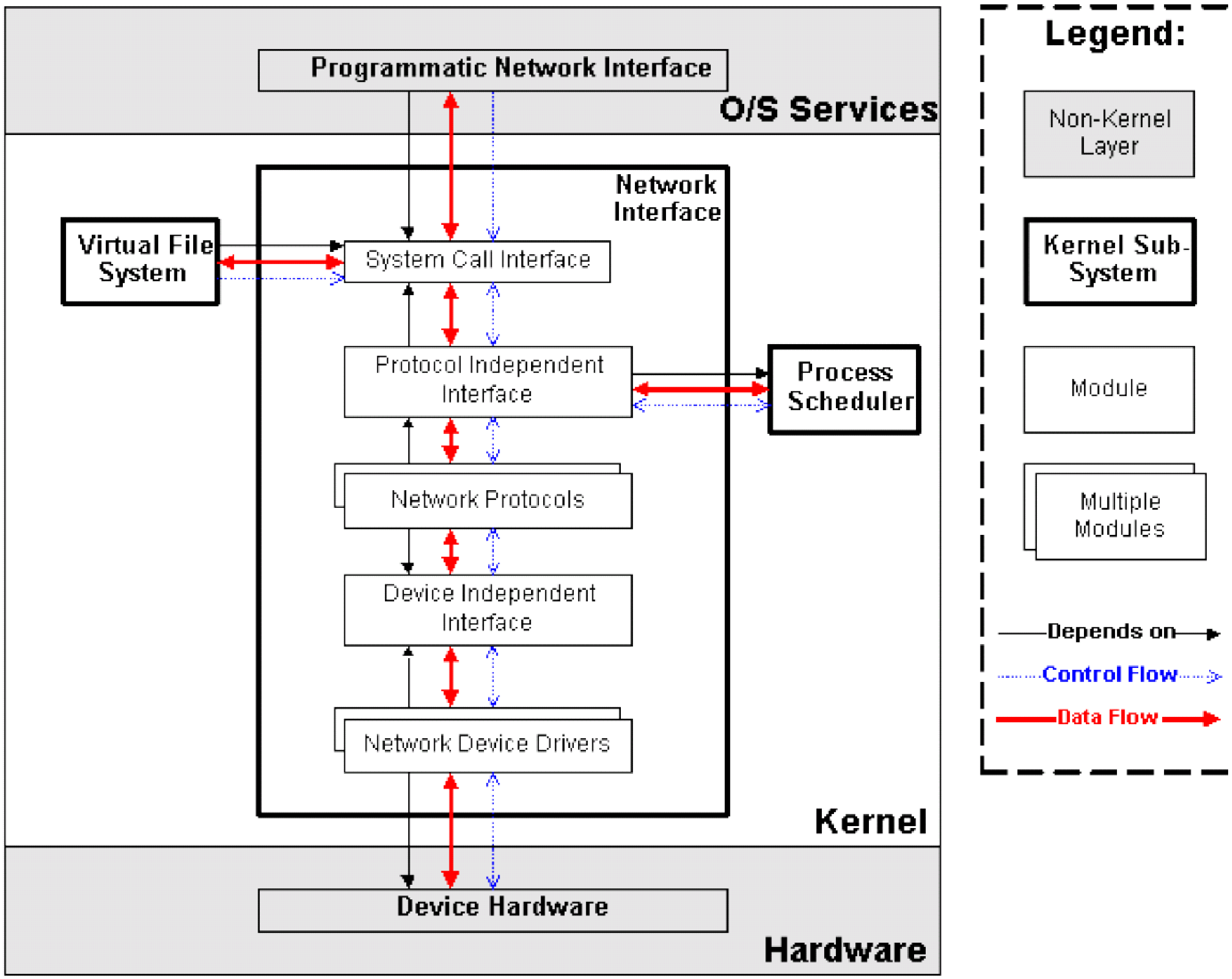
- Many hardware devices supported
- Many protocols can be used

- The Network subsystem abstracts both of these implementation details
  - User processes and other kernel subsystem access the network transparently

