Kernel Programming

Concurrency

- Kernel code does not run in simple environment like application s/w and must be written with the idea that many things can be happening at once.
- Multiple processes can be trying to use the ‘driver’ at the same time that our ‘driver’ is trying to do something else.
- In addition several other s/w abstractions (such as kernel timers) run asynchronously.
- Lastly (but not least) Linux run on SMP, with the result that your driver could be executing concurrently on more than one CPU.
- So Linux kernel, including driver code, must be reentrant – it must be capable of running in more than one context at the same time.

Mechanisms for Mutual Exclusion

- Using variables that are atomically incremented and decremented.
- Using spinlocks to enforce mutual exclusions.
- Using semaphore.

Atomic Bit Operations

- Atomic bit operations are very fast, since they perform the operation using a single machine instruction without disabling interrupts whenever the underlying platform can do that.
- They are guaranteed to be atomic even on SMP platforms.
- Data typing in these functions is architecture dependent.

void set_bit(nr, void *addr);

– Sets bit number nr in the data item pointed to by addr.

void clear_bit(nr, void *addr);

– Clears the specified bit in the data item lives at addr.

void change_bit(nr, void *addr);

– Toggles the bit.

int test_bit(nr, void *addr);

– This function is the only bit operation that doesn't need to be atomic; it simply returns the current value of the bit.

Atomic Bit Operations

int test_and_set_bit(nr, void *addr);

int test_and_clear_bit(nr, void *addr);

int test_and_change_bit(nr, void *addr);

– They also return the previous value of the bit.

Atomic Integer Operations

- Atomic integer operations are very fast, because they compile to a single machine instruction whenever possible.
- They are guaranteed to be atomic even on SMP platforms.
- An atomic_t holds an integer value on all supported architectures, you should not count on an atomic_t holding more than 24 bits.

```
void atomic_set(atomic_t *v, int i);   // = ATOMIC_INIT(0);
```

– Set the atomic variable v to the integer value i.

```
int atomic_read(atomic_t *v);
```

– Return the current value of variable pointed to by v.

```
void atomic_add(int i, atomic_t *v);
void atomic_sub(int i, atomic_t *v);
```

– Add to or subtract from i the atomic variable pointed to by v.

Atomic Integer Operations

void atomic_inc(atomic_t *v);
void atomic_dec(atomic_t *v);
– Increment or decrement an atomic variable.

int atomic_inc_and_test(atomic_t *v);
int atomic_dec_and_test(atomic_t *v);
– If, after the operation, the atomic value is 0, then the return value is true.

int atomic_add_return(int i, atomic_t *v);
int atomic_sub_return(int i, atomic_t *v);
int atomic_inc_return(atomic_t *v);
int atomic_dec_return(atomic_t *v);
– Return the new value of the atomic variable to the caller.

Seqlocks

- They work by allowing readers free access to the resource but requiring those readers to check for collisions with writers and, when such a collision happens, retry their access.

```
void seqlock_init(seqlock_t *lock);//seqlock_t lock=SEQLOCK_UNLOCKED;
```

– Initialize a seqlock.

```
unsigned int seq;
do {
    seq = read_seqbegin(&lock);
} while read_seqretry(&lock, seq);

write_seqlock(&lock);
write_sequnlock(&lock);
```

– _irq / _irq{save/restore} / _bh versions are also available.

Read-copy-update

- Works only when the critical data structure is accessed via pointer.
- When the data structure needs to be changed, the writing thread makes a copy, changes the copy, then aims the relevant pointer (which is seen by readers) at the new version.
- When the kernel is sure that no references to the old version remain, it can be freed. This happens when all processors have scheduled atleast once (all access to the data structure is atomic), after the data structure update.

Spinlocks

- They can be used only when the code in the critical section does not sleep.
- If the lock is not available, they do not cause the caller to sleep, instead they check for the lock to become available in a tight loop.
- On non-SMP platforms, they just disable preemption.

void spin_lock_init(spinlock_t *lock);//spinlock_t lock=SPIN_LOCK_UNLOCKED;
– Initialize a spinlock.

void spin_lock(spinlock_t *lock);
void spin_unlock(spinlock_t *lock);
– Lock and unlock a spinlock.

void spin_lock_irqsave(spinlock_t *lock, unsigned long flags);
void spin_lock_irq(spinlock_t *lock);
void spin_lock_bh(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);
– Lock and unlock a spinlock and additionally disbaling IRQs and BHs.

Reader/Writer Spinlocks

- They allow any number of readers in the critical section at a time but give the writers exclusive access.

  void rwlock_init(rwlock_t *lock);// rwlock_t lock=RW_LOCK_UNLOCKED;
  – Initialize a read-write spinlock.

  void {read/write}_lock(rwlock_t *lock);
  void {read/write}_lock_irqsave(rwlock_t *lock, unsigned long flags);
  void {read/write}_lock_irq(rwlock_t *lock);
  void {read/write}_lock_bh(rwlock_t *lock);
  void {read/write}_unlock(rwlock_t *lock);
  void {read/write}_unlock_irqrestore(rwlock_t *lock, unsigned long flags);
  void {read/write}_unlock_irq(rwlock_t *lock);
  void {read/write}_unlock_bh(rwlock_t *lock);
  – Locking and unlocking.

