Linux Device Driver Basics

By

-Amit Raddi

Session-3

Kernel “Embedded C” Programming

What to Expect?

- How to do programming in "Kernel C" for
  - Achieving Concurrency
  - Keeping Time
  - Providing Delays
  - Timer Control

Kernel Module/Driver Programming Pattern

```
static int __init start_module(void)
{
    register();
    alloc();
    create();
    start();
    lock();

    return 0;
}

static void __exit end_module(void)
{
    unlock();
    stop();
    destroy(); // delete()
    dealloc(); // free();
    unregister();

    return ;
}
```

Kernel Module/Driver Programming Pattern (contd.)

```
static int __init start_module(void)
{
   ret = register_this();
   if (ret == FAILED)
      return -1;
   ret = register_that();
   if (ret == FAILED)
   {
      unregister_this();
      return -1;
   }
   return 0;
}

static void __exit end_module(void)
{
   unregister_that();
   unregister_this();

   return ;
}
```

Kernel Threads

```
#include <linux/kthread.h>
#include <linux/sched.h>

int threadfn(void *data);

struct task_struct * kthread_create (int (* threadfn(void *data), void *data, const char namefmt[], ...);

int wake_up_process (struct task_struct * p);

struct task_struct * kthread_run (int (* threadfn(void *data), void *data, const char namefmt[], ...);

int kthread_stop ( struct task_struct *k);

int kthread_should_stop (void);

void kthread_bind (struct task_struct *k, unsigned int cpu);
```

Concurrency

Concurrency with Locking

- Mutexes
  - Header: <linux/mutex.h>
  - Type: struct mutex
  - APIs
    - DEFINE_MUTEX
    - mutex_is_locked
    - mutex_lock, mutex_trylock, mutex_unlock
- Semaphores
  - Header: <linux/semaphore.h>
  - Type: struct semaphore
  - APIs
    - sema_init
    - down, down_trylock, down_interruptible, up

Concurrency with Locking

Strict semantics to be followed for mutexes :

- only one task can hold the mutex at a time.
- only the owner can unlock the mutex.
- multiple unlocks are not permitted.
- recursive locking is not permitted.
- a mutex object must be initialized via the API.
- a mutex object must not be initialized via memset or copying.
- task may not exit with mutex held.
- memory areas where held locks reside must not be freed.
- held mutexes must not be reinitialized.
- mutexes may not be used in hardware or software interrupt contexts such as tasklets and timers

Concurrency w/ Locking (cont.)

- Spin Locks
  - Header <linux/spinlock.h>
  - Type: spinlock_t
  - APIs
    - spin_lock_init
    - spin_[try]lock, spin_unlock
- Reader-Writer Locks
  - Header: <linux/spinlock.h>
  - Type: rwlock_t
  - APIs
    - read_lock, read_unlock
    - write_lock, write_unlock

Concurrency without Locking

- Atomic Variables
  - Header: <asm-generic/atomic.h>
  - Type: atomic_t
  - Macros
    - ATOMIC_INIT
    - atomic_read, atomic_set
    - atomic_add, atomic_sub, atomic_inc, atomic_dec
    - atomic_xchg

Concurrency w/o Locking (cont.)

- Atomic Bit Operations
  - Header: <linux/bitops.h>
  - APIs
    - rol8, rol16, rol32, ror8, ror16, ror32
    - find_first_bit, find_first_zero_bit
    - find_last_bit
    - find_next_bit, find_next_zero_bit
  - Header: <asm-generic/bitops.h>
  - APIs
    - set_bit, clear_bit, change_bit
    - test_and_set_bit, test_and_clear_bit, test_and_change_bit

Wait Queues

- Wait Queues
  - Header: <linux/wait.h>
  - Wait Queue Head APIs
    - DECLARE_WAIT_QUEUE_HEAD(wq);
    - wait_event_interruptible(wq, cond);
    - wait_event_interruptible_timeout(wq, cond, timeout);
    - wake_up_interruptible(&wq);
    - ... (non-interruptible set)
  - Wait Queue APIs
    - DECLARE_WAITQUEUE(w, current);
    - add_wait_queue(&wq, &w);
    - remove_wait_queue(&wq, &w);