Network Architecture: modules

- One device driver module for each possible hardware device
- The device independent interface module provides a consistent view of all of the hardware devices
- The network protocol modules implement each of the possible network transport protocols
- The protocol independent interface module provides an interface that is independent of hardware devices and network protocols
  - Used by other kernel subsystems to access the network without dependencies on particular protocols or hardware

Network Architecture



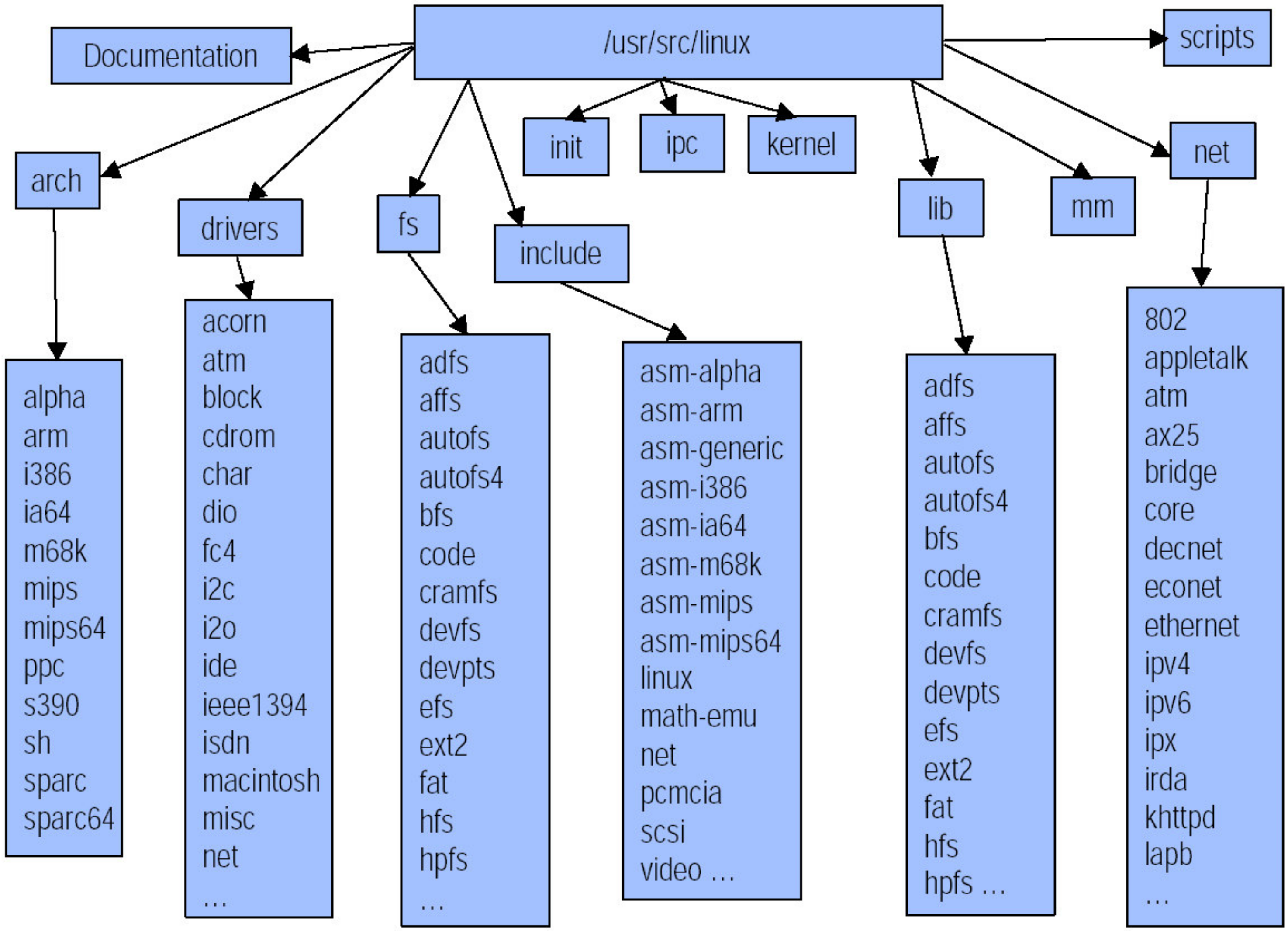
Overview

- Introduction
  - History
  - Linux Features
  - Anatomy of a Kernel
  - Linux Conceptual Architecture
- Details on the Linux Kernel
  - Source Tree
  - Recompiling
- Booting and Kernel Initialization
- System Calls

