Kernel Data Structures II and kernel module

Dongyoon Lee

Summary of last lectures

- Essential kernel data structures
  - list, hash table, red-black tree
- Design patterns of kernel data structures
  - Embedding its pointer structure
  - Tool box rather than a complete solution for generic service
  - Caller locks

Today's agenda

- Memory allocation in the kernel
- More kernel data structures
  - Radix tree
  - XArray
  - Bitmap
- Kernel module

Memory allocation in kernel

- Two types of memory allocation functions are provided
  - `kmalloc(size, gfp_mask)` - `kfree(address)`
  - `vmalloc(size)` - `vfree(address)`
- `gfp_mask` is used to specify
  - which types of pages can be allocated
  - whether the allocator can wait for more memory to be freed
- Frequently used `gfp_mask`
  - `GFP_KERNEL` : a caller *might sleep*
  - `GFP_ATOMIC` : a caller *will not sleep* → higher chance of failure

`kmalloc(size, gfp_mask)`

- Allocate virtually and *physically contiguous* memory
  - where physically contiguous memory is necessary
    - E.g., DMA, memory-mapped IO, performance in accessing
- The maximum allocatable size through one `kmalloc` is limited
  - 4MB on x86 (architecture dependent)

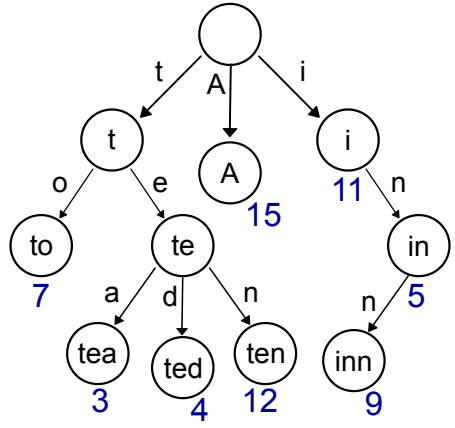
```
#include <linux/slab.h>
void my_function()
{
    char *my_string = (char *)kmalloc(128, GFP_KERNEL);
    my_struct my_struct_ptr = (my_struct *)kmalloc(sizeof(my_struct), GFP_KERNEL);
    /* ... */
    kfree(my_string);
    kfree(my_struct_ptr);
}
```

vmalloc(size)

- Allocate memory that is *virtually contiguous, but not physically* contiguous
- No size limit other than the amount of free RAM
  - Swapping is not supported for kernel memory
- Memory allocator might sleep to get more free memory
- Unit of allocation is a page (4KB)

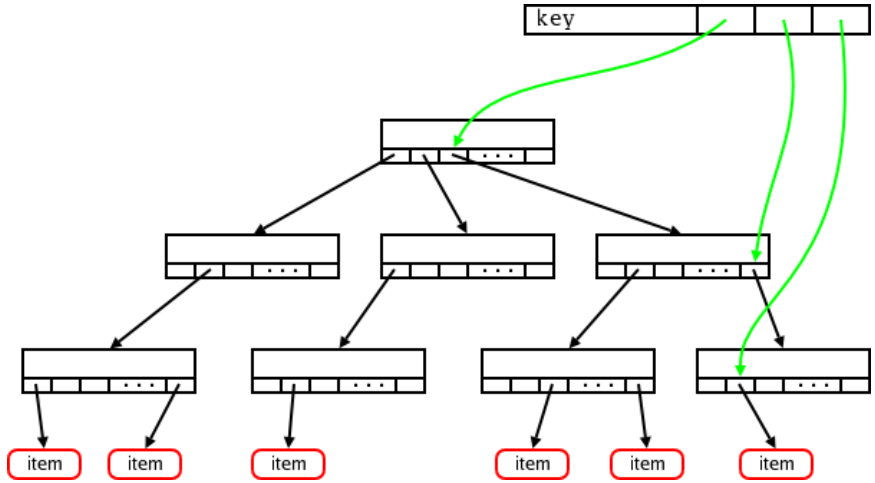
```
#include <linux/slab.h>
void my_function()
{
    char *my_string = (char *)vmalloc(128);
    my_struct my_struct_ptr = (my_struct *)vmalloc(sizeof(my_struct));
    /* ... */
    vfree(my_string);
    vfree(my_struct_ptr);
}
```

