Linux Kernel Module Programming

Anandkumar

July 11, 2021

The *managed* API is recommended:

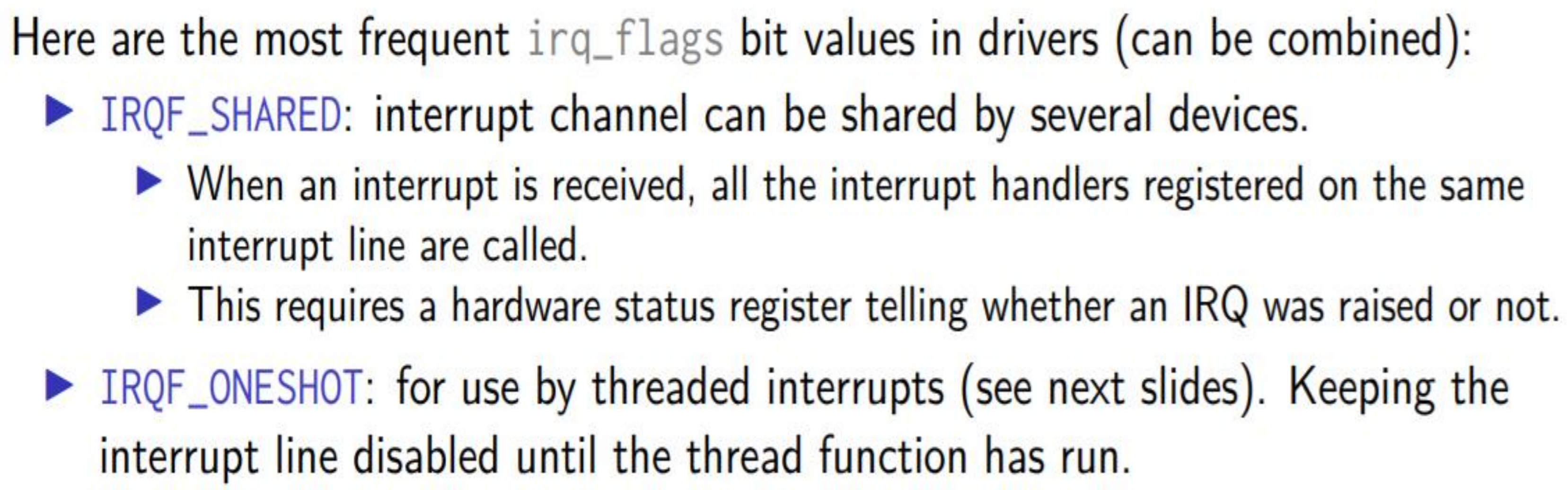
```
int devm_request_irq(struct device *dev, unsigned int irq, irq_handler_t handler,
                     unsigned long irq_flags, const char *devname, void *dev_id);
```

- `device` for automatic freeing at device or module release time.
- `irq` is the requested IRQ channel. For platform devices, use `platform_get_irq()` to retrieve the interrupt number.
- `handler` is a pointer to the IRQ handler function
- `irq_flags` are option masks (see next slide)
- `devname` is the registered name (for `/proc/interrupts`). For platform drivers, good idea to use `pdev->name` which allows to distinguish devices managed by the same driver (example: `44e0b000.i2c`).
- `dev_id` is an opaque pointer. It can typically be used to pass a pointer to a per-device data structure. It cannot be `NULL` as it is used as an identifier for freeing interrupts on a shared line.

```
void devm_free_irq(struct device *dev, unsigned int irq, void *dev_id);
```

- Explicitly release an interrupt handler. Done automatically in normal situations.

Defined in `include/linux/interrupt.h`



Here are the most frequent irq_flags bit values in drivers (can be combined):

- IRQF_SHARED: interrupt channel can be shared by several devices.
  - When an interrupt is received, all the interrupt handlers registered on the same interrupt line are called.
  - This requires a hardware status register telling whether an IRQ was raised or not.
- IRQF_ONESHOT: for use by threaded interrupts (see next slides). Keeping the interrupt line disabled until the thread function has run.

- No guarantee in which address space the system will be in when the interrupt occurs: can't transfer data to and from user space.
- Interrupt handler execution is managed by the CPU, not by the scheduler. Handlers can't run actions that may sleep, because there is nothing to resume their execution. In particular, need to allocate memory with `GFP_ATOMIC`.
- Interrupt handlers are run with all interrupts disabled on the local CPU (see https://lwn.net/Articles/380931). Therefore, they have to complete their job quickly enough, to avoiding blocking interrupts for too long.

```
           CPU0       CPU1       CPU2       CPU3
 17:    1005317          0          0          0 ARMCTRL-level   1 Edge      3f00b880.mailbox
 18:         36          0          0          0 ARMCTRL-level   2 Edge      VCHIQ doorbell
 40:          0          0          0          0 ARMCTRL-level  48 Edge      bcm2708_fb DMA
 42:     427715          0          0          0 ARMCTRL-level  50 Edge      DMA IRQ
 56:  478426356          0          0          0 ARMCTRL-level  64 Edge      dwc_otg, dwc_otg_pcd, dwc_otg_hcd:usb1
 80:     411468          0          0          0 ARMCTRL-level  88 Edge      mmc0
 81:        502          0          0          0 ARMCTRL-level  89 Edge      uart-pl011
161:          0          0          0          0 bcm2836-timer   0 Edge      arch_timer
162:   10963772    6378711   16583353    6406625 bcm2836-timer   1 Edge      arch_timer
165:          0          0          0          0 bcm2836-pmu     9 Edge      arm-pmu
FIQ:                                             usb_fiq
IPI0:         0          0          0          0 CPU wakeup interrupts
IPI1:         0          0          0          0 Timer broadcast interrupts
IPI2:   2625198    4404191    7634127    3993714 Rescheduling interrupts
IPI3:      3140      56405      49483      59648 Function call interrupts
IPI4:         0          0          0          0 CPU stop interrupts
IPI5:   2167923     477097    5350168     412699 IRQ work interrupts
IPI6:         0          0          0          0 completion interrupts
Err:          0
```

Note: interrupt numbers shown on the left-most column are virtual numbers when the Device Tree is used. The physical interrupt numbers can be found in `/sys/kernel/debug/irq/irqs/<nr>` files when `CONFIG_GENERIC_IRQ_DEBUGFS=y`.

- `irqreturn_t foo_interrupt(int irq, void *dev_id)`
  - `irq`, the IRQ number
  - `dev_id`, the per-device pointer that was passed to `devm_request_irq()`
- Return value
  - `IRQ_HANDLED`: recognized and handled interrupt
  - `IRQ_NONE`: used by the kernel to detect spurious interrupts, and disable the interrupt line if none of the interrupt handlers has handled the interrupt.
  - `IRQ_WAKE_THREAD`: handler requests to wake the handler thread (see next slides)

- Acknowledge the interrupt to the device (otherwise no more interrupts will be generated, or the interrupt will keep firing over and over again)
- Read/write data from/to the device
- Wake up any process waiting for such data, typically on a per-device wait queue: `wake_up_interruptible(&device_queue);`

The kernel also supports threaded interrupts:

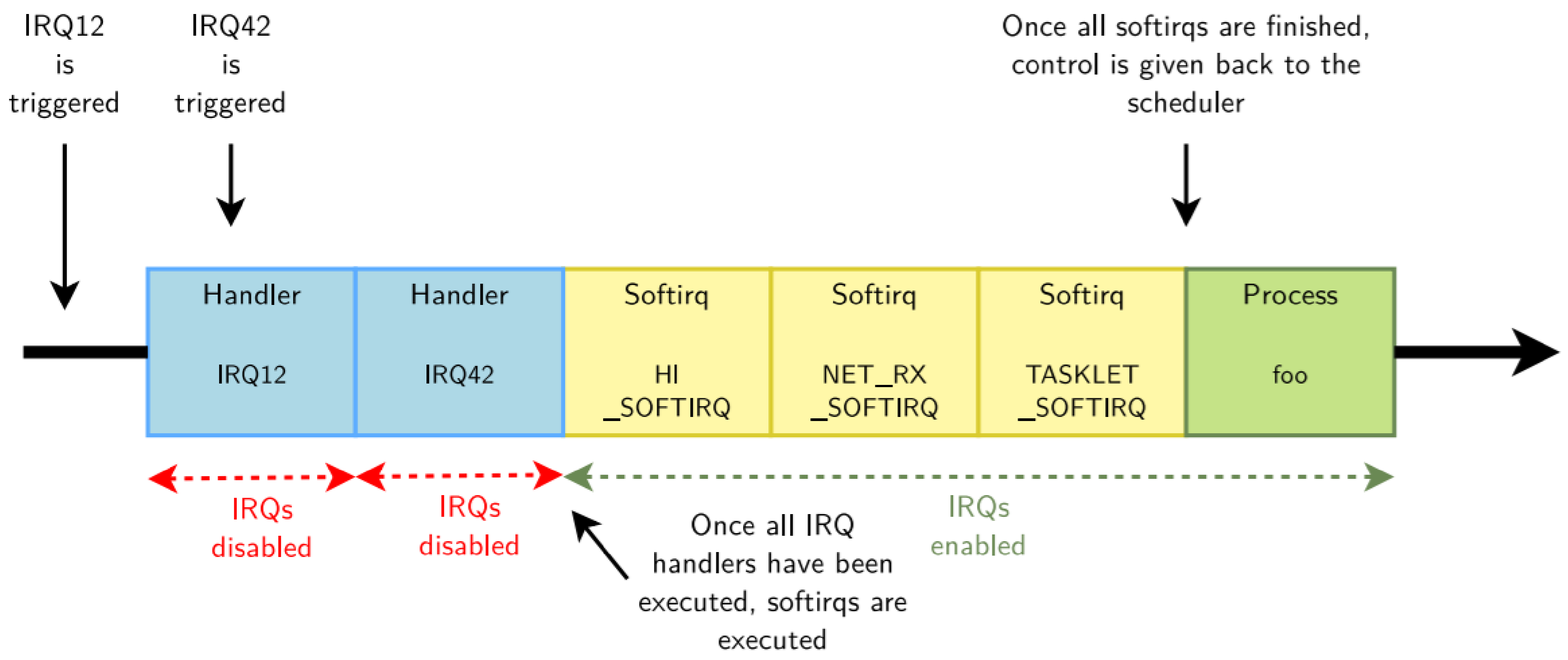
- The interrupt handler is executed inside a thread.
- Allows to block during the interrupt handler, which is often needed for I2C/SPI devices as the interrupt handler needs time to communicate with them.
- Allows to set a priority for the interrupt handler execution, which is useful for real-time usage of Linux

```
int devm_request_threaded_irq(struct device *dev, unsigned int irq,
                              irq_handler_t handler, irq_handler_t thread_fn,
                              unsigned long flags, const char *name,
                              void *dev);
```

- `handler`, "hard IRQ" handler
- `thread_fn`, executed in a thread

Splitting the execution of interrupt handlers in 2 parts

- Top half
  - This is the real interrupt handler, which should complete as quickly as possible since all interrupts are disabled. It takes the data out of the device and if substantial post-processing is needed, schedule a bottom half to handle it.
- Bottom half
  - Is the general Linux name for various mechanisms which allow to postpone the handling of interrupt-related work. Implemented in Linux as softirqs, tasklets or workqueues.


IRQ12 is triggered
IRQ42 is triggered
Once all softirqs are finished, control is given back to the scheduler
Handler IRQ12
Handler IRQ42
Softirq HI _SOFTIRQ
Softirq NET_RX _SOFTIRQ
Softirq TASKLET _SOFTIRQ
Process foo
IRQs disabled
IRQs disabled
Once all IRQ handlers have been executed, softirqs are executed
IRQs enabled

- Softirqs are a form of bottom half processing
- The softirqs handlers are executed with all interrupts enabled, and a given softirq handler can run simultaneously on multiple CPUs
- They are executed once all interrupt handlers have completed, before the kernel resumes scheduling processes, so sleeping is not allowed.
- The number of softirqs is fixed in the system, so softirqs are not directly used by drivers, but by complete kernel subsystems (network, etc.)
- The list of softirqs is defined in `include/linux/interrupt.h`: `HI_SOFTIRQ`, `TIMER_SOFTIRQ`, `NET_TX_SOFTIRQ`, `NET_RX_SOFTIRQ`, `BLOCK_SOFTIRQ`, `IRQ_POLL_SOFTIRQ`, `TASKLET_SOFTIRQ`, `SCHED_SOFTIRQ`, `HRTIMER_SOFTIRQ`, `RCU_SOFTIRQ`
- `HI_SOFTIRQ` and `TASKLET_SOFTIRQ` are used to execute tasklets

- Tasklets are executed within the `HI_SOFTIRQ` and `TASKLET_SOFTIRQ` softirqs. They are executed with all interrupts enabled, but a given tasklet is guaranteed to execute on a single CPU at a time.
- Tasklets are typically created with the `tasklet_init()` function, when your driver manages multiple devices, otherwise statically with `DECLARE_TASKLET()`. A tasklet is simply implemented as a function. Tasklets can easily be used by individual device drivers, as opposed to softirqs.
- The interrupt handler can schedule tasklet execution with:
  - `tasklet_schedule()` to get it executed in `TASKLET_SOFTIRQ`
  - `tasklet_hi_schedule()` to get it executed in `HI_SOFTIRQ` (highest priority)

```
/* The tasklet function */
static void atmel_sha_done_task(unsigned long data)
{
        struct atmel_sha_dev *dd = (struct atmel_sha_dev *)data;
        [...]
}

/* Probe function: registering the tasklet */
static int atmel_sha_probe(struct platform_device *pdev)
{
        struct atmel_sha_dev *sha_dd; /* Per device structure */
        [...]
        platform_set_drvdata(pdev, sha_dd);
        [...]
        tasklet_init(&sha_dd->done_task, atmel_sha_done_task,
                     (unsigned long)sha_dd);
        [...]
        err = devm_request_irq(&pdev->dev, sha_dd->irq, atmel_sha_irq,
                               IRQF_SHARED, "atmel-sha", sha_dd);
        [...]
}
```

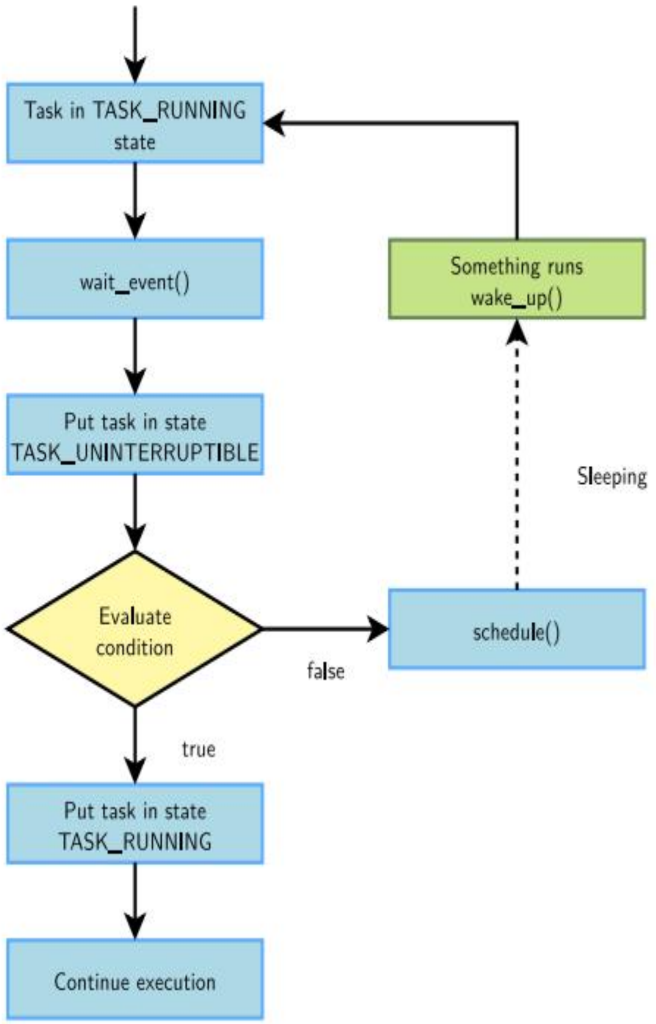
```c
/* Remove function: removing the tasklet */
static int atmel_sha_remove(struct platform_device *pdev)
{
        static struct atmel_sha_dev *sha_dd;
        sha_dd = platform_get_drvdata(pdev);
        [...]
        tasklet_kill(&sha_dd->done_task);
        [...]
}

/* Interrupt handler: triggering execution of the tasklet */
static irqreturn_t atmel_sha_irq(int irq, void *dev_id)
{
        struct atmel_sha_dev *sha_dd = dev_id;
        [...]
        tasklet_schedule(&sha_dd->done_task);
        [...]
}
```

Typically done by interrupt handlers when data sleeping processes are waiting for become available.

- `wake_up(&queue);`
  - Wakes up all processes in the wait queue
- `wake_up_interruptible(&queue);`
  - Wakes up all processes waiting in an interruptible sleep on the given queue

- `wait_event_interruptible()` puts a task in a non-exclusive wait.
  - All non-exclusive tasks are woken up by `wake_up()` / `wake_up_interruptible()`
- `wait_event_interruptible_exclusive()` puts a task in an exclusive wait.
  - `wake_up()` / `wake_up_interruptible()` wakes up all non-exclusive tasks and only one exclusive task
  - `wake_up_all()` / `wake_up_interruptible_all()` wakes up all non-exclusive and all exclusive tasks
- Exclusive sleeps are useful to avoid waking up multiple tasks when only one will be able to “consume” the event.
- Non-exclusive sleeps are useful when the event can “benefit” to multiple tasks.