Semaphores

- Puts caller to sleep if the resource is not available.
- The critical section can have code that sleeps.
- MUTEX operations are implemented as wrappers over the semaphore operations.

void sema_init(struct semaphore *sem, int val);
– Semaphore initialization.

void down(struct semaphore *sem);
int down_interruptible(struct semaphore *sem);
int down_trylock(struct semaphore *sem);
– Lock the semaphore.

void up(struct semaphore *sem);
– Unlock the semaphore.

Reader/Writer Semaphores

- Similar in concept to reader/writer spinlocks.

  void init_rwsem(struct rw_semaphore *sem);
  – Initialization.

  void down_{read/write}(struct rw_semaphore *sem);
  int down_{read/write}_trylock(struct rw_semaphore *sem);
  void up_{read/write}(struct rw_semaphore *sem);
  – Locking and unlocking.

Kernel Threads

- They do not have any user space context.

```
struct task_struct *ts;
void *arg;

int thread(void *data)
{
    while(1) {
      if(kthread_should_stop()) break;
    }
    return 0;
}

ts = kthread_run(thread, arg, “my_thread");
kthread_stop(ts);
```

Interrupt Handling

- Interrupts are a way for a device to let the processor know when something has happened.
- A driver is expected to request an interrupt channel before using it and to release it when finished.

```
int request_irq(unsigned int irq, irqreturn_t (*handler)(int, void *, struct pt_regs *),
    unsigned long flags, const char *dev_name, void *dev_id);
void free_irq(unsigned int irq, void *dev_id);
```

- Some devices “announce” which interrupt they’re going to use.
- Some devices require probing.

```
unsigned long probe_irq_on(void);
// Enable interrupt, generate interrupt, disable interrupt.
int probe_irq_off(unsigned long);
// Service the interrupt.
```

Interrupt Handling in Kernel

- The interrupt number is pushed on the stack and do_IRQ() is called.
- The first thing do_IRQ() does is to acknowledge the interrupt so that the interrupt controller can go on to other things.
- It then obtains a spinlock for the given IRQ number, thus preventing any other CPU from handling this IRQ.
- Then it looks up the handler(s) for this particular IRQ. If the handler is not of type "fast interrupt", then interrupts are enabled in hardware and then the handler is invoked.
- The spinlock is released, any pending software interrupts are handled, and do_IRQ() returns.
- Lastly, there is check for a possible rescheduling on the current CPU.

Interrupt Handler in Drivers

- Interrupt handlers do not execute in the context of a process. They use the stack of the process that got interrupted or a dedicated interrupt stack.
- They can not sleep.
- Servicing the interrupt often includes clearing the cause of the interrupt.
- They should return IRQ_HANDLED if there was a interrupt else IRQ_NONE.
- They can use the "struct pt_regs *" argument to find out what was the CPU doing when the interrupt occurred.
- They should defer all time consuming tasks to bottom halves.

Enabling/Disabling Interrupts

- They work by updating the mask for the specified IRQ in the programmable interrupt controller (PIC), thus disabling or enabling the specified IRQ across all processors.
- They can not be used for shared interrupts.

```
void disable_irq(int irq);
void disable_irq_nosync(int irq);
void enable_irq(int irq);
```

Bottom Half Processing

- The bottom half is a routine that is scheduled by the top half to be executed later, at a safer time.
- The big difference between the top half handler and the bottom half is that all interrupts are enabled during execution of the bottom half.
- Bottom half code that can sleep should use workqueues, else they should use tasklets.
- Tasklets are implemented over softirq (can be called from softirq kernel thread as well)
  - Tasklets may be scheduled to run multiple times, but tasklet scheduling is not cumulative, the tasklet runs only once, even if it is requested repeatedly before it is launched.
  - No tasklet ever runs in parallel with itself but, tasklets can run in parallel with other tasklets on SMP systems.
  - Tasklets are also guaranteed to run on the same CPU as the function that first schedules them, thus ensuring that a tasklet does not begin executing before the handler that schedules it has completed

Bottom Half Processing

- **Tasklet APIs**

```
void my_tasklet_handler(unsigned long);
unsigned long data;
DECLARE_TASKLET(my_tasklet, my_tasklet_handler, data);
tasklet_schedule(&my_tasklet);
```

- Workqueues invoke a function in the context of a special worker process which is a kernel thread (without any user space context).
- **Workqueue APIs**

```
struct work_struct work;
void my_wq_handler(void *);
void *data;
INIT_WORK(&work, my_wq_handler, data);
schedule_work(&work);
```

