Linux kernel programming

(& debugging)

Ottawa Linux Symposium
07/23/99

Jes Sorensen
European Laboratory
for Particle Physics - CERN
http://www.cern.ch/
Jes.Sorensen@cern.ch

Agenda

- Kernel intro + versions
- Basic kernel programming rules
- Source layout
- Kernel configuration & compile, modules
- Kernel bootstrap & init
- printk()
- Memory allocation
- Address spaces & correct access
- Device probing & resource allocation
- Interrupts, bottom half's, timers
- User space access
- Waiting inside the kernel (delays)

Agenda (continued)

- Locking mechanisms
- System calls & privileges
- Kernel threads
- ioctls, sysctl & /proc
- Portability across architectures
- Multi kernel version supporting code
- SMP, atomic access, spin locks
- Data alignment & performance
- initdata
- OOPS (and other aspects of debugging)
- Changes in 2.3.x
- Other information sources

Kernel intro & versions

- Stable (even) and developer (uneven):
  - 2.2.x current stable release (previous 2.0.x)
  - 2.3.x current developer release (expect radical changes every now and then)
  - This talk will mainly focus on 2.2.x
- Kernel maintained by Linus Torvalds, some modules available as separate patches from third parties (eg. PCMCIA, some non Intel architectures, etc.)
- Object Oriented C and assembly
- All code is GPL (Gnu general Public License) or GPL compatible. The normal BSD license is not GPL compatible! - Dual licensing is possible!

Basic rules

- No libc in the kernel, only few basic functions
- Use C (no C++ !!) whenever possible, only inline assembly for architecture specific
- Avoid unncessary #ifdef's (#ifdef <arch> !!)
- No floating point usage
- Only put things in the kernel that really need to go into the kernel (think twice!)
- Runtime options preferred to compile time options (flexibility)
- Use private header files for things that are not supposed to be public (ie. not in include/)
- Write in English (code and comments)

Kernel programming tools

- gcc + binutils (as, ld etc)
- gdb
- your favorite editor (emacs, vi etc.)
- grep
- find
- coffee
- diff
- patch

- Nothing special - really!

Kernel source layout

- **Documentation/**
- **Makefile**
- **arch/{alpha,i386,sparc,...}**
- **drivers/{block,char,net,scsi,...}**
- **fs/{ext2,fat,nfs,ntfs,ufs,...}**
- **include/{asm-<arch>,net,linux}**
- **init/**
- **ipc/**
- **kernel/**
- **lib/**
- **mm/**
- **modules/**
- **net/**
- **scripts/**

How to build a kernel

- Fresh build:
  - cd <source-dir>
  - make config (menuconfig/xconfig)
  - make dep
  - make install
  - make modules
  - make modules_install

- Applying patches:
  - cd <source-dir>
  - patch -p1 -d. < patch-file
  - make oldconfig
  - make dep
  - ....

Kernel bootstrap (1)

- arch/<arch>/kernel/head.S (assembly):
  - Initial minimum setup:
    - Shutdown remains of previous operating system
    - Initial mmu setup
    - Copy kernel to final location
    - Jump to start_kernel()

Kernel Bootstrap (2)

- init/main.c:
  - Initialize basic kernel modules:
    - architecture specific
    - memory management
    - interrupts/traps
    - scheduler
    - console
    - various caches (filesystem, directory etc)
    - virtual memory subsystem
    - SMP
    - busses (PCI, MCA, SBUS etc.)
    - devices
    - filesystems
  - mount root
  - exec init process

Kernel Bootstrap (3)

- Early initialization of memory subsystem, interrupts etc.
- Drivers, filesystems (what's the difference?) etc. will have a fully functional system - no need for special treatment for module vs statically compiled drivers

Device drivers and the VFS (1)

- Device drivers, file systems etc. (modules) are registered in the VFS (network devices are not directly in the VFS, but similar)
- Module can provide just a subset of the possible functions (no sense to mmap net-device)
- Most common functions & behavior:
  - **probe**/**init**: detect presence of hardware, allocate control structures, register driver
  - **open**: activate hardware/filesystem, allocate + init run time resources etc.
  - **close**: disable hardware/filesystem, release run time resources etc.