- `wait_event_interruptible()` puts a task in a non-exclusive wait.
  - All non-exclusive tasks are woken up by `wake_up()` / `wake_up_interruptible()`
- `wait_event_interruptible_exclusive()` puts a task in an exclusive wait.
  - `wake_up()` / `wake_up_interruptible()` wakes up all non-exclusive tasks and only one exclusive task
  - `wake_up_all()` / `wake_up_interruptible_all()` wakes up all non-exclusive and all exclusive tasks
- Exclusive sleeps are useful to avoid waking up multiple tasks when only one will be able to "consume" the event.
- Non-exclusive sleeps are useful when the event can "benefit" to multiple tasks.



The scheduler doesn't keep evaluating the sleeping condition!

- `wait_event(queue, cond);`

  The process is put in the `TASK_UNINTERRUPTIBLE` state.

- `wake_up(&queue);`

  All processes waiting in `queue` are woken up, so they get scheduled later and have the opportunity to evaluate the condition again and go back to sleep if it is not met.

See `include/linux/wait.h` for implementation details.

Wait Queues

```
#include <linux/sched.h>
wait_queue_head_t   my_queue;
init_waitqueue_head( &my_queue );
sleep_on( &my_queue );
wake_up( &my_queue );
```

But can’t unload driver if task stays asleep!

‘interruptible’ wait-queues

Device-driver modules should use:

```
wait_event_interruptible ( &my_queue,condition );
wake_up_interruptible( &my_queue );
```

Then tasks can be awakened by ‘signals’

Timeouts

- Ask the kernel to do it for you

```
#include <linux/wait.h>

long wait_event_timeout(wait_queue_head_t q, condition,
                        long timeout);
long wait_event_interruptible_timeout(wait_queue_head_t q,
                                      condition, long
   timeout);
```

  - Bounded sleep
  - **`timeout`**: in number of **`jiffies`** to wait, signed
  - If the timeout expires, return 0
  - If the call is interrupted, return the remaining jiffies

Timeouts

- Example

```
wait_queue_head_t wait;

init_waitqueue_head(&wait);
wait_event_interruptible_timeout(wait, 0, delay);
```

  - **`condition`** = 0 (no condition to wait for)
  - Execution resumes when
    - Someone calls **`wake_up()`**
    - Timeout expires

How ‘sleep’ works

Our driver defines an instance of a kernel data-structure called a ‘wait queue head’

It will be the ‘anchor’ for a linked list of ‘task_struct’ objects

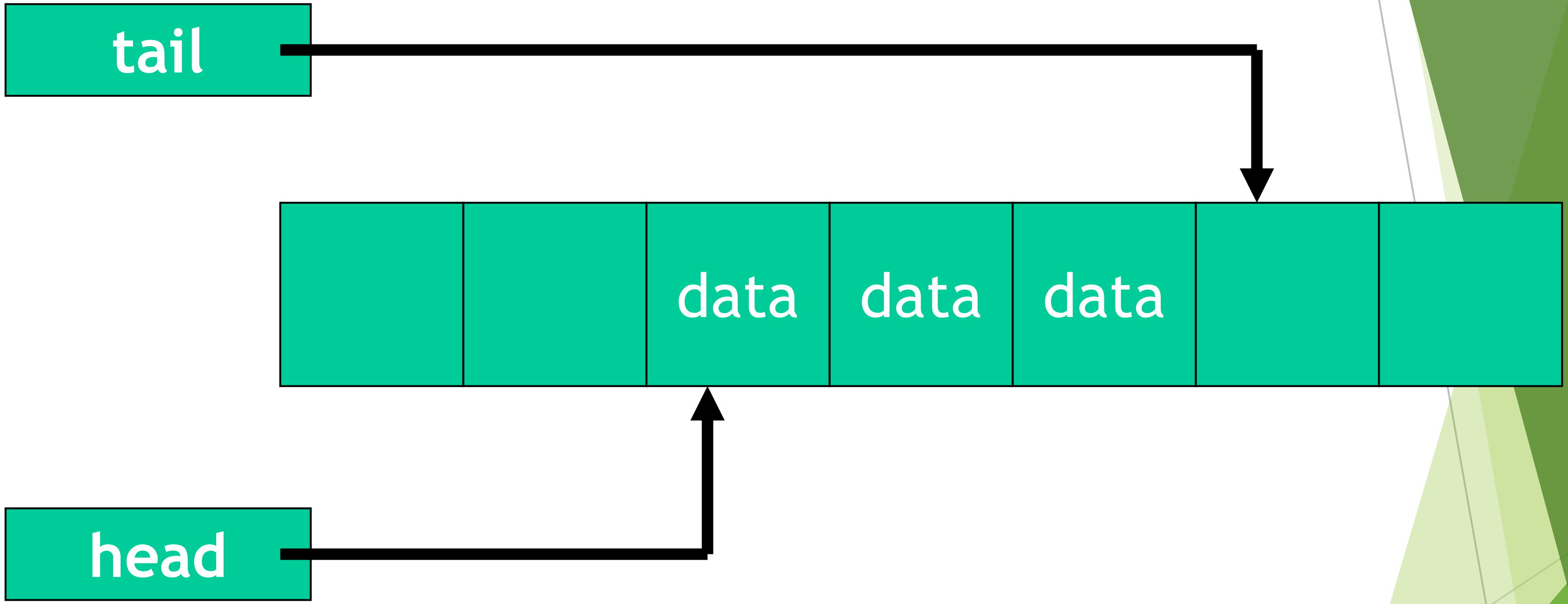
It will initially be an empty-list

If our driver wants to put a task to sleep, then its ‘task_struct’ will be taken off the runqueue and put onto our wait queue

How ‘wake up’ works

- If our driver detects that a task it had put to sleep (because no data-transfer could be done immediately) would now be allowed to proceed, it can execute a ‘wake up’ on its wait queue object
- All the task_struct objects that have been put onto that wait queue will be removed, and will be added to the CPU’s runqueue

How a ring buffer works



Application to a ringbuffer

- A first-in first-out data-structure (FIFO)
- Uses a storage-array of finite length
- Uses two array-indices: ‘head’ and ‘tail’
- Data is added at the current ‘tail’ position
- Data is removed from the ‘head’ position

Ringbuffer (continued)

One array-position is always left unused

Condition head == tail means “empty”

Condition tail == head-1 means “full”

Both ‘head’ and ‘tail’ will “wraparound”

Calculation: next = (next+1)%RINGSIZE;

‘write’ algorithm for ‘wait1.c’

```
while ( ringbuffer_is_full )
        {
        wait_event_interruptible ( &wq, condition !=0 );
        If ( signal_pending( current ) ) return –EINTR;
        }

Insert byte from user-space into ringbuffer;
wake_up_interruptible( &wq );
return 1;
```

‘read’ algorithm for ‘wait1.c’

```
while ( ringbuffer_is_empty )
        {
        wait_event_interruptible &wq,condition!=0 );
        If ( signal_pending( current ) ) return –EINTR;
        }

Remove byte from ringbuffer and store to
    user-space;
wake_up_interruptible( &wq );
return 1;
```

The other driver-methods

We can just omit definitions for other driver system-calls in this example (e.g., ‘open()’, ‘lseek()’, and ‘close()’) because suitable ‘default’ methods are available within the kernel for those cases in this example

Demonstration of ‘wait’

Quick demo: we can use I/O redirection

For demonstrating ‘write’ to /dev/wait1:

```
$ echo “Hello” > /dev/wait1
```

For demonstrating ‘read’ from /dev/wait1:

```
$ cat /dev/wait
```

- Workqueues are a general mechanism for deferring work. It is not limited in usage to handling interrupts. It can typically be used for background work which can be scheduled.
- The function registered as workqueue is executed in a thread, which means:
  - All interrupts are enabled
  - Sleeping is allowed
- A workqueue, usually allocated in a per-device structure, is registered with `INIT_WORK()` and typically triggered with `queue_work()`
- The complete API, in `include/linux/workqueue.h`, provides many other possibilities (creating its own workqueue threads, etc.)
- Example (`drivers/crypto/atmel-i2c`):

```
INIT_WORK(&work_data->work, atmel_i2c_work_handler);
schedule_work(&work_data->work);
```

Measuring Time Lapses

- Kernel keeps track of time via timer interrupts
  - Generated by the timing hardware
  - Programmed at boot time according to **`HZ`**
    - Architecture-dependent value defined in **`<linux/param.h>`**
    - Usually 100 to 1,000 interrupts per second
- Every time a timer interrupt occurs, a kernel counter called **`jiffies`** is incremented
  - Initialized to 0 at system boot

