Linux Internals

1. Booting

1.1 Building the Linux Kernel Image

- We discuss building specific to x86 architecture
- When the user types 'make zImage' or 'make bzImage' the resulting bootable kernel image is stored as arch/i386/boot/zImage or arch/i386/boot/bzImage respectively.
- The size of the bootsector is always 512 bytes
- The upper limit on the bzImage size produced at this step is about 2.5M for booting with LILO
- It is easy to build a broken kernel by just adding some large ".space" at the end of setup.S.

1.2 Booting: Overview

- Process steps for Intel 32 bit architectures
- BIOS selects the boot device.
- BIOS loads the bootsector from the boot device.
- Bootsector loads setup, decompression routines and compressed kernel image.
- The kernel is uncompressed in protected mode.
- Low-level initialisation is performed by asm code.
- High-level C initialisation.

1.3 Booting: BIOS POST

- The power supply starts the clock generator and asserts #POWERGOOD signal on the bus.
- CPU #RESET line is asserted (CPU now in real 8086 mode).
- %ds=%es=%fs=%gs=%ss=0, %cs=0xFFFF0000,%eip = 0x0000FFF0 (ROM BIOS POST code).
- All POST checks are performed with interrupts disabled.
- IVT (Interrupt Vector Table) initialised at address 0.
- The BIOS Bootstrap Loader function is invoked via **int 0x19**, with %dl containing the boot device 'drive number'. This loads track 0, sector 1 at physical address 0x7C00 (0x07C0:0000).

1.4 Booting: bootsector and setup

- The bootsector used to boot Linux kernel could be either:
  - Linux bootsector (arch/i386/boot/bootsect.S),
  - LILO (or other bootloader's) bootsector, or
  - no bootsector (loadlin etc)
- actual code is in bootsect.s

1.5 Using LILO as a bootloader

- There are several advantages in using a specialised bootloader (LILO) over a bare bones Linux bootsector:
  - Ability to choose between multiple Linux kernels or even multiple OSes.
  - Ability to pass kernel command line parameters (there is a patch called BCP that adds this ability to bare-bones bootsector+setup).
  - Ability to load much larger bzImage kernels - up to 2.5M vs 1M.

1.6 High level initialization

- By "high-level initialisation" we consider anything which is not directly related to bootstrap
- The following steps are performed:
  - Initialise segment values (%ds = %es = %fs = %gs = __KERNEL_DS = 0x18).
  - Initialise page tables.
  - Enable paging by setting PG bit in %cr0.
  - Zero-clean BSS (on SMP, only first CPU does this).
  - Copy the first 2k of bootup parameters (kernel commandline).
  - Check CPU type using EFLAGS and, if possible, cpuid, able to detect 386 and higher.
  - The first CPU calls start_kernel()

1.7 SMP Bootup on x86

- The steps for Symmetric Multiprocessing Boot are the following
  - On SMP, the BP goes through the normal sequence of bootsector, setup etc until it reaches the start_kernel(), and then on to smp_init()
  - The smp_boot_cpus() goes in a loop for each apicid (until NR_CPUS) and calls do_boot_cpu()
  - The boot CPU creates a copy of trampoline code for each CPU in low memory
  - The trampoline code simply sets %bx register to 1, enters protected mode and jumps to startup_32

1.8 Freeing initialization data and code

- When the operating system initialises itself, most of the code and data structures are never needed again
- Linux provides two macros to be used:
  - __init - for initialisation code
  - __initdata - for data
- There are two more macros which work in a similar manner, called __exit and __exitdata

1.9 Processing kernel command line

- The following are the steps to process the command line
  - LILO (or BCP) accepts the commandline using BIOS keyboard services and stores it at a well-known location in physical memory, as well as a signature saying that there is a valid commandline there.
  - arch/i386/kernel/head.S copies the first 2k of it out to the zeropage.
  - arch/i386/kernel/setup.c:parse_mem_cmdline() (called by setup_arch(), itself called by start_kernel()) copies 256 bytes from zeropage into saved_command_line which is displayed by /proc/cmdline.
  - We return to commandline in parse_options() (called by start_kernel()) which processes some "in-kernel" parameters (currently "init=" and environment/arguments for init) and passes each word to checksetup().
  - checksetup() goes through the code in ELF section .setup.init and invokes each function, passing it the word if it matches

2. Process and Interrupt Management

- 2.1 Task Structure and Process Table
- 2.2 Creation and termination of tasks and kernel threads
- 2.3 Linux Scheduler
- 2.4 Linux linked list implementation
- 2.5 Wait Queues

2.1 Task Structure and Process Table

- Every process under Linux is dynamically allocated a struct task_struct structure
- The maximum number of processes which can be created on Linux is limited only by the amount of physical memory
- The set of processes on the Linux system is represented as a collection of struct task_struct structures which are linked in two ways:
  - as a hashtable, hashed by pid, and
  - as a circular, doubly-linked list using p->next_task and p->prev_task pointers.