Device drivers and the VFS (2)

- **read**: prepare & provide data to caller, read from device etc.
- **write**: store/proces data provided by caller, write to device (net-devices have start_xmit)
- **mmap**: memory map raw device registers, control structures or actual device data
- **ioctl**: module specific io control; set buffer size, set volume, read device geometry etc. etc.
- **interrupt handler**: process hardware interrupts (modules supporting real hardware devices)
- **init_module**: call probe and other module init (loadable module specific)
- **cleanup_module**: release resources allocated by init_module and probe/init (loadable module specific)

printk() - can't live without it

- **printk()** - works more or less like printf() in user space. syslog priorities can be put in front of the content to be printed (recommended):

```
printk(KERN_INFO "%s: Firmware running\n",
    dev->name);
```

- printk() priorities:
  - **KERN_EMERG**
  - **KERN_ALERT**
  - **KERN_CRIT**
  - **KERN_ERR**
  - **KERN_WARNING**
  - **KERN_NOTICE**
  - **KERN_INFO**
  - **KERN_DEBUG**

Memory allocation

- kmalloc()/kfree():
  - Similar to malloc()/free(), however limited to a maximum request of 128KB
  - Guarantee contiguous block of physical memory
- get_free_pages(gfp, order) / free_pages(order):
  - Allocates power of 2 blocks of pages
  - **get_free_page(gfp)** == **get_free_pages(gfp, 0)**
  - Commonly used gfp masks:
    - **GFP_DMA** (to be accessibly by bus-master device)
    - **GFP_KERNEL**
- vmalloc()/vfree():
  - Allocates block of virtual memory which is not guaranteed to be contiguous physical memory

Kernel Address Spaces

- Physical memory
  - 'Real' host memory
- Bus shared memory (eg. PCI shared mem)
  - Use **ioremap()** to make accessible to kernel. Access with **read[bwl]()**/**write[bwl]()** as it may be presented in strange ways (eg. Alpha sparse memory)
- Virtual kernel memory
  - Memory as seen by the running kernel and in which the kernel is running
  - Virtual kernel address != physical address!!
- ISA/PCI style I/O address space
  - Only accessible via **in[bwl]()**/**out[bwl]()**

Address conversion

- Physical addresses - physical addresses are only of interest to the mm subsystem!!
  - **virt_to_phys()**: Convert a virtual kernel space address to a physical address
  - **phys_to_virt()**: Convert a physical address to a virtual kernel space address
- Bus addresses are physical addresses as seen by devices at the other end of a bus (PCI bus-master devices).
  - **virt_to_bus()**: Convert a kernel virtual address to a bus address
  - **bus_to_virt()**: Convert a bus address to a kernel virtual address
  - Bus addresses are the ones you will normally use in device drivers!
  - Bus address != physical address !!

Device Probing (ISA)

- Resource information for a device driver:
  - Interrupts
  - Device Shared Memory
  - I/O ports
- Probing for ISA devices:
  - Register layout for your device is known. Write to the assumed registers and read back and check if the device reacts as expected - if not try next I/O port
  - Interrupt probing: Enable one interrupt at a time, make the device generate an interrupt
  - Alternatively let the user specify the information
  - ISA device probing is dangerous!