Time Keeping

Time since Bootup

- tick – Kernel's unit of time. Also called jiffy
- HZ – ticks per second
  - Defined in Header: <linux/param.h>
  - Typically, 1000 for desktops, 100 for embedded systems
- 1 tick = 1ms (desktop), 10ms (embedded systems)
- Variables: jiffies & jiffies_64
  - Header: <linux/jiffies.h>
  - APIs
    - time_after, time_before, time_in_range, ...
    - get_jiffies_64, ...
    - msec_to_jiffies, timespec_to_jiffies, timeval_to_jiffies, ...
    - jiffies_to_msec, jiffies_to_timespec, jiffies_to_timeval, ...

Time since Bootup (cont.)

- Platform specific "Time Stamp Counter"
  - On x86
    - Header: <asm/msr.h>
    - API: rdtsc(ul low_tsc_ticks, ul high_tsc_ticks);
  - Getting it generically
    - Header: <linux/timex.h>
    - API: read_current_timer(unsigned long *timer_val);

Absolute Time

- Header: <linux/time.h>
- APIs
  - mktime(y, m, d, h, m, s) – Seconds since Epoch
  - void do_gettimeofday(struct timeval *tv);
  - struct timespec current_kernel_time(void);

Delays

Long Delays

- Busy wait: cpu_relax

```
while (time_before(jiffies, j1))
    cpu_relax();
```

- Yielding: schedule/schedule_timeout

```
while (time_before(jiffies, j1))
    schedule();
```

Short Delays but Busy Waiting

- Header: <linux/delay.h>
- Arch. specific Header: <asm/delay.h>
- APIs
  - void ndelay(unsigned long ndelays);
  - void udelay(unsigned long udelays);
  - void mdelay(unsigned long mdelays);

Long Delays: Back to Yielding

- Header: <linux/delay.h>
- APIs
  - void msleep(unsigned int millisecs);
  - unsigned long msleep_interruptible(unsigned int millisecs);
  - void ssleep(unsigned int secs);

Timers

Kernel Timers

- Back end of the various delays
- Header: <linux/timer.h>
- Type: struct timer_list
- APIs
  - void init_timer(struct timer_list *); /* Nullifies */
  - struct timer_list TIMER_INITIALIZER(f, t, p);
  - void add_timer(struct timer_list *);
  - void del_timer(struct timer_list *);
  - int mod_timer(struct timer_list *, unsigned long);
  - int del_timer_sync(struct timer_list *);