Radix tree (or trie, digital tree, prefix tree)



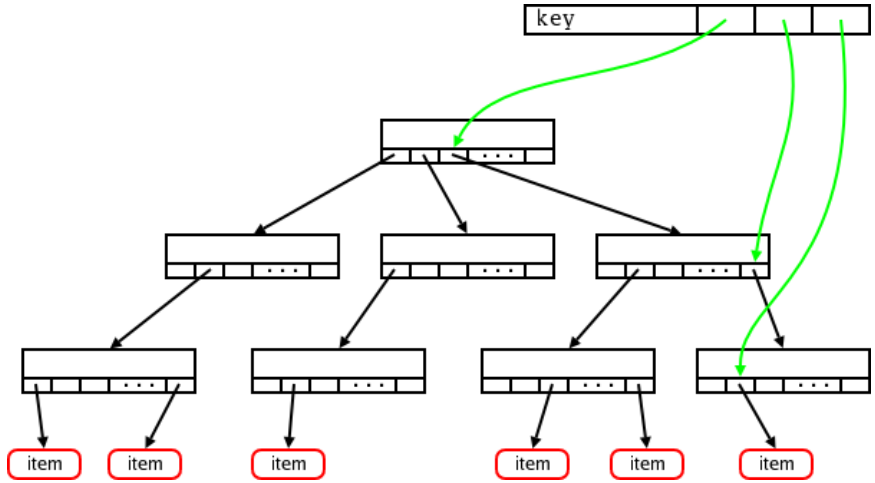
- The key at each node is compared chunk-of-bits by chunk-of-bits
- All descendents of a node have a common prefix
- Values are only associated with leaves
- Source: Wikipedia

Linux radix tree



- Mapping between `unsigned long` and `void *`
- Each node has 64 slots
- Slots are indexed by a 6-bit (2^6=64) portion of the key
- Source: LWN

Linux radix tree



- At leaves, a slot points to an address of data
- At non-leaf nodes, a slot points to another node in a lower layer
- Other metadata is also stored at each node:
  - **tags**, parent pointer, offset in parent, etc

Linux radix tree API

```
/* linux/include/linux/radix-tree.h, linux/lib/radix-tree.c */
#define RADIX_TREE_MAX_TAGS 3
#define RADIX_TREE_MAP_SIZE (1UL << 6)

/* Root of a radix tree */
struct radix_tree_root {
    gfp_t                   gfp_mask; /* used to allocate internal nodes */
    struct radix_tree_node  *rnode;
};

/* Radix tree internal node,
 * which is composed of slot and tag array */
struct radix_tree_node {
    unsigned char           offset;      /* Slot offset in parent */
    struct radix_tree_node *parent;      /* Used when ascending tree */
    void                   *slots[RADIX_TREE_MAP_SIZE];
    unsigned long           tags[RADIX_TREE_MAX_TAGS][RADIX_TREE_TAG_LONGS];
    /* ... */
};
```

- **Q: Is `radix_tree_node` embedded to user data like `list_head`?**

Linux radix tree API

```
/* Root of a radix tree */
struct radix_tree_root {
    gfp_t                   gfp_mask; /* used to allocate internal nodes */
    struct radix_tree_node  *rnode;
};

/* Radix tree internal node,
 * which is composed of slot and tag array */
struct radix_tree_node {
    unsigned char           offset;      /* Slot offset in parent */
    struct radix_tree_node *parent;      /* Used when ascending tree */
    void                   *slots[RADIX_TREE_MAP_SIZE];
    unsigned long           tags[RADIX_TREE_MAX_TAGS][RADIX_TREE_TAG_LONGS];
    /* ... */
};
```

- **Q: Is `radix_tree_node` embedded to user data like `list_head`?**
  - It is dynamically allocated when inserting an item.