The Linux Source Tree

- Can be found at http://www.kernel.org
- Even numbers = release versions (2.2, 2.4, 2.6, etc.)
- Odd numbers = development versions (2.3, 2.5, etc.)
- At the present, version 2.6.10 is available
- Size in the order of ~200Mb once unpacked
  - Most of the code are drivers
- Several "trees"
  - The official Linus Torvalds' tree is the one at kernel.org

An Overview of the tree



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, checksum, …
- `arch` as of 2.6
  - alpha, arm, arm26, cris, h8300, i386, ia64, m68k, m68knommu, mips, parisc, ppc, ppc64, s390, sh, sparc, sparc64, um, v850, x86_64

linux/drivers

- Largest amount of code in the kernel tree (~90Mb!)
  - **`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, 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`**, **`telephony`**, **`video`**
- Buses : **`fc4`**, **`i2c`**, **`nubus`**, **`pci`**, **`sbus`**, **`tc`**, **`usb`**
- Platforms : **`acorn`**, **`macintosh`**, **`s390`**, **`sgi`**

linux/fs

- VFS framework
  - exec.c, binfmt_*.c : to map 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
- Subdirectories for actual filesystems
  - **`ext2`**, **`ext3`**, **`FAT`**, **`NFS`**, **`proc`**, **`reiserfs`**

linux/include

- **asm-***
  - Architecture-dependent include subdirectories
- **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**, **sound**, **video**

linux/init

- version.c contains the version banner that prints at boot
- main.c is the architecture-independent boot code
  - `start_kernel` is the primary entry point
- initramfs.c, do_mounts*

linux/ipc

- System V IPC facilities
- If disabled at compile-time, util.c exports stubs that simply return -ENOSYS
- One file for each facility
  - sem.c for semaphores
  - shm.c for shared memory
  - msg.c for message queues

linux/kernel

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

linux/lib

- Kernel code cannot call standard C library routines
  - 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

linux/net

- Network layer support
- `core` contains the main files
- Different directories for different protocols and technologies
  - `bluetooth`, `irda`, `ipv4`, `ipv6`, `x25`, `sunrpc`
- TUNABLE lists the parameters that should be tunable at compile-time
- socket.c in an implementation of the SOCKET network access protocol (top level interface to the BSD socket paradigm)

linux/scripts

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

Tree Summary

- Kernel is the heart of the OS that executes with special hardware permission (kernel mode)
- "Core kernel" provides framework, data structures, support for drivers, modules, subsystems

- Linux is a UNIX-like modular kernel
- Architecture dependent source sub-trees live in **`/arch`**

Compiling the Linux Kernel

- What's needed
  - gcc and linker
  - Kernel sources
- The kernel is compiled in several steps (2.4)
  - make distclean *(removing all unnecessary files)*
  - make xconfig / menuconfig / config *(configuring)*
  - make dep *(resolving dependencies)*
  - make bzImage *(compiling the core)*
  - make modules *(compiling loadable modules)*
  - make modules_install *(installing modules)*
  - make install *(installing the kernel)*

2.6 compilation

- Completely restyled
  - Few significant messages instead of (very) verbose compilation reports
- Fewer steps
  - make distclean
  - make xconfig / menuconfig / config
  - make
    - Includes make vmlinux, make bzImage, make modules
    - No need of make dep
  - make modules_install
  - make install

What's new in 2.6

- µClinux integration (embedded systems)
- NUMA support (larger and larger servers)
- Hyperthreading
- Preemptible kernel (*Robert Love*)
- O(1) scheduler (*Ingo Molnar*)
- RMAP patch for VM (*Rik van Riel*)
- New Kernel Device Structure (kdev_t)
- Improved POSIX threading support (NGPT and NPLT)
- New Driver model and Unified Device Structure
- AMD 64-bit, PPC64
- ACPI, ALSA, USB 2.0

