Linux Kernel Hacking

aka

“Yvahk Xreary Unpxvat”

Almost Completely Based On The
2005 LCA Kernel Hacking Tutorial
By Rusty Russell And
Robert Love

With Minor Additions By
Muli Ben-Yehuda
mulix@mulix.org , muli@il.ibm.com

The Linux Kernel

- 18,000 files
- 15,000 source files
- 24 architectures
- A *lot* of code, but mostly in drivers and different architectures
- The greatest kernel *ever*

Directory Structure

- arch/
- crypto/
- Documentation
- drivers/
- fs/
- include/
- init/
- ipc/
- kernel/
- lib/
- mm/
- net/
- scripts/
- security/
- sound/
- usr/

Unpacking, Configuring, and Building

- `tar xvjf linux-2.6.12.tar.bz2`
- `cd linux-2.6.12/`
- `make [defconfig|oldconfig|…]`
- `make`

Beast of a Different Nature

- Written in GNU C and inline assembly
- No memory protection
- Very small stack
- Concurrency concerns: interrupts, preemption, SMP
- No libc
- No (easy use of) floating point
- Portability is a must: 64-bit clean, endian neutral

Entry Points to the Kernel

- Booting
  - `init/main.c`
- System calls/Traps
  - `arch/i386/kernel/entry.S`
- Interrupts
  - `arch/i386/kernel/irq.c`

System Calls

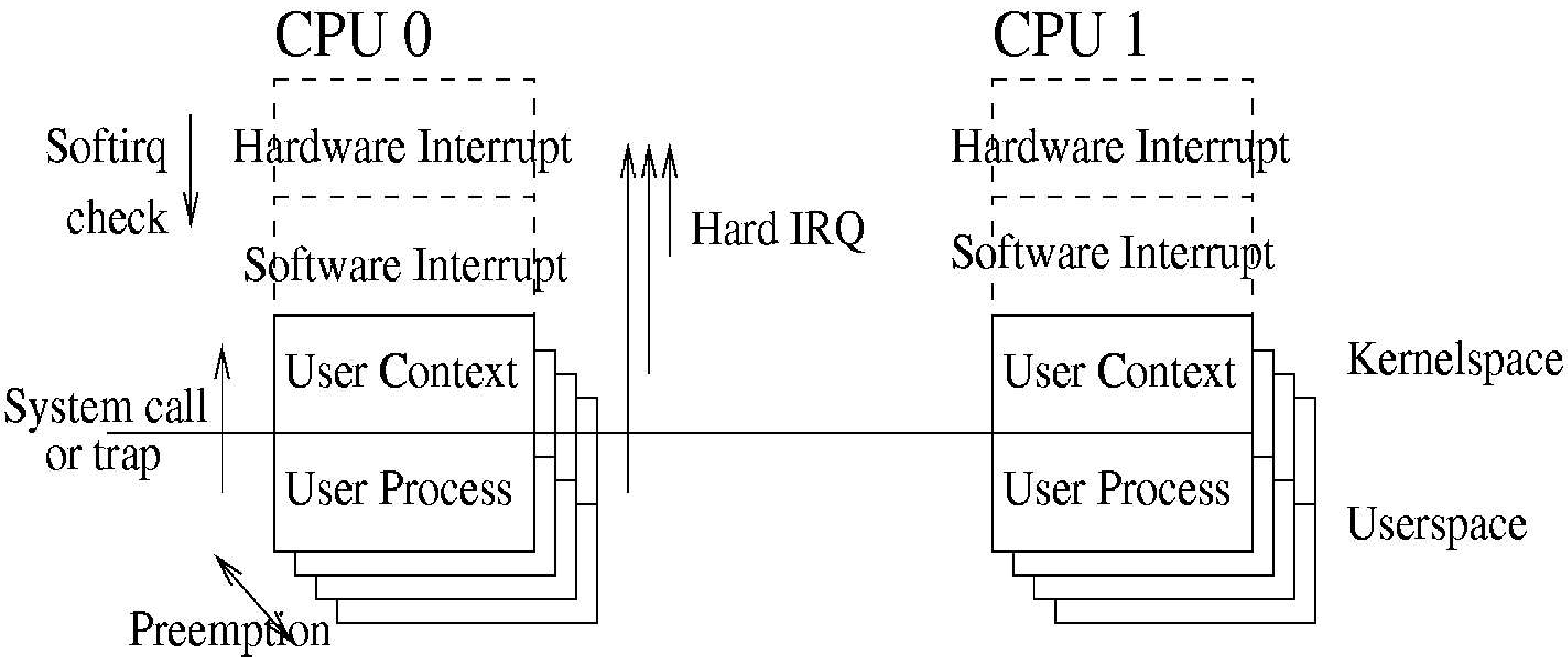
- System calls are the legal method of trapping into the kernel from user-space
- Syscalls are referenced by number
  - for x86, defined in `arch/i386/kernel/entry.S`
- Returning negative error code sets `errno`
  - example: `return -ENOMEM;`
- Syscalls run in the kernel in the context of the invoking process—called *process context*
- Defined with `asmlinkage` and return a `long`

User Context

- Processes only enter the kernel via trap: system call, page fault, other exceptions
- `current` is a pointer to the process's task structure (process descriptor)
  - e.g. `current->pid` is the pid
- Process can *sleep* if needed: Block on a wait queue and wait until event occurs
  - Done by calling `schedule()`
    - explicitly or implicitly with f.e. a semaphore

