Laboratory 5: „Kernel threads and synchronization mechanisms” (one week)

Arkadiusz Chrobot, PhD
Karol Tomaszewski, PhD

April 22, 2024

Contents

Introduction **1**

1. Kernel threads **1**
1.1. API description 1
1.2. Example 3

2. Synchronization tools **5**
2.1. Indivisible operations 5
2.2. Mutexes 9
2.3. Completion variables 12
2.4. Sequential locks 14
2.5. RCU 17

Exercises **20**

Introduction

This guide contains information about the kernel threads and some of the synchronization mechanisms. Chapter 1 describes the kernel threads, and Chapter 2 presents the main synchronization mechanisms, i.e. indivisible (atomic) operations, mutexes, completion variables, sequential locks and the RCU mechanism. The last chapter of this document contains a list of tasks to be solved as part of the laboratory.

1. Kernel threads

Kernel threads are created in the kernel space, which means that they share all resources with other kernel subsystems in the operating system. Such threads usually fall into one of two categories:

1. threads activated by a specific kernel subsystems that perform certain tasks and go into a waiting state,
2. threads activated by a time mechanism every specified period of time that perform certain tasks and go into a waiting state.

The execution of both types of threads is therefore similar. Both types of threads go into `TASK_RUNNING` state due to an external event, than perform some work e.g. related to monitoring the state of resources, and then go into a waiting state - usually the `TASK_INTERRUPTIBLE` state. This action is cyclical. Kernel threads are started when the operating system starts or when the module in which they are programmed is loaded. They are finished when removing the module or shutting down the operating system. Putting such threads into the waiting state is possible because they are executed in the *process context*. Like user tasks, they are prioritized, but not necessarily pre-empted. The pre-emption possibility depends on the kernel configuration with which it was compiled. If the thread pre-emption option in the kernel build configuration is turned off, then all kernel threads will remain active as long as they do not release the processor themselves. Therefore, it is important that the kernel thread, after doing its work puts itself into the waiting state and calls the processor scheduler to regroup tasks. Information about running kernel threads can be obtained using the `ps aux` command. The kernel thread names in the list displayed by this command are enclosed in square brackets.

1.1. API description

Kernel threads are implemented in the form of functions that have the following prototype:

```
int thread_function(void *data)
```

The definition of this function should contain code that will be executed by the thread. These functions will be called *thread functions*. The following functions and macros have been defined to handle threads in the Linux kernel:

`kthread_create(threadfn, data, namefmt, arg…)` - this macro creates a thread. It takes three call arguments. The first is the pointer to the thread function, the second is the pointer to a data for this function and the third argument is the name of the thread, which can contain formatting strings, as in the case of the `printf()` function. If this is the case, then the third argument should be followed by other arguments whose types match the format strings in the name. The macro returns a pointer to the descriptor of the created thread, i.e. a structure of type `struct task_struct`. The created thread is inactive by default (in the `TASK_INTERRUPTIBLE` state). To activate it, the `wake_up_process()` function have to be called with a pointer to the thread descriptor given as the call argument. The `wake_up_process()` function will return `1` if the thread was successfully activated or `0` if it has already been active.

`kthread_run(threadfn, data, namefmt, …)` - this macro creates a thread and activates it. The macro takes the same arguments as `kthread_create` and returns values of the same type. **Those two macros must not be used with the same thread function.**

`void kthread_bind(struct task_struct *k, unsigned int cpu)` - this function allows to determine on which processor, in a computer with many processors, the thread will be executed. It takes two call arguments: a pointer to a thread descriptor and a processor identifier, which is a natural number. The first processor has an identifier value of `0`. The function returns nothing.

`int kthread_stop(struct task_struct *k)` - function called to terminate the kernel thread. As a call argument, it takes a pointer to the thread descriptor, and returns the value returned by the thread function or the value of `-EINTR` if the thread has never been active.

`bool kthread_should_stop(void)` - this function is called inside of the thread function and returns `true` if the `kthread_stop()` function was called for this thread, and `false` otherwise. If this function returns `true`, then the thread function should end.

The described subprograms are not the only ones designed to handle kernel threads, but they are a necessary minimum. They are declared in the `linux/kthread.h` header file.

