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.

```c
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:

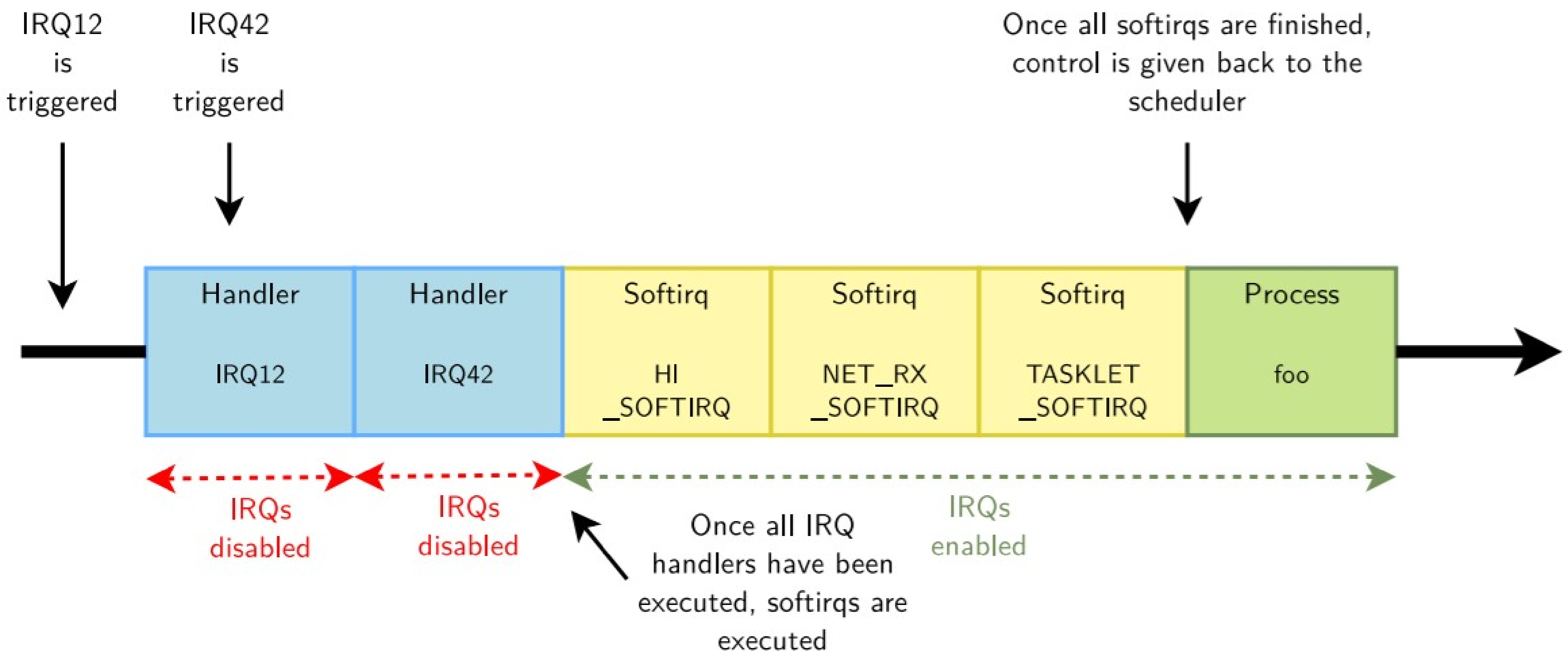
- The interrupt handler is executed inside a thread.
- Allows to block during the interrupt handler, which is often needed for I2C/SPI devices as the interrupt handler needs time to communicate with them.
- Allows to set a priority for the interrupt handler execution, which is useful for real-time usage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

```c
/* 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);
        [...]
