kthreads

Kernel Modules

- Kernel modules are event driven
  - They respond to system calls / procfs I/O
- How do you get them to handle multiple tasks at once
  - Service new requests
  - Schedule an elevator

kthreads

- Multi-threading technique done in the kernel
- Multiple execution points working on the same process at the same time
  - Assuming multi-core
  - For single-core its perceived to be at the same time
- Similar to user level pthreads
  - One or more pthreads will map to a single kthread

kthread_run

- `#include <linux/kthread.h>`
- `kthread_run(threadfn, data, namefmt, …)`
- Creates a new thread and tells it to run
  - `Threadfn` is the function name to run
  - `Data` is a pointer to the function arguments
  - `Namefmt` is the name of the thread (in `ps`)
    - Specified in a printf formatting string
- Returns a task_struct

kthread_stop

- int kthread_stop(struct task_struct *kthread);
- Tells `kthread` to stop
  - Sets `kthread->kthread_should_stop` to true
  - Wakes the thread
  - Waits for the thread to exit
- Returns the result of the thread function

Scheduling

- You need to make sure to block kthread when not doing anything
- Otherwise it will continue to run and eat resources with nothing to do
- A couple of common ways
  - schedule()
  - ssleep()
- Can use these or others
  - Look up the header files and definitions of these functions in lxr as a starting place

schedule

- #include <linux/sched.h>
- void schedule(void)
- Blocks the kthread for a preset interval

ssleep

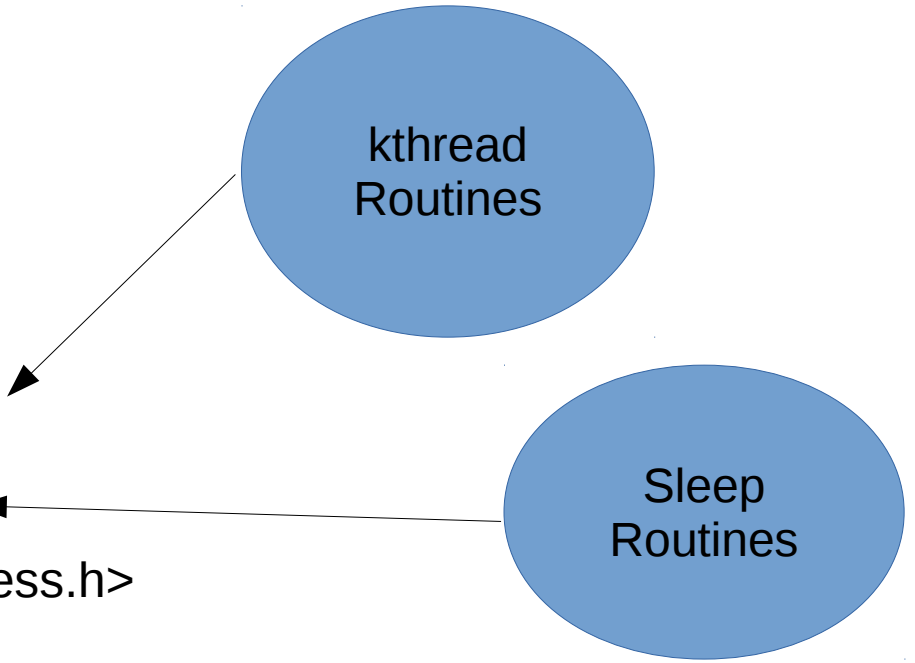
- #include <linux/delay.h>
- void ssleep(unsigned int seconds)
- Blocks the kthread for the specified number of seconds

Example

- Simple counter module
  - We'll use the hello proc module as a template
- kthread to increment counter once a second
- /proc/counter returns number of seconds since loaded
- Start counter on insert
- Stop counter on remove
- Note, you can not use this for part2!

