#### Time, Delays, and Deferred Work

In the Linux kernel, managing time and scheduling work to be executed after a certain delay or at a specific interval is crucial. This is important for various reasons, such as implementing timeouts, scheduling periodic tasks, or deferring work to be executed later to avoid blocking the current execution context.

#### Kernel Timers

Kernel timers are mechanisms that allow you to schedule functions to be executed at a later time. They are often used to implement timeouts and periodic tasks.

### Timer Initialization: init\_timer()

The init\_timer() function is used to initialize a timer. In more recent kernels, the function timer\_setup() is used instead. It sets up a timer structure to be used later.

**Prototype:**

void timer\_setup(struct timer\_list \*timer, void (\*callback)(struct timer\_list \*), unsigned long flags);

* timer: Pointer to the timer\_list structure.
* callback: Function to be called when the timer expires.
* flags: Timer flags.

### Adding and Modifying Timers: add\_timer(), mod\_timer()

* add\_timer(): Adds a timer to the list of active timers.

**Prototype:**

void add\_timer(struct timer\_list \*timer);

* timer: Pointer to the timer\_list structure.
* mod\_timer(): Modifies an existing timer to expire at a new time.

**Prototype:**

int mod\_timer(struct timer\_list \*timer, unsigned long expires);

* timer: Pointer to the timer\_list structure.
* expires: The new expiration time in jiffies.

### Timekeeping in the Kernel

The Linux kernel keeps track of time using several mechanisms, such as jiffies, high-resolution timers, and time of day. Jiffies are a global variable that counts the number of timer interrupts since the system booted.

### Real-Time Example and Code

Let's go through an example that demonstrates the initialization, addition, and modification of a kernel timer.

1. **Initialization of a Timer**

#include <linux/module.h>

#include <linux/kernel.h>

#include <linux/timer.h>

#include <linux/jiffies.h>

static struct timer\_list my\_timer;

void my\_timer\_callback(struct timer\_list \*timer)

{

printk(KERN\_INFO "Timer expired\n");

}

static int \_\_init my\_module\_init(void)

{

printk(KERN\_INFO "Initializing module\n");

timer\_setup(&my\_timer, my\_timer\_callback, 0);

my\_timer.expires = jiffies + msecs\_to\_jiffies(200);

add\_timer(&my\_timer);

return 0;

}

static void \_\_exit my\_module\_exit(void)

{

printk(KERN\_INFO "Exiting module\n");

del\_timer(&my\_timer);

}

module\_init(my\_module\_init);

module\_exit(my\_module\_exit);

MODULE\_LICENSE("GPL");

MODULE\_AUTHOR("Author");

MODULE\_DESCRIPTION("A simple timer example");

1. **Modifying a Timer**

To modify the timer to expire at a new time, use mod\_timer():

void my\_timer\_callback(struct timer\_list \*timer)

{

printk(KERN\_INFO "Timer expired, modifying to expire in 300ms\n");

mod\_timer(&my\_timer, jiffies + msecs\_to\_jiffies(300));

}

This modifies the timer to expire 300ms after the current time.

### Assignment 1: Basic Timer Implementation

Create a kernel module that initializes a timer to expire after a certain delay and prints a message.

#include <linux/module.h>

#include <linux/kernel.h>

#include <linux/timer.h>

#include <linux/jiffies.h>

static struct timer\_list my\_timer;

void my\_timer\_callback(struct timer\_list \*timer)

{

printk(KERN\_INFO "Timer expired and callback function executed.\n");

}

static int \_\_init my\_module\_init(void)

{

printk(KERN\_INFO "Initializing the basic timer module.\n");

timer\_setup(&my\_timer, my\_timer\_callback, 0);

// Set the timer to expire after 500ms (500ms \* HZ / 1000)

my\_timer.expires = jiffies + msecs\_to\_jiffies(500);

add\_timer(&my\_timer);

printk(KERN\_INFO "Timer added to expire in 500ms.\n");

return 0;

}

static void \_\_exit my\_module\_exit(void)

{

printk(KERN\_INFO "Exiting the basic timer module.\n");

del\_timer(&my\_timer);

}

module\_init(my\_module\_init);

module\_exit(my\_module\_exit);

MODULE\_LICENSE("GPL");

MODULE\_AUTHOR("Your Name");

MODULE\_DESCRIPTION("A basic timer example module");

1. **Periodic Timer:**

Modify the callback function to reschedule the timer:

void my\_timer\_callback(struct timer\_list \*timer)

{

printk(KERN\_INFO "Timer expired, rescheduling\n");

mod\_timer(&my\_timer, jiffies + msecs\_to\_jiffies(200));

}

Complete example below

#include <linux/module.h>

#include <linux/kernel.h>

#include <linux/timer.h>

#include <linux/jiffies.h>

static struct timer\_list my\_timer;

void my\_timer\_callback(struct timer\_list \*timer)

{

printk(KERN\_INFO "Timer expired, rescheduling for another 500ms.\n");

mod\_timer(&my\_timer, jiffies + msecs\_to\_jiffies(500));

}

static int \_\_init my\_module\_init(void)

{

printk(KERN\_INFO "Initializing the periodic timer module.\n");

timer\_setup(&my\_timer, my\_timer\_callback, 0);

my\_timer.expires = jiffies + msecs\_to\_jiffies(500);

add\_timer(&my\_timer);

printk(KERN\_INFO "Periodic timer added to expire in 500ms.\n");

return 0;

}

static void \_\_exit my\_module\_exit(void)

{

printk(KERN\_INFO "Exiting the periodic timer module.\n");

del\_timer(&my\_timer);