}
```

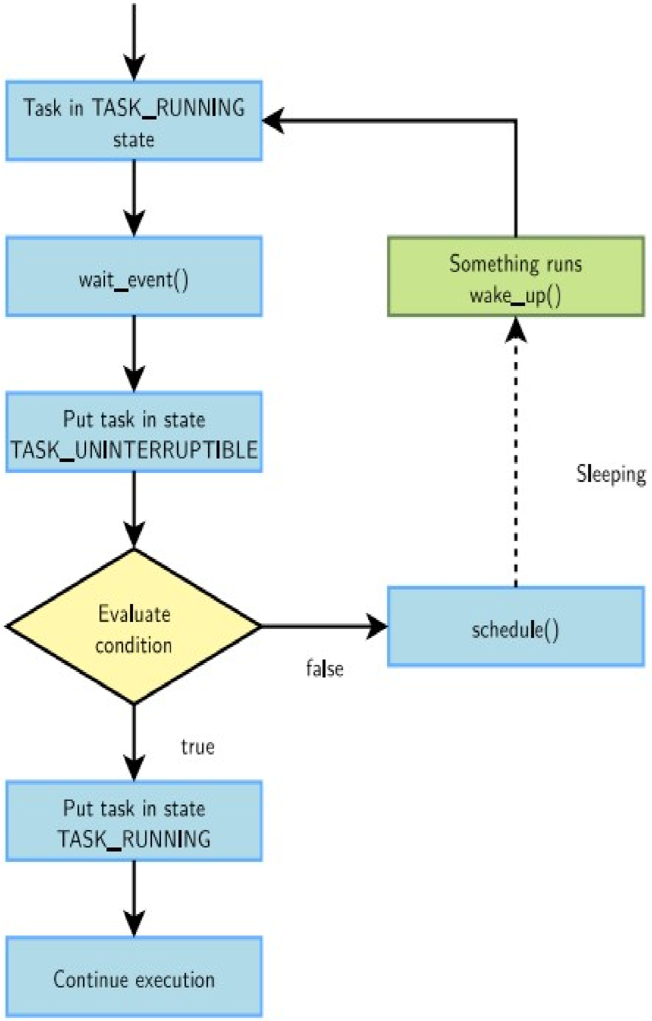
```c
/* Remove function: removing the tasklet */
static int atmel_sha_remove(struct platform_device *pdev)
{
	static struct atmel_sha_dev *sha_dd;
	sha_dd = platform_get_drvdata(pdev);
	[...]
	tasklet_kill(&sha_dd->done_task);
	[...]