PCI device probing

Manufacturer and product codes known for device:

```
struct pci_dev *pdev = NULL;
struct my_private *priv = kmalloc(sizeof(struct my private),
                                  GFP_KERNEL);

if (!pci_present())
    return -ENODEV;

while ((pdev = pci_find_dev(PCI_VENDOR_ID_ESSENTIAL,
                            PCI_DEVICE_ID_ESSENTIAL_ROADRUNNER,
                            pdev))) {

    priv->irq = pdev->irq;
    priv->io = pdev->base_address[0] & PCI_BASE_ADDRESS_IO_MASK;
    priv->shared1 = pdev->base_address[1];
    priv->shared2 = pdev->base_address[2];

    ....

    my_device_init();

    ....
}
```


Interrupt Allocation

- Interrupts (irqs) are reserved with **request_irq()**. Sharing irqs is recommended for sane devices (PCI etc).
- ISA does not support shared irqs!
- Linux 2.0.x does not support shared irqs!

```
static void my_dev_init(struct device *dev, struct my_priv *priv)
{
    if (request_irq(dev->irq, my_int, SA_SHIRQ, priv->name, dev)) {
        printk(KERN_WARNING "%s: Requested IRQ %d is busy\n",
               dev->name, dev->irq);
        return -EAGAIN;
    }
 ...

static void my_int(int irq, void *dev_id, struct pt_regs *ptregs)
{
    struct device *dev = (struct device *)dev_id;
    priv = (struct my_priv *)dev->priv;

    if (!(inb(priv->io + MY_IRQ_REG) & MY_INT_PENDING))
        return;
```

Resource Allocation (2)

- There is no method for reserving device shared memory! Multiple drivers can map the same shared memory region and access it simultanously -- implement your own locking if necessary!
- I/O ports are allocated with **request_region()**:

```
request_region(my_ioaddr, MY_REGION_SIZE, dev->name);
```


Interrupt handlers

- Interrupts 'block' other parts of the kernel while running (and other interrupts) - problem for timing critical things (serial etc.)

- Reduce the work in the interrupt handlers to a bare minimum:

    1) acknowledge the interrupt
    2) do your work
    3) get out of the interrupt handler

bottom half handlers

- Low priority non-timing-critical work that can run with other interrupts enabled should be put in a 'bottom half' handler
- Use a generic bottom half handler if possible or create one for a specific purpose if necessary (number of bh's is limited)
- bh's are run last by the interrupt handler with all interrupts re-enabled
- See include/linux/interrupt.h for bh's currently defined

Using the generic bh handler

```
struct tq_struct immediate;

static void my_bh_handler(struct my_data *data) {
    ... do something ...
}

void my_queue_bh(struct my_data *data) {

    /* Initialize task queue structures */
    immediate.next = NULL;
    immediate.sync = 0;
    immediate.routine = (void *)(void *)my_bh_handler;
    immediate.data = data;

    queue_task(&immediate, &tq_immediate);
    mark_bh(IMMEDIATE_BH);
```

Timers

- The kernel has a global timer known as 'jiffies' ticking at **HZ** ticks per second (HZ = 100 on most architectures)
- To avoid busy waiting use the timer software interrupt (example while waiting for a disk drive motor to spin up)
- Once a timer has expired it will not run again
- Do not add the same timer again before it has expired!

```
struct timer_list mytimer;

static void my_timer_func(unsigned long data) {
    printk(KERN_WARNING "My timer expired!\n");
}

void my_set_timer(int delay, struct my_data *data) {
    init_timer(&mytimer);
    mytimer.data = (unsigned long)data;
    mytimer.function = my_timer_func;
    mytimer.expires = jiffies + (delay * HZ);
    add_timer(&mytimer);

!! Alternative for modifying a running timer: !!
    mod_timer(&mytimer, delay * HZ);
```


User space access

- Range copy similar to BSD copyin/copyout:
  - **copy_from_user(*to, *from, size)**
  - **copy_to_user(*to, *from, size)**
- Single word (char/short/int(/long)):
  - **get_user(dest, *useraddr)**
  - **put_user(source, *useraddr)**
- Single word (non address checking)
  - **__get_user(dest, *useraddr)**
  - **__put_user(source, *useraddr)**

- Checking versions all return zero for succesful access, non-zero for an error (**copy_{to,from}_user()** returns the amount of data not copied on error)

