EMBEDDED DEVICE DRIVERS

Linux Device Drivers on Beaglebone Black

LKM: Kernel Threads

- Like user space, Linux kernel supported threading
  - Threads are called kernel threads (kthreads)
  - Run purely in kernel space
    - Hence, user space thread management framework **does not** apply
      - Such as pthreads, etc.

- Kernel thread usage steps
  - Creation
  - Start
  - Stop – by others / self
  - Other functions – waiting, completion, etc.

- Header file:

  #include <linux/kthread.h>

LKM: kthread creation/stop

- Creation - thread has to be “woken up” to start running
  *struct task_struct ***kthread_create***(int (*thread_func)(void *data), const char namefmt[], …);*

  - *thread_func: Function to run in the created thread; API:* ***int (*func)(void *);***
  - *namefmt: Printf style name for the thread*

  - Returns:
    - ***task_struct **** on success
    - ***ERR_PTR(-ENOMEM)*** on failure
- Wake up
  int ***wake_up_process****(struct task_struct *thread);*

- Combined create-and-wake-up/run wrapper
  *struct task_struct ***kthread_run***(int (*thread_func)(void *data), void *data, const char namefmt[]);*

- Stop
  int ***kthread_stop****(struct task_struct *thread);*
  - Sets the ***kthread_stop*** variable of **thread**

LKM: kthread self-exit

- Each kthread has an internal variable
  - ***kthread_stop***
    - Set/Reset by somebody other than this kthread (ideally!)

- Long-running kthread should
  - Check value of its ***kthread_stop*** variable
    - ***kthread_should_stop()***
      - If non-zero, it should return

- A kthread can end itself; optionally returning a value

  *void **do_exit**(long retcode);*
    - Used in “detached” mode (from ***kthread*** parent)

LKM: Completion (kthread)

- Linux kernel has a mechanism
  - To indicate whether some work is done / thread can self-exit

- Called completion
  - Header: ***#include <linux/completion.h>***
  - Variable: *struct* ***completion*** *my_cmpln;*
  - APIs:
    - Creating:

      Static: ***DECLARE_COMPLETION****(my_cmpln);*

      Dynamic: ***init_completion****(struct completion *my_cmpln);*
    - Waiting:

      void ***wait_for_completion****(struct completion *my_cmpln);*
    - Completing:

      void ***complete****(struct completion *my_cmpln);*
    - Checking status:

      bool ***completion_done****(struct completion *my_cmpln);*

LKM: kthread Exercise #1

- Refer ***mod8*** directory
  - ***mod81.c*** contains code for basic kthreads
    - Study the way threads are created in the kernel
      - And how it differs from user space!
    - Compile and load the module on BBB
    - Observe ***dmesg*** output
  - ***mod82.c*** contains code for kthreads with completion
    - Study how a wait thread can be created
      - And made to wait for completion
      - And made to self-exit upon getting the completion
    - Compile and load the module on BBB
    - Test read with ***cat /dev/cdac_dev***
      - Observe ***dmesg*** output
    - Unload the module
      - Observe ***dmesg*** output

LKM: Sharing b/w kthreads

- We know things go downhill
  - When resources are shared
  - Between threads
  - Without proper synchronization
- kthreads suffer from the same malaise
  - Hence we need thread-level synchronization primitives
    - We already know (from user space)
      - Semaphores
      - **Mutexes**
    - Kernel has some more
      - **Spinlocks**
      - Read-write locks
      - Seqlocks
      - Atomic variables

LKM: kthread Exercise #2

- Refer ***mod8*** directory
  - ***mod83.c*** contains code for unprotected shared resources
    - We share an integer and a char buffer b/w 2 kthreads
    - Compile and load the module on BBB
    - Observe ***dmesg*** output
      - What do you see as far as kthread-1's prints are concerned?
  - ***mod84.c*** contains code for protection using a mutex
    - Study how we create a mutex, init it, and perform locks/unlocks
    - Compile and load the module on BBB
    - Observe ***dmesg*** output
      - What do you observe in terms of kthread prints?

LKM: Spinlocks

- In the mutex case
  - When a kthread is trying to acquire a lock
    - If the mutex is not available
      - It goes to sleep
      - Wakes when it gets the mutex lock

- Spinlocks are similar to mutex
  - Single-owner locks
  - Locks / unlock operations

  - But the kthread wanting to lock a spinlock
    - **Spins** – busy spin of the underlying CPU
    - Till it gets the lock
    - No sleep induced

  - Advantage: A fast and simple locking mechanism
  - Restrictions: Waits have to be really short!