Using the **`jiffies`** Counter

- Must treat **`jiffies`** as read-only
- Example

```
#include <linux/jiffies.h>

unsigned long j, stamp_1, stamp_half, stamp_n;

j = jiffies; /* read the current value */
stamp_1 = j + HZ; /* 1 second in the future */
stamp_half = j + HZ/2; /* half a second */
stamp_n = j + n*HZ/1000; /* n milliseconds */
```

Using the `jiffies` Counter

- Jiffies may wrap - use these macro functions

```
#include <linux/jiffies.h>

/* check if a is after b */
int time_after(unsigned long a, unsigned long b);

/* check if a is before b */
int time_before(unsigned long a, unsigned long b);

/* check if a is after or equal to b */
int time_after_eq(unsigned long a, unsigned long b);

/* check if a is before or equal to b */
int time_before_eq(unsigned long a, unsigned long b);
```

Using the **`jiffies`** Counter

- 32-bit counter wraps around every 50 days
- To exchange time representations, call

```
#include <linux/time.h>

unsigned long timespec_to_jiffies(struct timespec
void jiffies_to_timespec(unsigned long jiffies,
                         struct timespec *value);
unsigned long timeval_to_jiffies(struct timeval *value);
void jiffies_to_timeval(unsigned long jiffies,
                        struct timeval *value);
```

```
struct timespec {
  time_t tv_sec;
  long tv_nsec;
};
```

```
struct timeval {
  time_t tv_sec;
  susecond_t tv_usec;
};
```

- Return number of seconds sinc
  Jan 1, 1970

Knowing the Current Time

- **`jiffies`** represents only the time since the last boot
- To obtain wall-clock time, use

```
#include <linux/time.h>

/* near microsecond resolution */
void do_gettimeofday(struct timeval *tv);

/* based on xtime, near jiffy resolution */
struct timespec current_kernel_time(void);
```

Using the **`jiffies`** Counter

- To access the 64-bit counter **`jiffie_64`** on 32-bit machines, call

```
#include <linux/jiffies.h>
u64 get_jiffies_64(void);
```

Processor-Specific Registers

- To obtain high-resolution timing
  - Need to access the CPU cycle counter register
    - Incremented once per clock cycle
    - Platform-dependent
      - Register may not exist
      - May not be readable from user space
      - May not be writable
        - Resetting this counter discouraged
        - Other users/CPUs might rely on it for synchronizations
      - May be 64-bit or 32-bit wide
        - Need to worry about overflows for 32-bit counters

Processor-Specific Registers

- Timestamp counter (TSC)
  - Introduced with the Pentium
  - 64-bit register that counts CPU clock cycles
  - Readable from both kernel space and user space

Processor-Specific Registers

- To access the counter, include **`<asm/msr.h>`** and use the following marcos

```
/* read into two 32-bit variables */
rdtsc(low32,high32);

/* read low half into a 32-bit variable */
rdtscl(low32);

/* read into a 64-bit long long variable */
rdtscll(var64);
```

- 1-GHz CPU overflows the low half of the counter every 4.2 seconds

Processor-Specific Registers

- To measure the execution of the instruction itself

```
unsigned long ini, end;
rdtscl(ini); rdtscl(end);
printk("time lapse: %li\n", end - ini);
```

- Broken example
  - Need to use `long long`
  - Need to deal with wrap around

Processor-Specific Registers

- Linux offers an architecture-independent function to access the cycle counter

```
#include <linux/tsc.h>
cycles_t get_cycles(void);
```

  - Returns 0 on platforms that have no cycle-counter register

Other Alternatives

- Non-busy-wait alternatives for millisecond or longer delays

```
#include <linux/delay.h>

void msleep(unsigned int millisecs);
unsigned long msleep_interruptible(unsigned int millisecs);
void ssleep(unsigned int seconds);
```

- **`msleep`** and **`ssleep`** are not interruptible
- **`msleeps_interruptible`** returns the remaining milliseconds

Short Delays

```
#include <linux/delay.h>

void ndelay(unsigned long nsecs); /* nanoseconds */
void udelay(unsigned long usecs); /* microseconds */
void mdelay(unsigned long msecs); /* milliseconds */
```

- Perform busy waiting

Kernel Timers

- A *kernel timer* schedules a function to run at a specified time, without blocking the current process
  - E.g., polling a device at regular intervals

Kernel Timers

- The scheduled function is run as a software interrupt
  - Needs to observe constraints imposed on this *interrupt/atomic context*
    - Not associated with any user-level process
      - No access to user space
      - The **`current`** pointer is not meaningful
    - No sleeping or scheduling may be performed
      - No calls to **`schedule()`**, **`wait_event()`**, **`kmalloc(…, GFP_KERNEL)`**, or semaphores

Kernel Timers

- To check if a piece of code is running in special contexts, call
  - `int in_interrupt();`
    - Returns nonzero if the CPU is running in either a hardware or software interrupt context
  - `int in_atomic();`
    - Returns nonzero if the CPU is running in an atomic context
      - Scheduling is not allowed
      - Access to user space is forbidden (can cause scheduling to happen)

Kernel Timers

- Both defined in **`<asm/hardirq.h>`**
- More on kernel timers
  - A task can reregister itself (e.g., polling)
  - Reregistered timer tries to run on the same CPU
  - A potential source of race conditions, even on uniprocessor systems
    - Need to protect data structures accessed by the timer function (via atomic types or spinlocks)

The Timer API

- Basic building blocks

```
#include <linux/timer.h>

struct timer_list {
  /* ... */
  unsigned long expires;
  void (*function) (unsigned long);
  unsigned long data;
};

void add_timer(struct timer_list *timer);
int del_timer(struct timer_list *timer);
```

`jiffies` value when the timer is expected to run

Called with data as argument; pointer cast to **`unsigned long`**

Various Delayed Execution Methods

	Interruptible during the wait	No busy waiting	Good precision for Fine-grained delay	Scheduled task can access user space	Can sleep inside the scheduled task
Busy waiting	Maybe	No	No	Yes	Yes
Yielding the processor	Yes	Maybe	No	Yes	Yes
Timeouts	Maybe	Yes	Yes	Yes	Yes
msleep ssleep	No	Yes	No	Yes	Yes
msleep_interruptible	Yes	Yes	No	Yes	Yes
ndelay udelay mdelay	No	No	Maybe	Yes	Yes
Kernel timers	Yes	Yes	Yes	No	No
Tasklets	Yes	Yes	No	No	No
Workqueues	Yes	Yes	Yes	No	Yes

Kernel Memory Allocator

KMA Subsystem Goals

- Must be fast (this is crucial)
- Should minimize memory waste
- Try to avoid memory fragmentation
- Cooperate with other kernel subsystems

