Kernel Critical Sections

Advanced Embedded Linux Development

with **Dan Walkes**

University of Colorado **Boulder**

Learning objectives:

Understand what a Critical Section is.
Understand options for handling Critical Sections in the Linux Kernel.
Introduce Linux Kernel Semaphores

Critical Sections/Atomic

- Goal of Critical Section: Make operations atomic that might not be atomic.
- What do we mean by atomic?
  - Entire operation happens at once as far as threads of execution are concerned (ie in a single assembly instruction).
  - Which are atomic?
    - a |= b;
      - Typically Not
    - a++;
      - Typically Not
    - a = b;
      - May or may not be depending on architecture/variable sizes

Critical Sections Primitives Select

- Critical Section includes code that can be executed by only 1 thread at one time.
- Different locking primitives are used to support critical sections
  - semaphores & mutexes - Work with/use sleep
  - spinlocks - Work in all cases (including when you can't sleep)
- Is it safe to sleep in this Critical Section?
  - Reasons it's not safe to sleep:
    - May be called from interrupt handlers
    - Latency requirements
    - Holding other critical resources
- Will my critical function sleep **or possibly call a function that may sleep**?

Critical Sections Primitives Select

- Are we already performing an operation which could sleep?

```
/* follow the list up to the right position */
dptr = sculld_follow(dev, item);
if (!dptr->data) {
        dptr->data = kmalloc(qset * sizeof(void *), GFP_KERNEL);
        if (!dptr->data)
                goto nomem;
        memset(dptr->data, 0, qset * sizeof(char *));
}
```

Critical Sections Primitives Select

- kmalloc can sleep

Name

kmalloc — allocate memory

Synopsis

```
void * kmalloc ( size_t size,
                 gfp_t flags);
```

Description

kmalloc is the normal method of allocating memory for objects smaller than page size in the kernel.

The *flags* argument may be one of:

GFP_USER - Allocate memory on behalf of user. May sleep.

GFP_KERNEL - Allocate normal kernel ram. May sleep.

Linux Device Drivers 3rd Edition Chapter 5
https://www.kernel.org/doc/htmldocs/kernel-api/API-kmalloc.html

Semaphores

- Correct implementation when:
  - You need mutual exclusion access
  - The process may sleep while the semaphore is held
    - It’s safe for the code to sleep in the critical section.
    - The critical section calls functions which may sleep.

Semaphores

- Semaphore implementation
  - Integer value + functions P() and V()
    - P() when value > 0:
      - Value is decremented, process continues
    - P() when value <= 0:
      - Process blocks until value >0
    - V()
      - Increments value

Semaphores as a Mutex

- Initialize value to 1
- Ensures exactly 1 caller owns the semaphore and associated mutex resource.
- P() = lock() and V() = unlock()
- The way described in the book to use a binary semaphore as a mutex works but is the old way
  - The kernel now we has dedicated mutex operations

Initialize Semaphores/Mutex

- Option 1: Define directly using DEFINE_MUTEX(name_of_mutex)
  - Typically means this is a global

```
#include <asm/irq.h>

static DEFINE MUTEX(ssp_lock);
static LIST_HEAD(ssp_list);

struct ssp_device *pxa_ssp_request(int port, const char *label)
{
        struct ssp_device *ssp = NULL;

        mutex_lock(&ssp_lock);
```

Linux Device Drivers 3rd Edition Chapter 5
https://elixir.bootlin.com/linux/v5.3.6/source/Documentation/locking/mutex-design.rst
https://elixir.bootlin.com/linux/v5.3.6/source/arch/arm/plat-pxa/ssp.c#L35