Waiting inside the kernel

- Long delay loops inside the kernel are a bad solution! - To be avoided whenever possible!
- Use a wait queue to wait for events, like device ready etc. (described later)
- Use **udelay()** for short delays (usecs)
- If absolutely necessary (for jiffies waiting):
  - **time_before()**, **time_before_eq()**
  - **time_after()**, t**ime_after_eq()**

```
void some_secs_delay(unsigned long delay)
{
    unsigned long myjif;

    myjif = jiffies + delay * HZ;
    while (time_before(jiffies, myjif));
}
```


Locking mechanisms

- Linux provides various locking mechanisms used to protect data structures etc:
  - semaphore
  - wait queue
  - spin lock + read/write lock (SMP special - covered later)
- All mechanisms are thread + SMP safe
- Classic 'disable all interrupts' (sledgehammer approach) is to be avoided whenever possible - never in hot path code

Semaphores

- The semaphore is a traditional mutex
- If a semaphore is 'taken' the task will sleep until it becomes available (unlike spin locks)
- Use semaphores in 'task-related' code such as system calls etc. Rarely used in device drivers and never in interrupt handlers!
- Take semaphore with **down()** or **down_interruptible()** - for killable tasks use **down_interruptible()** (ie. in syscalls)
- release a semaphore with **up()**
- Init semaphore with **MUTEX** or **MUTEX_LOCKED**

Semaphore example

```
struct semaphore my_sem = MUTEX;

asmlinkage int sys_my_syscall(struct my_struct *userdata)
{
    int errno = 0;
    struct my_struct mydata;

    if (down_interruptible(&my_sem))
        return -ERESTARTSYS;

    if (copy_from_user(userdata, &mydata,
                       sizeof (struct my_struct)))
        errno = -EFAULT;

    ... do something with data ...

    up(&my_sem);
    return errno;
}
```

Wait queues

- Wait queues are used to make a task sleep until a certain event happens, eg. device receives data and an interrupt is generated
- Exist as both interruptible and non-interruptible
- Used with poll() VFS function call

```
struct wait_queue *myqueue = NULL;

void wait_on_my_queue_inside_kernel(){
    interruptible_sleep_on(&myqueue);
}

void my_event_notify(){
    wake_up_interruptible(&myqueue);
}
    /* poll() example */
unsigned int my_poll(struct my_device *dev,
                     struct file *file, poll_table *wait){
    int mask = 0;
    poll_wait(file, &myqueue, wait);
    if (some_condition)
        mask |= (POLLIN | POLLRDNORM);
    return mask;
}
```


System calls

- System calls are defined by a unique syscall number in include/asm/unistd.h
- A pointer to the system call function is put at the corresponding position in sys_call_table normally found in arch/<arch>/kernel/entry.S
- API defined between user and kernel space

```
arch/i386/kernel/entry.S:
ENTRY(sys_call_table)
 .long SYMBOL_NAME(sys_ni_syscall)
 .long SYMBOL_NAME(sys_exit)
 .long SYMBOL_NAME(sys_fork)

include/asm-i386/unistd.h:
#define __NR_exit    1
#define __NR_fork    2

arch/i386/kernel/process.c:
asmlinkage int sys_fork(struct pt_regs regs)
{
    return do_fork(SIGCHLD, regs.esp, &regs);
}
```

Privileges

- Linux provides POSIX (draft) defined capabilities support
- Used to determine whether a user proces is authorized to perform a task in the kernel, (eg. reboot or set the address on a network interface)
- See include/linux/capability.h for capabilities

```
asmlinkage int sys_reboot(int magic1, int magic2,
                          int cmd, void * arg)
{
    char buffer[256];

    /* We only trust the superuser with rebooting the system. */
    if (!capable(CAP_SYS_BOOT))
        return -EPERM;
```