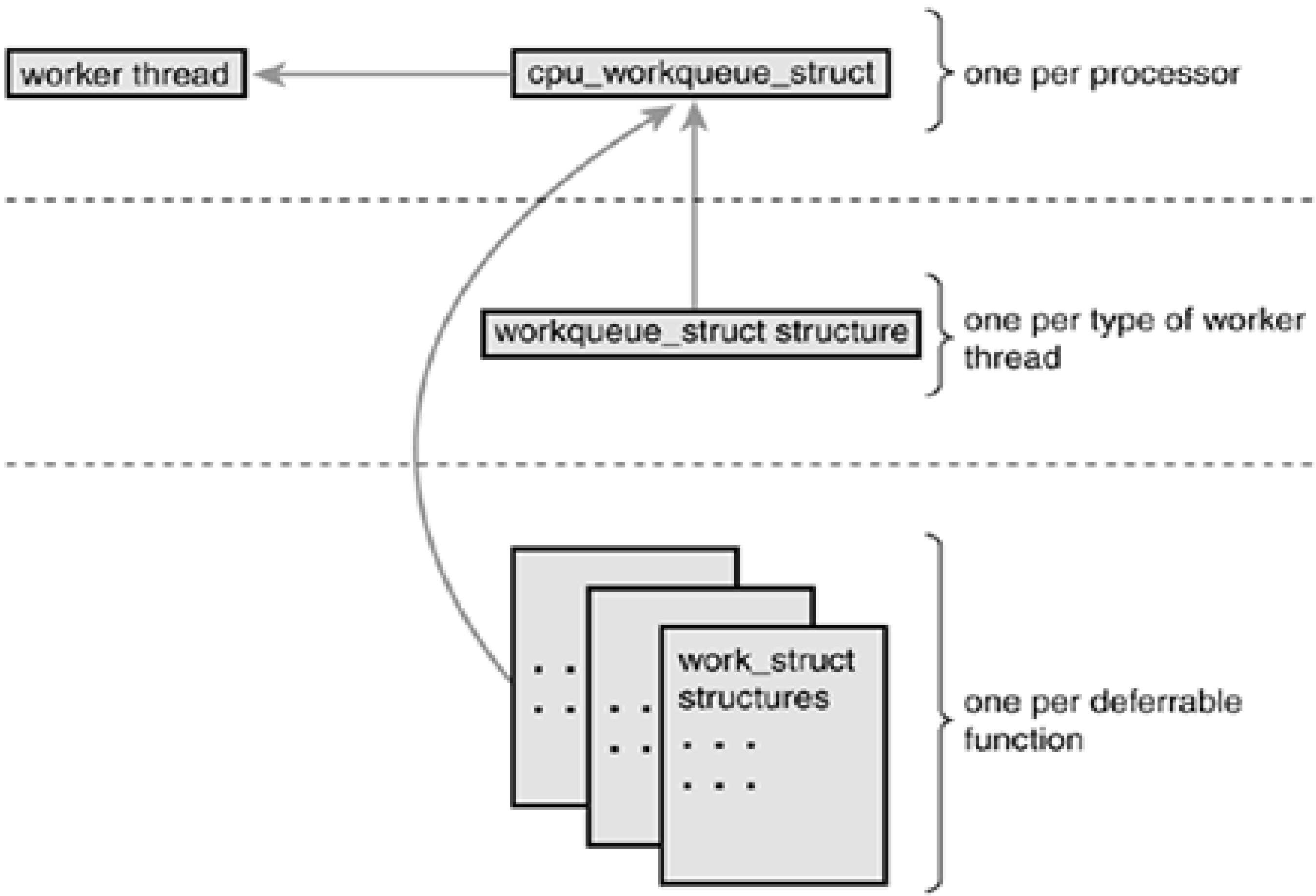
Tasklets

- Timers without specific Timing
- Header: <linux/interrupt.h>
- Type: struct tasklet_struct
- APIs
  - void tasklet_init(struct tasklet_struct *t, void (*func)(unsigned long), unsigned long data);
  - void tasklet_kill(struct tasklet_struct *t);
  - DECLARE_TASKLET(name, func, data);
  - tasklet_enable(t), tasklet_disable(t)
  - tasklet_[hi_]schedule(t);

Work Queues

- In context of "Special Kernel Thread"
- Header: <linux/workqueue.h>
- Types: struct workqueue_struct, struct work_struct
- Work Queue APIs
  - q = create_workqueue(name);
  - q = create_singlethread_workqueue(name);
  - flush_workqueue(q);
  - destroy_workqueue(q);
- Work APIs
  - DECLARE_WORK(w, void (*function)(void *), void *data);
  - INIT_WORK(w, void (*function)(void *), void *data);
- Combined APIs
  - int queue_work(q, &w);
  - int queue_delayed_work(q, &w, d);
  - int cancel_delayed_work(&w);
- Global Shared Work Queue API
  - schedule_work(&w);