Hyperthreading

- Ability of a single processor to masquerade at the hardware level as two or more processors
  - Currently only built into modern Pentium 4 processors but applicable elsewhere
- Performance boosts in many circumstances
- Adds scheduling complexity and other issues
- The scheduler now knows how to recognize and optimize processor loads across both real and virtual processors

Preemption

- In previous versions it was NOT possible to preempt a task running in kernel mode
  - User processes enter in kernel mode after a syscall
- Not all sections of the kernel code can be preempted!
  - Critical sections of code must be protected against preemption
    - Hardware specific sections
  - Locking mechanism
    - `preempt_disable()`, `preempt_enable()`
      - Nested functions: they operate on a counter
    - `get_cpu()`, `put_cpu()`
      - They call previous functions plus `smp_processor_id()`

O(1) scheduler

- In SMP systems, recalculation loop for timeslices causes processors to enter idle state waiting for all the processes to expire their timeslice
  - Bouncing processes: processes waiting for a CPU are re-scheduled on different queues
- O(1) algorithm maintains data structures on a per-CPU basis
  - Active array: processes with some timeslice left
  - Expired array: processes with no more timeslices
  - When active array is empty access pointers are switched
- Better load balancing on SMP systems

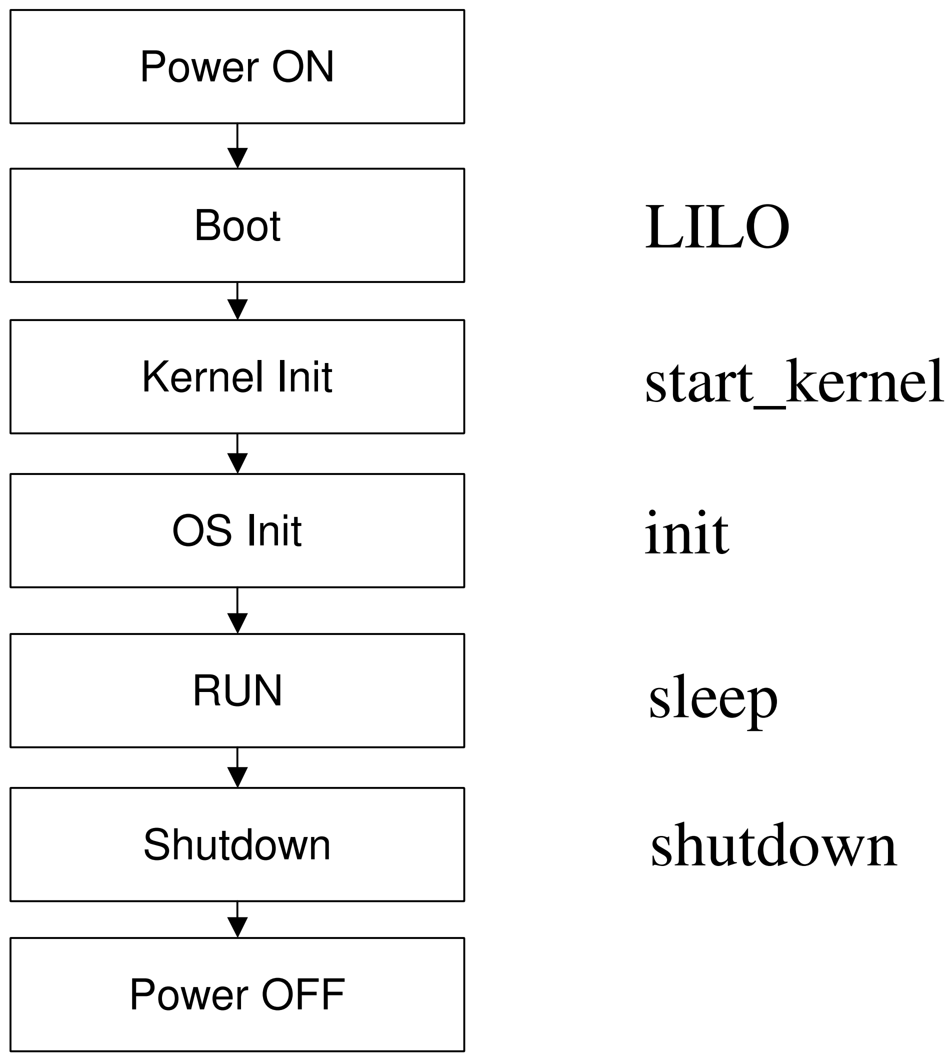
Reverse mapping patch

- Virtual-to-physical page translation thanks to Task List and Memory Map data structures
  - Pages can be shared among processes
- Identifying a page to overwrite means scanning the whole structure
  - Reference count = 0 -> the page can be overwritten
- `pte_chain` data structure that reports for every page references it has
  - A page to be overwritten can be easily identified
  - Overhead in terms of wasted space

Overview

- Introduction
  - History
  - Linux Features
  - Anatomy of a Kernel
  - Linux Conceptual Architecture
- Details on the Linux Kernel
  - Source Tree
  - Recompiling
- Booting and Kernel Initialization
- System Calls

System Lifecycle



Boot terminology

- Loader
  - Program that moves bits from disk (usually) to memory and then transfers CPU control to the newly "loaded" bits (executable)
- Bootloader / Bootstrap
  - Program that loads the "first program" (the kernel)
- Boot PROM / PROM Monitor / BIOS
  - Persistent code that is "already loaded" on power-up
- Boot Manager