In order for the thread to release the processor and wait for it to be activated by other kernel code, it may be needed to define a wait queue and add the thread to it. This queue is a structure of type `wait_queue_head_t`. Functions and macros related to its support are declared in the `linux/wait.h` header file. Here is a list describing some of them:

`init_waitqueue_head(q)` - macro that is responsible for initiating the wait queue. As a call argument, it takes the pointer to this queue.

`DEFINE_WAIT(name)` - macro that defines the waiting queue element that can be added to it.

`void add_wait_queue(wait_queue_head_t *q, wait_queue_t *wait)` - this function adds the element indicated by the `wait` argument to the wait queue indicated by the `q` argument.

`void prepare_to_wait(wait_queue_head_t *q, wait_queue_t *wait, int state)` - the function, if necessary, adds the element indicated by its `wait` argument to the wait queue indicated by the `q` argument, and then puts the kernel thread or user task that called it in state determined by the `state` argument. The argument used for this purpose is usually a constant variable that specifies the waiting state of the thread or task.

`void finish_wait(wait_queue_head_t *q, wait_queue_t *wait)` - this function puts the kernel thread or user task that called it into `TASK_RUNNING` state and removes the element pointed to by its `wait` argument from the queue indicated by the `q` argument.

`wake_up(x)` - a macro that activates (wakes up) a single thread waiting in the queue. As a call argument, it takes the pointer to the waiting queue.

`wake_up_all(x)` - a macro that wakes up all threads waiting in the wait queue, whose pointer is given as its call argument.

1.2. Example

Listing 1 contains the source code of the kernel module, which creates two threads. One of the threads is activated every second and its only task is to activate the second thread, which places the message in the kernel buffer.

Listing 1: Example module with two threads

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

enum thread_index {WAKING_THREAD, SIMPLE_THREAD};

static struct threads_structure
{
        struct task_struct *thread[2];
} threads;

static wait_queue_head_t wait_queue;
static bool condition;

static int simple_thread(void *data)
{
        DEFINE_WAIT(wait);
        for(;;) {
                add_wait_queue(&wait_queue,&wait);
                while(!condition) {
                        prepare_to_wait(&wait_queue,&wait,TASK_INTERRUPTIBLE);
                        if(kthread_should_stop())
                                return 0;
                        printk(KERN_INFO "[simple_thread]: awake\n");
                        schedule();
                }
                condition=false;
                finish_wait(&wait_queue,&wait);
        }
}

static int waking_thread(void *data)
{
        for(;;) {
                if(kthread_should_stop())
                        return 0;
                set_current_state(TASK_INTERRUPTIBLE);
                if(schedule_timeout(1*HZ))
                        printk(KERN_INFO "Signal received!\n");
                condition=true;
                wake_up(&wait_queue);
        }

}

static int __init threads_init(void)
{
        init_waitqueue_head(&wait_queue);
        threads.thread[SIMPLE_THREAD] = kthread_run(simple_thread,NULL,"simple_thread");
        threads.thread[WAKING_THREAD] = kthread_run(waking_thread,NULL,"waking_thread");
        return 0;
}

```