Interrupt Context

- *Interrupt handlers* run in response to *interrupts*
- Interrupt handlers run in *interrupt context*
- There is not an associated process
    - `current` is not valid
    - Cannot sleep
- Handlers run with their interrupt line disabled
- Speed is crucial, as the handler interrupted previously executing code

CPU 0
CPU 1
Softirq
check
Hardware Interrupt
Software Interrupt
Hard IRQ
Hardware Interrupt
Software Interrupt
User Context
User Context
Kernelspace
System call
or trap
User Process
User Process
Userspace
Preemption

QEMU

- Full system simulator
- Simulates (in software) a complete computer system (CPU, RAM and peripheral devices)
- Excellent for kernel debugging (although there's no replacement for having real live hardware)
- We are using a copy of QEMU that has the LR3K device

LR3K

- LR3K is a “ROT13” encryption device
- This is how LR3K is hooked into QEMU

```
+void pci_lr3000_init(PCIBus *bus)
+{
+  struct lr3000 *d;
+  uint8_t *pci_conf;
+
+  d = container_of(pci_register_device(bus, "LR3000", sizeof(*d),
+                                  -1, NULL, NULL),
+                struct lr3000, dev);
+  pci_conf = d->dev.config;
+  pci_conf[0x00] = 0x11; // Manufacturer
+  pci_conf[0x01] = 0x11;
+  pci_conf[0x02] = 0x01;
+  pci_conf[0x03] = 0x02;
+  pci_conf[0x0a] = 0x00; // Network and computing encryption device
+  pci_conf[0x0b] = 0x10;
+  pci_conf[0x0e] = 0x00; // header_type
+  pci_conf[0x3d] = 1; // interrupt pin 0
+
+  pci_register_io_region(&d->dev, 0, 0x100,
+                      PCI_ADDRESS_SPACE_IO, lr3000_map);
+}
```

LR3K continued

- Registering with QEMU for IO space accesses

```
#include <linux/init.h>
+static void lr3000_map(PCIDevice *pci_dev, int region_num,
+                       uint32_t addr, uint32_t size, int type)
+{
+  struct lr3000 *l = container_of(pci_dev, struct lr3000, dev);
+
+  /* First twelve bytes are simple. */
+  register_ioport_write(addr, 12, 4, ioport_write_src_dst_len, l);
+  register_ioport_read(addr, 12, 4, ioport_read_src_dst_len, l);
+
+  /* Write-only control word. */
+  register_ioport_write(addr + 12, 4, 4, ioport_write_control, l);
+
+  /* Read only result word. */
+  register_ioport_read(addr + 16, 4, 4, ioport_read_result, l);
+
+  /* Read only SHA. */
+  register_ioport_read(addr + 20, MD4_DIGEST_SIZE, 1, ioport_read_md4,l);
+}
```

Your Mission

- Write first kernel module
- Add Makefile info
- Add Kconfig info
- Build it
- Load it
- Unload it
- Start hacking the SCSI layer

Hello World Module

- Save this as `drivers/crypto/l3rk.c` in your kernel source tree:

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

static int hello_init(void)
{
   printk(KERN_ALERT "Hello, world!\n");
   return 0;
}

static void hello_exit(void)
{
   printk(KERN_ALERT "Goodbye, world!\n");
}

module_init(hello_init);
module_exit(hello_exit);
```

Hello World Makefile

- The kernel build process is vastly improved in 2.6
- Constructing kernel Makefiles is easy
  - E.g., `drivers/crypto/Makefile`
  - Add to it:

```
obj-$(CONFIG_CRYPTO_LR3K) += lr3k.o
```

- Save your module as `drivers/crypto/lr3k.o`

Configuration Entries

- Edit `drivers/crypto/Kconfig`
- Add a new entry:

```
config CRYPTO_LR3K
tristate “Support for the Love-Rusty 3000”
help
   The greatest crypto card in the word.
```

Building

- `make oldconfig`
- Set `CONFIG_CRYPTO_LR3K=m`
- `make`

Installing Modules

- As root:

```
mkdir /mnt/qemu
mount -o loop -o offset=32256 image.img /mnt/qemu
make INSTALL_MOD_PATH=/mnt/qemu modules_install
umount /mnt/qemu
```

Loading and Unloading Modules

- Modules are built as `name.ko`
- Run qemu:
  - `qemu -hda ../debian -kernel arch/i386/boot/bzImage -append “ro root=/dev/hda1”`
- Login as root/root to qemu, load the module with “`modprobe name`”
- As root, unload with “`modprobe -r name`”

Making it Easy

- Let's write a script to automate this stuff, name it `test-kernel`:

  -
    ```
    #! /bin/sh

    mount -o loop -o offset=32256 ../debian /mnt/qemu

    make INSTALL_MOD_PATH=/mnt/qemu modules_install

    umount /mnt/qemu

    qemu -hda ../debian -kernel arch/i386/boot/bzImage -append “ro root=/dev/hda1”
    ```

Tainted Kernel?

- You probably saw an error about a tainted kernel
- Add the line
  `MODULE_LICENSE(“GPL”);`
  to the end of the file
- Go and add your names as the author, too:
  `MODULE_AUTHOR(“Lennon, McCartney”);`
- And some nice comments at the top of the file, listing your copyright and license

What was that printk() thing?