Work Queues

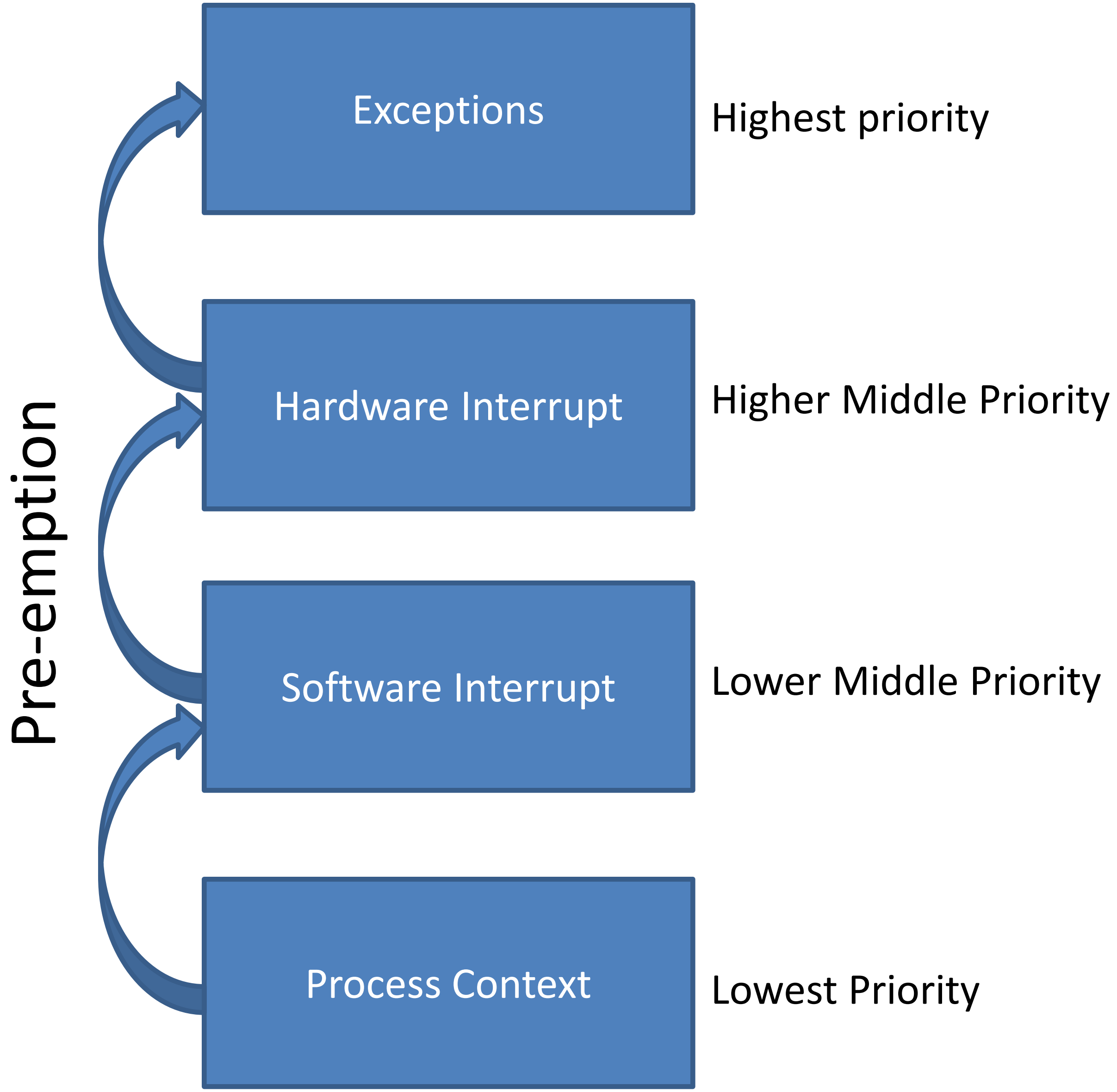


The relationship between work, work queues, and the worker threads

Tasklet Vs. Work-Queue

	Tasklet	Work-Queue
Context	Runs in software interrupt context as it is built using SoftIRQ.	Runs in kernel-space process context.
Sleep	Can not sleep as it runs in interrupt context and it can not be rescheduled.	Can sleep as it runs in process context and it can be rescheduled.
Execution priority	Executes faster. Runs at higher priority than Work-Queue as it runs at software interrupt priority level.	Executes slower. Runs at lower priority than Tasklet as it runs at process priority level.
Function atomicity	Tasklet function must be atomic. It must run in one go.	Work-Queue function need not be atomic. It need not run in one go. It may sleep or block at some function call or lock.
Lock usage	Must use Spinlock as it can not be blocked.	May use Spinlock or Mutex lock as it can be blocked.
Execution	More than one tasklet of same type can not run simultaneously. Runs serially one after another on a multi-processor/ multi-core system.	More than one Work-Queue of same type can run simultaneously. May run parallelly on different processors/cores on a multi-processor/ multi-core system.
Usage	Suitable for high-speed device drivers (USB, PCIe, Ethernet etc)	Suitable for low-speed device drivers (UART, I2C, SPI etc)