```
54 static void __exit threads_exit(void)
55 {
56         kthread_stop(threads.thread[WAKING_THREAD]);
57         kthread_stop(threads.thread[SIMPLE_THREAD]);
58 }
59
60 module_init(threads_init);
61 module_exit(threads_exit);
62
63 MODULE_LICENSE("GPL");
64 MODULE_DESCRIPTION("An example of using the kernel linux threads.");
65 MODULE_AUTHOR("Arkadiusz Chrobot <a.chrobot@tu.kielce.pl>");
66 MODULE_VERSION("1.0");
```

Lines 2 and 3 of the source code include header files related to thread and queue support. In line 5 the enumeration type has been defined, whose elements will be used as constant variables defining the indexes of array elements in the `threads` structure. This structure is declared in line 10 of the source code, and its type is defined earlier (`struct thread_structure`). The `threads` structure contains a single field, which is an array of pointers for process descriptors. It will store the addresses of the descriptors created by the kernel thread module. The address of the periodically wake-up thread descriptor will be stored in an element of this table specified by the value of the `WAKING_THREAD` constant variable. The address of the thread descriptor activated by the wake-up thread will be stored in the element specified by the value of the `SIMPLE_THREAD` variable. In line 12 the `wait_queue` is declared. In line 13, a `bool` variable named `condition` is declared, whose value will determine whether the thread can finish its waiting in the queue.

Lines 15-30 contain the definition of the `simple_thread()` function, which is the function performed within the thread activated by the another thread started by the module. On line 17, a queue element named `wait` has been declared, that will represent the thread performing this function on the wait queue. Lines 18-19 contain an „infinite" `for` loop. As described in the previous chapter, threads usually do their work cyclically, which is why this loop was used in the thread function. The first thing done in this loop is adding the thread represented by the `wait` variable to the queue (line 19). In lines 20-26, there is a `while` loop that ends when the value of `condition` variable will become `true`. Inside this loop, the thread is added to the queue and changes its status to the waiting state (`TASK_INTERRUPTIBLE`), if it was not already in it. Please note that this does not necessarily means stopping the thread execution. It then calls the `kthread_should_stop()` function to see if it should end. If this function returns `true`, then the thread function returns `0` and exits. Otherwise, the thread adds a message to the kernel buffer and then calls the `schedule()` function, which is a processor planner that takes away the processor from one thread and passes it to another task (user process, user thread, or other kernel thread). Only after completing these steps will the thread be in a waiting state. If the return from `schedule()` function occurred, then the thread was awakened and the planner gave him the processor. The thread performs the next iteration of the `while` loop to ensure that the `condition` variable is `true`, which means it can actually end the waiting. Kernel threads can only be awakened for two reasons: either they must end their activity, or an event that they are waiting for has occurred. In this example module, the `condition` value is used to distinguish between these two causes. After the `while` loop completes, the thread sets the value of `condition` to `false` (line 27), changes its status to `TASK_RUNNING`, then removes from the queue (line 28), and finally performs the next iteration of the `for` loop.

Lines 32-44 contain the definition of the `waking_thread()` function, which will be executed within the thread activated cyclically after a specified time. This function is built differently than the previous one. Only one loop is executed in it. This is an „infinite" `for` loop. Inside this loop, the thread first checks if it should end (lines 35-36), and if so, then the thread function returns `0` and exits. Otherwise, the thread changes its state to `TASK_INTERRUPTIBLE` by calling the `set_current_state()` function, which sets the state of the currently executed task to the one specified by its call argument. After that, the thread calls the `schedule_timeout()` function. The purpose of this function is to delay the execution of task for a period of time specified by its call argument. To make other tasks able to use the processor that is currently assigned to the thread, this function puts the thread to sleep and puts it on the queue waiting for the time specified in its call, thereby releasing the processor for other tasks. All these activities are contained the function body. The waiting time in the sample module is determined by the constant

variable named HZ. Its value depends on the hardware platform on which Linux is running, and it always determines the frequency of the system clock, i.e. how many interrupt signals it will generate in 1 second. In this module, the thread will wait for activation for 1 second[1]. If the `schedule_timeout()` returned and the returning value is `0`, it means that the time for which the thread was delayed has already expired[2]. However, if the function returned a non-zero value, it means that the `kthread_stop()` function was called on the thread. In this case, the thread places the appropriate message in the kernel buffer. The necessity to stop the thread will be handled at the beginning of the next `for` loop iteration by the conditional statement in lines 35 and 36. After waking up, the thread is in a TASK_RUNNING state. In line 40 it changes the value of `condition` variable to `true` and in line 41 it calls the `wake_up()` function for the queue in which (probably) the first of the threads is waiting to wake up. After doing this, the `waking_thread()` function goes to the next iteration of the `for` loop.

In the module constructor (lines 46-52), a waiting queue is initiated (line 48) and two threads are created (lines 49-50). The first one (activated by event) will be visible on the task list under the name *simple_thread*, and the second (activated by the period of time) under the name *waking_thread*. The addresses of the descriptors of these threads are stored in the `thread` table of the `threads` structure. Please note that after creating both threads are activated.