- `printk()` is the kernel's version of `printf()`
- Use is the same except for the optional use of a kernel log level
  - e.g. `KERN_WARNING` and `KERN_DEBUG`
  - `printk(KERN_INFO “My dog smells\n”);`
- You can always call `printk()`
  - As we will see, most kernel interfaces are not so robust

Module Initialization and Exit

- `module_init()` marks a module's init function
- Invoked by the kernel when the module is loaded
- Returns zero on success, negative error code on failure
  - Standard kernel convention

- `module_exit()` marks a module's exit function
- Invoked by the kernel when the module is unloaded
- No return value

Registering a PCI Device

- PCI devices are registered via

```
pci_register_driver(struct pci_driver *dev)
```

- PCI devices are unregistered via

```
pci_unregister_driver(struct pci_driver *dev)
```

struct pci_driver

- the `pci_driver` structure defines properties of a PCI driver
- Example:

```
static struct pci_driver foo_driver {
.name = “foo”,
.probe = foo_probe,
.remove = foo_remove,
.id_table = foo_tbl,
};
```

Complete PCI Device Registration and Init

-
  ```
  static struct pci_driver foo_dev {
    .owner = THIS_MODULE,
    .name = “foo”,
    .probe = foo_probe,
    .remove = foo_remove,
    .id_table = foo_id_tbl,
  };

  static int foo_init(void)
  {
    return pci_register_driver(&foo_dev);
  }

  static void foo_exit(void)
  {
    pci_unregister_driver(&foo_dev);
  }

  module_init(foo_init);
  module_exit(foo_exit);
  ```

PCI Probe Function

- Invoked in response to kernel detecting device
- Example:

```
static int foo_probe(struct pci_driver *pdev,
                     const struct pci_device_id *id)
{
/* tell the kernel that we are alive */
pci_enable_device(pdev);

/* init the physical hardware ... */

printk(KERN_INFO “Foo driver is loaded!\n”);

return 0;
}
```

PCI Device Table

- Describes to the PCI layer the devices that this driver supports
- Defined in `<linux/mod_devicetable.h>` as

```
struct pci_device_id {
__u32 vendor, device;       /* Vendor and device ID or PCI_ANY_ID*/
__u32 subvendor, subdevice;  /* Subsystem ID's or PCI_ANY_ID */
__u32 class, class_mask;/* (class,subclass,prog-if) triplet */
kernel_ulong_t driver_data;  /* Data private to the driver */
};
```

- `PCI_ANY_ID` means “anything matches”
- The PCI layer will automatically calls your probe function for any matching device

PCI Device Table Example

- Example:

```
static struct pci_device_id foo_id[] = {
  { 0x1111, 0x0201, PCI_ANY_ID, PCI_ANY_ID, 0, 0, 0 },
  { 0 }
};
```

- This driver is saying that it supports a device with a vendor ID of 0x1111, a device ID of 0x0201, with any subvendor and subdevice ID's
- Array is zero-terminated

Your Mission: Compile

- Fill out an array of `pci_device_id` structures to identify the LR3K
  - `#include <linux/mod_devicetable.h>`
- Define a `pci_driver` structure
  - Include `id_table` and a `name` field
  - `#include <linux/pci.h>`
- Call `pci_register_driver()` and `pci_unregister_driver()`

Your Mission: Compile and Test

- Write a probe function
  - Call `pci_enable_device()` in the right place
  - Add your probe to the `pci_driver` structure
  - `printk()` something charismatic in the probe

Summary

- `pci_driver` structure
- `pci_device_id` structure
- Probe and Remove functions
- Registering and Unregistering PCI drivers

What's Next?

- Talking to devices
- Virtual versus Physical Memory
- PCI I/O space
- Writing to and reading from PCI I/O mappings

PCI I/O Space

- PCI provides into own I/O memory space
- Can be mapped into virtual memory
- This allows device drivers to read from and write to PCI device's memory (regions, mappings, etc.) via pointers to normal memory addresses

pci_iomap

- Defined in `<asm/iomap.h>` as

  ```
  void * pci_iomap(struct pci_dev *pdev, int bar, unsigned int max)
  ```

- `bar` holds the BAR
  - Base Address Register
  - Where to start the mapping
  - Often zero
- `max` is the amount to map (i.e. size)
- Unmap with `pci_iounmap(struct pci_dev *pdev, void *iomap)`

pci_iomap Example

- Example:

```
void *iomap;

iomap = pci_iomap(pdev, 0, sizeof (struct foo));
if (!iomap)
  /* error */
```

Reading from and Writing to I/O Memory

- `int ioread32(void *iomap)`
  - Reads and returns the 32-bit word starting at the given address
- `void iowrite32(u32 word, void *iomap)`
  - Writes `word` to `iomap`
- Both defined in `<asm/iomap.h>`

I/O Examples

- Example:

```
void *iomap;
char buf[] = “dog”;
int foo;

iomap = pci_iomap(pdev, 0, sizeof(struct my_regs));
if (!iomap)
  return -EFAULT;

/* write the physical LOCATION of “buf” */
iowrite32(virt_to_phys(buf), iomem);

/* read the second word */
foo = ioread32(iomem + 4);
```

Your Mission: Compile and Test

- Create a C structure that contains the layout of the register structure
  - Per your data sheet
  - Name it `lr3k_regs`
- Use `pci_iomap()` to map the registers from the LR3K device
  - `include/asm-generic/iomap.h`
- Print out the `result` register