  - Program that lets you choose the "first program" to load

LILO: LInux LOader

- A versatile boot manager that supports:
  - Choice between different Linux kernels
  - Boot time kernel parameters
  - Booting non-Linux kernels
- Main characteristics
  - Lives in MBR or partition boot sector
  - Has no knowledge of filesystem structure
  - Builds a sector "map file" (block map) to find kernel
- /sbin/lilo is the "map installer"
- /etc/lilo.conf is the configuration file for LILO
- Only for x86 architectures!

An alternative: GRUB

- Provides the same features of LILO
- Doesn't need to be rerun when adding a new kernel
- Compliant with the Multiboot specification
- Chaining functions
- Future support for different architectures
- `/boot/grub/grub.conf` is the configuration file for GRUB

/sbin/init

- Ancestor of all processes (except idle/swapper process)
- Controls transitions between "runlevels":
  - 0: shutdown
  - 1: single-user
  - 2: multi-user (no NFS)
  - 3: full multi-user
  - 5: X11
  - 6: reboot
- Executes startup/shutdown scripts for each runlevel

Shutdown

- Use `/bin/shutdown` to avoid data loss and filesystem corruption
- Shutdown inhibits login, asks init to send SIGTERM to all processes, then SIGKILL
- Low-level commands: halt, reboot, poweroff
  - Use -h, -r or -p options to shutdown instead
- Ctrl-Alt-Delete "Vulcan neck pinch":
  - defined by a line in `/etc/inittab`
    - ca::ctrlaltdel:/sbin/shutdown -t3 -r now
  - `/etc/shutdown.allow`

Advanced Boot Concepts

- Initial ramdisk (initrd) : two-stage boot for flexibility
  - First mount "initial" ramdisk as root
  - Execute linuxrc to perform additional setup, configuration
  - Finally mount "real" root and continue
  - Documentation/initrd.txt or "man initrd" for details
- Net booting
  - Remote root (Diskless-root-HOWTO)
  - Diskless boot (Diskless-HOWTO)

Booting summary (1/2)

- Bootstrapping a system is a complex, device-dependent process
  - It involves transition from hardware, to firmware, to software
- Booting within the constraints of the Intel architecture is especially complex
  - Usually involves firmware support (BIOS) and a boot manager (LILO)

Booting summary (2/2)

- `/sbin/lilo`
  - reads configuration information
  - writes a boot sector and block map files used during boot
- `start_kernel` is Linux "main"
  - sets up process context before spawning process 0 (idle) and process 1 (init)
- The `init()` function performs high-level initialization before executing the user-level init process