}

module\_init(my\_module\_init);

module\_exit(my\_module\_exit);

MODULE\_LICENSE("GPL");

MODULE\_AUTHOR("Your Name");

MODULE\_DESCRIPTION("A periodic timer example module");

1. **Dynamic Timer Interval via Sysfs:**

#include <linux/module.h>

#include <linux/kernel.h>

#include <linux/timer.h>

#include <linux/jiffies.h>

#include <linux/kobject.h>

#include <linux/sysfs.h>

static struct timer\_list my\_timer;

static unsigned int timer\_interval = 200; // Default interval in ms

static struct kobject \*timer\_kobj;

void my\_timer\_callback(struct timer\_list \*timer)

{

printk(KERN\_INFO "Timer expired, rescheduling for %u ms\n", timer\_interval);

mod\_timer(&my\_timer, jiffies + msecs\_to\_jiffies(timer\_interval));

}

static ssize\_t timer\_interval\_show(struct kobject \*kobj, struct kobj\_attribute \*attr, char \*buf)

{

return sprintf(buf, "%u\n", timer\_interval);

}

static ssize\_t timer\_interval\_store(struct kobject \*kobj, struct kobj\_attribute \*attr, const char \*buf, size\_t count)

{

sscanf(buf, "%u", &timer\_interval);

return count;

}

static struct kobj\_attribute timer\_interval\_attr = \_\_ATTR(timer\_interval, 0660, timer\_interval\_show, timer\_interval\_store);

static int \_\_init my\_module\_init(void)

{

int retval;

printk(KERN\_INFO "Initializing module\n");

timer\_kobj = kobject\_create\_and\_add("timer", kernel\_kobj);

if (!timer\_kobj)

return -ENOMEM;

retval = sysfs\_create\_file(timer\_kobj, &timer\_interval\_attr.attr);

if (retval)

kobject\_put(timer\_kobj);

timer\_setup(&my\_timer, my\_timer\_callback, 0);

mod\_timer(&my\_timer, jiffies + msecs\_to\_jiffies(timer\_interval));

return retval;

}

static void \_\_exit my\_module\_exit(void)

{

printk(KERN\_INFO "Exiting module\n");

del\_timer(&my\_timer);

kobject\_put(timer\_kobj);

}

module\_init(my\_module\_init);

module\_exit(my\_module\_exit);

MODULE\_LICENSE("GPL");

MODULE\_AUTHOR("Author");

MODULE\_DESCRIPTION("A dynamic timer interval example");d

### Tasklets

Tasklets are a mechanism in the Linux kernel for handling bottom-half processing asynchronously in interrupt context. They are typically used for tasks that need to be executed quickly and are often scheduled from interrupt handlers.

#### Tasklet Initialization: tasklet\_init()

The tasklet\_init() function initializes a tasklet structure.

**Prototype:**

void tasklet\_init(struct tasklet\_struct \*t, void (\*func)(unsigned long), unsigned long data);

* t: Pointer to the tasklet\_struct structure.
* func: Function to be called when the tasklet runs.
* data: Data to be passed to the function.

### Example: Tasklet Implementation

#include <linux/module.h>

#include <linux/kernel.h>

#include <linux/interrupt.h>

static struct tasklet\_struct my\_tasklet;

void my\_tasklet\_function(unsigned long data)

{

printk(KERN\_INFO "Tasklet function executed with data: %lu\n", data);

}

static int \_\_init my\_module\_init(void)

{

printk(KERN\_INFO "Initializing the tasklet module.\n");

tasklet\_init(&my\_tasklet, my\_tasklet\_function, 42);

tasklet\_schedule(&my\_tasklet);

return 0;

}

static void \_\_exit my\_module\_exit(void)

{

printk(KERN\_INFO "Exiting the tasklet module.\n");

tasklet\_kill(&my\_tasklet);

}

module\_init(my\_module\_init);

module\_exit(my\_module\_exit);

MODULE\_LICENSE("GPL");

MODULE\_AUTHOR("Your Name");

MODULE\_DESCRIPTION("A tasklet example module");

### Workqueues

Workqueues are another mechanism in the Linux kernel for handling deferred work that needs to be executed asynchronously outside of interrupt context. They are suitable for tasks that may take longer to execute or involve complex operations.

#### Workqueue Initialization: create\_workqueue()

The create\_workqueue() function initializes a workqueue.

**Prototype:**

struct workqueue\_struct \*create\_workqueue(const char \*name);

### Example: Workqueue Implementation

#include <linux/module.h>

#include <linux/kernel.h>

#include <linux/workqueue.h>

static struct workqueue\_struct \*my\_workqueue;

static void my\_work\_function(struct work\_struct \*work)

{

printk(KERN\_INFO "Work function executed.\n");

}

static int \_\_init my\_module\_init(void)

{

printk(KERN\_INFO "Initializing the workqueue module.\n");

my\_workqueue = create\_workqueue("my\_workqueue");

if (!my\_workqueue) {

printk(KERN\_ERR "Failed to create workqueue.\n");

return -ENOMEM;

}

queue\_work(my\_workqueue, &my\_work);

return 0;

}

static void \_\_exit my\_module\_exit(void)

{

printk(KERN\_INFO "Exiting the workqueue module.\n");

if (my\_workqueue)

destroy\_workqueue(my\_workqueue);

}

module\_init(my\_module\_init);

module\_exit(my\_module\_exit);

MODULE\_LICENSE("GPL");

MODULE\_AUTHOR("Your Name");

MODULE\_DESCRIPTION("A workqueue example module");