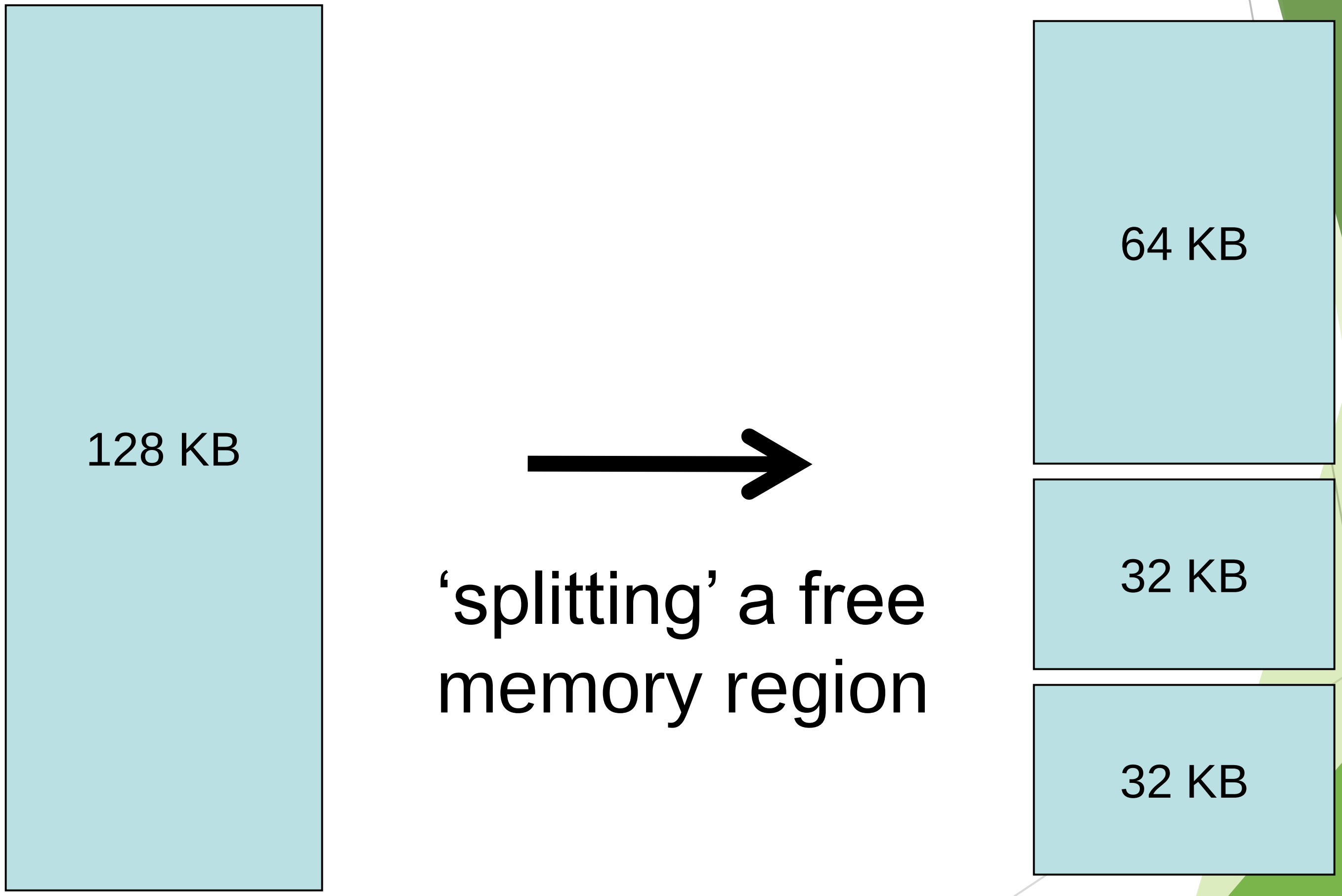
‘Layered’ software structure

At the lowest level, the kernel allocates and frees ‘blocks’ of contiguous pages of phyical memory:

```
struct page *
__alloc_pages( zonelist_t        *zonelist,
               unsigned long   order );
```

(The number of pages in a ‘block’ is a power of 2.)

The zoned buddy allocator



'splitting' a free memory region

block allocation sizes

- Smallest block is 4 KB (i.e., one page)

  order = 0

- Largest block is 128 KB (i.e., 32 pages)

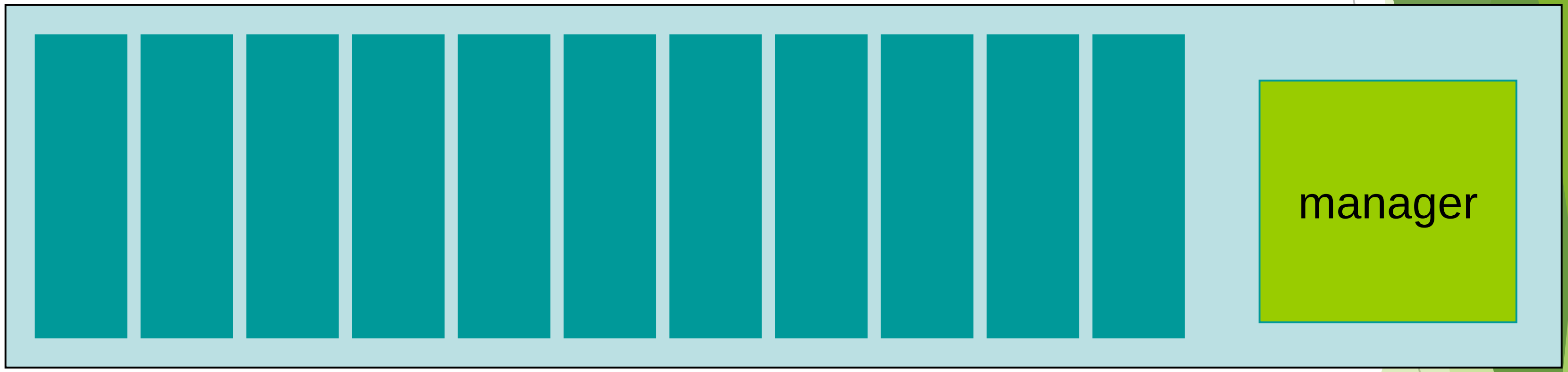
  order = 5

Inefficiency of small requests

- Many requests are for less than a full page
- Wasteful to allocate an entire page!
- So Linux uses a ‘slab allocator’ subsystem

Idea of a ‘slab cache’

kmem_cache_create()



The memory block contains several equal-sized ‘slabs’ (together with a data-structure used to ‘manage’ them)

Allocation Flags

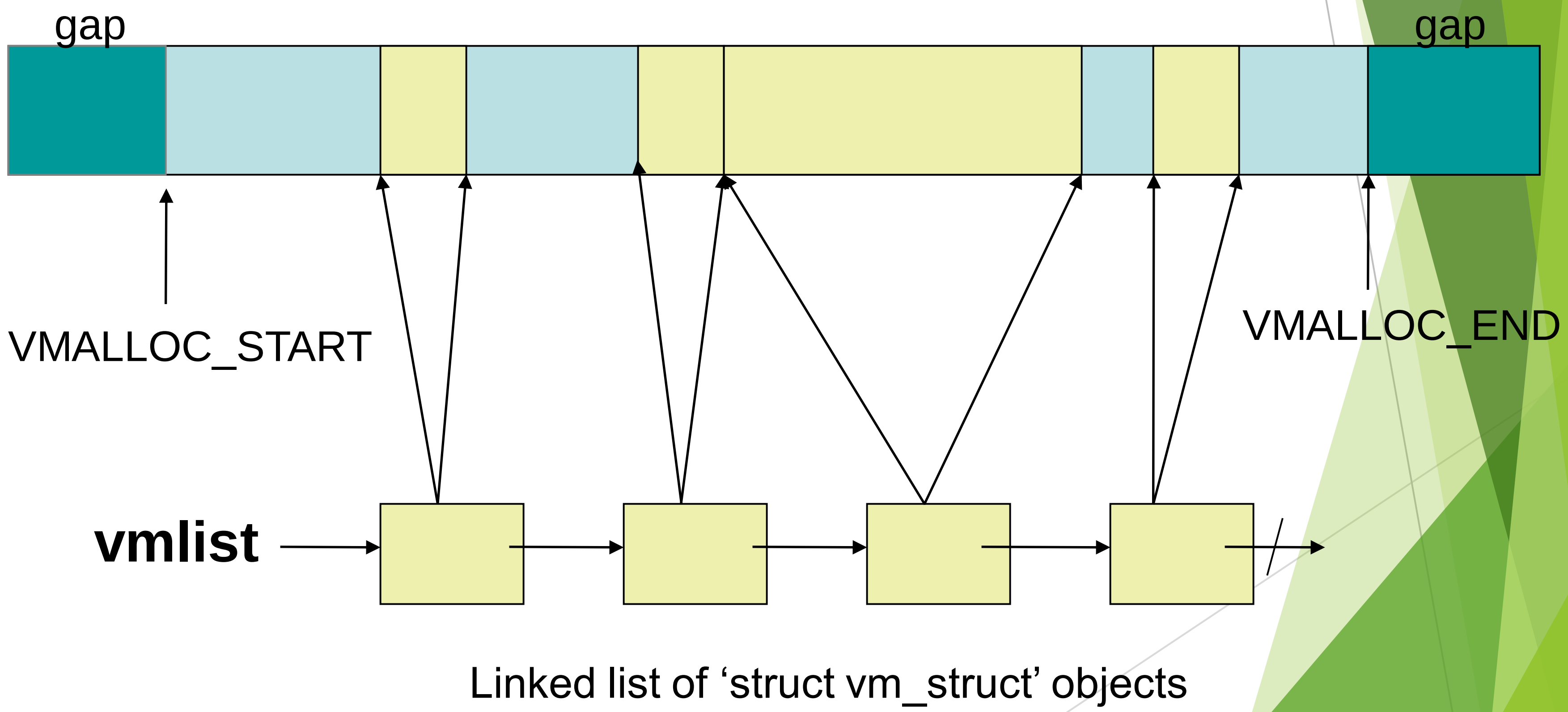
```
__get_free_pages( flags, order );
```

- GFP_KERNEL (might sleep)
- GFP_ATOMIC (will not sleep)
- GFP_USER (low priority)
- __GFP_DMA (below 16MB)
- __GFP_HIGHMEM (from high_memory)

Virtual memory allocations

- Want to allocate a larger-sized block?
- Don’t need physically contiguous pages?
- You can use the ‘vmalloc()’ function

The VMALLOC address-region



‘struct vm_struct’

```
struct vm_struct {
            unsigned long           flags;
            void                          *addr;
            unsigned long           size;
            struct vm_struct  *next;
                  };
```

Defined in <include/linux/vmalloc.h>

Physical Pages

- MMU manages memory in pages
  - 4K on 32-bit
  - 8K on 64-bit
- Every physical page has a `struct page`
  - `flags`: dirty, locked, etc.
  - `count`: usage count, access via `page_count()`
  - `virtual`: address in virtual memory

Zones

Zones represent hardware constraints

- What part of memory can be accessed by DMA?
- Is physical addr space > virtual addr space?