Overview

- Introduction
  - History
  - Linux Features
  - Anatomy of a Kernel
  - Linux Conceptual Architecture
- Details on the Linux Kernel
  - Source Tree
  - Recompiling
- Booting and Kernel Initialization
- System Calls

System Calls

- Interfaces between user-level processes and hardware devices
  - CPU, memory, disks, …
- 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 Programming Interface)
  - Function definition specifying how to obtain a service
- A system call is an explicit request to the kernel made via a software interrupt
- Standard C library (libc) contains wrapper routines that make system calls
  - E.g., **`malloc`** and **`free`** are libc routines that use the **`brk`** system call
- POSIX-compliant : having a standard set of APIs
- Non-UNIX systems can be POSIX-compliant if they offer the required set of APIs

Linux System Calls (1/2)

- Invoked by executing `int $0x80`
  - Programmed exception vector number 128
  - CPU switches to kernel mode and executes a kernel function
- Calling process passes to the `syscall` a number identifying system call in `eax` register (on Intel processors)
- Syscall handler is responsible for:
  - saving registers on kernel mode stack
  - invoking syscall service routine
  - exiting by calling `ret_from_sys_call()`

Linux System Calls (2/2)

- 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)
  - *n*-th entry contains service routine address of syscall *n*

Initializing System Calls

- `trap_init()` called during kernel initialization sets up the IDT (Interrupt Descriptor Table) entry corresponding to vector 128

  ```
  set_system_gate(0x80, &system_call);
  ```

- A system gate descriptor is placed in the IDT, identifying address of `system_call` routine
  - Does not disable maskable interrupts
  - Sets the DPL (Descriptor Privilege Level) to 3
    - Allows User Mode processes to invoke exception handlers (i.e. syscall routines)

The `system_call()` function

- Saves syscall number and CPU registers used by exception handler on the stack, except those automatically saved by Control Unit
- Checks for valid system call
- Invokes specific service routine associated with syscall number (contained in eax)

  ```
  call *sys_call_table(0, %eax, 4)
  ```

- Return code of system call is stored in **`eax`**

Parameter Passing

- On the 32-bit Intel 80x86 6 registers are used to store syscall parameters
  - **`eax`** (syscall number)
  - **`ebx`**, **`ecx`**, **`edx`**, **`esi`**, **`edi`** store parameters to syscall service routine identified by syscall number

Wrapper Routines

- Kernel code (e.g., kernel threads) cannot use library routines
- `_syscall0` … `_syscall5` macros define wrapper routines for system calls with up to 5 parameters
  - E.g.: `_syscall3(int write,int fd, const char* buf,unsigned int count)`

Linux files related to syscalls

- **`arch/i386/kernel/entry.S`**
  - System call and low-level fault handling routines
- **`include/asm-i386/unistd.h`**
  - System call numbers and macros
- **`kernel/sys.c`**
  - System call service routines

arch/i386/kernel/entry.S

```
.data
ENTRY(sys_call_table)
.long SYMBOL_NAME(sys_ni_syscall) /* 0 old
  "setup()" system              call*/
  .long SYMBOL_NAME(sys_exit)
  .long SYMBOL_NAME(sys_fork)
  .long SYMBOL_NAME(sys_read)
  .long SYMBOL_NAME(sys_write)
```

- Add system calls by appending entry to sys_call_table

  `.long SYMBOL_NAME(sys_my_system_call)`

include/asm-i386/unistd.h

- Each system call needs a number in the system call table
  - E.g., `#define __NR_write 4`
  - `#define __NR_my_system_call nnn`, where `nnn` is next free entry in system call table

kernel/sys.c

- Service routine bodies are defined here
  - E.g.,

```
asmlinkage retval sys_my_system_call
  (parameters) {
    // body of service routine
    return retval;
   }
```


Useful documents

- Daniel P. Bovet, Marco Cesati – Understanding the Linux Kernel
- Documentation directory in the kernel tree source
- LDP : Linux Documentation Project
- kernelnewbies.org/status/latest.html
- Joe Pranevic – The Wonderful World of Linux 2.6
- Dave Jones – 2.6: what's to expect
- Anand K Santhaman – Towards Linux 2.6