Linux Device Drivers explained

Giuseppe Calderaro

Imagination technologies ltd.

29th October 2009

Table of contents

Linux system architecture

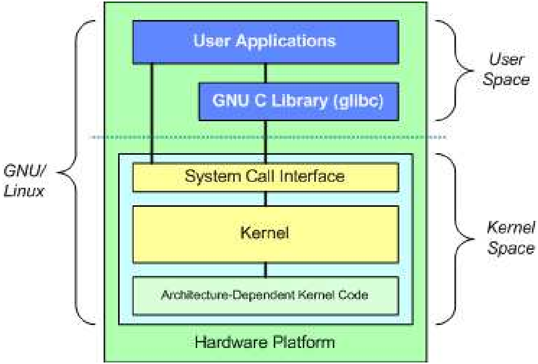

Figure: Linux system architecture

Linux kernel architecture

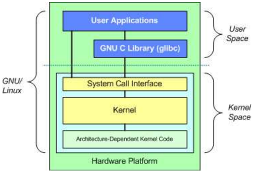

Figure: Linux system architecture

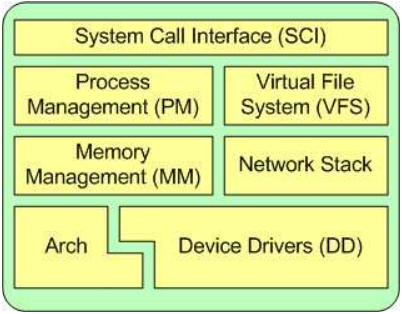

Figure: Linux kernel architecture

A polite module...


```
#include <linux/module.h>

int example_init(void)
{
        printk(KERN_CRIT "Hello world! :-)\n");

        return 0;
}

void example_exit(void)
{
        printk(KERN_CRIT "Goodbye cruel world... :-(\n");
}

module_init(example_init);
module_exit(example_exit);

MODULE_LICENSE("GPL");
```

...and his Makefile

```
ifndef KERNELRELEASE
        LINUX ?= /usr/src/linux
        PWD := $(shell pwd)
all:
        $(MAKE) -C $(LINUX) SUBDIRS=$(PWD) modules
clean:
        rm -rf *.o *.ko *~ *.symvers *.mod.c *.order
else
obj-m := example.o
endif
```

printk

- printf("I am printf!");
- printk("I am printk!");

printk

- printf("I am printf!");
- printk("I am printk!");
- printf prototype: int printf(const char *format, ...);
- printk prototype: int printk(const char *fmt, ...);

printk

- printf("I am printf!");
- printk("I am printk!");
- printf prototype: int printf(const char *format, ...);
- printk prototype: int printk(const char *fmt, ...);

Log level values

- **#define** KERN_EMERG "<0>"

  #define KERN_ALERT "<1>"

Log level values

- **#define** KERN_EMERG "<0>"

  #define KERN_ALERT "<1>"

Log level values


```
#define KERN_EMERG   "<0>" /* system is unusable         */
#define KERN_ALERT   "<1>" /* action must be taken       */
#define KERN_CRIT    "<2>" /* critical conditions        */
#define KERN_ERR     "<3>" /* error conditions           */
#define KERN_WARNING "<4>" /* warning conditions         */
#define KERN_NOTICE  "<5>" /* normal but significant     */
#define KERN_INFO    "<6>" /* informational              */
#define KERN_DEBUG   "<7>" /* debug-level messages       */
```

mmm... proc... what's that?!?

- [giuseppe@wopr]$ cat /proc/sys/kernel/printk
  3 4 1 7
- Current Default Minimum Boottime

mmm... proc... what's that?!?

- [giuseppe@wopr]$ cat /proc/sys/kernel/printk
  3 4 1 7
- Current Default Minimum Boottime
- echo 8 ¿ /proc/sys/kernel/printk
- if the value is set to 8, all messages, including debugging ones, are displayed.

mmm... proc... what's that?!?

- [giuseppe@wopr]$ cat /proc/sys/kernel/printk
  3 4 1 7
- Current Default Minimum Boottime
- echo 8 ¿ /proc/sys/kernel/printk
- if the value is set to 8, all messages, including debugging ones, are displayed.

