

Overview

- This Lecture
 - Debugging & Concurrency in kernel
 - Source: LDD ch4 & ch5, ELDD ch2, ch21

Kernel debugging options

- There are some configuration options for supporting debugging in the kernel
 - CONFIG_DEBUG_KERNEL
 - CONFIG_DEBUG_SLAB/SLUB
 - CONFIG_DEBUG_PAGEALLOC
 - CONFIG_DEBUG_SPINLOCK
 - CONFIG_DEBUG_SPINLOCK_SLEEP
 - CONFIG_DEBUG_INFO
 - CONFIG_MAGIC_SYSRQ
 - CONFIG_DEBUG_STACKOVERFLOW
 - CONFIG_DEBUG_STACK_USAGE

Kernel debugging options (cont.)

- CONFIG_KALLSYMS
- CONFIG_IKCONFIG
- CONFIG_IKCONFIG_PROC
- CONFIG_DEBUG_DRIVER
- CONFIG_INPUT_EVBUG
- CONFIG_PROFILING
-

Debugging by printing

- Use **printk**
- Be aware of eight priority levels
 - KERN_EMERG, KERN_ALERT, KERN_CRIT, KERN_ERR, KERN_WARNING, KERN_NOTICE, KERN_INFO, KERN_DEBUG
- Mind the console level
 - Not all printk messages appear at the console
 - Check `/proc/sys/kernel/printk`
- You may need a separate console for debugging
 - Use `ioctl(TIOCLINUX)` to redirect the console message
- `klogd` is used to read all kernel messages from `/proc/kmsg`

Turn debugging messages on/off

- You may turn on/off debugging message using conditional compilation

```
#undef PDEBUG          /* undef it, just in case */
#ifdef SCULL_DEBUG
# ifdef __KERNEL__
    /* This one if debugging is on, and kernel space */
#   define PDEBUG(fmt, args...) printk( KERN_DEBUG "scull: " fmt,
    ## args)
# else /* This one for user space */
#   define PDEBUG(fmt, args...) fprintf(stderr, fmt, ## args)
# endif
#else
# define PDEBUG(fmt, args...) /* not debugging: nothing */
#endif
#undef PDEBUGG
#define PDEBUGG(fmt, args...) /* nothing: it's a placeholder */
```

Rate limit

- Use `printk_ratelimit` to limit printing rate
if (`printk_ratelimit()`)
`printk(KERN_NOTICE "The printer is still on
fire\n");`
- Print device numbers
 - `int print_dev_t(char *buffer, dev_t dev);`
 - `char *format_dev_t(char *buffer, dev_t dev);`

Debugging by querying

- Use /proc to query important variables
- Create a /proc entry
 - struct proc_dir_entry *proc_create_data(const char *name, mode_t mode, struct proc_dir_entry *base, struct file_operations *proc_fops, void *data);
 - Example
 - proc_create_data("scullmem", 0 /* default mode */, NULL /* parent dir */, &test_proc_fops, NULL /* driver data */);
 - remove_proc_entry("scullmem", NULL /* parent dir */);

Querying with ioctl

- Use undocumented ioctl commands to expose internal status of driver programs.
- More details for ioctl will be addressed later (refer to chapter 6 of LDD)

Debugging by watching

- Watch behavior of user space applications
 - Use `strace`
 - ...

Debugging system faults

- Check oops message
 - Important fields: EIP, stack, call trace
 - Examples
- System hangs
 - Use `schedule` call
 - Use the magic **SysRq** key
 - Use “`echo 1 > /proc/sys/kernel/sysrq`” to enable it
- They are for advanced kernel programmers!