}

/* 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

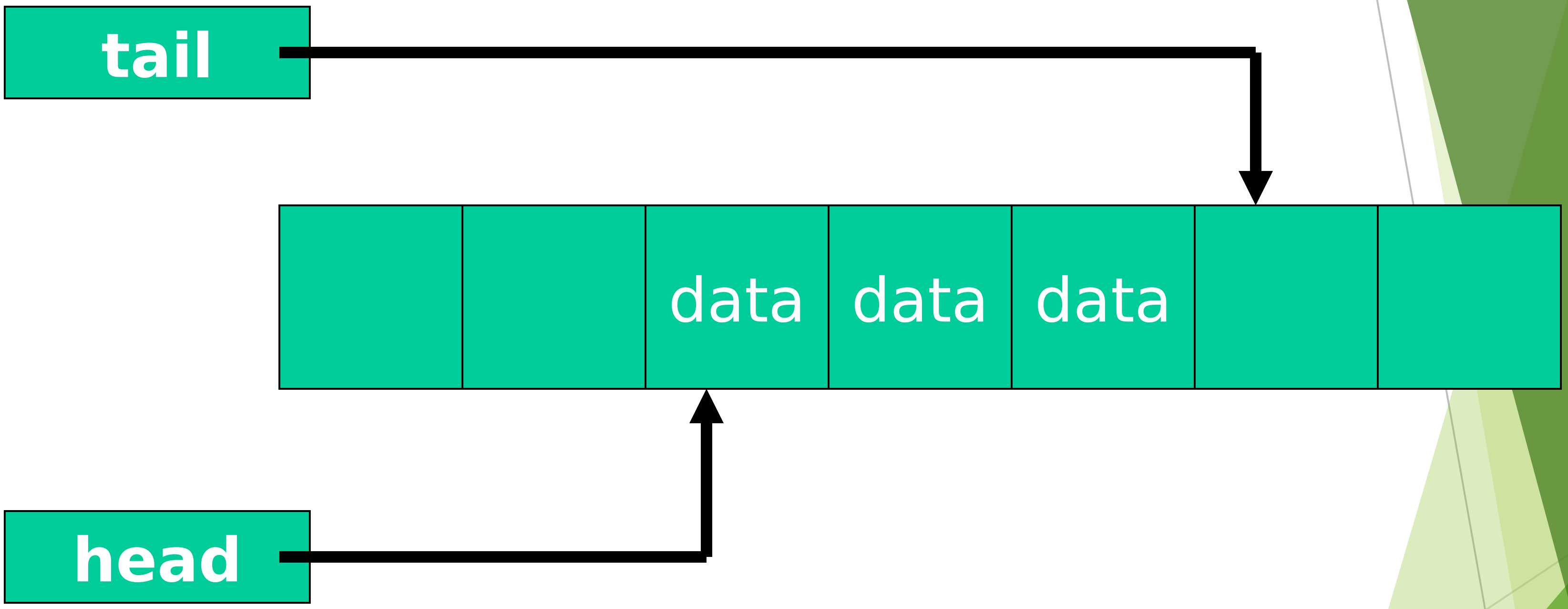
It will initially be an empty-list

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;
};
```

- Return number of seconds sinc

  Jan 1, 1970

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