Linux radix tree API

```
/* Declare and initialize a radix tree
 * @gfp_mask: how memory allocations are to be performed
 *            (e.g., GFP_KERNEL, GFP_ATOMIC, GFP_FS, etc) */
RADIX_TREE(name, gfp_mask);

/* Initialize a radix tree at runtime */
struct radix_tree_root my_tree;
INIT_RADIX_TREE(my_tree, gfp_mask);

/* Insert an item into the radix tree at position @index.
 * @root:       radix tree root
 * @index:      index key
 * @item:       item to insert */
int radix_tree_insert(struct radix_tree_root *root,
        unsigned long index, void *item);
```

- **Q: What happens if memory allocation fails?**

Linux radix tree API

- When failure to insert an item into a radix tree can be a significant problem, use `radix_tree_preload`

```
/* 1. Allocate sufficient memory (using the given gfp_mask) to guarantee
 * that the next radix tree insertion cannot fail. When successful,
 * it disables preemption so the pre-allocated memory can be used for
 * subsequent radix_tree_insert() operations. */
int radix_tree_preload(gfp_t gfp_mask);

/* 2. Insert an item into the radix tree at position @index.
 * @root:  radix tree root
 * @index: index key
 * @item:  item to insert */
int radix_tree_insert(struct radix_tree_root *root,
        unsigned long index, void *item);

/* 3. Enable preemption again. */
void radix_tree_preload_end(void);
```

Linux radix tree API

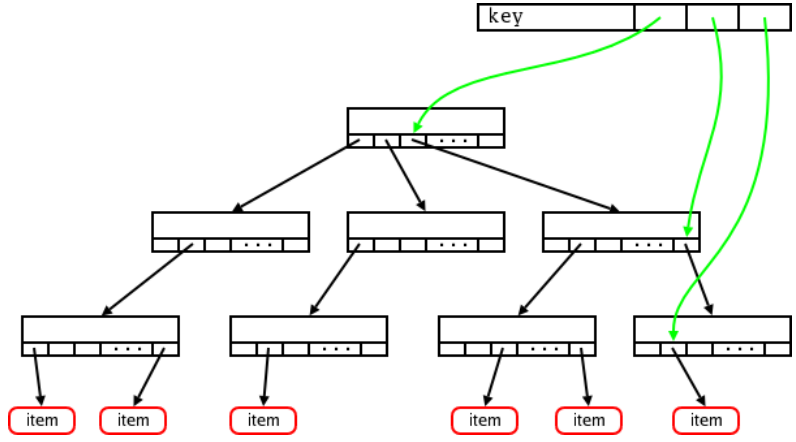
```
/* Remove the entry at @index from the radix tree rooted at @root.
 * @root:  radix tree root
 * @index: index key
 * Return: The deleted entry, or NULL if it was not present. */
void *radix_tree_delete(struct radix_tree_root *root, unsigned long index);

/* radix_tree_lookup    -    perform lookup operation on a radix tree
 * @root:  radix tree root
 * @index: index key
 * Return: data pointer corresponding to the position @index */
void *radix_tree_lookup(const struct radix_tree_root *root, unsigned long index);

/* radix_tree_lookup_slot    -    lookup a slot in a radix tree
 * @root:  radix tree root
 * @index: index key
 * Return: the slot corresponding to the position @index */
void __rcu **radix_tree_lookup_slot(const struct radix_tree_root *root,
                unsigned long index);
```

Linux radix tree API

```
/* radix_tree_gang_lookup - perform multiple lookup on a radix tree
 * @root:        radix tree root
 * @results:     where the results of the lookup are placed
 * @first_index: start the lookup from this key
 * @max_items:   place up to this many items at *results
 *
 * Performs an index-ascending scan of the tree for present items.  Places
 * them at *@results and returns the number of items which were placed at
 * *@results. */
unsigned int
radix_tree_gang_lookup(const struct radix_tree_root *root, void **results,
            unsigned long first_index, unsigned int max_items);
```