Sysfs

- The device model in 2.6 kernel provides a single mechanism for representing devices and describing their topology in the system. Such a system provides several benefits
  - A mechanism for providing common facilities, such as reference counting
  - The capability to enumerate all the devices in the system, view their status, and see to what bus they attach
  - The capability to generate a complete and valid tree of the entire device structure of the system, including all buses and interconnections
  - The capability to link devices to their drivers and vice versa
  - The capability to categorize devices by their class, such as input device, without the need to understand the physical device topology
  - The ability to walk the tree of devices from the leaves up to the root, powering down devices in the correct order

Sysfs

- At the heart of the device model is the kobject. It provides basic facilities, such as reference counting, a name, and a parent pointer, allowing the creation of a hierarchy of objects.
- ktypes describe properties shared by kobjects of a related type.
- ksets aggregate collections of related kobjects. The distinction between ktypes and ksets is kept to allow kobjects of identical ktypes to be grouped into different ksets.
- Subsystems represent high-level concepts in the kernel and are a collection of one or more ksets.
- The sysfs filesystem is an in-memory virtual filesystem that provides a view of the kobject object hierarchy. Using attributes, kobjects can export files that allow kernel variables to be read from and optionally written to.
- The Kernel Event Layer implements a kernel-to-user notification system on top of kobjects.

Procfs

- Files in procfs are used to configure kernel parameters, look at kernel structures, glean statistics from device drivers, and get general system information.
- Procfs is a pseudo filesystem. This means that files resident in procfs are not associated with physical storage devices such as hard disks. Instead, data in those files is dynamically created on demand by the corresponding entry points in the kernel.
- To get a first feel of the capabilities of procfs, examine the contents of
  - /proc/cpuinfo
  - /proc/meminfo,
  - /proc/interrupts
  - /proc/bus/usb/devices
  - /proc/stat

Procfs

- Certain kernel parameters can be changed at runtime by writing to files under /proc/sys/
  - For example, you can change kernel printk log levels by echoing a new set of values to /proc/sys/kernel/printk
  - Many utilities (such as ps) and system performance monitoring tools (such as sysstat) internally derive information from files residing under /proc

USB Drivers

- struct urb *usb_alloc_urb(int iso_packets, int mem_flags);
- void usb_free_urb(struct urb *urb);

- struct usb_device *interface_to_usbdev(struct usb_interface *intf);

- unsigned int usb_sndctrlpipe(struct usb_device *dev, unsigned int endpoint);
- unsigned int usb_rcvctrlpipe(struct usb_device *dev, unsigned int endpoint);
- unsigned int usb_sndbulkpipe(struct usb_device *dev, unsigned int endpoint);
- unsigned int usb_rcvbulkpipe(struct usb_device *dev, unsigned int endpoint);
- unsigned int usb_sndintpipe(struct usb_device *dev, unsigned int endpoint);
- unsigned int usb_rcvintpipe(struct usb_device *dev, unsigned int endpoint);

USB Drivers

- void usb_fill_int_urb(struct urb *urb, struct usb_device *dev, unsigned int pipe, void *transfer_buffer, int buffer_length, usb_complete_t complete, void *context, int interval);
- void usb_fill_bulk_urb(struct urb *urb, struct usb_device *dev, unsigned int pipe, void *transfer_buffer, int buffer_length, usb_complete_t complete, void *context);
- void usb_fill_control_urb(struct urb *urb, struct usb_device *dev, unsigned int pipe, unsigned char *setup_packet, void *transfer_buffer, int buffer_length, usb_complete_t complete, void *context);

- int usb_submit_urb(struct urb *urb, int mem_flags);
- int usb_kill_urb(struct urb *urb);
- int usb_unlink_urb(struct urb *urb);

USB Drivers

```
urb->dev = dev;
urb->context = context;
urb->pipe = usb_{rcv/snd}isocpipe(dev, endp);
urb->interval = 1;
urb->transfer_flags = URB_ISO_ASAP;
urb->transfer_buffer = buffer;
urb->complete = complete;
urb->number_of_packets = FRAMES_PER_DESC;
urb->transfer_buffer_length = FRAMES_PER_DESC;
for (j=0; j < FRAMES_PER_DESC; j++) {
    urb->iso_frame_desc[j].offset = j;
    urb->iso_frame_desc[j].length = 1;
}
```

USB Drivers

```
static struct usb_device_id dummy_table [ ] = {
    { USB_DEVICE(USB_DUMMY_VENDOR_ID,
    USB_DUMMY_PRODUCT_ID) },
    { }
};

static struct usb_driver dummy_driver = {
    .owner = THIS_MODULE,
    .name = "dummy",
    .id_table = dummy_table,
    .probe = dummy_probe,
    .disconnect = dummy_disconnect,
};
```

USB Drivers

- USB_DEVICE(vendor, product);
- USB_DEVICE_INFO(class, subclass, protocol);
- USB_INTERFACE_INFO(class, subclass, protocol);

- int (*probe) (struct usb_interface *intf, const struct usb_device_id *id);
- void (*disconnect) (struct usb_interface *intf);

- int usb_register(struct usb_driver *d);
- void usb_deregister(struct usb_driver *d);

- void usb_set_intfdata(struct usb_interface *intf, void *data);
- void *usb_get_intfdata(struct usb_interface *intf);