Initialize Semaphores/Mutex

- Option 2: Include in a structure, then initialize using mutex_init()

```
struct scull_dev *scull_devices;        /* allocated in scull_init_module */
```

```
int scull_init_module(void)
{
```

```
    /* Initialize each device. */
    for (i = 0; i < scull_nr_devs; i++) {
            scull_devices[i].quantum = scull_quantum;
            scull_devices[i].qset = scull_qset;
            mutex_init(&scull_devices[i].lock);
            scull_setup_cdev(&scull_devices[i], i);
    }
```

```
struct scull_dev {
        struct scull_qset *data;  /* Pointer to first quantum set */
        int quantum;              /* the current quantum size */
        int qset;                 /* the current array size */
        unsigned long size;       /* amount of data stored here */
        unsigned int access_key;  /* used by sculluid and scullpriv */
        struct mutex lock;     /* mutual exclusion semaphore     */
        struct cdev cdev;         /* Char device structure            */
};
```

Linux Device Drivers 3rd Edition Chapter 5
https://elixir.bootlin.com/linux/v5.3.6/source/Documentation/locking/mutex-design.rst
https://github.com/cu-ecen-5013/ldd3/blob/master/scull/main.c#L653
https://github.com/cu-ecen-5013/ldd3/blob/master/scull/scull.h#L93

Obtaining Semaphore with P() (mutex lock())

```
void down(struct semaphore *sem);
int down_interruptible(struct semaphore *sem);
int down_trylock(struct semaphore *sem);
```

- down() - blocks until semaphore is available
- down_interruptible() - same but can be interrupted by the user - 0 return means you obtained, non-zero means the call was interrupted.
  - Typically you will want to use interruptible versions.
  - When would a process be interrupted?
    - SIGTERM or SIGKILL

When not to use interruptible

It is worth repeating that driver writers should almost always use the *interruptible* instances of these functions/macros. The noninterruptible version exists for the small number of situations in which signals cannot be dealt with, for example, when waiting for a data page to be retrieved from swap space. Most drivers do not present such special situations.

- Removed in 3rd edition

A process can sleep in two different modes, interruptible and uninterruptible. In an interruptible sleep, the process could be woken up for processing of signals. In an uninterruptible sleep, the process could not be woken up other than by issuing an explicit wake_up. Interruptible sleep is the preferred way of sleeping, unless there is a situation in which signals cannot be handled at all, such as device I/O.

- 2005 article

Linux Device Drivers 2nd Edition Chapter 5
https://www.linuxjournal.com/article/8144

Additional Interruptible Guidance

Given the highly obnoxious nature of unkillable processes, one would think that interruptible sleeps should be used whenever possible. The problem with that idea is that, in many cases, the introduction of interruptible sleeps is likely to lead to application bugs. As recently noted by Alan Cox:

> Unix tradition (and thus almost all applications) believe file store writes to be non signal interruptible. It would not be safe or practical to change that guarantee.

```
> Applications should not assume that write() (or other syscalls) can't
> return EINTR.  Not all filesystems have a bounded-time backing store.

Unix tradition (and thus almost all applications) believe file store
writes to be non signal interruptible. It would not be safe or practical
to change that guarantee.

Alan
```

- 2008 article/guidance
- Outcome: consider whether an application using your driver would be likely to handle signals properly
- File I/O (especially in filesystem drivers) should be non-interruptible to work with existing application assumptions.
- Everything else should be interruptible if possible.

https://lwn.net/Articles/288056/
https://lwn.net/Articles/288062/

Obtaining Semaphore with P() (mutex lock())

```
void down(struct semaphore *sem);
int down_interruptible(struct semaphore *sem);
int down_trylock(struct semaphore *sem);
```

- down_trylock() - never sleeps
  - 0 means you obtained
  - 1 means someone else is holding.
- Thread which completed “down” successfully “holds”, has “taken out” or has “acquired” the semaphore.

Releasing Semaphore with V() (mutex unlock()

```
void up(struct semaphore *sem);
```

- On return of up() your thread no longer holds the semaphore (and/or controlled access to the associated resource).
- 1 call to down should result in exactly 1 call to up
- What about error conditions?
  - Must be released in error paths.