Linux radix tree API

- **tags**: specific bits can be set on items in the trees (0, 1, 2)
  - E.g., set the status of memory pages, which are dirty or under writeback



Linux radix tree API

```
/* radix_tree_tag_set - set a tag on a radix tree node
 * @root:       radix tree root
 * @index:      index key
 * @tag:        tag index
 *
 * Set the search tag (which must be < RADIX_TREE_MAX_TAGS)
 * corresponding to @index in the radix tree.  From
 * the root all the way down to the leaf node.
 *
 * Returns the address of the tagged item. */
void *radix_tree_tag_set(struct radix_tree_root *root,
            unsigned long index, unsigned int tag);

/* radix_tree_tag_clear - clear a tag on a radix tree node
 *
 * Clear the search tag corresponding to @index in the radix tree.
 * If this causes the leaf node to have no tags set then clear the tag
 * in the next-to-leaf node, etc.
 *
 * Returns the address of the tagged item on success, else NULL. */
void *radix_tree_tag_clear(struct radix_tree_root *root,
            unsigned long index, unsigned int tag);
```

Linux radix tree API

```
/* radix_tree_tag_get - get a tag on a radix tree node
 * @root:       radix tree root
 * @index:      index key
 * @tag:        tag index (< RADIX_TREE_MAX_TAGS)
 *
 * Return values:
 *  0: tag not present or not set
 *  1: tag set */
int radix_tree_tag_get(const struct radix_tree_root *root,
            unsigned long index, unsigned int tag);

/* radix_tree_tagged - test whether any items in the tree are tagged
 * @root:       radix tree root
 * @tag:        tag to test  */
int radix_tree_tagged(const struct radix_tree_root *root,
            unsigned int tag);
```

Linux radix tree API

```
/* radix_tree_gang_lookup_tag - perform multiple lookup on a radix tree
 *                              based on a tag
 * @root:       radix tree root
 * @results:    where the results of the lookup are placed
 * @first_index:    start the lookup from this key
 * @max_items:  place up to this many items at *results
 * @tag:        the tag index (< RADIX_TREE_MAX_TAGS)
 *
 * Performs an index-ascending scan of the tree for present items which
 * have the tag indexed by @tag set.  Places the items at *@results and
 * returns the number of items which were placed at *@results.
 */
unsigned int
radix_tree_gang_lookup_tag(const struct radix_tree_root *root, void **results,
        unsigned long first_index, unsigned int max_items,
        unsigned int tag);
```

Linux radix tree example

- The most important user is the page cache
  - Every time we look up a page in a file, we consult the radix tree to see if the page is already in the cache
  - Use tags to maintain the status of page (e.g., `PAGECACHE_TAG_DIRTY` or `PAGECACHE_TAG_WRITEBACK`)

Linux radix tree example

```
/* linux/include/linux/fs.h */

/* inode: a metadata of a file */
struct inode {
    umode_t                 i_mode;
    struct super_block     *i_sb;
    struct address_space *i_mapping;
};

/* address_space: a page cache of a file */
struct address_space {
    struct inode            *host;      /* owner: inode, block_device */
    struct radix_tree_root  page_tree;  /* radix tree of all pages
                                         * (i.e., page cache of an inode) */
    spinlock_t              tree_lock;  /* and lock protecting it */
};
```

Linux radix tree example

- Shared memory virtual file system
  - shared memory among process (`shmget()` and `shmat()`)
  - `tmpfs` memory file system

```
/* linux/fs/inode.c */
/* page_tree is initialized at associated address_space is inialized */
void address_space_init_once(struct address_space *mapping)
{
    INIT_RADIX_TREE(&mapping->page_tree, GFP_ATOMIC | __GFP_ACCOUNT);
}

/* linux/mm/shmem.c */
/* Radix operations are performed on page_tree for file system operations */
static int shmem_add_to_page_cache(struct page *page,
    struct address_space *mapping, pgoff_t index, void *expected)
{
    error = radix_tree_insert(&mapping->page_tree, index, page);
}
```

