1 Linked list and seeking the device, process sleeping

1.1 Linked list

The kernel API provides a standard API for a circular doubly-linked list. The elementary data structure is:

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

struct list_head {
  struct list_head *next;
  struct list_head *prev;
};
```

In a data node struct, list_head will always be the first member of the struct, so that its base address is the same as the data node struct itself. For example:

```
struct my_struct {
  struct list_head list;
  int val;
};
```

The list_head struct must be initialized before use:

```
LIST_HEAD(my_list);
```

or

```
struct my_struct me = {
  .list = LIST_HEAD_INIT(me.list);
  .val = 0;
};
```

or at runtime:

```
struct list_head list;

INIT_LIST_HEAD(&list);
```

Basically, the list initialization macro sets both **prev** and **next** field to point to itself.

Here are some functions provided by the Linux kernel API to manipulate lists:

```
void list_add(struct list_head *new, struct list_head *head);
```

inserts an element pointed to by **new** after the element pointed by **head**

```
void list_add_tail(struct list_head *new, struct list_head *head);
```

inserts an element pointed to by **new** at the end of the list pointed by **head**

```
void list_del(struct list_head *entry);
```

removes the element pointed by **entry** from its list

```
void list_del_init(struct list_head *entry);
```

removes the element pointed by **entry** from its list and reinitialize the list head entry. This function must be used when a node removed from a list will be re-inserted into a different list.

```
int list_empty(struct list_head *head);
```

test whether a list pointed by head is empty

```
void list_splice(struct list_head *list, struct list_head *head);
```

joins two lists together, by inserting the new list list at head in the first list

The kernel API also provides the following helper macros:

```
list_entry(ptr, type, member);
```

returns pointer to the data structure of the **type** indicated in the second argument, from the list of which head is pointed to by the first argument **ptr**, and of which the **member** we desire is given by the third argument

```
list_for_each(struct list_head *pos, struct list_head *head);
```

iterates the list forward. **pos** points to the struct list_head of the current node, and **head** points to the head of the list.

Since **pos** points to the struct list_head of the current node only rather than the current node itself,

```
list_entry(ptr, type, member);
```

must be used to get the pointer to the current node to access elements of the current node.

For example:

```
static LIST_HEAD(my_list);

struct my_entry {
  struct list_head clist;
  int val;
};


void print_list() {
  struct list_head *ptr;
  struct my_entry *curr;

  list_for_each(ptr, &my_list) {
    curr = list_entry(ptr, struct my_entry, clist);
    printk(KERN_INFO "val = %d\n", curr->val);
  }
}
```

```
list_for_each_prev(pos, head);
```

as in list_for_each(), but iterates the list backward.

```
list_for_each_entry(pos, head, member);
```

directly returns the data structure (entry pointer) that encloses the list_head node, so the code can be simplified by eliminating the need to call list_entry() to get the pointer to the current node.

Here is the snippet above using list_for_each_entry():

```
static LIST_HEAD(my_list);

struct my_entry {
  struct list_head clist;
  int val;
};


void foo() {
  struct my_entry *curr;

  list_for_each_entry(curr, &my_list, clist) {
    printk(KERN_INFO "val = %d\n", curr->val);
  }
}
```

```
list_for_each_safe(struct list_head *pos, struct list_head *tmp, struct list_head *head);
```

handles the case where one is removing the list entry. **pos** points to the curr node (serves as the iterator); **tmp** is an extra pointer of type struct list_head * that is used by the macro for temporary storage, and **head** points to the head of your linked list.

Important
This function must be used when the current node **pos** may be deleted

Here is an example of how to use this function to delete all nodes in a list:

```
static LIST_HEAD(my_list);

struct my_entry {
  struct list_head clist;
  int val;
};

void del_list(void) {
  struct list_head *pos;
  struct list_head *tmp;
  struct my_entry *curr;

  list_for_each_safe(pos, tmp, &my_list) {
    curr = list_entry(pos, struct my_entry, clist);
    list_del(&curr->clist);
    printk(KERN_INFO "(exit): val %d removed\n", curr->val);
    kfree(curr);
  }
}
```

```
list_for_each_entry_safe(pos, tmp, head, member)
```

like list_for_each_safe(), except that **pos** now points to the node itself, saving you from calling list_entry(). Here is the above code with list_for_each_entry_safe() used instead:

```
static LIST_HEAD(my_list);

struct my_entry {
  struct list_head clist;
  int val;
};

void del_list(void) {
  struct my_entry *curr;
  struct my_entry *tmp;

  list_for_each_entry_safe(curr, tmp, &my_list, clist) {
    list_del(&curr->clist);
    printk(KERN_INFO "(exit): val %d removed\n", curr->val);
    kfree(curr);
  }
}
```

For more information, please read:

- Writing Linux Device Drivers Chapter 7.7 P. 81

Important
For the following exercises you should use the skeleton code for Lab 3. Download the skeleton code tar ball from the resources page if you have not done so.

1.1.1 Exercise: Basic linked-list

Write a module that sets up a doubly linked circular list of data structures. Each element contains an integer.

For this lab, elements should be allocated using:

void *kmalloc(size_t size, int flags); with flags set to GFP_KERNEL. kmalloc() is normally used to allocate small memory pieces. First, inserts elements to the list, then traverse the list and print the value of each element.

Then at the cleanup function, delete all the elements. Remember to use list_for_each_safe() for this purpose.

Don't forget to free the memory using kfree() that you have allocated using kmalloc()

You only need to touch the functions my_init() and my_exit().

1.1.2 Exercise: Finding tainted kernel modules

All modules loaded on the system are linked in a list that can be accessed from any module:

```
struct module {
  .....
  struct list_head list;
  .....
  char name[MODULE_NAME_LEN];
  .....
  unsigned int taints;
  .....
};
```

Write a module that walks through this list and prints out the value of taints of all modules.

You can begin from THIS_MODULE.

The skeleton code already list the details from THIS_MODULE (your module), then when you run the list_for_each_entry() loop, it will loop through the rest of the loaded modules.

1.1.3 Exercise: Enabling seeking of a device

The lseek() system call allows user-space programs to change the current read/write position in a file (i.e. our device).

Now your task is to implement:

```
loff_t mycdev_lseek(struct file *file, loff_t offset, int whence)
```

struct file contains I/O related metadata of your device, and **file→f_pos** gives the current file position.

First, you need to find out the type of request by looking at **whence**, which can be one of:

SEEK_SET	position file relative to the beginning of the file
SEEK_CUR	position file relative to the current position
SEEK_END	position file relateive to the end of the file

After you calculate the new offset, make sure it is within the address range f the file (i.e. not negative, and not above the maximum ramdisk size). If within range, set file>f_pos to the calculated offset and return the calculated offset, otherwise, return -EINVAL.

Don't forget to register mycdev_lseek() in mycdrv_fops

Finally, you can use the program **seek_test** included in the tarball to test the correct functioning of your driver.

Before you can run **seek_test**, you need to create a node for your module first:

```
$ sudo mknod /dev/mycdrv c 700 0
```

1.2 Suspending process - wait queue

Sometimes a process needs to wait for a condition to be fulfilled (e.g. wait for data to arrive on a peripheral device). In this case, the kernel provides wait queues for putting a task (process) to sleep until the condition it waits for is ready (become true), then it becomes necessary to wake up.

The data structure used by wait queues is of the type wait_queue_head_t. It is declared and initialized by:

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

wait_queue_head_t wq;
init_waitqueue_head(&wq);
```

If the wait queue is statically declared, it can be declared and initialized with the macro:

```
DECLARE_WAIT_QUEUE_HEAD(wq);
```

A wait queue must be initiaized before it can be used.

These macros put a task to sleep:

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

wait_event                (wait_queue_head_t wq, int condition);
wait_event_interruptible  (wait_queue_head_t wq, int condition);
wait_event_killable       (wait_queue_head_t wq, int condition);
```

and these corresponding functions wakes up tasks on the wait queue:

```
void wake_up                (wait_queue_head_t *wq);
void wake_up_interruptible  (wait_queue_head_t *wq);
```

In general, the interruptible version of wait_event should be used, which returns 0 if it returns due to a wakeup call and -ERESTARTSYS if it returns due to a signal arriving. The other versions are used in critical sections of a kernel, where it is unacceptable to be scheduled out while holding a lock. The killable version is only interrupted when the process is killed.

The proper wake up call should be paired with the originating sleep call, except wait_event_killable() shoud be paired with wake_up()

When you use the interruptible forms, you will always have to check upon awakening whether you woke up because a signal has arrived, or there was an explicit wakeup call. The signal_pending(current) macro can be used for this purpose.

This is a simple example of wait queue:

```
#include <linux/sched.h>
DECLARE_WAIT_QUEUE_HEAD(wq);

static int func1( ... ) {
  ...
  printk(KERN_INFO "task%i (%s) going to sleep\n", current->pid, current->comm);
  wait_event_interruptible(wq, atomic_read(&dataready));
  printk(KERN_INFO "awoken %i (%s)\n", current->pid, current->comm);
  if (signal_pending(current))
    return -ERESTARTSYS;
  ...
  atomic_set(&dataready, 0);
}

static int func2( ... ) {
  ...
  printk(KERN_INFO "task %i (%s) awakening sleepers...\n", current->pid, current->comm);
  atomic_set(&dataready, 1);
  wake_up_interruptible(&wq);
  ...
}
```

The sleeping tasks we dealt with are non-exclusive and the API we discussed above will wake up allsleeping tasks. However if more than one task is waiting for exclusive access to a resource (one that only one can use at a time), then this kind of wake up is inefficient and leads to the thundering herd problem, where all sleepers are woken up, but only one of the sleepers can get the resource and the others must be put back to sleep.

To address the thundering herd problem, we need an exclusive sleeping system that only wakes up one task from the wait queue at a time. Exclusive wait can be set up by using this macro:

```
wait_event_interruptible_exclusive (wait_queue_head_t wq, int condition);
```

The usual wake up functions mentioned above can be used, but now only one task will be woken up. If more than one tasks from an exclusive wait queue need to be waken up at one time, these functions can be used:

```
void wake_up_all                  (wait_queue_head_t *wq);
void wake_up_interruptible_all    (wait_queue_head_t *wq);
void wake_up_nr                   (wait_queue_head_t *wq, int nr);
void wake_up_sync_nr              (wait_queue_head_t *wq, int nr);
void wake_up_interruptible_nr     (wait_queue_head_t *wq, int nr);
void wake_up_interruptible_sync_nr (wait_queue_head_t *wq, int nr);
```

The functions with the suffix **_all** will wake up all tasks, but those with the suffix **_nr** will only wake up nr tasks.

1.2.1 Exercise: Using wait queues

Start from the skeleton code in **03/3.2.1/wait_event.c** and get it to use wait queues.

Have the read() function go to sleep until woken by a write() function. (You could also try reversing read and write).

You may want to open up two windows and read in one window and then write in the other window.

Try putting more than one process to sleep, i.e. run your test read program more than once simutaneously before running the write program to awaken them. If you keep track of the pid's you should be able to detect what order processes are woken.

You should use exclusive wait, i.e. wait_event_interruptible_exclusive(), and any global variables used in the logical condition should be atomic.