Kernel threads

- Kernel threads are 'ordinary' processes running in kernel space
- Scheduled as normal processes
- Use for things that must be run regularly (daemon type tasks)
- Can run system calls as normal user space processes
- Can run executables from disk with **execve()** and the thread then becomes a normal user process

ioctl's

- ioctl's are generally used to get/set configuration and get info about general system settings
- slow, not for performance critical code
- registered by a device driver, filesystem etc.
- Uses unique identifiers (ex. SIOSIFFLAGS)

```
int my_ioctl(struct device *dev, struct ifreq *rq, int cmd){
    int error = -EOPNOTSUPP;

    switch(cmd){
    case SIOCDST:
        ... do something ...
        if (ok)
            error = 0;
        break;
    case SIOCDSTE:
        ... do something else ....
        if (ok)
            error = 0;
        break;
    }
    return error;
}
```


sysctl

- sysctl is an interface to make tuning parameters run time configurable
- Make parameters sysctl configurable if possible - compile time options are evil!
- Examples of sysctl parameters:
  - Socket buffer sizes (default+max)
  - Max shared mem
  - VM settings (page cache tuning etc).
- Entries registered in **kernel/sysctl.c, include/linux/sysctl.h** for defines
- Default handlers for strings and integer lists (read-only and read-write)
- Special cases covered with own handlers

sysctl example

```
int bdflush_min[N_PARAM] = {0, 10, 5, 25, 0, 1*HZ, 1*HZ, 1, 1};
int bdflush_max[N_PARAM] = {100,5000,2000,2000,100,600*HZ,
                            600*HZ,2047, 5};
int bdf_prm = {40, 500, 64, 256, 15, 30*HZ, 5*HZ, 1884, 2};

static ctl_table vm_table[] = {
    {VM_FREEPG, "freepages",
    &freepages, sizeof(freepages_t),
    0644, NULL, &proc_dointvec},

    {VM_BDFLUSH, "bdflush",
    &bdf_prm, 9*sizeof(int),
    0600, NULL, &proc_dointvec_minmax,
    &sysctl_intvec, NULL,
    &bdflush_min, &bdflush_max},

    {0}
};
```


/proc

- /proc is a virtual filesystem
  - provides kernel information to user space
  - good alternative to ioctl's for getting general system info
  - can be used to set certain system settings
  - /proc/sys interface to sysctl (also accessible via sysctl binary)
- /proc entries are registered dynamically by kernel modules - see **fs/proc/root.c**, **fs/proc/array.c** and **include/linux/proc_fs.h**

```
static struct proc_dir_entry proc_root_partitions = {
    PROC_PARTITIONS, 10, "partitions",
    S_IFREG | S_IRUGO, 1, 0, 0,
    0, &proc_array_inode_operations
};

void proc_root_init(){
    proc_base_init();
    proc_register(&proc_root, &proc_root_loadavg);
    proc_register(&proc_root, &proc_root_partitions);
    ....
```

/proc output function

- output function called from fs/proc/array.c

```
static long get_root_array(char * page, int type, char **start,
                           off_t offset, unsigned long length){
    switch (type) {
    case PROC_PARTITIONS:
        return get_partition_list(page);
    ....

int get_partition_list(char * page){
    struct gendisk *p;
    char buf[32];
    int n, len;

    len = sprintf(page, "major minor  #blocks  name\n\n");
    for (p = gendisk_head; p; p = p->next) {
        for (n=0; n < (p->nr_real << p->minor_shift); n++) {
            if (p->part[n].nr_sects && len < PAGE_SIZE - 80) {
                len += sprintf(page+len, "%4d  %4d %10d %s\n",
                p->major, n, p->sizes[n], disk_name(p, n, buf));
            }
        }
    }
    return len;
}
```


Multi architecture & portability

- Linux is a multi architecture OS
- Writing the code to be cross-architecture is very important
- Do not expect 'normal' data types to have a fixed size or byte order
- If data types need to match hardware expectations use fixed sized types
- Byte order conversion macros are available
- Not all archs guarantee cache coherency
- Not all archs guarantee write ordering
- Not all archs have their devices attached to the I/O ports of an ISA/PCI bus