XArray

- A nicer API wrapper for linux radix tree (merged to 4.19)
- An automatically resizing array of pointers indexed by an unsigned long
- Entries may have up to three tag bits (get/set/clear)
- You can iterate over entries
- You can extract a batch of entries
- Embeds a spinlock
- Loads are store-free using RCU

XArray API

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

/** Define an XArray */
DEFINE_XARRAY(array_name);
/* or */
struct xarray array;
xa_init(&array);

/** Storing a value into an XArray is done with: */
void *xa_store(struct xarray *xa, unsigned long index, void *entry,
    gfp_t gfp);

/** An entry can be removed by calling: */
void *xa_erase(struct xarray *xa, unsigned long index);

/** Storing a value only if the current value stored there matches old: */
void *xa_cmpxchg(struct xarray *xa, unsigned long index, void *old,
    void *entry, gfp_t gfp);
```

XArray API

```
/** Fetching a value from an XArray is done with xa_load(): */
void *xa_load(struct xarray *xa, unsigned long index);

/** Up to three single-bit tags can be set on any non-null XArray
entry; they are managed with: */
void xa_set_tag(struct xarray *xa, unsigned long index, xa_tag_t tag);
void xa_clear_tag(struct xarray *xa, unsigned long index, xa_tag_t tag);
bool xa_get_tag(struct xarray *xa, unsigned long index, xa_tag_t tag);

/** Iterate over present entries in an XArray: */
xa_for_each(xa, index, entry) {
    /* Process "entry" */
}

/** Iterate over marked entries in an XArray: */
xa_for_each_marked(xa, index, entry, filter) {
    /* Process "entry" which marked with "filter" */
}
```

Linux XArray example

```
/* linux/include/linux/fs.h */

/* inode: a metadata of a file */
struct inode {
    umode_t               i_mode;
    struct super_block   *i_sb;
    struct address_space *i_mapping;
};

/* address_space: a page cache of a file */
struct address_space {
    struct inode          *host;   /* owner: inode, block_device */
    struct xarray         i_pages; /* XArray of all pages
                                    * (i.e., page cache of an inode) */
};
```

Linux bitmap

- A bit array that consumes one or more `unsigned long`
- Using in many places in kernel
  - a set of online/offline processors for systems which support hot-plug cpu (more about this you can read in the cpumasks part)
  - a set of allocated IRQs during initialization of the Linux kernel

Linux bitmap

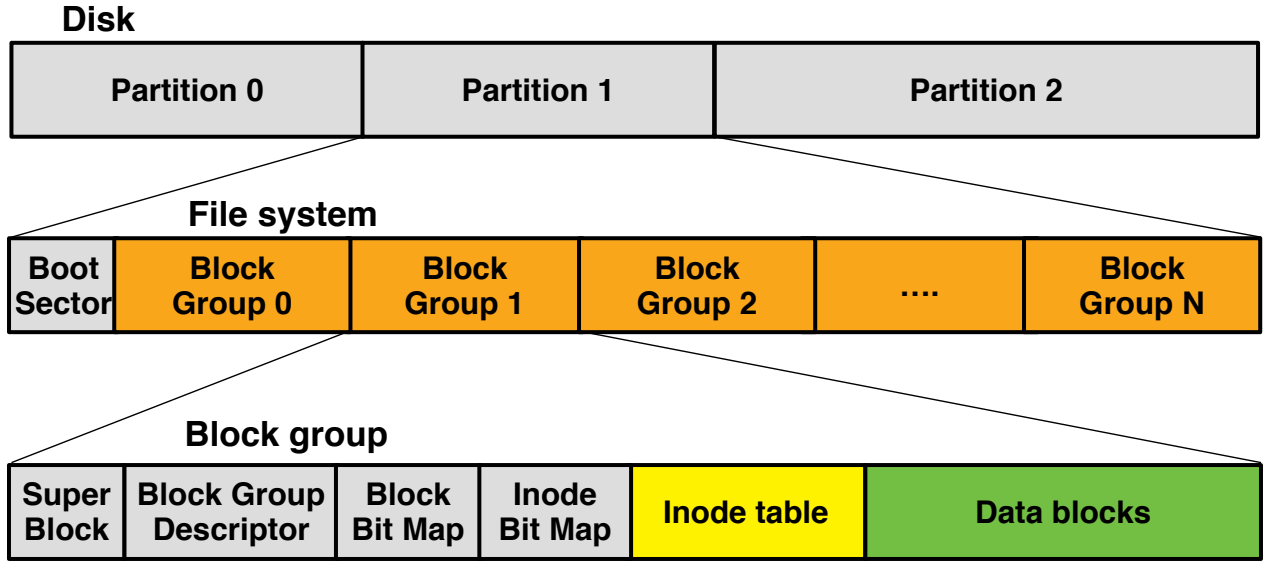
```
/* linux/include/linux/bitmap.h
 * linux/lib/bitmap.c
 * arch/x86/include/asm/bitops.h */