Linux zones on i386 architecture:

Zone	Description	Physical Addr
ZONE_DMA	DMA-able pages	0-16M
ZONE_NORMAL	Normally addressable.	16-896M
ZONE_HIGHMEM	Dynamically mapped pages	>896M

Allocating Pages

`struct page *alloc_pages(mask, order)`

- Allocates 2^{order} contiguous physical pages.
- Returns pointer to 1st page, NULL on error.
- Logical addr: `page_address(struct page *page)`

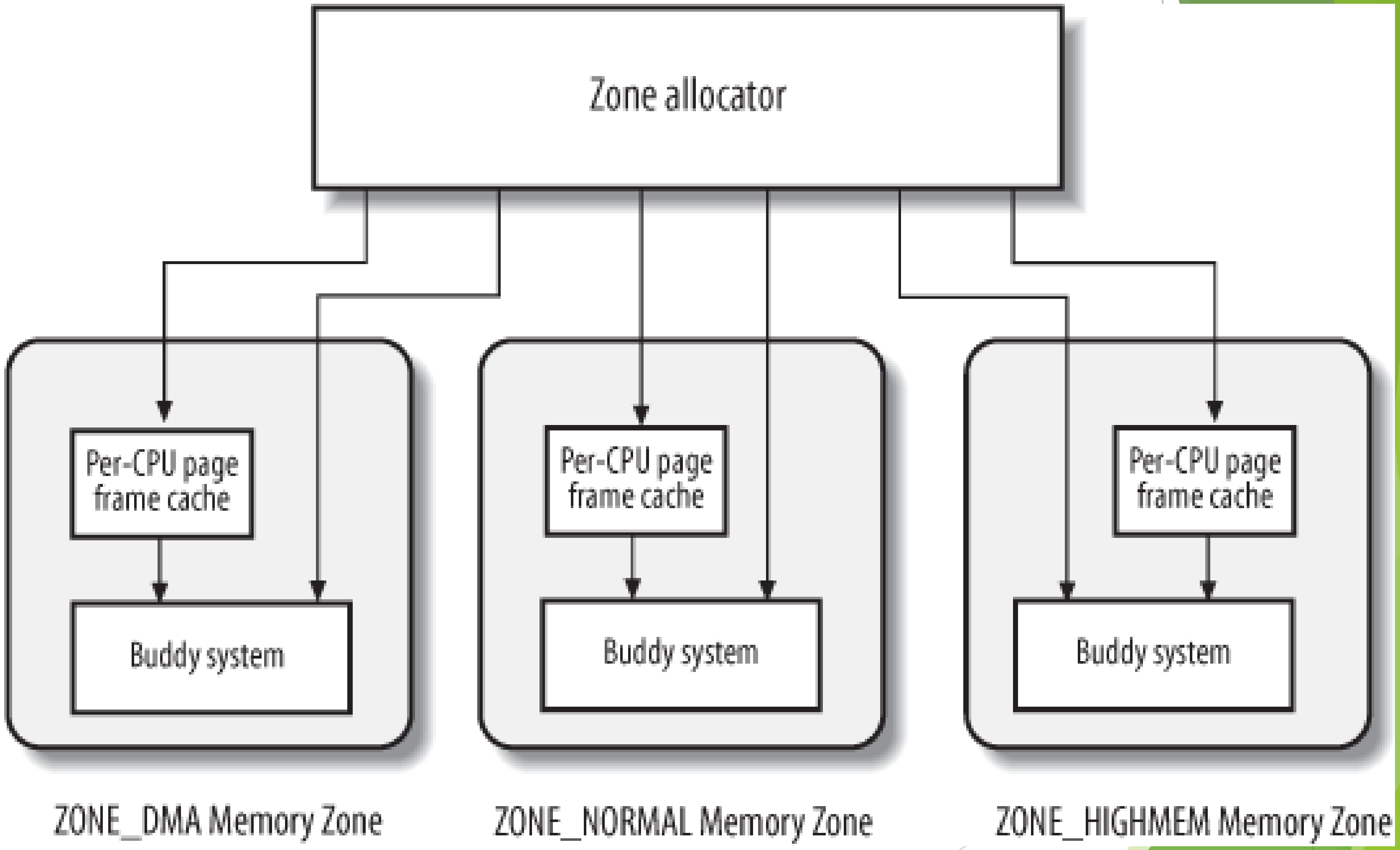
Variants

- `__get_free_pages`: returns logical addr instead
- `alloc_page`: allocate a single page
- `__get_free_page`: get logical addr of single page
- `get_zeroed_page`: like above, but clears page.

External Fragmentation

- The Problem
  - Free page frames scattered throughout mem.
  - How can we allocate large contiguous blocks?
- Solutions
  - Virtually map the blocks to be contiguous.
  - Track contiguous blocks, avoiding breaking up large contiguous blocks if possible.

Zone Allocator



Buddy System

- Maintains 11 lists of free page frames
  - Consist of groups of 2^n pages, n=0..10
- Allocation Algorithm for block of size k
  - Allocate block from list number k.
  - If none available, break a (k+1) block into two k blocks, allocating one, putting one in list k.
- Deallocation Algorithm for size k block
  - Find buddy block of size k.
  - If contiguous buddy, merge + put on (k+1) list.

Per-CPU Page Frame Cache

- Kernel often allocates single pages.
- Two per-CPU caches
  - Hot cache
  - Cold cache

kmalloc()

```
void *kmalloc(size_t size, int flags)
```

- Sizes in bytes, not pages.
- Returns ptr to *at least* size bytes of memory.
- On error, returns NULL.

Example:

```
struct felis *ptr;
ptr = kmalloc(sizeof(struct felis),
  GFP_KERNEL);
if (ptr == NULL)
    /* Handle error */
```

`gfp_mask` Flags

Action Modifiers

- __GFP_WAIT: Allocator can sleep
- __GFP_HIGH: Allocator can access emergency pools.
- __GFP_IO: Allocator can start disk I/O.
- __GFP_FS: Allocator can start filesystem I/O.
- __GFP_REPEAT: Repeat if fails.
- __GFP_NOFAIL: Repeat indefinitely until success.
- __GFP_NORETRY: Allocator will never retry.

Zone Modifiers

- __GFP_DMA
- __GFP_HIGHMEM

`gfp_mask` Type Flags

`GFP_ATOMIC`: Use when cannot sleep.

`GFP_NOIO`: Used in block code.

`GFP_NOFS`: Used in filesystem code.

`GFP_KERNEL`: Normal alloc, may block.

`GFP_USER`: Normal alloc, may block.

`GFP_HIGHUSER`: Highmem, may block.

`GFP_DMA`: DMA zone allocation.

kfree()

```
void kfree(const void *ptr)
```

- Releases mem allocated with `kmalloc()`.
- Must call once for every kmalloc().

Example:

```
char *buf;
buf = kmalloc(BUF_SZ, GFP_KERNEL);
if (buf == NULL)
   /* deal with error */
/* Do something with buf */
kfree(buf);
```

vmalloc()

`void *vmalloc(unsigned long size)`

- Allocates virtually contiguous memory.
- May or may not be physically contiguous.
- Only hardware devs require physical contiguous.

`kmalloc()` **vs.** `vmalloc()`

- `kmalloc()` results in higher performance.
- `vmalloc()` can provide larger allocations.

Slab Allocator

Single cache strategy for kernel objects.

- **Object**: frequently used data struct, e.g. inode
- **Cache**: store for single type of kernel object.
- **Slab**: Container for cached objects.

Older kernels used individual object caches.

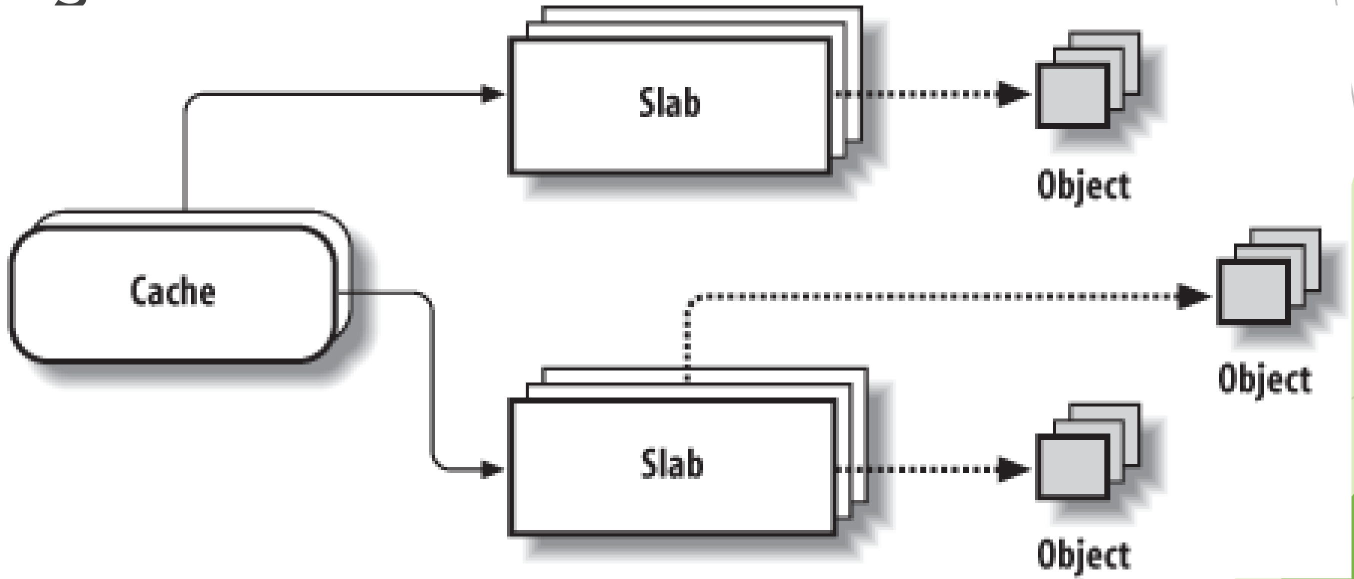
- How could kernel manage when memory low?