2.2 Creation and termination of tasks and kernel threads

- Under Linux, there are three kinds of processes:
  - The idle thread(s),
  - kernel threads,
  - user tasks.
- The idle thread is created at compile time for the first CPU; it is then "manually" created for each CPU by means of arch-specific fork_by_hand()
- User tasks are created by means of **clone(2)** or **fork(2)** system calls
- There are several ways for tasks to terminate:
  - by making **exit(2)** system call;
  - by being delivered a signal with default disposition to die;
  - by being forced to die under certain exceptions;
  - by calling **bdflush(2)** with func == 1

2.3 Linux Scheduler

- The job of a scheduler is to arbitrate access to the current CPU between multiple processes
- The fields of task structure relevant to scheduler include
  - p->need_resched: this field is set if schedule() should be invoked at the 'next opportunity'.
  - p->counter: number of clock ticks left to run in this scheduling slice, decremented by a timer. When this field becomes lower than or equal to zero, it is reset to 0 and p->need_resched is set. This is also sometimes called 'dynamic priority' of a process because it can change by itself.
  - p->priority: the process' static priority, only changed through well-known system calls like **nice(2)**, POSIX.1b **sched_setparam(2)** or 4.4BSD/SVR4 **setpriority(2)**.
  - p->rt_priority: realtime priority
  - p->policy: the scheduling policy, specifies which scheduling class the task belongs to

2.4 Linux linked list implementation

- Before we go on to examine implementation of wait queues, we must acquaint ourselves with the Linux standard doubly-linked list implementation
- The fundamental data structure here is struct list_head
- struct list_head

```
{
    struct list_head *next, *prev;
};
```

2.5 Wait Queues

- When a process requests the kernel to do something which is currently impossible but that may become possible later, the process is put to sleep and is woken up when the request is more likely to be satisfied. One of the kernel mechanisms used for this is called a 'wait queue'.
- you can define your own waitqueue and use add/remove_wait_queue to add and remove yourself from it
- wake_up/wake_up_interruptible can be used to wake up when needed
- An example of autonomous waitqueue usage is interaction between user process requesting data via **read(2)** system call

3. Virtual Filesystem (VFS)

- 3.1 Inode Caches and Interaction with Dcache
- 3.2 Filesystem Registration/Unregistration
- 3.3 File Descriptor Management
- 3.4 File Structure Management
- 3.5 Superblock and Mountpoint Management

3.1 Inode Caches and Interaction with Dcache

- In order to support multiple filesystems, Linux contains a special kernel interface level called VFS (Virtual Filesystem Switch).
- Linux inode cache is implemented in a single file, fs/inode.c, which consists of 977 lines of code
- The type lists are anchored from inode->i_list, the hashtable from inode->i_hash
- All these lists are protected by a single spinlock: inode_lock.

3.2 Filesystem Registration/Unregistration

- The Linux kernel provides a mechanism for new filesystems to be written with minimum effort
- All that is needed is to fill in a struct file_system_type structure and register it with the VFS using the register_filesystem()
- The job of the read_super() function is to fill in the fields of the superblock, allocate root inode and initialise any fs-private information associated with this mounted instance of the filesystem

3.3 File Descriptor Management

- Under Linux there are several levels of indirection between user file descriptor and the kernel inode structure
- When a process makes **open(2)** system call, the kernel returns a small non-negative integer which can be used for subsequent I/O operations on this file.
- This integer is an index into an array of pointers to struct file
- Each file structure points to a dentry via file->f_dentry
- And each dentry points to an inode via dentry->d_inode.
- Each task contains a field tsk->files which is a pointer to struct files_struct

3.4 File Structure Management

- The file structure is declared in include/linux/fs.h
- Let us look at the various fields of struct file:
  - **f_list** , **f_dentry** , **f_vfsmnt** , **f_flags** etc
- file_operations structure which contains the methods that can be invoked on files

3.5 Superblock and Mountpoint Management

- Under Linux, information about mounted filesystems is kept in two separate structures - super_block and vfsmount
- struct super_block is declared in include/linux/fs.h
- The various fields in the super_block structure are
  - **s_list** , **s_dev** , **s_dirty** , **s_files** etc
- The superblock operations are described in the super_operations structure

4. IPC mechanisms

- 5.1 Semaphores
- 5.2 Message queues
- 5.3 Shared Memory
- 5.4 Linux IPC Primitives

4.1 Semaphores

- The functions described in this section implement the user level semaphore mechanisms
- System calls are…
  - sys_semget()
  - sys_semctl()
  - sys_semop()
- The following structures are used specifically for semaphore support
  - struct sem_array
  - struct sembuf

4.2 Message queues

- Message System Call Interfaces
  - sys_msgget()
  - sys_msgctl()
  - sys_msgsnd()
  - sys_msgrcv()
- Message Specific Structures
  - struct msg_queue
  - struct msqid_ds