/* Declare an array named 'name' of just enough unsigned longs to
 * contain all bit positions from 0 to 'bits' - 1 */
#define DECLARE_BITMAP(name,bits) \
    unsigned long name[BITS_TO_LONGS(bits)]

/* set_bit - Atomically set a bit in memory
 * @nr: the bit to set
 * @addr: the address to start counting from */
void set_bit(long nr, volatile unsigned long *addr);
void clear_bit(long nr, volatile unsigned long *addr);
void change_bit(long nr, volatile unsigned long *addr);

/* clear nbits from dst */
void bitmap_zero(unsigned long *dst, unsigned int nbits);
void bitmap_fill(unsigned long *dst, unsigned int nbits);
```

Linux bitmap

```
/* find_first_bit - find the first set bit in a memory region
 * @addr: The address to start the search at
 * @size: The maximum number of bits to search
 *
 * Returns the bit number of the first set bit.
 * If no bits are set, returns @size.
 */
unsigned long find_first_bit(const unsigned long *addr, unsigned long size);
unsigned long find_first_zero_bit(const unsigned long *addr, unsigned long size);

/* iterate bitmap */
#define for_each_set_bit(bit, addr, size) \
    for ((bit) = find_first_bit((addr), (size));        \
         (bit) < (size);                    \
         (bit) = find_next_bit((addr), (size), (bit) + 1))
#define for_each_set_bit_from(bit, addr, size)   ...
#define for_each_clear_bit(bit, addr, size)      ...
#define for_each_clear_bit_from(bit, addr, size) ...
```

Linux bitmap example

- Free inode/disk block management in ext2/3/4 file system



Kernel modules

- Modules are pieces of kernel code that can be **dynamically loaded and unloaded at runtime** → No need to reboot
- Appeared in Linux 1.2 (1995)
- Numerous Linux features can be compiled as modules
  - Selection in the configuration .config file

```
# linux/.config
# CONFIG_XEN_PV is not set
CONFIG_KVM_GUEST=y  # built-in to kernel binary executable, vmlinux
CONFIG_XFS_FS=m     # kernel module
```

Benefit of kernel modules

- No reboot → saves a lot of time when developing/debugging
- No need to compile the entire kernel
- Saves memory and CPU time by running on-demand
- No performance difference between module and built-in kernel code
- Help identifying buggy code
  - E.g., identifying a buggy driver compiled as a module by selectively running them

Writing a kernel module

- Module is linked against the entire kernel
- Module can access all of the kernel global symbols
  - `EXPORT_SYMBOL(function or variable name)`
- To avoid namespace pollution and involuntary reuse of variables names
  - Put prefix of your module name to symbols:

    `my_module_func_a()`
  - Use `static` if a symbol is not global
- Kernel symbols list are at `/proc/kallsyms`

Writing a kernel module