Linux data types

bits/type	32 bit arch	64 bit arch
char	8	8
short	16	16
int	32	32
long	32	64
long long*	64	64

(* depreciated inside the kernel)

Name	Exported *	Size(bits)
s8 /u8	__s8 /__u8	8
s16/u16	__s16/__u16	16
s32/u32	__s32/__u32	32
s64/u64**	__s64/__u64	64

(* for export in header files
- no POSIX type conflict)
(** 64 bit architectures only)

Pointers and bit operations

- Cast/store pointers to/in unsigned long only!

- Bit operations always operate on longest native entity: unsigned long (ie. 32 or 64 bit)!
- Never use bit operations on char/short! (bitops on char * is a classic mistake)
- Atomic and SMP safe
  - **set_bit(int bitnr, void * addr)**
  - **clear_bit(int bitnr, void *addr)**
  - **test_bit(int bitnr, void *addr)**
  - **test_and_set_bit(int bitnr, void *addr)**

Kernel versions

- Some features are not available in all kernel versions, data structures may get new elements etc.
- To make a driver multi-version portable one can check the kernel version
- The kernel defines **LINUX_VERSION_CODE**, (major << 16) + (minor << 8) + release

```
#if (LINUX_VERSION_CODE > 0x20155)

#include <linux/init.h>

#else

#define __init
#define __initdata

#endif
```


Byte order handling

- To check for byte order:
  - **#ifdef __BIG_ENDIAN**
  - **#ifdef __LITTLE_ENDIAN**
- Byte order conversion:
  - **cpu_to_le{32,16}(), le{32,16}_to_cpu()**
  - **cpu_to_be{32,16}(), be{32,16}_to_cpu()**
- 64 bit versions available on 64 bit architectures only
- Classic network macros available as well **ntohl()**, **htonl()**, **ntohs()**, **htons()** - use these for networking only!

Cache coherency

- Some architectures are not cache coherent, and/or do not have data+instruction cache inter connected
- Explicit cache handling in software required:
  - Data segment to be accessed via DMA
- A code segment previously accessed as data by a CPU (copied with memcpy() etc)
- Cache handling functions (virtual addresses):
  - **flush_page_to_ram(u_long pageaddr)**
  - **flush_icache_range(u_long start, u_long end)**

Write ordering

- Some archs do not guarantee write ordering (Alpha + PPC) and gcc may cache data in registers (even x86!)
- **mb()** explicitly ensures write ordering
- **volatile** != write ordering
- Wrong vs correct example:

```
static inline void write_wd33c93_cmd(wd33c93_regs *regp,
                                     unsigned char cmd)
{
    writeb(WD_COMMAND, &regp->SASR);
    writeb(cmd, &regp->SCMD);
}

static inline void write_wd33c93_cmd(wd33c93_regs *regp,
                                     unsigned char cmd)
{
    writeb(WD_COMMAND, &regp->SASR);
    mb();
    writeb(cmd, &regp->SCMD);
    mb();
}
```

SMP

- Two or more processors may try to access a data structure at the same time
- Control concurrent access to avoid race conditions and their dangerous results
- Avoid 'just' disabling all interrupts, it is slow as it requires synchronization across all processors!
- Alternative solutions:
  - atomic data types (**atomit_t** with atomic operations, use for counters etc.)
  - **xchg()** atomic swap of data at a given address with a new value - works for 8, 16, 32 (& 64 on 64 bit archs) data types
  - spin locks

spin locks

- Linux provides two versions of spin locks, a normal mutex lock and a read/write lock
- spin locks are fast but do busy wait!
- Use to restrict access to critical data structures, device registers etc.
- Usable in interrupt handlers
- Never call anything that may sleep or take a page fault while holding a spin lock
- Available with or without disabling interrupts on local processor (inside irq handler, the irq is not reenabled - use non-disable version):
  - **spin_lock()/spin_unlock()**
  - **spin_lock_irqsave()/spin_unlock_irqrestore()**