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

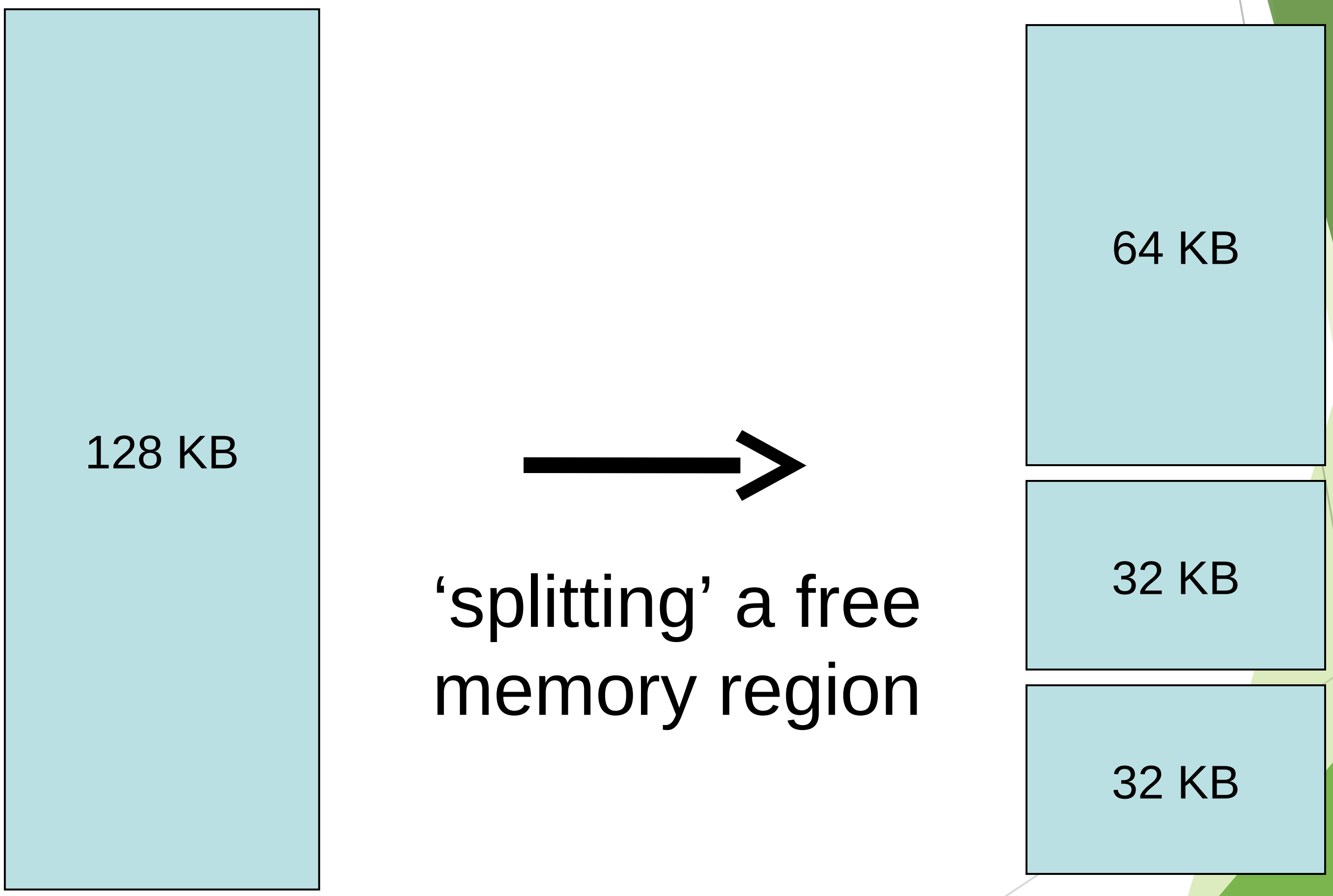
‘Layered’ software structure

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



block allocation sizes

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