```
#include <linux/module.h>   /* Needed by all modules */
#include <linux/kernel.h>   /* KERN_INFO */
#include <linux/init.h>     /* Init and exit macros */

static int answer = 42;

static int __init lkp_init(void)
{
    printk(KERN_INFO "Module loaded ...\n");
    printk(KERN_INFO "The answer is %d ...\n", answer);
    return 0; /* return 0 on success, something else on error */
}

static void __exit lkp_exit(void)
{
    printk(KERN_INFO "Module exiting ...\n");
}

module_init(lkp_init); /* lkp_init() will be called at loading the module */
module_exit(lkp_exit); /* lkp_exit() will be called at unloading the module */

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Dongyoon Lee <dongyoon@cs.stonybrook.edu");
MODULE_DESCRIPTION("Sample kernel module");
```

Building a kernel module

- Source code of a module is out of the kernel source
- Put a Makefile in the module source directory
- After compilation, the compiled module is the file with `.ko` extension

```
# let's assume the module C file is named lkp.c
obj-m := lkp.o
# obj-m += lkp2.o # add multiple files if necessary

CONFIG_MODULE_SIG=n
KDIR := /path/to/kernel/sources/root/directory
# KDIR := /lib/modules/$(shell uname -r)/build
PWD := $(shell pwd)

all: lkp.c # add lkp2.c if necessary
    make -C $(KDIR) M=$(PWD) modules

clean:
    make -C $(KDIR) M=$(PWD) clean
```

Launching a kernel module

- Needs `root` privileges because you are executing kernel code!
- Loading a kernel module with `insmod`
  - `sudo insmod file.ko`
  - Module is loaded and init function is executed
- Note that a module is compiled against a specific kernel version and will not load on another kernel
  - This check can be bypassed through a mechanism called `modversions` but it can be dangerous

Launching a kernel module

- Remove the module with `rmmod`
  - `sudo rmmod file`
  - or `sudo rmmod file.ko`
  - Module exit function is called before unloading
- `make modules_install` from the kernel sources installs the modules in a standard location
  - `/lib/modules/<kernel version>/`

Launching a kernel module

- These installed modules can be loaded using `modprobe`
  - `sudo modprobe <module name>` ← no need to give a file name
- Contrary to `insmod`, `modprobe` handles module dependencies
  - Dependency list generated in `/lib/modules/<kernel version/modules.dep`
- Unload a module using `modprobe -r <module name>`
- Such installed modules can be loaded automatically at boot time by editing `/etc/modules` or the files in `/etc/modprobe.d`

Module parameters ~= command line arguments for module

```
#include <linux/module.h>
/* ... */
static int int_param = 42; /* default value */
static char *string_param = "default value";

module_param(int_param, int, 0);
MODULE_PARM_DESC(int_param, "A sample integer kernel module parameter");
module_param(string_param, charp, S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH);
MODULE_PARM_DESC(string_param, "Another parameter, a string");

static int __init lkp_init(void)
{
    printk(KERN_INFO "Int param: %d\n", int_param);
    printk(KERN_INFO "String param: %s \n", string_param);
        /* ... */
}
```

- sudo insmod lkp.ko int_param=12 string_param=“hello”

Getting module information

- `modinfo [module name | file name]`

```
modinfo my_module.ko
filename:     /tmp/test/my_module.ko
description: Sample kernel module
author:      Dongyoon Lee <dongyoon@cs.stonybrook.edu>
license:     GPL
srcversion:  A5ADE92B1C81DCC4F774A37
depends:
vermagic:    4.8.0-34-generic SMP mod_unload modversions
parm:        int_param:A sample integer kernel module parameter (int)
parm:        string_param:Another parameter, a string (charp)
```

- `lsmod` : list currently running modules

Further readings

- LWN: Trees I: Radix trees
- Bit arrays and bit operations in the Linux kernel
- LWN: The XArray data structure
- XArray API
- The design and implementation of the XArray
- LWN: How to get rid of mmap_sem
- 2.6. Passing Command Line Arguments to a Module
- Building External Modules

Next lecture

- Kernel debugging techniques