spin lock example

```
void ace_spin_init(struct ace_private *ap){
#ifdef __SMP__
    spin_lock_init(&ap->lock);
#endif
}

static void ace_interrupt(int irq, void *dev_id,
                          struct pt_regs *ptregs){
    struct ace_private *ap;
    struct ace_regs *regs;
    struct device *dev = (struct device *)dev_id;

    ap = (struct ace_private *)dev->priv;
    regs = ap->regs;

    spin_lock(&ap->lock);
    /*
     * In case of PCI shared interrupts or spurious interrupts,
     * we want to make sure it is actually our interrupt before
     * spending any time in here.
     */
    if (!(readl(&regs->HostCtrl) & IN_INT)){
        spin_unlock(&ap->lock);
        return;
    }
    ....
```


Data alignment

- Data types are not the same size on all archs
- Compiler does not perform the same level of padding in data structures on all archs
- Do not expect the compiler to align entries on four byte boundaries!
- Explicitly pad chars/shorts if it is required that the following entry is on a certain boundary
- Pack data in a sensible matter in data structures, ie. longs together with longs, shorts with shorts whenever possible

Data alignment (2)

- Data structures with elements accessed concurrently by two processors or CPU + device may cause cache line stalls
- If elements are independent (eg. indices for a TX and a RX ring for a network device) perform explicit cache line alignment

```
struct something {
    volatile u32 evt_prd
                 __attribute__ ((aligned (L1_CACHE_BYTES)));
    volatile u32 rx_ret_prd
                 __attribute__ ((aligned (L1_CACHE_BYTES)));
    volatile u32 tx_csm
                 __attribute__ ((aligned (L1_CACHE_BYTES)));
}
```


__initdata

- Data structures and functions that are only used at boot time can be marked as **__initdata** and **__init** functions.
- Will be put in a special ELF section and added to the pool of free memory after the kernel has booted
- Good for device driver probe functions and firmware data

```
static int probed __initdata = 0;

int __init some_device_probe (struct device *dev))
{
    if (probed)
        return -ENODEV;
    probed++;

    ....
}
```

OOPS

- The kernel produces a register + stack dump in case of a detected crash inside the kernel (NULL pointer access etc)
- When possible dumps will appear in syslog, otherwise only on the console
- Stack trace addresses can be converted to symbolic addresses using **ksymoops** and the symbol table generated by the kernel compile (**System.map**)
- Symbol tables for loadable modules can be obtained when the modules are loaded:
  - **insmod -m <module-name> > symfile**
- Keep your System.map!

OOPS example (1)

```
Unable to handle kernel NULL pointer dereference at virtual address 00000000
current->tss.cr3 = 073d1000, %cr3 = 073d1000
*pde = 00000000
Oops: 0002
CPU:    1
EIP:    0010:[<c01b2ff8>]
EFLAGS: 00010286
eax: c01b2fe8   ebx: c7f80000   ecx: c733df48   edx: c020c2cc
esi: c020c2cc   edi: c020c2cc   ebp: c8000000   esp: c733dedc
ds: 0018   es: 0018   ss: 0018
Process ifconfig (pid: 397, process nr: 29, stackpage=c733d000)
Stack: 00000000 00001043 ffffffff c015e83d c020c2cc 00001002 c020c2cc 00000000
       c015f386 c020c2cc c7ad7ae0 00001043 c7ad7b04 c733df58 c017a950 c020c2cc
       00001043 00008914 00008914 bffffb34 c7570b80 00001043 00000000 00000001
Call Trace: [<c015e83d>] [<c015f386>] [<c017a950>] [<c017c7cb>] [<c015b121>]
Code: c7 05 00 00 00 00 42 00 00 00 83 bb c4 76 01 00 00 75 1d 8b
```