Using debuggers

- Using gdb
 - `gdb /usr/src/linux/vmlinux /proc/kcore`
 - But the changing variables are found the same
 - Use the command `core-file /proc/kcore` to get updated.
 - For modules, the `add-symbol-file` command should be used.
 - `add-symbol-file scull.ko 0xd0832000 -s .bss 0xd0837100 -s .data 0xd0836be0`
- Using kdb and kgdb (need special patches)
- QEMU, Linux Trace Toolkit, and Dynamic Probes

Data race

- Data race condition
 - occurs when a variable is accessed concurrent R/W operations and one of them is write
- Semaphores, mutexes and locks are used to prevent data races
- Linux semaphores
 - `void sema_init(struct semaphore *sem, int val);`
 - `DECLARE_MUTEX(name);`
 - `DECLARE_MUTEX_LOCKED(name);`
 - `Void init_MUTEX(struct semaphore *sem);`
 - `void init_MUTEX_LOCKED(struct semaphore *sem);`

Linux semaphores

- Operations on semaphores
 - void down(struct semaphore *sem);
 - int down_interruptible(struct semaphore *sem);
 - int down_trylock(struct semaphore *sem);
 - void up(struct semaphore *sem);
- Reader/writer semaphores
 - Allow multiple readers access the resource simultaneously.
- Completion
 - A faster constructor for synchronization between processes/threads

Mutex

- Mutex API
 - `DEFINE_MUTEX(name);`
 - `mutex_init(mutex);`
 - `void mutex_lock(struct mutex *lock);`
 - `int mutex_lock_interruptible(struct mutex *lock);`
 - `int mutex_trylock(struct mutex *lock);`
 - `void mutex_unlock(struct mutex *lock);`
 - `int mutex_is_locked(struct mutex *lock);`
 - `void mutex_lock_nested(struct mutex *lock, unsigned int subclass);`
 - `int mutex_lock_interruptible_nested(struct mutex *lock, unsigned int subclass);`
 - `int atomic_dec_and_mutex_lock(atomic_t *cnt, struct mutex *lock);`
 - Why mutex? [http://www.kernel.org/doc/Documentation/mutex-](http://www.kernel.org/doc/Documentation/mutex-design.txt)

Spinlocks

- Spinlock is a busy waiting lock
 - Used to protect quick accesses to resources
- Spinlock API
 - `spinlock_t my_lock = SPIN_LOCK_UNLOCKED;`
 - `void spin_lock_init(spinlock_t *lock);`
 - `void spin_lock(spinlock_t *lock);`
 - `void spin_unlock(spinlock_t *lock);`
- When a spinlock is acquired, the critical section shouldn't lose CPU, meaning the critical section should be atomic
 - Functions such as `copy_to_user` shouldn't be called.

Spinlocks (cont.)