Execution Priority



Helper Interfaces

Other Helper Interfaces in Latest Kernels

- User Mode Helper
- Linked Lists
- Hash Lists
- Notifier Chains
- Completion Interface
- Kthread Helpers

What to Expect?

- How to do programming in "Kernel C" for
  - Achieving Concurrency
    - With & without Locking
    - Wait Queues
  - Keeping Time
    - Relative & Absolute
  - Providing Delays
    - Long and Short
    - Busy Wait and Yielding
  - Timer Control
    - Kernel Timers
    - Tasklets
    - Work Queues

Any Queries?

Linux Kernel logging

- Kernel provides central logging facility.
- Klogd: daemon for collecting the kernel logs.
- Generally all the printk() logs will be stored in /proc/kmsg file(kernel buffer).
- Can associate priorities to printk().
- Klogd will collect the logs from that buffer and redirects based on the priorities.
- High priority logs goes to console, and rest of the kernel logs goes to kernel bufer.
- Dmesg also collects the logs from kernel buffer(/proc/kmsg) and dumps on to console.

Aricent

Linux Kernel logging..

- Available priorities (include/linux/kernel.h):

```
#define KERN_EMERG    "<0>"  /* system is unusable*/
#define KERN_ALERT    "<1>"  /* action must be taken immediately */

#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 condition */
#define KERN_INFO     "<6>"  /* informational*/
#define KERN_DEBUG    "<7>"  /* debug-level messages */
```

- Default priority is KERN_DEBUG (7).
- Example:

```
void func(void)
{
    printk("<4> func invoked\n");
}
```

Communication to/from Kernel

- Communication bet'n U-space to K-space:
  1. Using file operations (use write() call or ioctl() from applications) which inturn use system calls
  2. Sysfs interace.
- Communication bet'n Kernel to user space apps:
  1. copy_to_user() & Copy_from_user() routines
  2. Signals. – asynchronous communication from kernel.

Communication to/from Kernel..

Signals introduction:

- Asynchronous messages delivered to a process by the signaling subsystem of kernel, when some even occurs.
- Each signal identified by a number, from 1 to 31.(Linux is having 32 signals)
- Signals that report exceptions:

  Ex: SIGILL -- Execution of Illegal Instruction.

  SIGSEGV– occurs when program tries to read/write unauthorized memory.
- Termination signals:

  Ex: SIGKILL-- Immediate program termination.

  SIGINT -- control+c (to terminate running process).

Communication to/from Kernel…

Applications can register a custom signal handler using **signal()** routine.

Here is a short code snippet demonstrating how to use it.