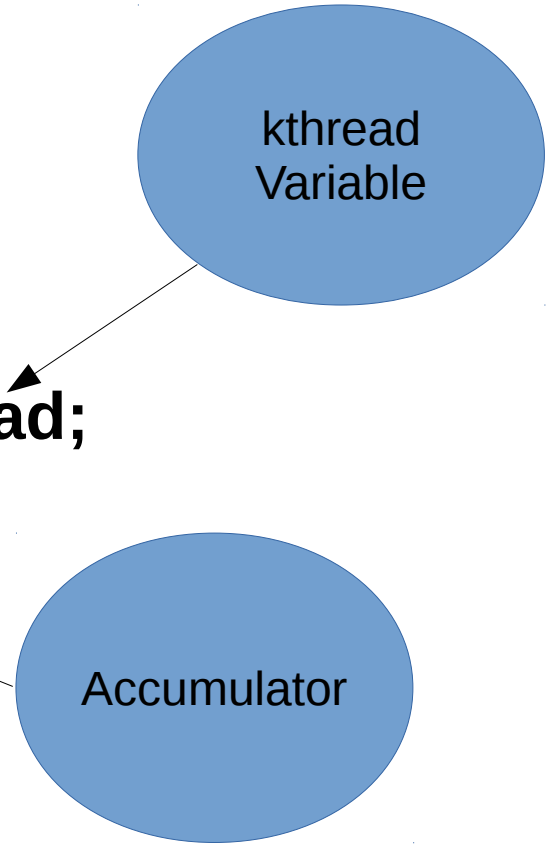
Headers

```
#include <linux/init.h>
#include <linux/module.h>
#include <linux/proc_fs.h>
#include <linux/slab.h>
#include <linux/string.h>
#include <linux/kthread.h>
#include <linux/delay.h>
#include <asm-generic/uaccess.h>

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Britton");
MODULE_DESCRIPTION("Simple module featuring proc read");
```

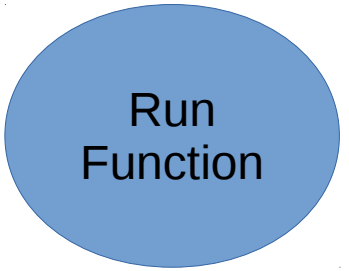


Globals

```
#define ENTRY_NAME "counter"
#define PERMS 0644
#define PARENT NULL
static struct file_operations fops;

static struct task_struct *kthread;
static int counter;
static char *message;
static int read_p;
```



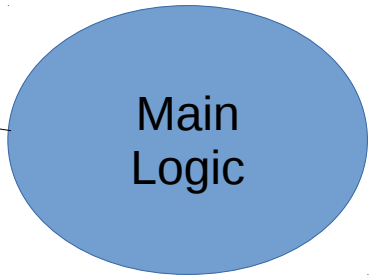
Thread Run




```
int counter_run(void *data) {
    while (!kthread_should_stop()) {
        ssleep(1);
        counter += 1;
    }
    printk("The counter thread has terminated\n");
    return counter;
}
```

Thread Run

```
int counter_run(void *data) {
    while (!kthread_should_stop()) {
        ssleep(1);
        counter += 1;
    }
    printk("The counter thread has terminated\n");
    return counter;
}
```

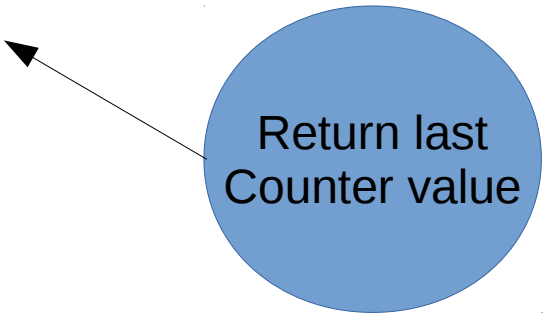


Thread Run


```
int counter_run(void *data) {
    while (!kthread_should_stop()) {
        ssleep(1);
        counter += 1;
    }
    printk("The counter thread has terminated\n");
    return counter;
}
```

Thread Run

```
int counter_run(void *data) {
    while (!kthread_should_stop()) {
        ssleep(1);
        counter += 1;
    }
    printk("The counter thread has terminated\n");
    return counter;
}
```



Proc Open

```
int counter_proc_open(struct inode *sp_inode, struct file *sp_file) {
    printk("proc called open\n");

    read_p = 1;
    message = kmalloc(sizeof(char) * 20, __GFP_WAIT | __GFP_IO |
    __GFP_FS);
    if (message == NULL) {
        printk("ERROR, counter_proc_open");
        return -ENOMEM;
    }
    sprintf(message, "The counter is now at: %d\n", counter);
    return 0;
}
```