Slab Allocator Organization

There is one cache for each object type.

Caches consist of one or more slabs.

Slabs have one or more contiguous memory pages.



Slab States

Full

Has no free objects.

Partial

Some free. Allocation starts with partial slabs.

Empty

Contains no allocated objects.

Slab Algorithm

1. Selects cache for appropriate object type.
   – Minimizes internal fragmentation.
2. Allocate from 1st partial slab in cache.
   – Reduces page allocations/deallocations.
3. If no partial slab, allocate from empty slab.
4. If no empty slab, allocate new slab to cache.

Which allocation method to use?

- Many allocs and deallocs.
  - Slab allocator.
- Need memory in page sizes.
  - alloc_pages()
- Need high memory.
  - alloc_pages().
- Default
  - kmalloc()
- Don’t need contiguous pages.
  - vmalloc()

KThread

- To work with threads you would need the header file linux/kthread.h A thread is created by the call to the function

  struct task_struct *kthread_create(int (*function)(void *data),void *data, const char name[], ...)

- The function takes the following arguments:
  - **function:** The function that the thread has to execute
  - **data** : The "data" to be passed to the function
  - **name:** The name by which the process will be recognised in the kernel.

KThread

- A thread created using the above call does not run but only gets created.
- To run the thread we need to call the function "wake_up_process" passing the thread id that is returned by "kthread_create".
- When the wake_up_process is called the function passed to kthread_create gets executed.
- To stop a thread that is running we need to call the kthread_stop(struct task_struct *threadid) where threadid is the same that is returned by the kthread_create call.

Concurrency and Race Conditions

- *Race condition*: result of uncontrolled access to shared data

```
→ if (!dptr->data[s_pos]) {
    dptr->data[s_pos] = kmalloc(quantum, GFP_KERNEL);
    if (!dptr->data[s_pos]) {
      goto out;
    }
  }
```

- *Race condition*: result of uncontrolled access to shared data

```
if (!dptr->data[s_pos]) {
  dptr->data[s_pos] = kmalloc(quantum, GFP_KERNEL);
  if (!dptr->data[s_pos]) {
    goto out;
  }
}
```

Pitfalls in **`logic`**

- *Race condition*: result of uncontrolled access to shared data

```
if (!dptr->data[s_pos]) {
  dptr->data[s_pos] = kmalloc(quantum, GFP_KERNEL);
  if (!dptr->data[s_pos]) {
    goto out;
  }
}
```

- *Race condition*: result of uncontrolled access to shared data

```
if (!dptr->data[s_pos]) {
  dptr->data[s_pos] = kmalloc(quantum, GFP_KERNEL);
  if (!dptr->data[s_pos]) {
    goto out;
  }
}
```

Memory leak

Concurrency and Its Management

- Sources of concurrency
  - Multiple user-space processes
  - Multiple CPUs
  - Device interrupts
  - Workqueues
  - Tasklets
  - Timers

Concurrency and Its Management

- Some guiding principles
  - Avoid shared resources whenever possible
    - Avoid global variables
  - Apply *locking* and *mutual exclusion* principles to protect shared resources
    - Implications to device drivers
      - No object can be made available to the kernel until it can function properly
      - References to such objects must be tracked for proper removal

Semaphores and Mutexes

- *Atomic operation*: all or nothing from the perspective of other threads
- *Critical section*: code that can be executed by only one thread at a time
  - Not all critical sections are the same
    - Some involve accesses from interrupt handlers
    - Some have latency constraints
    - Some might hold critical resources

Semaphores and Mutexes

- *Semaphores*: an integer combined with P and V operations
  - Call P to enter a critical section
    - If semaphore value > 0, it is decremented
    - If semaphore value == 0, wait
  - Call V to exit a critical section
    - Increments the value of the semaphore
    - Waits up processes that are waiting
  - For mutual exclusions (*mutex*), semaphore values are initialized to 1

The Linux Semaphore Implementation

- **`#include <asm/semaphore.h>`**
- To declare and initialize a semaphore, call

```
void sema_init(struct semaphore *sem, int val);
```

  - Can also call two macros

```
DECLARE_MUTEX(name);  /* initialized to 1 */
DECLARE_MUTEX_LOCKED(name);  /* initialized to 0 */
```

- To initialize a dynamically allocated semaphore, call

```
void init_MUTEX(struct semaphore *sem);
void init_MUTEX_LOCKED(struct semaphore *sem);
```

The Linux Semaphore Implementation

- For the P function, call

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

- **`down`**
  - Just waits for the critical section
    - Until the cows come home
  - A good way to create unkillable process

The Linux Semaphore Implementation

- **`down_interruptible`**
  - Almost always the one to use
  - Allows a user-space process waiting on a semaphore to be interrupted by the user
  - Returns a nonzero value if the operation is interrupted
    - No longer holds the semaphore

The Linux Semaphore Implementation

- **`down_trylock`**
  - Never sleeps
  - Returns immediately with a nonzero value if the semaphore is not available
- For the V function, call

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

  - Remember to call V in error paths

Using Semaphores

```
struct semaphore sem;      /* mutual exclusion semaphore */

init_MUTEX(&sem);

if (down_interruptible(&sem))
  return -ERESTARTSYS;
```

- If **`down_interruptible`** returns nonzero
  - Undo visible changes if any
    - If cannot undo, return **`-EINTR`**
  - Returns **`-ERESTARTSYS`**
  - Higher kernel layers will either restart or return **`-EINTR`**

Using Semaphores

```
up(&sem);
return retval;
```

Reader/Writer Semaphores

- Allow multiple concurrent readers
  - Single writer (for infrequent writes)
  - Too many writers can lead to reader *starvation* (unbounded waiting)
- **`#include <linux/rwsem.h>`**
- Do not follow the return value convention
- Not interruptible

New Mutex Implementation

- Replacing 90% of semaphores
  - Currently architecture-dependent code
- Initialization
  - `DEFINE_MUTEX(name)`
  - `void mutex_init(struct mutex *lock);`
- Various routines
  - `void mutex_lock(struct mutex *lock);`
  - `int mutex_lock_interruptible(struct mutex *lock);`
  - `int mutex_lock_killable(struct mutex *lock);`
  - `void mutex_unlock(struct mutex *lock);`

Completions

- A common pattern in kernel programming
  - Start a new thread
  - Wait for that activity to complete
  - E.g., RAID
- To use completions
  - **`#include <linux/completion.h>`**

Completions

- To create a completion

```
DECLARE_COMPLETION(my_completion);
```

  - Or

```
struct completion my_completion;
init_completion(&my_completion);
```

- To wait for the completion, call

```
void wait_for_completion(struct  completion *c);
void wait_for_completion_interruptible(struct
  completion *c);
void wait_for_completion_timeout(struct  completion
  *c, unsigned long timeout);
```

Completions

- To signal a completion event, call one of the following

```
/* wake up one waiting thread */
void complete(struct completion *c);

/* wake up multiple waiting threads */
/* need to call INIT_COMPLETION(struct completion c)
   to reuse the completion structure */
void complete_all(struct completion *c);
```

Completions

- Example

```
DECLARE_COMPLETION(comp);

ssize_t complete_read(struct file *filp, char __user *buf,
                      size_t count, loff_t *pos) {
  printk(KERN_DEBUG "process %i (%s) going to sleep\n",
         current->pid, current->comm);
  wait_for_completion(&comp);
  printk(KERN_DEBUG "awoken %i (%s)\n", current->pid,
         current->comm);
  return 0; /* EOF */
}
```