LKM: Spinlock API (simple)

- Creation
  - Static

    DEFINE_SPINLOCK*(my_spinlock);*
  - Dynamic

    spinlock_t my_spinlock;

    spin_lock_init*(&my_spinlock);*

- Locking

  spin_lock*(spinlock_t *lock);*

  spin_trylock*(spinlock_t *lock);*

- Unlocking

  spin_unlock*(spinlock_t *lock);*

- Checking status

  spin_is_locked*(spinlock_t *lock);*

LKM: Spinlock API (interrupts)

- Between bottom halves
  - Use simple APIs

- Between process context and bottom half

  spin_lock_bh*(rwlock_t *lock);*
  spin_unlock_bh*(rwlock_t *lock);*
  - Soft interrupts on CPU with the lock are disabled

- Between IRQ and bottom halves

  spin_lock_irq*(rwlock_t * lock);*
  spin_unlock_irq*(rwlock_t * lock);*
  - Disable all IRQs before locking, restore all on unlock

  spin_lock_irqsave*(spinlock_t * lock, unsigned long flags);*
  spin_unlock_irqrestore*(spinlock_t * lock, unsigned long flags);*
  - Save IRQ state before locking, restore exactly on unlock

LKM: kthread Exercise #3

- Refer ***mod8*** directory
  - ***mod85.c*** contains code for spinlock protected resource
    - Again, we share an integer and a char buffer b/w 2 kthreads
    - But we protect using spinlocks
      - Remember we cannot sleep when we have the lock now!
      - *Try sleeping and see what happens!*
    - Compile and load the module on BBB
    - Observe ***dmesg*** output
      - What do you see as far as kthread-1's prints are concerned?

LKM: Read-write Spinlock

- Let's assume a shared resource
  - With multiple readers
  - And 1 single writer

- If we use a spinlock
  - Everybody waits
    - Write has to be **exclusive**
    - But what about read?
  - Readers don't really change the resource
    - So multiple readers can actually read at the same time
  - Thus:
    - Write and read have to be mutually exclusive
    - Writes have to be mutually exclusive
    - Any number of reads can be together!

- This situation is handled by a different primitive
  - Called **Read Write Spinlock (RWLock)**

LKM: RWLock API (simple)

- Creation
  - Static

    DEFINE_RWLOCK*(my_rwlock);*
  - Dynamic

    rwlock_t my_rwlock;
    rw_lock_init*(&my_rwlock);*

- Write Locking / Unlocking

  write_lock*(rwlock_t *lock);*
  write_unlock*(rwlock_t *lock);*

- Read Locking / Unlocking

  read_lock*(rwlock_t *lock);*
  read_unlock*(rwlock_t *lock);*

LKM: RWLock API (interrupts)

- Between bottom halves
  - Use simple APIs

- Between process context and bottom half

  ***write_lock_bh**(rwlock_t *lock);*
  ***write_unlock_bh**(rwlock_t *lock);*
  ***read_lock_bh**(rwlock_t *lock);*
  ***read_unlock_bh**(rwlock_t *lock);*

  - Soft interrupts on CPU with the lock are disabled

- Between IRQ and bottom halves

  ***write_lock_irq**(rwlock_t * lock);*
  ***write_unlock_irq**(rwlock_t * lock);*
  ***read_lock_irq**(rwlock_t * lock);*
  ***read_unlock_irq**(rwlock_t * lock);*

  - Disable all IRQs before locking, restore all on unlock

  ***write_lock_irqsave**(rwlock_t * lock, unsigned long flags);*
  ***write_unlock_irqrestore**(rwlock_t * lock, unsigned long flags);*
  ***read_lock_irqsave**(rwlock_t * lock, unsigned long flags);*
  ***read_unlock_irqrestore**(rwlock_t * lock, unsigned long flags);*

  - Save IRQ state before locking, restore exactly on unlock

LKM: kthread Exercise #4

- Refer ***mod8*** directory
  - ***mod86.c*** contains code for rwlock protected resource
    - Again, we share an integer b/w 1 write and 2 read threads
    - But we protect using read write locks
    - Compile and load the module on BBB
    - Observe ***dmesg*** output
      - What do you see as far as wthread's prints are concerned?
      - What do you see as far as rthreads' prints are concerned?
    - Change the relative sleep time between read and write
      - Note your observations

LKM: Sequential Lock

- The issue with the Read Write Lock (RWLock)
  - It favours readers
  - Over writers

- Multiple readers
  - Can starve writers
  - Make writers wait for long times

- What if we wanted to favour writers?

- This situation is handled by a different primitive
  - Called **Sequential Lock (SeqLock)**

LKM: SeqLock API (simple)

- Creation

  seqlock_t my_seqlock;
  ***seq_lock_init**(&my_seqlock);*

- Only the Writer takes the SeqLock

  ***write_seqlock**(seqlock_t *lock);*
  ***write_sequnlock**(seqlock_t *lock);*

- Reader does not take a lock
  - Rather it does a sequence begin – which yields an int (seq_no)

  *unsigned int seq_no = **read_seqbegin**(seqlock_t *lock);*
  - Then it retries the sequence – comparing the earlier seq_no

  *int **read_seqretry**(seqlock_t *lock, unsigned int seq_no);*

LKM: kthread Exercise #5

- Refer ***mod8*** directory
  - ***mod87.c*** contains code for seqlock protected resource
    - Again, we share an integer b/w 1 write and 1 read thread
    - But we protect using seq locks
    - Compile and load the module on BBB
    - Observe ***dmesg*** output
      - What do you see as far as wthread's prints are concerned?
      - What do you see as far as rthreads' prints are concerned?
    - Change the relative sleep time between read and write
      - Note your observations

THANK YOU!