Other debugging techniques

- Use the /proc filesystem
- Use ioctl method
- Debugging by watching (strace, ltrace)
- Oops messages (continues...)
- gdb vmlinux /proc/kcore
- kdb kernel debugger
- kgdb kernel debugger (in vanilla)
- User mode linux (!?!)
- Linux Trace Toolkit
- Kernel probes (good good)
- ice kernel debugger (waited for 4891AD)

OOOOOOPS!


```
Unable to handle kernel NULL pointer dereference at virtual address 00000000
 printing eip:
d083a064
Oops: 0002 [#1]
SMP
CPU:    0
EIP:    0060:[<d083a064>]    Not tainted
EFLAGS: 00010246    (2.6.6)
EIP is at faulty_write+0x4/0x10 [faulty]
eax: 00000000   ebx: 00000000   ecx: 00000000 edx: 00000000
esi: cf8b2460   edi: cf8b2480   ebp: 00000005 esp: c31c5f74
ds: 007b   es: 007b   ss: 0068
Process bash (pid: 2086, threadinfo=c31c4000 task=cfa0a6c0)
Stack: c0150558 cf8b2460 080e9408 00000005 cf8b2480 00000000 cf8b2460 cf8b2460
       fffffff7 080e9408 c31c4000 c0150682 cf8b2460 080e9408 00000005 cf8b2480
       00000000 00000001 00000005 c0103f8f 00000001 080e9408 00000005 00000005
Call Trace:
 [<c0150558>] vfs_write+0xb8/0x130
 [<c0150682>] sys_write+0x42/0x70
 [<c0103f8f>] syscall_call+0x7/0xb
Code: 89 15 00 00 00 00 c3 90 8d 74 26 00 83 ec 0c b8 00 a6 83 d0
```

Semaphores and mutexes


```c
#include <asm/semaphore.h>

/* Semaphore initialisation. */
void sema_init(struct semaphore *sem, int val);
DECLARE_MUTEX(name);
DECLARE_MUTEX_LOCKED(name);
void init_MUTEX(struct semaphore *sem);
void init_MUTEX_LOCKED(struct semaphore *sem);

/* P() - Probeer te verlagen (try to reduce). */
void down(struct semaphore *sem);
int down_interruptible(struct semaphore *sem);
int down_trylock(struct semaphore *sem);

/* V() - Verhogen (increase). */
void up(struct semaphore *sem);
```

Reader/writer semaphores


```c
#include <linux/rwsem.h>

/* Semaphore initialisation.  */
void init_rwsem(struct rw_semaphore *sem);

/* P() - Probeer te verlagen (try to reduce). Only for reading */
void down_read(struct rw_semaphore *sem);
int down_read_trylock(struct rw_semaphore *sem);

/* V() - Verhogen (increase). Only for reading!!! */
void up_read(struct rw_semaphore *sem);

/* P() - Probeer te verlagen (try to reduce). I can write... */
void down_write(struct rw_semaphore *sem);
int down_write_trylock(struct rw_semaphore *sem);

/* V() - Verhogen (increase). Writing writing... */
void up_write(struct rw_semaphore *sem);
/* Change a write lock in a read lock.  */
void downgrade_write(struct rw_semaphore *sem);
```

Comple...

A common pattern in kernel programming involves initiating some activity outside of the current thread, then waiting for that activity to complete.

```
struct semaphore sem;

init_MUTEX_LOCKED(&sem);
start_external_task(&sem);
down(&sem):

void external_task(struct semaphore *sem)
{
  [...snip...];

  up(sem);
}
```

...tions


```c
#include <linux/completion.h>

/* Statically defined completion. */
DECLARE_COMPLETION(my_completion);

/* Dinamically defined completion. */
struct completion my_completion;
/* ... */
init_completion(&my_completion);

/* Wait for something to complete. */
void wait_for_completion(struct completion *c);

/* Complete! */
void complete(struct completion *c);
void complete_all(struct completion *c);
```