Completions

- Example

```
ssize_t complete_write(struct file *filp,
                       const char __user *buf, size_t count,
                       loff_t *pos) {
  printk(KERN_DEBUG
         "process %i (%s) awakening the  readers...\n",
         current->pid, current->comm);
  complete(&comp);
  return count; /* succeed, to avoid retrial */
}
```

Spinlocks

- Used in code that cannot sleep
  - (e.g., interrupt handlers)
  - Better performance than semaphores
- Usually implemented as a single bit
  - If the lock is available, the bit is set and the code continues
  - If the lock is taken, the code enters a tight loop
    - Repeatedly checks the lock until it become available

Spinlocks

- Actual implementation varies for different architectures
- Protect a process from other CPUs and interrupts
    - Usually do nothing on uniprocessor machines
        - Exception: changing the IRQ masking status

Introduction to Spinlock API

- `#include <linux/spinlock.h>`
- To initialize, declare

```
spinlock_t my_lock = SPIN_LOCK_UNLOCKED;
```

  - Or call

```
void spin_lock_init(spinlock_t *lock);
```

- To acquire a lock, call

```
void spin_lock(spinlock_t *lock);
```

  - Spinlock waits are uninterruptible
- To release a lock, call

```
void spin_unlock(spinlock_t *lock);
```

Spinlocks and ***Atomic Context***

- While holding a spinlock, be atomic
  - Do not sleep or relinquish the processor
    - Examples of calls that can sleep
      - Copying data to or from user space
        - User-space page may need to be on disk…
      - Memory allocation
        - Memory might not be available
  - Disable interrupts (on the local CPU) as needed
- Hold spinlocks for the minimum time possible

The Spinlock Functions

- Four functions to acquire a spinlock

```
void spin_lock(spinlock_t *lock);

/* disables interrupts on the local CPU */
void spin_lock_irqsave(spinlock_t *lock,
                       unsigned long flags);

/* used as the only process that disables interrupts */
void spin_lock_irq(spinlock_t *lock);

/* disables software interrupts; leaves hardware
  interrupts enabled (e.g. tasklets)*/
void spin_lock_bh(spinlock_t *lock);
```

The Spinlock Functions

- Four functions to release a spinlock

```
void spin_unlock(spinlock_t *lock);

/* need to use the same flags variable for locking */
/* need to call spin_lock_irqsave and
   spin_unlock_irqrestore in the same function, or your
   code may break on some architectures */
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

- Analogous to the reader/writer semaphores
  - Allow multiple readers to enter a critical section
  - Provide exclusive access for writers
- **`#include <linux/spinlock.h>`**

Reader/Writer Spinlocks

- To declare and initialize, there are two ways

```
/* static way */
rwlock_t my_rwlock = RW_LOCK_UNLOCKED;

/* dynamic way */
rwlock_t my_rwlock;
rwlock_init(&my_rwlock);
```

Reader/Writer Spinlocks

- Similar functions are available

```
void read_lock(rwlock_t *lock);
void read_lock_irqsave(rwlock_t *lock, unsigned long flags);
void read_lock_irq(rwlock_t *lock);
void read_lock_bh(rwlock_t *lock);

void read_unlock(rwlock_t *lock);
void read_unlock_irqrestore(rwlock_t *lock,
                            unsigned long flags);
void read_unlock_irq(rwlock_t *lock);
void read_unlock_bh(rwlock_t *lock);
```

Reader/Writer Spinlocks

- Similar functions are available

```
void write_lock(rwlock_t *lock);
void write_lock_irqsave(rwlock_t *lock, unsigned long flags);
void write_lock_irq(rwlock_t *lock);
void write_lock_bh(rwlock_t *lock);

void write_unlock(rwlock_t *lock);
void write_unlock_irqrestore(rwlock_t *lock,
                             unsigned long flags);
void write_unlock_irq(rwlock_t *lock);
void write_unlock_bh(rwlock_t *lock);
```

Locking Traps

- It is very hard to manage concurrency
- What can possibly go wrong?

Ambiguous Rules

- Shared data structure D, protected by lock L

```
function A() {
  lock(&L);
  /* call function B() that accesses D */
  unlock(&L);
}
```

- If `function B()` calls `lock(&L)`, we have a deadlock

Ambiguous Rules

- Solution
  - Have clear entry points to access data structures
  - Document assumptions about locking

Lock Ordering Rules

```
function A() {
 → lock(&L1);
   lock(&L2);
   /* access D */
   unlock(&L2);
   unlock(&L1)
}
```

```
function B() {
→  lock(&L2);
   lock(&L1);
   /* access D */
   unlock(&L1);
   unlock(&L2)
}
```

- Multiple locks should always be acquired in the same order
- Easier said than done

Lock Ordering Rules

```
function A() {
 → lock(&L1);
   X();
   unlock(&L1)
}
```

```
function B() {
 → lock(&L2);
   Y();
   unlock(&L2)
}
```

```
function X() {
   lock(&L2);
   /* access D */
   unlock(&L2);
}
```

```
function Y() {
   lock(&L1);
   /* access D */
   unlock(&L1);
}
```

Lock Ordering Rules

- Rules of thumb
  - Take a lock that is local to your code before taking a lock belonging to a more central part of the kernel (more contentious)
  - Obtain the semaphore first before taking the spinlock
  - Calling a semaphore (which can sleep) inside a spinlock can lead to deadlocks

Fine- Versus Coarse-Grained Locking

- Coarse-grained locking
  - Poor concurrency
- Fine-grained locking
  - Need to know which one to acquire
    - And which order to acquire
- At the device driver level
  - Start with coarse-grained locking
  - Refine the granularity as contention arises
  - Can enable lockstat to check lock holding time

BKL

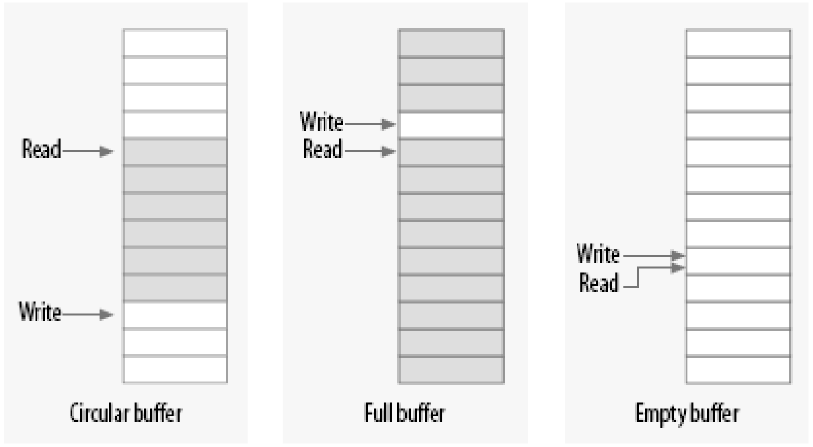
- Kernel used to have “big kernel lock”
  - Giant spinlock introduced in Linux 2.0
  - Only one CPU could be executing locked kernel code at any time
- BKL has been removed
  - https://lwn.net/Articles/384855/
  - https://www.linux.com/learn/tutorials/447301:whats-new-in-linux-2639-ding-dong-the-big-kernel-lock-is-dead

Alternatives to Locking

- Lock-free algorithms
- Atomic variables
- Bit operations
- seqlocks
- Read-copy-update (RCU)

Lock-Free Algorithms

- Circular buffer
  - Producer places data into one end of an array
    - When the end of the array is reached, the producer wraps back
  - Consumer removes data from the other end



Lock-Free Algorithms

- Producer and consumer can access buffer concurrently without race conditions
  - Always store the value before updating the index into the array
  - Need to make sure that producer/consumer indices do not overrun each other
- A generic circular buffer is available in 3.2.36
  - See **`<linux/kfifo.h>`**

Atomic Variables

- If the shared resource is an integer value
  - Locking is too expensive
- The kernel provides atomic types
  - **`atomic_t`** - integer
  - **`atomic64_t`** – long integer
- Both types must be accessed through special functions (See **`<asm/atomic.h>`**)
  - SMP safe

Atomic Variables

- Atomic variables might not be sufficient
  - **`atomic_sub(amount, &account1);`**
  - **`atomic_add(amount, &account2);`**
- A higher level locking must be used

Bit Operations

- Atomic bit operations
  - See **`<asm/bitops.h>`**
  - SMP safe

seqlocks

- Designed to protect small, simple, and frequently accessed resource
  - (e.g., computation that requires multiple consistent values)
- Write access is rare but fast
  - Must obtain an exclusive lock
- Allow readers free access to the resource
  - Check for collisions with writers
  - Retry as needed
  - Not for protecting pointers

Read-Copy-Update (RCU)

- Rarely used in device drivers
- Assumptions
  - Reads are common
  - Writes are rare
  - Resources accessed via pointers
    - All references to those resources held by atomic code

Read-Copy-Update

- Basic idea
  - The writing thread makes a copy
  - Make changes to the copy
  - Switch a few pointers to commit changes
  - Deallocate the old version when all references to the old version are gone