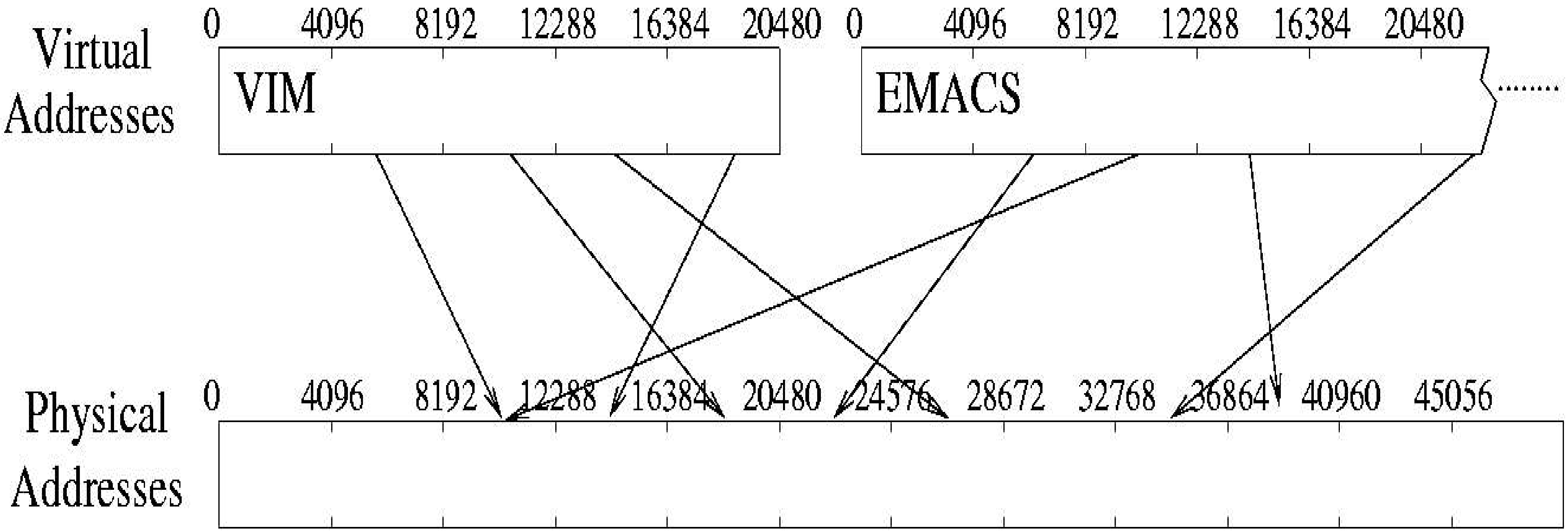
Summary

- PCI I/O space
- `pci_iomap()`
- `iowrite32()` and `ioread32()`

Virtual versus Physical Memory

- Modern machines implement virtual memory
  - User-space is used to dealing with virtual addresses
    - Addresses zero to 0xffffffff
    - Cannot see other process's memory
  - Memory is divided up into pages (4K)
    - Hardware transparently maps addresses onto physical memory
- Virtual memory has many benefits: large separate virtual address spaces, demand paging, protection, sharing physical memory

Virtual Address Spaces



Getting physical address of virtual memory

- The kernel knows about physical memory and page tables
- `unsigned long virt_to_phys(void *addr)`
  - Declared in `<asm/io.h>`
  - Given the virtual address `addr`, returns the backing location in physical memory
  - Not very portable and full of other problems but good for testing
- Physical devices will want physical addresses

Your Mission: Compile and Test

- Declare input and output test buffers on the stack
  - `char in[] = “Something funny”;`
  - `char out[sizeof (in)];`
- Program the `src`, `dst`, and `len` registers
- Write `LR3K_IN_ACTIVATE` to `control`
- Loop until not busy
- Print output

Your Mission: Compile and Test

- Handle errors
  - `pci_enable_device()` can fail
  - `pci_iomap()` can fail
  - Hardware can fail and `result` register may indicate error
- Cleanup in response to errors
- Return valid error values
  - See `include/asm-generic/errno-base.h`

Dynamic Memory Allocations

- `kmalloc(size, flag)`
  - Just like `malloc()`: allocates at least `size` bytes
  - `flag` stipulates the type of allocation
    - `GFP_KERNEL` says the caller is able to wait for the memory to become available (the usual case)
    - `GFP_ATOMIC` is used for special cases
- `kfree(buf)`

kmalloc() Example

-
  ```
  struct foo *f;

  f = kmalloc(sizeof (struct foo), GFP_KERNEL);
  if (!f)
    /* handle error */

  /* ... */

  kfree(f);
  ```

Storing Data in your pci_driver

- You can stuff personal data in the pci_driver structure and retrieve it later
- `void pci_set_drvdata(struct pci_driver *, void *)`
  - Associates the given data with the given `pci_driver` structure
- `void * pci_get_drvdata(struct pci_driver *)`
  - Returns the data associated with the given `pci_driver` structure

PCI Remove Function

- Kernel calls this function when device is removed
- Stored in the `remove` field of `pci_driver`
- Example:

```
static void foo_remove(struct pci_driver *pdev)
{
/* shutdown the device ... */

pci_disable_device(pdev);
}
```

- Your probe function can use `pci_set_drvdata()` to store data that you need during remove

Your Mission: Compile and Test

- Declare a `struct lr3k` containing the PCI I/O map pointer
- Allocate this structure dynamically using `kmalloc()` in your probe routine
- Use `pci_set_drvdata()` to store the `lr3k` structure
- Write a remove function that uses `pci_get_drvdata()` and cleans up
- Add the remove to `pci_driver`