4.3 Shared Memory

- Shared Memory System Call Interfaces
  - sys_shmget()
  - sys_shmctl()
  - sys_shmat()
  - sys_shmdt()
- Shared Memory Support Structures
  - struct shm_info
  - struct shmid64_ds

4.4 Linux IPC Primitives

- Generic Linux IPC Primitives used with Semaphores, Messages,and Shared Memory
  - ipc_alloc()
  - ipc_addid()
  - ipc_lock()
- Generic IPC Structures used with Semaphores,Messages, and Shared Memory
  - struct kern_ipc_perm
  - struct ipc_id

Embedded Linux

Overview

- **Embedded Linux** is the designation for Linux-based operating systems that are used as
  - Cell phones
  - Personal digital assitants
  - Media player handsets
  - Consumer electronic devices etc.
- Other embedded applications too…
  - Networking equipment
  - Machine control
  - Industrial automation
  - Navigation equipment

Differences

- Embedded Linux can be characterized as different from desktop and server versions of Linux
- Limited resources in terms of RAM, secondary storage
- Frequently involves flashdisk based memories instead of hard disks
- Embedded Linux is tailor made specific to required application and target hardware
- Optimizations are done in order to make the OS Real Time OS (RTOS)

Advantages

- the source code can be modified and redistributed
- relatively small footprint (a typical installation may require less than two megabytes of memory)
- no royalty or licensing costs
- mature and stable
- large support base

The Linux Foundation

- The Linux Foundation manages the ELC Platform specification
- ELC is Embedded Linux Consortium
- It defines the set of standard interfaces to be employed while developing embedded Linux

System Enviroments

- Minimal System Environment: Requires no user interaction. Only minor storage
- Intermediate System Enviroment: Adds support for massive file storage, Asynchronous I/O etc
- Full System Environment: A Full, multipurporse Linux

Enviroment Function Group Table

- R – Required, P – Optional (POSIX), L – Optional (LSB 1.2)

	Minimal SE	Intermediate SE	Full SE
NGROUPS_MAX			>=8
_POSIX_CHOWN_RESTRICTED			R
_POSIX_FSYNC	R	R	R
_POSIX_JOB_CONTROL			R
_POSIX_MESSAGE_PASSING	R	R	R
_POSIX_NO_TRUNC	R	R	R
_POSIX_REGEXP			R
_POSIX_READER_WRITER_LOCKS	R	R	R
_POSIX_SAVED_IDS			R
_POSIX_VDISABLE			R

Environment Function Group Table(2)

	Minimal SE	Intermediate SE	Full SE
ELC_ASYNCHRONOUS_IO		R	R
ELC_C_LANG_JUMP		R	R
ELC_C_LANG_MATH			R
ELC_C_LANG_SUPPORT	R	R	R
ELC_C_LANG_SUPPORT_R	R	R	R
ELC_C_LIB_EXT		R	R
ELC_DEVICE_IO		R	R
ELC_DEVICE_SPECIFIC			R
ELC_DEVICE_SPECIFIC_R			R
ELC_DYNAMIC_LINKING		R	R
ELC_FD_MGMT		R	R

Interface Function Groups

- Each group's elements will be separated to indicate the specification upon which they are based:
  - POSIX.1-2001 is a reference to IEEE POSIX 1003.1-2001, including Rationale
  - LSB1.2 is a reference to Linux Standard Base Version 1.2.0
  - SUSv3 is a reference to the Single UNIX Specification, Version 3
- Threads
  - LSB 1.2 Based
  - POSIX Based

Feature Functions and Macros

- A conforming implementation shall make available an <elcstd.h> header, defining the symbolic constants and types described in this section. The actual values of the constants are unspecified except as shown.
- The following symbolic constants shall be defined in <elcstd.h>:
- _ELCPS_VERSION
- Long integer value indicating version of ELCPS to which the implementation conforms. For implementations conforming to this particular version, the value shall be 200212L.

Constants

_ELCPS_MINIMAL_ENV

The implementation supports the Minimal System Environment. If this symbol has a value other than -1 or 0, it shall have the value 200212L.

_ELCPS_INTERMEDIATE_ENV

The implementation supports the Intermediate System Environment. If this symbol has a value other than -1 or 0, it shall have the value 200212L.

_ELCPS_FULL_ENV

The implementation supports the Full System Environment. If this symbol has a value other than -1 or 0, it shall have the value 200212L.

_ELC_ASYNCHRONOUS_IO

The implementation supports the Asynchronous I/O interface group. If this symbol has a value other than -1 or 0, it shall have the value 200212L.

_ELC_C_LANG_JUMP

The implementation supports the ISO C Library Jump Functions interface group. If this symbol has a value other than -1 or 0, it shall have the value 200212L.

_ELC_C_LANG_MATH

The implementation supports the Math Functions interface group. If this symbol has a value other than -1 or 0, it shall have the value 200212L.

References

- The Embedded Linux Consortium

  http://www.linux-foundation.org/en/ELC/ELCPS
- The Linux Kernel FAQs
  - http://www.faqs.org