  order = 0

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

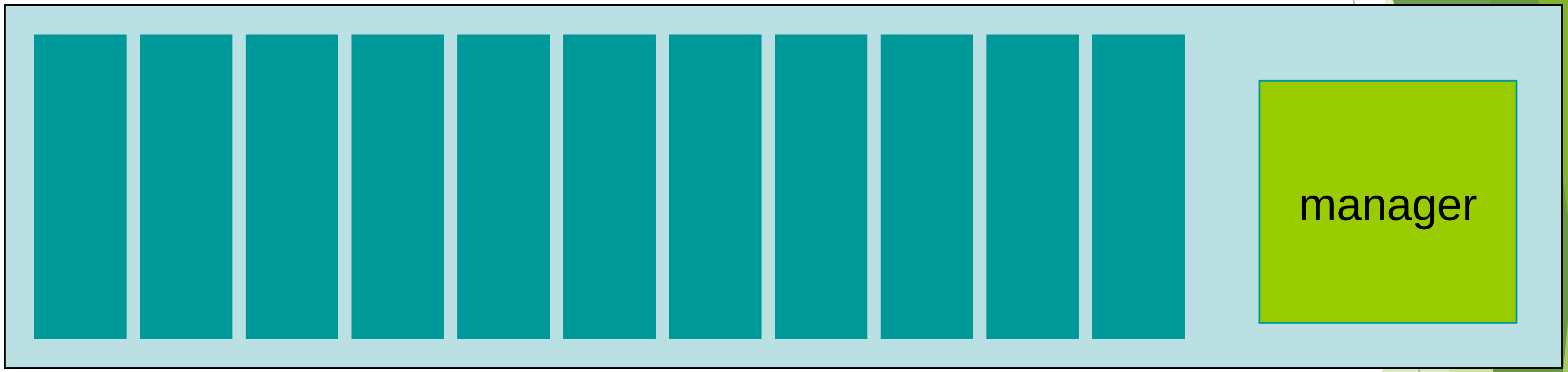
  order = 5

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

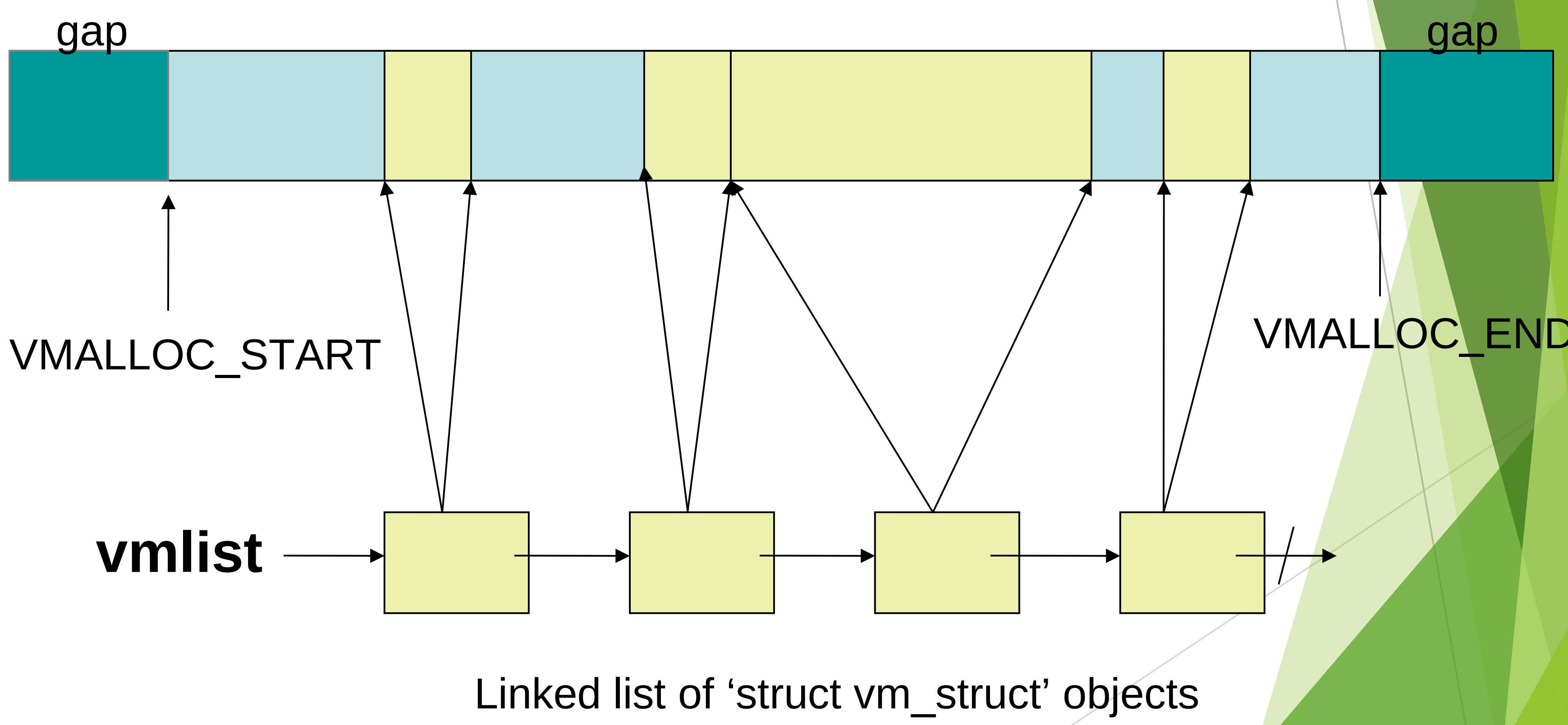
__get_free_pages(flags, order);

- 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