Spinlocks and...


```
#include <linux/spinlock.h>

/* Statically defined spin_lock. */
spinlock_t my_lock = SPIN_LOCK_UNLOCKED;

/* Dinamically defined spin_lock. */
void spin_lock_init(spinlock_t *lock);

/* Enter critical section. */
void spin_lock(spinlock_t *lock);
void spin_lock_irqsave(spinlock_t *lock, unsigned long flags);
void spin_lock_irq(spinlock_t *lock);
void spin_lock_bh(spinlock_t *lock);

/* Exit critical section. */
void spin_unlock(spinlock_t *lock);
void spin_unlock_irqrestore(spinlock_t *lock, unsigned long flags);
void spin_unlock_irq(spinlock_t *lock);
void spin_unlock_bh(spinlock_t *lock);

/* Nonblocking spinlock operations. */
int spin_trylock(spinlock_t *lock);
int spin_trylock_bh(spinlock_t *lock);
```

...Atomic Context!

In atomic context (interrupt context, softirq context)
YOU CAN'T:

- Schedule
- Sleep
- Use semaphores (they could go to sleep)

and

YOU MUST USE SPINLOCKS!

Otherwise:

BUG: scheduling while atomic
kernel panic, not syncing. Aiiie, killing interrupt handler...

Reader/Writer Spinlocks


```
#include <linux/spinlock.h>

rwlock_t my_rwlock = RW_LOCK_UNLOCKED; /* Static way */

rwlock_t my_rwlock;
rwlock_init(&my_rwlock);  /* Dynamic way */

void read_lock(rwlock_t *lock);
void read_lock_irqsave(rwlock_t *lock, unsigned long flags);
void read_lock_irq(rwlock_t *lock);
void read_lock_bh(rwlock_t *lock);

void read_unlock(rwlock_t *lock);
void read_unlock_irqrestore(rwlock_t *lock, unsigned long flags);
void read_unlock_irq(rwlock_t *lock);
void read_unlock_bh(rwlock_t *lock);

void write_lock(rwlock_t *lock);
void write_lock_irqsave(rwlock_t *lock, unsigned long flags);
void write_lock_irq(rwlock_t *lock);
void write_lock_bh(rwlock_t *lock);
int write_trylock(rwlock_t *lock);

void write_unlock(rwlock_t *lock);
void write_unlock_irqrestore(rwlock_t *lock, unsigned long flags);
void write_unlock_irq(rwlock_t *lock);
void write_unlock_bh(rwlock_t *lock);
```

Lock-Free Algorithms

- Circular buffers
- Atomic variables
- Bit operations
- seqlocks (continues...)
- Read-Copy update (continues...)

seqlocks

Read access works by obtaining an (unsigned) integer sequence value on entry into the critical section. On exit, that sequence value is compared with the current value; if there is a mismatch, the read access must be retried. As a result, reader code has a form like the following:

```
seqlock_t lock;
seqlock_init(lock);
unsigned int seq;

do {
  seq = read_seqbegin(&lock);
  /* Do what you need to do */
 } while read_seqretry(&lock, seq);
}
```

Read-Copy-Update


```
struct my_stuff *stuff;
rcu_read_lock( );
stuff = find_the_stuff(args...);
do_something_with(stuff);
rcu_read_unlock( );
```

http://www.rdrop.com/users/paulmck/rclock/intro/rclock_intro.html

The real story of kmalloc


```
/* user space. */
void *malloc(size_t size);

/* kernel space. */
#include <linux/slab.h>
void *kmalloc(size_t size, int flags);
```

Table: Flags:

Allocation priorities	Allocation flags
GFP_ATOMIC	__GFP_DMA
GFP_KERNEL	__GFP_HIGHMEM
GFP_USER	__GFP_COLD
GFP_HIGHUSER	__GFP_NOWARN
GFP_NOIO	__GFP_HIGH
GFP_NOFS	__GFP_REPEAT
	__GFP_NOFAIL
	__GFP_NORETRY