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>

void sig_handler(int signum)
{
    printf("Received signal %d\n", signum);
}

int main()
{
    signal(SIGINT, sig_handler);
    sleep(10); // This is your chance to press CTRL-C
    return 0;
}
```

This nice and small application registers its own SIGINT signal. Try compiling this small program. See what is happening when you run it and press CTRL-C.

Books for Ref..

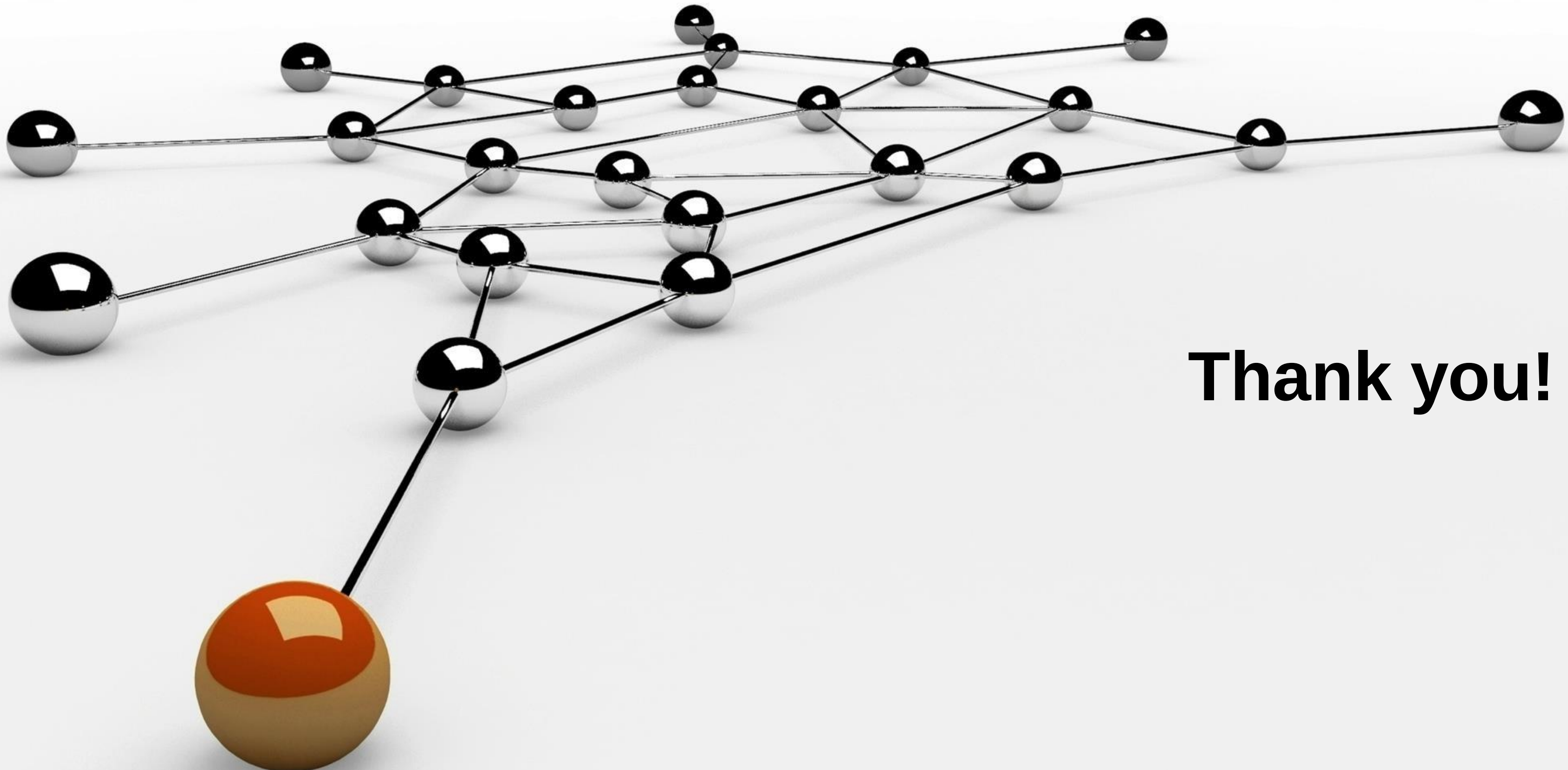
Books:

- Understanding the Linux Kernel, D. P. Bovet and M. Cesati, O'Reilly & Associates, 2000.
- Linux Core Kernel – Commentary, In-Depth Code Annotation, S. Maxwell, Coriolis Open Press, 1999.
- The Linux Kernel, Version 0.8-3, D. A Rusling, 1998.
- Linux Kernel Internals, 2nd edition, M. Beck et al., Addison-Wesley, 1998.
- Linux Kernel, R. Card et al., John Wiley & Sons, 1998.
- Linux Device Drivers, 3rd Edition, Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman Published by O'Reilly Media, Inc., 1005

Questions ??

Aricent®

Engineering excellence. Sourced.



Thank you!

www.aricent.com