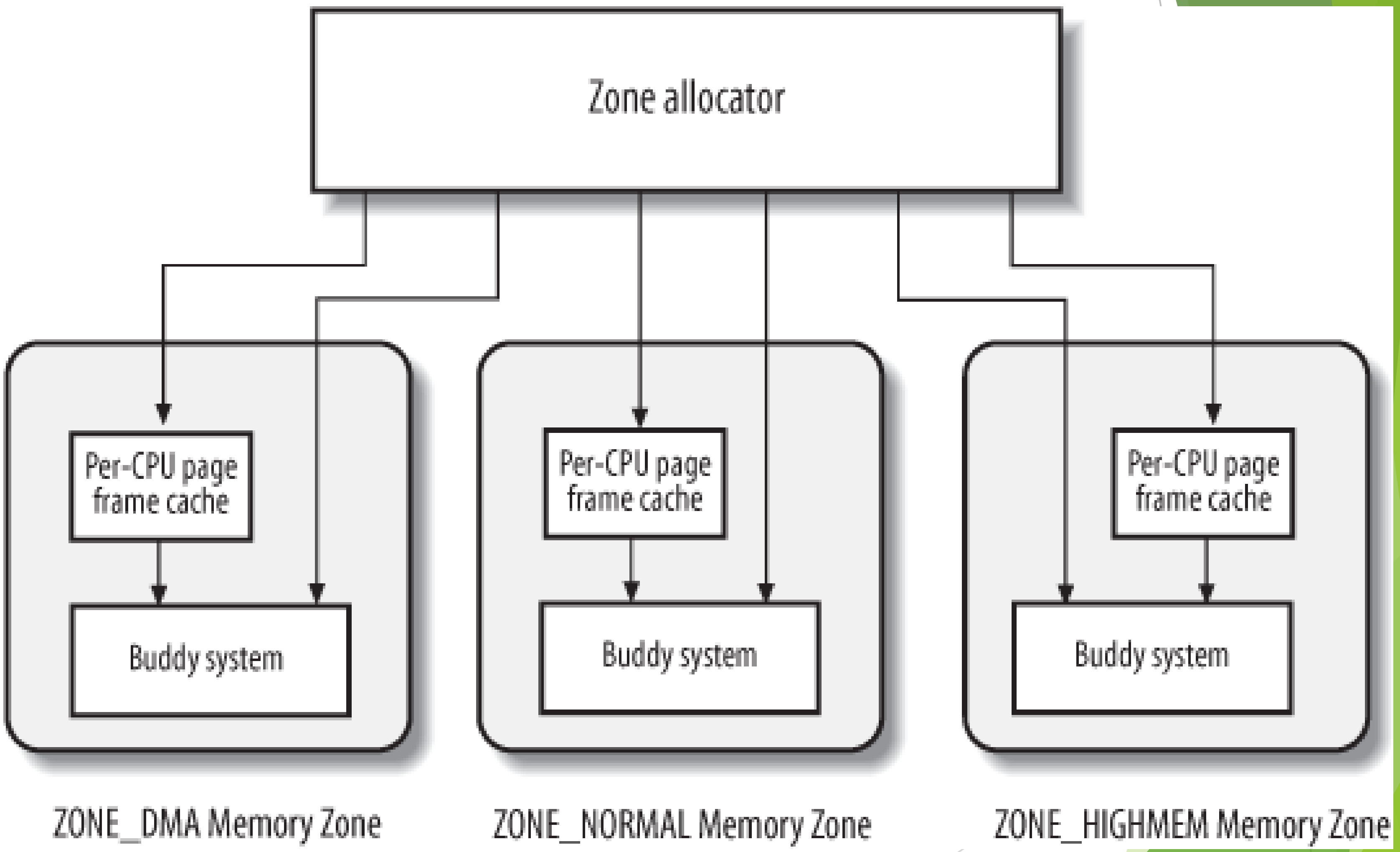
The Problem

- 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.

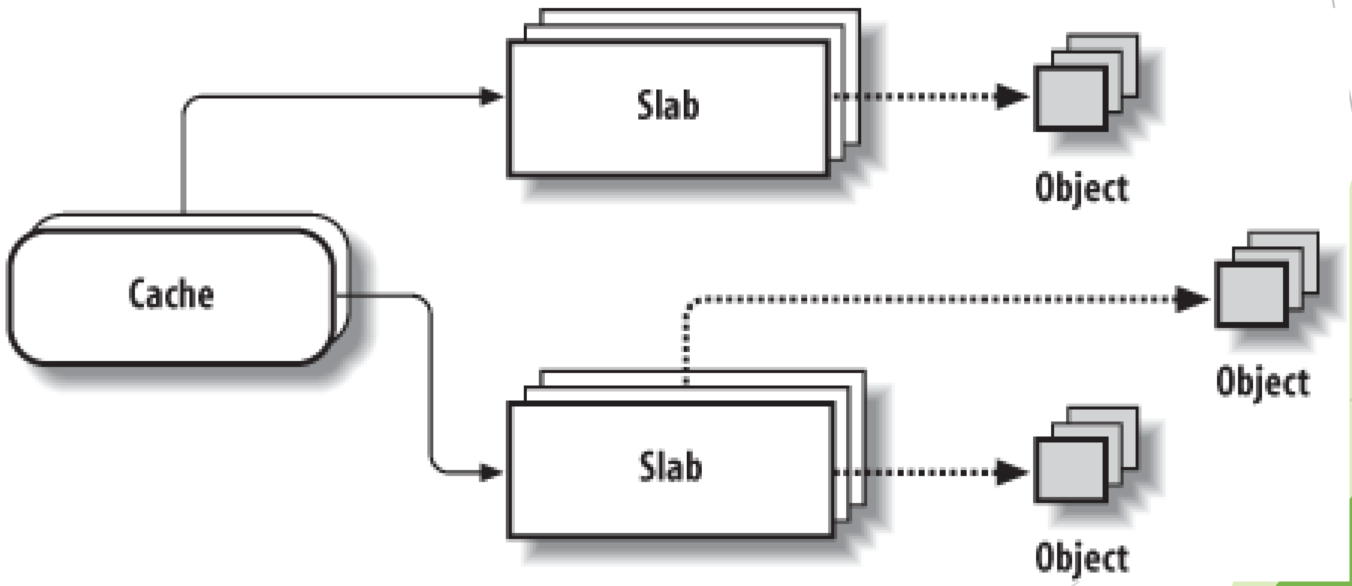
- How could kernel manage when memory low?

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()