Kernel driver prog. day 5

Presented by
Hans de Goede

Bit arrays

- Linux has a notion of so called bit arrays, these are represented as an array of unsigned longs
- These may contain more then sizeof(long) bits / be larger then one unsigned long
- Helper macros for these are:
  - BITS_TO_LONGS
  - BIT_WORD
  - BIT_MASK

Atomic bit operations (1)

- set_bit(int nr, volatile unsigned long * addr);
- clear_bit(int nr, volatile unsigned long * addr);
- Note these functions do NOT give any re-ordering or memory barries guarantees

Re-ordering

- Given the following C-code:

```
a = 5;
b = 7;
func();
c = 8;
```

- The compiler is free to generate code for:

```
b = 7;
a = 5;
func();
c = 8;
```

Memory barriers (1)

- If 2 cpu-cores are executing code accessing the same memory address; and
- CPU-1 writes 0xdeadbeaf to that address; immediately followed by;
- CPU-2 reading that address; then
- CPU-2 may or may not read 0xdeadbeaf
- Because the write / read operations may be re-ordered by the memory subsystem

fedora™

Memory barriers (2)

- CPU-2 seeing 0xdeadbeaf can be assured by:
- Using a general memory barrier instruction after the write on CPU-1; and
- A general memory barrier before the read on CPU-2; and
- Ensuring that the general memory barrier on CPU-2 executes after the general memory barrier on CPU-1

Memory barriers (3)

- Linux mutexes / spinlocks imply memory barriers taking care of this for you, so normally you do not need to worry about this as long as you use proper locking
- For much more details see:

https://www.kernel.org/doc/Documentation/memory-barriers.txt

fedora™

Atomic bit operations (2)

- int test_and_set_bit(int nr, volatile unsigned long * addr);
- int test_and_clear_bit(int nr, volatile unsigned long * addr);
- Note these functions imply a memory-barrier, but do not give any re-ordering guarantees

Workqueues

- It is useful to schedule some work to be done in another thread, the main reasons for this:
  - Quickly complete something which another task is waiting on to unblock that task
  - Schedule work which involves sleeping from an atomic context
- The Linux kernel has a mechanism called workqueues for this

Workqueue example (1)

```
#include <linux/workqueue.h>
#define READ_ERROR 0
#define WRITE_ERROR 1
struct driver_data {
    struct work_struct error_recovery_work;
    unsigned long flags;
}
```

Workqueue example (2)

```
irq_return_t driver_irq(int irq, void *dev_id) {
    struct driver_data *d = dev_id;
    if (read_status_flags(d) & ST_FL_READ_ERR)
        set_bit(READ_ERROR, &d→flags);
    if (read_status_flags(d) & ST_FL_READ_ERR)
        set_bit(READ_ERROR, &d→flags);
    schedule_work(&d→error_recovery_work);
    return IRQ_HANDLED;
}
```

fedora

Workqueue example (3)

```
static void error_recovery(struct work_struct *w) {
    struct driver_data *d =
        container_of(w, struct driver_data,
                                    error_recovery_work);
    if (test_and_clear_bit(READ_ERROR, &d→flags))
        do_read_error_recovery(d);
    if (test_and_clear_bit(WRITE_ERROR, &d→flags))
        do_write_error_recovery(d);
}
```

fedora

Workqueue example (4)

```
int driver_probe(...) {
    ...
    INIT_WORK(&d→error_recovery_work,
                    error_recovery);
    ...
}
```

fedora

Questions?

Contact:
hdegoede@redhat.com

Hands on: Coding time!

Contact:
hdegoede@redhat.com