Your Mission: Compile and Test

- Handle endianness

  -
    ```
    #define write_register(value, iomem, member) \
        iowrite32(cpu_to_le32(value),        \
            (iomem) + offsetof(struct lr3k_regs, member))

    #define read_register(iomem, member) \
     le32_to_cpu(ioread32((iomem) + \
        offsetof(struct lr3k_regs, member)))
    ```
  - Use these everywhere instead of hardcoded writes and reads

Summary

- Physical versus Virtual Memory
- `virt_to_phys()`
- `kmalloc()` and `kfree()`
- PCI remove functions
- `pci_set_drvdata()` and `pci_get_drvdata()`

DMA'able Memory

- Often a device needs to write directly to memory
  - This is called *DMA*, *direct memory access*
  - Memory capable of undergoing DMA is called *DMA-capable*
- A concern when both devices and processors are accessing memory is coherence
  - Processors have caches
  - Does a processor read following a device write return the correct data?

Allocating DMA'able Coherent Memory without Shame

- `void * dma_alloc_coherent(dev, size, dma, flag)`
  - `dev` is your device (`pci_dev->dev`)
  - `size` is the size in bytes of the allocation
  - `dma` is a `dma_addr_t *`
    - Filled in with the physical address
  - `flag` is the allocation flags, same as `kmalloc()`
  - `<asm/dma-mapping.h>`
  - Returns the virtual address

Freeing DMA'able Coherent Memory

- `dma_free_coherent(dev, size, buf, dma)`
  - `dev` is the device (`&pci_dev->dev`)
  - `size` is the size, in bytes, of the allocation
  - `buf` is a pointer to the memory to free, previously returned by `dma_alloc_coherent()`
  - `dma` is the `dma_addr_t`

Allocating and Freeing DMA'able Coherent Memory Example

-

```
dma_addr_t dma;
void *buf;

/* allocate 4 KB */
buf = dma_alloc_coherent(&pdev->dev, PAGE_SIZE,
                                     &dma,
GFP_KERNEL);

memcpy(buf, “hello”, 5); /* buf is virt addr */
iowrite32(dma, iomem+4); /* dma is phys addr */

dma_free_coherent(&pdev->dev, PAGE_SIZE,
                          buf, dma);
```

Your Mission: Compile and Test

- Doing `virt_to_phys()` does not give a valid PCI-visible address on all architectures
- Doing DMA on the stack is problematic on some architectures
- Use `dma_alloc_coherent()` to allocate the buffers for reading and writing
- Read and write from these buffers and not via `virt_to_phys()`

Registering an Interrupt Handler

- Interrupt Handlers are registered via

```
request_irq(irq, handler, flags,
                 name, dev_id)
```

  - declared in `<linux/interrupt.h>`
  - `irq` is the interrupt number requested
  - `handler` is a pointer to the interrupt handler
  - `flags` is a bit mask of options
  - `name` is the name of the interrupt
  - `dev_id` is a unique identifier
  - `<linux/interrupt.h>`

Interrupt Handler

- Interrupt handlers must match the prototype:

```
irqreturn_t handler(int irq, void *data, struct pt_regs *regs)
```

- `irq` is the interrupt number
- `data` is the dev_id value given during registration
- `regs` is a copy of the register contents (almost totally worthless)

Interrupt Handler's Retval

- `irqreturn_t` is a special return type
- Only two legit values
  - `IRQ_HANDLED` on success (or indeterminate)
  - `IRQ_NONE` on failure
- Used by the kernel to detect spurious interrupts

Flags

- Flags parameter provides options relating to the new interrupt handler
- `SA_INTERRUPT`
- `SA_SHIRQ`
- `SA_SAMPLE_RANDOM`

Freeing an Interrupt Handler

- An interrupt handler is removed from a given interrupt line via `free_irq(irq, dev_id)`

Probing for Interrupts

- Older devices needed to be probed
- Or have the interrupt provided
    - Poking, probing, guessing
    - Not pretty
- Modern bus architectures make this easy
- PCI detects and assigns interrupt number automatically
- Kernel interface makes it easy

Registering a PCI Device

- Recall that PCI devices are registered via

  `pci_register_driver(struct pci_driver *pdev)`

- Automatically determines the interrupt number and places it in `pdev->irq`
- Simple

Your Mission: Compile and Test

- Register an interrupt handler with `request_irq()`
- Write an interrupt handler that simply calls `printk()` with a limerick
- Set the `LR3K_IN_IRQ_ON_DMA_COMPLETE` flag along with the existing `LR3K_IN_ACTIVATE` flag

Summary

- DMA-capable memory
- `dma_alloc_coherent()` and `dma_free_coherent()`
- Interrupts, interrupts handlers, probing
- Registering an interrupt handler

Sleeping and Waking Up

- Processes can go to sleep, suspending execution and allowing other processes to run, until some event occurs that wakes them up
  - For example a consumer might sleep until the producer creates more data, at which time the producer would wake the consumer up

Sleeping

- The fundamental way of sleeping is to just call `schedule()`
  - Selects next task to run with `state==TASK_RUNNING`
    - May be the current task
    - If no other runnable tasks, selects the idle task
- Usually need better control, though, so wrappers are used
  - `ssleep(n)` sleeps for `n` seconds or until woken up

Waking another process up

- Wake up a sleeping process with `wake_up_process(struct task_struct *)`
  - Returns task's state to `TASK_RUNNING` so it will be run on the next `schedule()`
- Recall that `current` is the `task_struct` of the currently running process

Your Mission: Compile and Test

- Have the interrupt handler wake up the waiting task when the work is complete
- After activating the device, sleep using `ssleep()` for three seconds until the interrupt hits and wakes you up

Bug?

- Did you notice that the reader always slept for three seconds?
  - The interrupt is too fast. Our awesome hardware and qemu respond immediately.
  - The interrupt hits before the `ssleep()` is invoked, and thus the process sleeps unconditionally for three seconds.
    - program `activate` register
    - interrupt hits and sets `current->state=TASK_RUNNING`
    - current calls `ssleep()`
      - `ssleep()` sets `current->state=TASK_UNINTERRUPTIBLE`
      - calls `schedule()`
    - Wakes up three seconds later

Process States

- Processes are in various states: running, sleeping, zombie, etc.
- `TASK_RUNNING` means running or runnable
- `TASK_UNINTERRUPTIBLE` means sleeping (and not responding to signals)
- `set_current_state(foo)` sets the state of the current process to `foo`
  - e.g. `set_current_state(TASK_UNINTERRUPTIBLE)`
- `ssleep()` did this automatically

Your Mission: Compile and Test

- `schedule_timeout(n)` sleeps for `n` clock ticks or until woken up
  - There are `HZ` clock ticks in a second
  - Does *not* automatically set the state
  - Example:

```
set_current_state(TASK_UNINTERRUPTIBLE);
/* optionally do stuff ... */
schedule_timeout(HZ);
```

- Can you now fix the bug?

Summary

- Processes can sleep via `schedule()`
- Processes have states
- Most code uses wrappers that perform other functions too, such as `ssleep()` or `schedule_timeout()`
- `wake_up_process()` wakes a process up

Your Mission: Compile and Test

- Check the device `result` register in the interrupt handler
  - If it is not our interrupt, return `IRQ_NONE`
- Remove the test code from the probe routine
  - Leave card I/O mapped and setup but do not use card in probe routine

Linked Lists

- Data structure for dynamically linking multiple objects together
- Kernel provides a nice circular double linked list interface in `<linux/list.h>`
- Unique implementation: Pointers for each node go *inside* of the object
  - No list of pointers as in classic linked lists
  - Just a bunch of objects that point to each other

Creating a linked list

- Create a global list head:
```
static LIST_HEAD(my_list);
```
- Add a list_head structure to your object:
```
static struct foo {
struct list_head list;
/* ... */
}
```

Adding to and Removing from the list

- Add:
  `list_add(&foo->list, &my_list);`
  - Adds `foo->list` as a node to `my_list`
- Remove:
  `list_del(&foo->list);`
  - Removes `foo->list` from whatever list it may be in

Walking the List

- Walk a list:

```
struct foo *i;

list_for_each_entry(i, &my_list, list) {
  /* 'i' points to a node in the list */
}
```

  - `i` is a temp iterator holding each node
  - `my_list` is the global list head
  - `list` is the name of the list variable inside of each `foo` structure (recall that we named the variable “list”)

Your Mission: Compile and Test

- What if a user's machine has more than one LR3K card?
- Add support for multiple cards to the driver
  - Create a global list of cards
  - Add card on PCI probe
  - Remove card on PCI remove
  - Implement a `find_card()` that returns first entry in list (or `NULL`)

What's Next

- Need a way for user-space to access the card
- The Unix-way is to provide a device file that can be read from or written to

Devices

- Device files abstract device drivers and other special kernel interfaces as normal files
  - example: `/dev/null` and `/dev/hda1`
  - Accessed via normal Unix system calls: read, write, open, close
- Block devices
- Character devices
- Misc devices

Character Devices

- Abstraction for devices that are accessed sequentially
  - Character by character
  - No seeking
- Files are opened, read from, written to, and then closed
- Examples: `/dev/zero`, `/dev/input/mice`

Creating a New Character Device

- What you need:
  - A major number, such as 42, or a carefree attitude that allows you to not even care what the major number is
  - A name, such as “lettuce”
  - A pointer to a structure defining a bunch of function pointers that implement the various system calls that act on the device

register_chrdev()

-
  ```
  int register_chrdev(unsigned int major,
              const char *name,
              struct file_operations *fops)
  ```
- `major` is the request major number
  - If `major` is zero, a major is automatically assigned
- `name` is the name of the character device
- Returns the assigned major number on success or a negative error code on failure

register_chrdev() example

- Example:

```
int retval;

retval = register_chrdev(0, “foo”, &fops);

if (retval < 0)
     /* error */
else
     /* 'retval' is the major */
```

Unregistering a Char Device

- `unregister_chrdev(major, name)`
  - Unregisters the previously registered device with the given major and name
- Example:

```
unregister_chrdev(major, “foo”);
```

File Operations Table

- Defined in `<linux/fs.h>`
- Contains function pointers to various VFS functions, such as read, write, open, close, etc.
  - A *lot* of operations
    - Do *not* need to define all of them
    - *Do* need to define some of them

struct file_operations

- Example (only some of the fields):

```
struct file_operations {
  struct module *owner;
  ssize_t (*read) (struct file *, char *,
                              size_t, loff_t *);
  ssize_t (*write) (struct file *, const char *,
                                 size_t, loff_t *);
  int (*open) (struct inode *, struct file *);
  int (*release) (struct inode *, struct file *);
}
```

Defining your own file operations

- For example, When the user does an `open()` on your device, the `open` method is invoked in response
- Example:

```
static struct file_operations fops = {
.owner   = THIS_MODULE,
.read    = my_read,
.write   = my_write,
.open    = my_open,
.release = my_release,
};
```

An Open Function

- Called when the device is opened
  - Initializes device, etc.
-
```
struct foo_module { void *buf; /* ... */ };

static int my_open(struct inode *inode, struct file *file)
{
struct foo_module *f;

f = kmalloc(sizeof (struct foo_module), GFP_KERNEL);
if (!f)
    return -ENOMEM;
f->buf = kmalloc(PAGE_SIZE, GFP_KERNEL);
if (!f->buf) {
    kfree(f);
    return -ENOMEM;
}
memset(f->buf, 0, PAGE_SIZE);
file->private_data = f;

return 0;
}
```

A Release Function

- Called on final close
- Cleans up reference to open device
- Example:

```
static int my_release(struct inode *inode,
                      struct file *file)
{
struct foo_module *f;

f= = file->private_data;

kfree(f->buf);
kfree(f);
return 0;
}
```

Your Mission: Compile and Test

- Create a character device
  - Register it on module init
  - Unregister it on module exit
- Implement open
  - Use your find card function to find a card
    - Return -ENODEV if no card exists, otherwise zero
- Implement release
  - Just return zero

Your Mission: Compile and Test

- Enhance your open routine to dynamically allocate, populate, and stuff in `file->private_data`:

  -
    ```
    struct lr3k_file {
      struct lr3k *lr3k;
      unsigned int size;
      char *inpage, *outpage;
      dma_addr_t in, out;
      };
    ```
- Clean up and free in release
  - Hint: You need a new member in `struct lr3k`

Getting Data To and From the User

- `int copy_to_user(void *dst, void *src, size_t size)`
  - `<asm/uaccess.h>`
  - Copies `size` bytes from `src` in kernel-space to `dst` in user-space
  - Returns zero on success, number of bytes not copied, or negative error code
- `int copy_from_user(void *dst, void *src, size_t size)`
  - `<asm/uaccess.h>`
  - Copies `size` bytes from `src` in user-space to `dst` in kernel-space
  - Returns zero on success, number of bytes not copied, or negative error code

A Write Function

- Example:

```
static int my_read(struct file *file, char *data,
                   size_t size, loff_t *off)
{
  struct foo_module *f;

  f = file->private_data;

  if (size > PAGE_SIZE)
       size = PAGE_SIZE;

  if (copy_to_user(f->buf, data, size))
       return -EFAULT;

  return size;
}
```

A Read Function

- Example:

```
static int my_read(struct file *file, char *data,
                   size_t size, loff_t *off)
{
  struct foo_module *f;

  f = file->private_data;

  if (size > PAGE_SIZE)
       size = PAGE_SIZE;

  if (copy_to_user(data, f->buf, size))
       return -EFAULT;

  return size;
}
```

Your Mission: Compile and Test

- Implement write function
  - copy data into `inpage`
  - set `size` for read
  - program card to encrypt `inpage`
- Implement read function
  - copy up to `size` bytes from `outpage`
  - reset `size` to zero
- Put these in the `file_operations` structure

Interrupts, Processes, SMP

- Everything running in the kernel shares the same memory: share globals and static vars.
  - Multiple processes can be playing with the same data:
    - eg. one reads from a file while another writes to it.
  - Interrupt handlers can interrupt and play with data while a process is also playing with it.
  - Other CPUs can be playing with data while we are playing with it (`CONFIG_SMP` only).
- These are usually called *races*.

Interrupts, Processes, SMP

- Consider “`i++`” in C.
- In PPC assembler, this becomes:

  -
    ```
    lwz r9,0(r3)    # Load contents of R3 + 0 into R9
    addi r9,r9,1    # Add one to R9
    stw r9,0(r3)    # Put contents of R9 back into R3 + 0
    ```

Interrupts, Processes, SMP

- Race with an interrupt:

  -
    ```
    lwz r9,0(r3)   # Load contents of R3 + 0 into R9
    ****INTERRUPT****
     ...
     lwz r9,0(r3)   # Load contents of R3 + 0 into R9
     addi r9,r9,1   # Add one to R9
     stw r9,0(r3)   # Put contents of R9 back into R3+0
     ...
    ****RETURN FROM INTERRUPT***
    addi r9,r9,1   # Add one to R9
    stw r9,0(r3)   # Put contents of R9 back into R3 + 0
    ```

