The Linux Kernel: Introduction

History

- UNIX: 1969 Thompson & Ritchie AT&T Bell Labs.
- Commercial Vendors: Sun, HP, IBM, SGI, DEC.
- GNU: 1984 Richard Stallman, FSF.
- POSIX: 1986 IEEE Portable Operating System UNIX.
- Linux: 1991 Linus Torvalds Intel 386 (i386).
- Open Source: GPL.
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.

What’s a Kernel?

- AKA: 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).
- Etc…etc…
Kernel Design Goals

- Performance: efficiency, speed.
  - Utilize resources to capacity with low overhead.
- Stability: robustness, resilience.
  - Uptime, graceful degradation.
- Capability: features, flexibility, compatibility.
  - Security, protection.
    - Protect users from each other & system from bad users.
- Portability.
- Extensibility.

Example “Core” Kernel
Architectural Approaches

- Monolithic.
- Layered.
- Modularized.
- Micro-kernel.
- Virtual machine.
**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 2.4:
  - alpha, arm, i386, ia64, m68k, mips, mips64.
  - ppc, s390, sh, sparc, sparc64.

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

 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 – semaphores.
  - shm.c – shared memory.
  - msg.c – message queues.
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`, `timer.c`.
  - `printk.c`, `info.c`, `panic.c`, `sysctl.c`, `sys.c`.

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.
  - `vscanf.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`, `mmremap.c`, `mlock.c`, `mprotect.c`.

### linux/scripts

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

- Linux is a modular, UNIX-like monolithic kernel.
- 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.
- Architecture dependent source sub-trees live in /arch.

Booting and Kernel Initialization
System Lifecycle: Ups & Downs

Power on → LIO → start kernel → init → sleep? (h) → shutdown → Power off

Boot: Program that loads the “first program” (the kernel).

Bootloader / Bootstrap: Program that moves bits from disk (usually) to memory and then transfers CPU control to the newly “loaded” bits (executable).

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.

Boot Terminology
LILO: LInux LOader

- A versatile boot manager that supports:
  - Choice of Linux kernels.
  - Boot time kernel parameters.
  - Booting non-Linux kernels.
  - A variety of configurations.

- Characteristics:
  - Lives in MBR or partition boot sector.
  - Has no knowledge of filesystem structure so…
  - Builds a sector “map file” (block map) to find kernel.

- `/sbin/lilo` – “map installer”.
- `/etc/lilo.conf` is lilo configuration file.

Example lilo.conf File

```
boot=/dev/hda
map=/boot/map
install=/boot/boot.b
prompt
timeout=50
default=linux

image=/boot/vmlinuz-2.2.12-20
  label=linux
  initrd=/boot/initrd-2.2.12-20.img
  read-only
  root=/dev/hda1
```
/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.
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.
  - See Documentation/initrd.txt for details.
  - Also see “man initrd”.

- Net booting:
  - Remote root (Diskless-root-HOWTO).
  - Diskless boot (Diskless-HOWTO).

Summary

- Bootstrapping a system is a complex, device-dependent process that involves transition from hardware, to firmware, to software.
- Booting within the constraints of the Intel architecture is especially complex and usually involves firmware support (BIOS) and a boot manager (LILO).
- /sbin/lilo is a “map installer” that reads configuration information and writes a boot sector and block map files used during boot.
- start_kernel is Linux “main” and 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.
System Calls

- 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 defn 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.
- 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)

Invoked by executing `int $0x80`.
- Programmed exception vector number 128.
- CPU switches to kernel mode & executes a kernel function.
- Calling process passes syscall number identifying system call in `eax` register (on Intel processors).
- Syscall handler responsible for:
  - Saving registers on kernel mode stack.
  - Invoking syscall service routine.
  - Exiting by calling `ret_from_sys_call()`.
Linux System Calls (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).
  - nth 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 descriptor privilege level (DPL) to 3:
    - Allows User Mode processes to invoke exception handlers (i.e. syscall routines).
The system_call() Function

- Saves syscall number & 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):
  - \texttt{call *sys\_call\_table(0, \%eax, 4)}
- Return code of system call is stored in \texttt{eax}.

Parameter Passing

- On the 32-bit Intel 80x86:
  - 6 registers are used to store syscall parameters.
    - \texttt{eax} (syscall number).
    - \texttt{ebx}, \texttt{ecx}, \texttt{edx}, \texttt{esi}, \texttt{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)

Example: “Hello, world!”

```assembly
.data
    .section .data
msg:
    .string "Hello, world!\n"
len = . - msg
len = . - msg

.text
    .section .text
    .global _start
    _start:
# write our string to stdout
  movl $len, %edx       # third argument: message length
  movl $msg, %ecx       # second argument: pointer to message to write
  movl $1, %ebx         # first argument: file handle (stdout)
  movl $4, %eax         # system call number (sys_write)
  int $0x80             # call kernel
# and exit
  movl $0, %ebx         # first argument: exit code
  movl $1, %eax         # system call number (sys_exit)
  int $0x80             # call kernel
```

Linux Files Relating to Syscalls

- Main files:
  - 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;
    }


Kernel Modules

- Modules can be compiled and dynamically linked into kernel address space.
  - 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 too!
Example: “Hello, world!”

```c
#define MODULE
#include <linux/module.h>
int init_module(void) {
    printk("<1>Hello, world!\n");
    return 0;
}

void cleanup_module(void) {
    printk("<1>Goodbye cruel world  \n");
}
```

Using Modules

- Module object file is installed in running kernel using `insmod module_name`.
  - Loads module into kernel address space and links unresolved symbols in module to symbol table of running kernel.
The Kernel Symbol Table

- Symbols accessible to kernel-loadable modules appear in `/proc/ksyms`.
  - `register_symtab` registers a symbol table in the kernel's main table.

- Real hackers export symbols from the kernel by modifying `kernel/ksyms.c`

Project Suggestions (1)

- Real-Time thread library.
- Scheduler activations in Linux.
- A Linux “upcall” mechanism.
- Real-Time memory allocator / garbage collector.
- A distributed shared memory system.
- A QoS-based socket library.
- An event-based mechanism for implementing adaptive systems.
- DWCS packet scheduling.
- A heap-based priority scheduler for Linux.
Project Suggestions (2)

- µS resolution timers for Linux.
- Porting the Bandwidth-Broker to Linux.
- A QoS Management framework like QuO or Dionisys.
- A Real-Time communications protocol.
- A feedback-control system for flow/error/rate/congestion control.
- “Active Messages” for Linux.
- A thread continuation mechanism.
- A thread migration / load-balancing system.