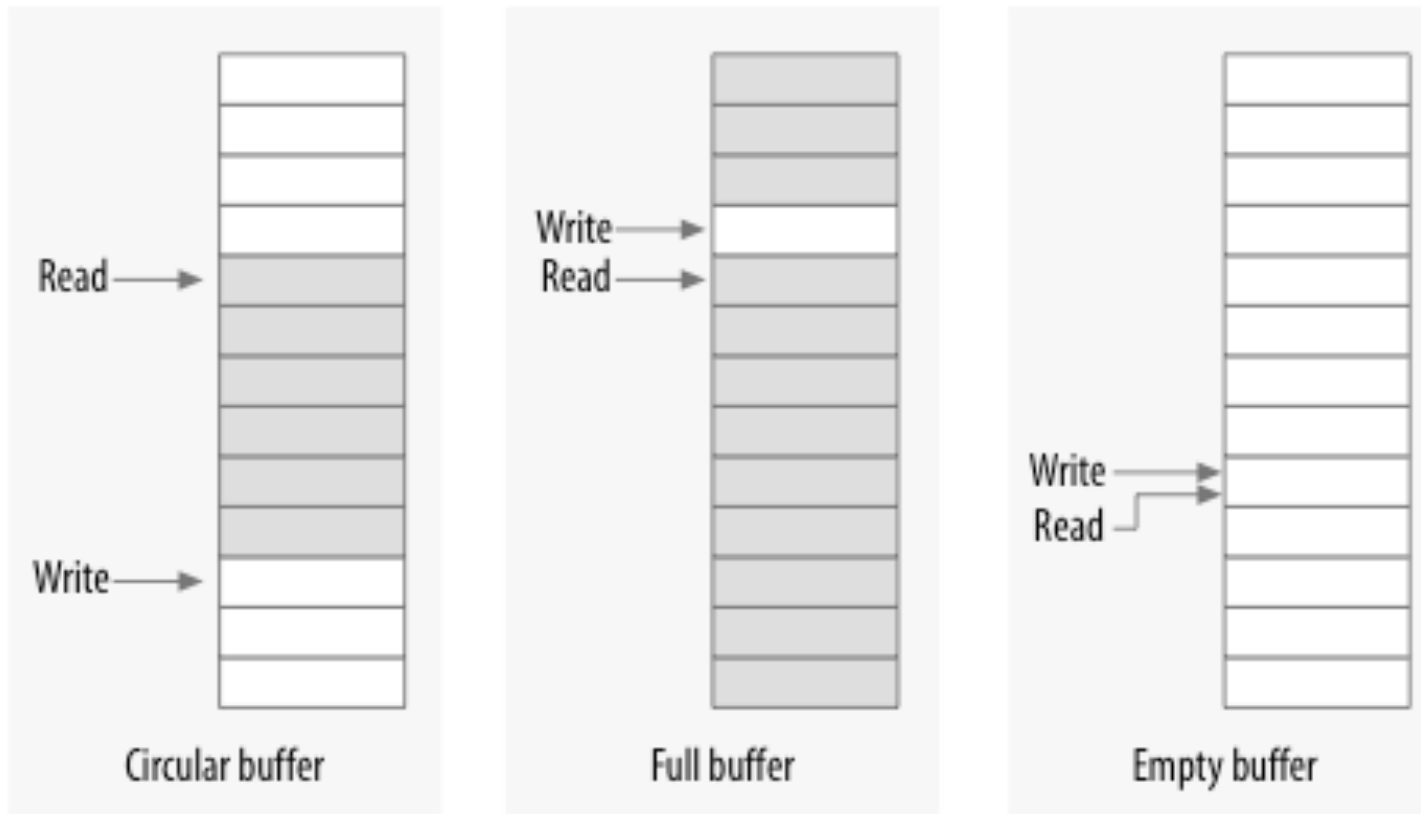
- More spinlock functions
 - void spin_lock(spinlock_t *lock);
 - void spin_lock_irqsave(spinlock_t *lock, unsigned long flags);
 - void spin_lock_irq(spinlock_t *lock);
 - void spin_lock_bh(spinlock_t *lock)
 - void spin_unlock(spinlock_t *lock);
 - void spin_unlock_irqrestore(spinlock_t *lock, unsigned long flags);
 - void spin_unlock_irq(spinlock_t *lock);
 - void spin_unlock_bh(spinlock_t *lock);
- Reader/Writer spinlocks

Tips for locking

- Locking order
 - Always the same order for acquiring multiple locks
- Locking granularity
 - As fine as possible so that parallelism is maximized, but there is a tradeoff between locking overhead and parallelism.
- Don't protect unnecessary code sections
- ...

Lock-free data structure

- Circular buffer



Other lock-free algorithms

- Using CAS (compare-and-swap)
 - For integers n and n' and a memory location a
 - CAS(n, a, n')
 - If the value at address a is n , write the value of n' to address a ; return true
 - Otherwise, return false
- Lock-free algorithm for shared stack

Shared stack

- Example
 - Push an item onto a lock-free stack (using linked list)
- Pseudo-code
 - Push(new){
 - do {
 - T' = Top;
 - new->next = T';
 - ret = CAS(T', &Top, new);
 - } while(!ret);
 - }

Atomic variables

- Use type *atomic_t*
- Functions and macros
 - void atomic_set(atomic_t *v, int i);
 - atomic_t v = ATOMIC_INIT(i);
 - int atomic_read(atomic_t *v);
 - void atomic_add(int i, atomic_t *v);
 - void atomic_sub(int i, atomic_t *v);
 - void atomic_inc(atomic_t *v);
 - void atomic_dec(atomic_t *v);
 - int atomic_inc_and_test(atomic_t *v);
 - int atomic_dec_and_test(atomic_t *v);
 - int atomic_sub_and_test(int i, atomic_t *v);

Other solutions for data race

- Bit operations
 - Linux kernel provides atomic functions for bit-wise operations
- Seqlocks
 - For situations where the resource to be protected is small, simple, and frequently accessed, and where write access is rare but must be fast.
- Read-Copy-Update (RCU)
 - for situations where reads are common and writes are rare