Proc Read

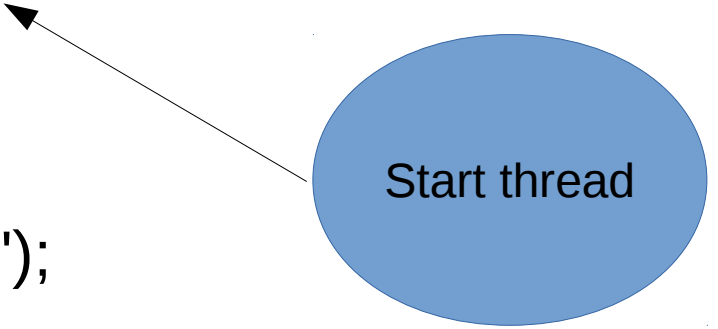
```
ssize_t counter_proc_read(struct file *sp_file, char __user
*buf, size_t size, loff_t *offset) {
    int len = strlen(message);

    read_p = !read_p;
    if (read_p) {
        return 0;
    }

    printk("proc called read\n");
    copy_to_user(buf, message, len);
    return len;
}
```

Proc Close

```
int counter_proc_release(struct inode
*sp_inode, struct file *sp_file) {
    printk("proc called release\n");
    kfree(message);
    return 0;
}
```

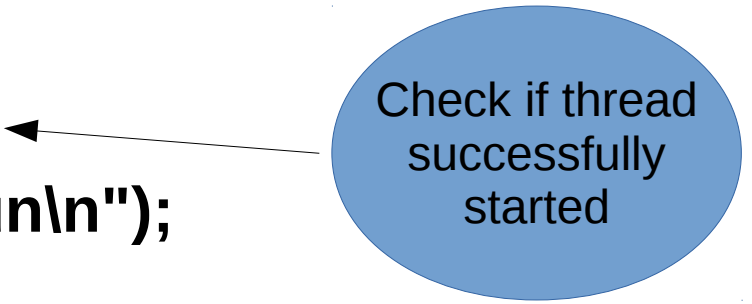
Module Init

```
static int counter_init(void) {
    printk("/proc/%s create\n", ENTRY_NAME);

    kthread = kthread_run(counter_run, NULL,
    "counter");
    if (IS_ERR(kthread)) {
        printk("ERROR! kthread_run\n");
        return PTR_ERR(kthread);
    }
```



Module Init

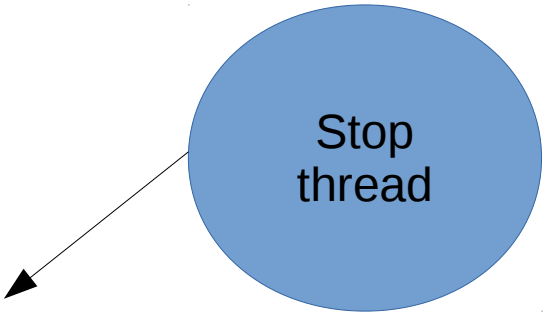
```
static int counter_init(void) {
    printk("/proc/%s create\n", ENTRY_NAME);

    kthread = kthread_run(counter_run, NULL,
    "counter");
    if (IS_ERR(kthread)) {
        printk("ERROR! kthread_run\n");
        return PTR_ERR(kthread);
    }
```



Module Init

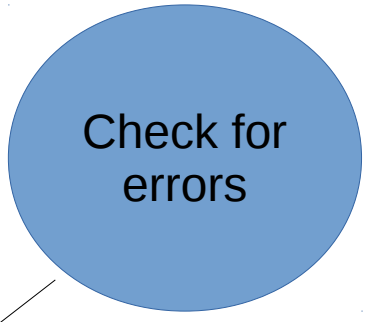
```
    fops.open = counter_proc_open;
    fops.read = counter_proc_read;
    fops.release = counter_proc_release;

    if (!proc_create(ENTRY_NAME, PERMS, NULL, &fops)) {
        printk("ERROR! proc_create\n");
        remove_proc_entry(ENTRY_NAME, NULL);
        return -ENOMEM;
    }
    return 0;
}
module_init(counter_init);
```

Module Exit




```
static void counter_exit(void) {
    int ret = kthread_stop(kthread);
    if (ret != -EINTR)
        printk("Counter thread has stopped\n");
    remove_proc_entry(ENTRY_NAME, NULL);
    printk("Removing /proc/%s.\n", ENTRY_NAME);
}
module_exit(counter_exit);
```

Module Exit




```
static void counter_exit(void) {
    int ret = kthread_stop(kthread);
    if (ret != -EINTR)
        printk("Counter thread has stopped\n");
    remove_proc_entry(ENTRY_NAME, NULL);
    printk("Removing /proc/%s.\n", ENTRY_NAME);
}
module_exit(counter_exit);
```