Lookaside caches

A device driver often ends up allocating many objects of the same size, over and over.

```
/* Create cache. */
mem_cache_t *kmem_cache_create(const char *name, size_t size,
                               size_t offset,
                               unsigned long flags,
                               void (*constructor)(void *, kmem_cache_t *,
                                                   unsigned long flags),
                               void (*destructor)(void *, kmem_cache_t *,
                                                  unsigned long flags));

/* Allocate object. */
void *kmem_cache_alloc(kmem_cache_t *cache, int flags);

/* Deallocate object. */
void kmem_cache_free(kmem_cache_t *cache, const void *obj);

/* Destroy cache. */
int kmem_cache_destroy(kmem_cache_t *cache);
```

Memory pools

There are places in the kernel where memory allocations cannot be allowed to fail. As a way of guaranteeing allocations in those situations, the kernel developers cre- ated an abstraction known as a memory pool (or "mempool"). A memory pool is really just a form of a lookaside cache that tries to always keep a list of free memory around for use in emergencies.

get_free_page and co


```
/* Returns a pointer to a new page and fills the page with zeros. */
get_zeroed_page(unsigned int flags);

/* Similar to get_zeroed_page, but doesn't clear the page. */
__get_free_page(unsigned int flags);

/* Allocates and returns a pointer to the first byte of a memory area that is
   potentially several (physically contiguous) pages long but doesn't zero the a
*/
__get_free_pages(unsigned int flags, unsigned int order);

/* Free Page(s). */
void free_page(unsigned long addr);
void free_pages(unsigned long addr, unsigned long order);
```

vmalloc and friends

vmalloc allocates a contiguous memory region in the virtual address space. Although the pages are not consecutive in physical memory the kernel sees them as a contiguous range of addresses.
vmalloc is described here because it is one of the fundamental Linux memory allocation mechanisms.
We should note, however, that use of vmalloc is discouraged in most situations. Memory obtained from vmalloc is slightly less efficient to work with, and, on some architectures, the amount of address space set aside for vmalloc is relatively small.
Code that uses vmalloc is likely to get a chilly reception if submitted for inclusion in the kernel. If possible, you should work directly with individual pages rather than trying to smooth things over with vmalloc.

vmalloc api


```
#include <linux/vmalloc.h>

/* Allocates memory and creates page tables. */
void *vmalloc(unsigned long size);
void vfree(void * addr);

/* Used to map device memory. */
void *ioremap(unsigned long offset, unsigned long size);
void iounmap(void * addr);
```

Communicating with hardware

```
#define ISA_BASE    0xA0000
#define ISA_MAX     0x100000  /* for general memory access */

io_base = ioremap(ISA_BASE, ISA_MAX - ISA_BASE);

/* Read...  */
unsigned int ioread8(void *addr);
unsigned int ioread16(void *addr);
unsigned int ioread32(void *addr);

void ioread8_rep(void *addr, void *buf, unsigned long count);
void ioread16_rep(void *addr, void *buf, unsigned long count);
void ioread32_rep(void *addr, void *buf, unsigned long count);

/* ...Write */
void iowrite8(u8 value, void *addr);
void iowrite16(u16 value, void *addr);
void iowrite32(u32 value, void *addr);

void iowrite8_rep(void *addr, const void *buf, unsigned long count);
void iowrite16_rep(void *addr, const void *buf, unsigned long count);
void iowrite32_rep(void *addr, const void *buf, unsigned long count);
```

Installing an interrupt handler


```c
/* Register interrupt line. */
int request_irq(unsigned int irq,
                irqreturn_t (*handler)(int, void *, struct pt_regs *),
                unsigned long flags,
                const char *dev_name,
                void *dev_id);

/* Free interrupt line. */
void free_irq(unsigned int irq, void *dev_id);
```

Table: Flags:

Interrupt flags:
SA_INTERRUPT
SA_SHIRQ
SA_SAMPLE_RANDOM

Top and bottom halves

One of the main problems with interrupt handling is how to perform lengthy tasks
within a handler.
In the typical scenario, the top half saves device data to a device-specific buffer,
schedules its bottom half, and exits: this operation is very fast.
The bottom half then performs whatever other work is required, such as awakening processes, starting up another I/O operation, and so on.
This setup permits the top half to service a new interrupt while the bottom half is still working.

Kernel data types

arch	char	short	int	long	ptr	longlong	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

Title