Interrupts, Processes, SMP

- Race with another process by being preempted:

  -
    ```
    lwz r9,0(r3)   # Load contents of R3 + 0 into R9
    ****Process 1 kicked off CPU. Process 2:****
    ...
    lwz r9,0(r3)   # Load contents of R3 + 0 into R9
    addi r9,r9,1   # Add one to R9
    stw r9,0(r3)   # Put contents of R9 back into R3+0
    ...
    ****Process 1 returns to CPU***
    addi r9,r9,1   # Add one to R9
    stw r9,0(r3)   # Put contents of R9 back into R3 + 0
    ```

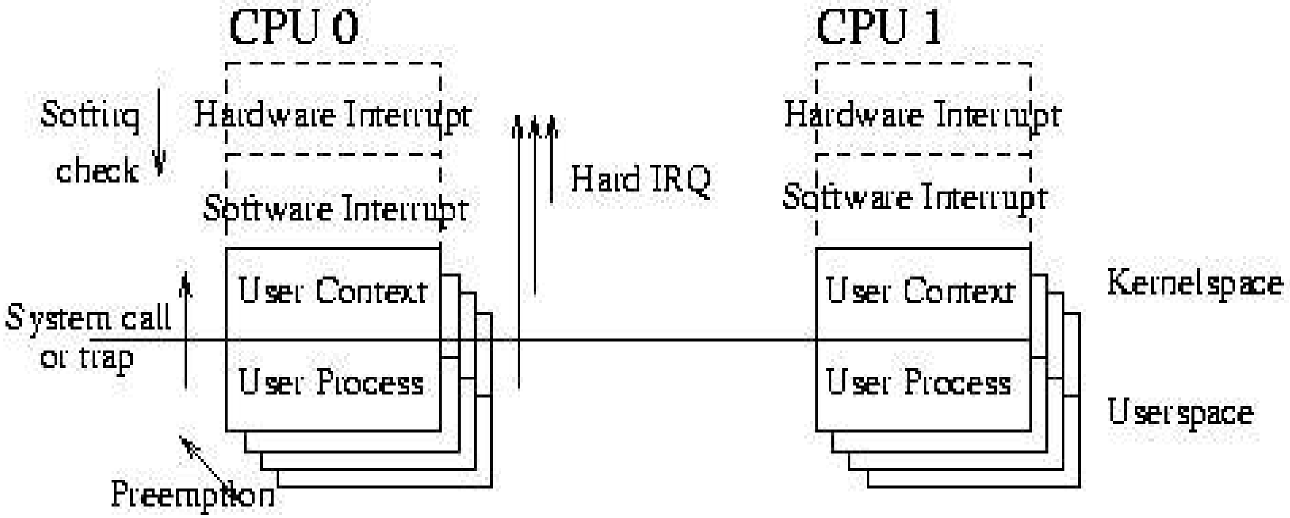
Interrupts, Processes, SMP

- Race with another CPU:

  -
    ```
    CPU 1             CPU 2
                   ...
    ...            lwz r9,0(r3)
    lwz r9,0(r3)   addi r9,r9,1
    addi r9,r9,1   stw r9,0(r3)
    stw r9,0(r3)   ...
    ...
    ```

Who Can Race Me?

- Here is a diagram of who can run at the same time:
  - Worry about those beside you and above you.



What Can I Do About It?

- We can prevent changes on this CPU:
  - Stop hardware interrupts:
    - `local_irq_disable() / ...enable()`
`local_irq_save(flags) / ...restore(flags)`
  - Stop softirqs (aka. bottom halves):
    - `local_bh_disable() / ...enable()`
  - Stop other processes from running:
    - `preempt_disable() / ...enable()`

What Can I Do About It?

- eg: an interrupt handler increments a variable set in user context:

  -
    ```
    static int i;

    static irqreturn_t irq_handler(...)
    {
     i++;
    }
    ...
    static void myfunc(void)
    {
     local_irq_disable();
     i++;
     local_irq_enable();
    }
    ```

What Can I Do About It?

- To protect data from other CPUs, we need locks.
- “spinlocks” can be used everywhere
  - You will spin until you get it.
  - You can't sleep/call schedule() while holding one.
- Simple interface:
  - `static DEFINE_SPINLOCK(lock);`
  - `void spin_lock(spinlock_t *lock);`
  - `void spin_unlock(spinlock *lock);`

What Can I Do About It?

- spinlock combo meal deals available:
  - `spin_lock_bh(lock)/ ...unlock_bh`
  - `spin_lock_irq(lock)/ ...unlock_irq`
  - `spin_lock_irqsave(lock, flags)`
    `/ spin_unlock_irqrestore(lock, flags)`

What Can I Do About It?

- eg. interrupts handler increments variable (SMP-safe version):

    -
      ```
      static int i;
      static DEFINE_SPINLOCK(i_lock);

      static irqreturn_t irq_handler(...)
      {
       spin_lock(&i_lock);
       i++;
       spin_unlock(&i_lock);
      }
      ...
      static void myfunc(void)
      {
       spin_lock_irq(&i_lock);
       i++;
       spin_unlock_irq(&i_lock);
      }
      ```

What Can I Do About It?

- We can use the generic _irqsave version:

```
static increment_i(void)
{
 unsigned long flags;
 spin_lock_irqsave(&i_lock, flags);
 i++;
 spin_unlock_irqrestore(&i_lock, flags);
}

static irqreturn_t irq_handler(...)
{
 increment_i();
}
...
static void myfunc(void)
{
 increment_i();
}
```

Who Can Race Me?

			Soft Interrupts		
	User-space	User-context	Tasklet	Softirq	IRQ
Same one runs simultaneously on other CPU?	No	No	No	Yes	No
Same type runs simultaneously on other CPU?	Yes	Yes	Yes	Yes	Yes
Interrupted by same type?	Yes	Yes*	No	No	Yes
Interrupted by soft interrupts?	Yes	Yes	No	No	No
Interrupted by hard interrupts?	Yes	Yes	Yes	Yes	Yes

Find the Races!

- Checklist:
  - For each piece of data / object:
    - What pieces of code read or write that data?
    - Can those pieces of code run at the same time?
    - If so, you need some locking.
  - Don't have to worry about:
    - Stack variables (everyone gets their own stack)
    - Things you've created but not put anywhere?

Your Mission: Identify

- How many races can you find in the driver?
  - Which ones need a lock?
    - A spinlock or semaphore?
  - Which ones need irq-disabling?
  - Which ones need something else?