OOPS example (2)

```
[jes@hippie linux]$ ksymoops -m /boot/Symbol.map-2.2.3 < oops-file

Unable to handle kernel NULL pointer dereference at virtual address 00000000
current->tss.cr3 = 073d1000, %cr3 = 073d1000
*pde = 00000000
Oops: 0002
CPU:    1
EIP:    0010:[<c01b2ff8>]
EFLAGS: 00010286
eax: c01b2fe8   ebx: c7f80000   ecx: c733df48   edx: c020c2cc
esi: c020c2cc   edi: c020c2cc   ebp: c8000000   esp: c733dedc
ds: 0018   es: 0018   ss: 0018
Process ifconfig (pid: 397, process nr: 29, stackpage=c733d000)
Stack: 00000000 00001043 ffffffff c015e83d c020c2cc 00001002 c020c2cc 00000000
       c015f386 c020c2cc c7ad7ae0 00001043 c7ad7b04 c733df58 c017a950 c020c2cc
       00001043 00008914 00008914 bffffb34 c7570b80 00001043 00000000 00000001
Call Trace: [<c015e83d>] [<c015f386>] [<c017a950>] [<c017c7cb>] [<c015b121>] [<c012f910>]
Code: c7 05 00 00 00 00 42 00 00 00 83 bb c4 76 01 00 00 75 1d 8b

>>EIP: c01b2ff8 <ace_open+10/130>
Trace: c015e83d <dev_open+25/80>
Trace: c015f386 <dev_change_flags+4a/f8>
Trace: c017a950 <devinet_ioctl+244/544>
Trace: c017c7cb <inet_ioctl+33f/3a0>
Trace: c015b121 <sock_ioctl+21/28>
Trace: c012f910 <sys_ioctl+14c/178>
Trace: c0108c00 <system_call+34/38>
Code:  c01b2ff8 <ace_open+10/130>  00000000 <_EIP>: <===
Code:  c01b2ff8 <ace_open+10/130>     0: c7 05 00 00 00  movl $0x42,0x0 <===
Code:  c01b2ffd <ace_open+15/130>     5: 00 42 00 00 00
Code:  c01b3002 <ace_open+1a/130>     a: 83 bb c4 76 01  cmpl $0x0,0x176c4(%ebx)
Code:  c01b3007 <ace_open+1f/130>     f: 00 00
Code:  c01b3009 <ace_open+21/130>    11: 75 1d           jne   c01b3028 <ace_open+40/130>
Code:  c01b300b <ace_open+23/130>    13: 8b 00           movl (%eax),%eax
```


Changes in 2.3.x

- Mainly the declaration of sempahores and wait queues
  - semaphores:

```
Before:
    struct semaphore sem = MUTEX_LOCKED;
    somestruct->sem = MUTEX;

Now:
    DECLARE_MUTEX_LOCKED(sem);
    init_MUTEX(&somestruct->sem);
```

  - wait queues:

```
Before:
    struct wait_queue wait = {current, NULL};
    struct wait_queue *xdc_wait = NULL;
    init_waitqueue(&thread->wqueue);

Now:
    DECLARE_WAITQUEUE(wait, current);
    DECLARE_WAIT_QUEUE_HEAD(xdc_wait);
    init_waitqueue_head(&thread->wqueue);
```

Other sources of info

- The **Documentation**/ directory contains a lot of valuable information
- Linux Documentation Project (LDP) http://metalab.unc.edu/LDP/
- The Linux Kernel, David A Rusling, 1996-1999,
  - ftp://metalab.unc.edu
  - /pub/Linux/docs/linux-doc-project/linux-kernel/
- Linux Kernel Internals, 2nd. edition, Beck et al. Addison Wesley 1998
- Linux Device Drivers, Alessandro Rubini, O'Reilly & Associates Inc. 1998
- Use the source Luke