In the module destructor (lines 54-58) the `kthread_stop()` function is called for both threads, which signals them to quit their work. Values returned by this function calls are ignored.

It is easier to watch the messages placed in the kernel buffer by calling the `dmesg` with the `-w -d` options. The first option instructs this command to wait for the next message to appear on the screen, and the second option causes `dmesg` to place information about how much time elapsed between subsequent messages placed in the buffer. The execution of the `dmesg` command called like that can be terminated by pressing the `Ctrl+C` key combination.

2. Synchronization tools

Same as with user tasks, kernel threads may also cause a hazard situations when they use shared resources. To prevent it, a synchronization mechanisms implemented by kernel programmers should be used in kernel modules. This section describes some of them. They can be divided into two categories: general purpose measures (indivisible operations, mutexes and signal variables) and those focused on solving the readers-writers problem (sequential locks, RCU mechanism).

2.1. Indivisible operations

If the resources shared by threads and other kernel code fragments are integer variables or even a single bits in a word, then the *indivisible (atomic) operations* - implemented in the form of macros and functions - can be used to protect them. Two abstract data types have been defined in the `linux/types.h` header file: `atomic_t` and `atomic64_t`. The variables of these types store 32-bit and 64-bit integers, respectively. The operations defined for these types are performed in an indivisible (*atomic*) manner, which in this case means two things:

1. no other operation on the variable of any of the aforementioned types can start until the operation currently executed on this variable is finished
2. these operations will never be interrupted, they will always be executed to the end.

The `atomic_t` type has the following indivisible operations in the form of macros and functions:

`ATOMIC_INIT(i)` - macro that is used to initialize a variable of the `atomic_t` type. It takes one call argument, which is the value to be assigned to the variable of the mentioned type. This value can be written directly or in the form of a constant, expression or a variable. The value returned by this macro should be assigned to a variable of the `atomic_t` type.

[1] Multiplying the HZ variable by 1 is unnecessary in this case, but it presents how to change this period of time using a different value.

[2] The measurement of this time may not be accurate, so the thread can be woken up a few clock cycles sooner or later.

`int atomic_read(const atomic_t *v)` - a function that reads the value of an `atomic_t` variable passed to it and returns it as an `int`.

`void atomic_set(atomic_t *v, int i)` - a function that gives the `atomic_t` variable pointed by the first call argument, the `int` value given by the second argument.

`void atomic_add(int i, atomic_t *v)` - the function adds the value of type `int` given by the first call argument to the variable of type `atomic_t` pointed by the second argument. The result of the operation is stored in an `atomic_t` variable.

`void atomic_sub(int i, atomic_t *v)` - the function subtracts the value of type `int` given by the first call argument from the variable of type `atomic_t` pointed by the second argument. The result of the operation is stored in an `atomic_t` variable.

`void atomic_inc(atomic_t *v)` - this function increases the value of the `atomic_t` variable pointed by call argument by one and does not return anything.

`void atomic_dec(atomic_t *v)` - this function decreases the value of the `atomic_t` variable pointed by call argument by one and does not return anything.

`int atomic_sub_and_test(int i, atomic_t *v)` - function that subtracts the number given by its first call argument from the value of the `atomic_t` variable pointed by the second argument. The result of subtraction is stored in a variable of `atomic_t` type, and the function returns a non-zero value if the result of this subtraction is `0` and zero otherwise.

`int atomic_add_negative(int i, atomic_t *v)` - the function adds the number given by the first call argument to a variable of `atomic_t` type pointed by the second argument. The result of the operation is stored in a variable of `atomic_t` type, and the function returns a non-zero value if it was negative, or zero otherwise.

`int atomic_add_return(int i, atomic_t *v)` - the function adds the number given by the first call argument to a variable of `atomic_t` type pointed by the second argument. The result of the operation is stored in a variable of `atomic_t` type and returned by a function.

`int atomic_sub_return(int i, atomic_t *v)` - the function subtracts the number given by the first call argument from a variable of `atomic_t` type pointed by the second argument. The result of the operation is stored in a variable of `atomic_t` type and returned by a function.

`atomic_inc_return(v)` - a macro that increases the value of the `atomic_t` variable pointed as a call argument by one and returns the resulting value.

`atomic_dec_return(v)` - a macro that decreases the value of the `atomic_t` variable pointed as a call argument by one and returns the resulting value.

`int atomic_dec_and_test(atomic_t *v)` - the function decreases the value of a variable of `atomic_t` type pointed by a call argument by one and returns zero if the result is different from `0`, or returns a non-zero value otherwise.

`int atomic_inc_and_test(atomic_t *v)` - the function increases the value of a variable of `atomic_t` type pointed by a call argument by one and returns zero if the result is different from `0`, or returns a non-zero value otherwise.

`atomic_inc_return(v)` - this macro increases by one the value of the `atomic_t` variable pointed by a call argument and returns the result of this action.

`atomic_dec_return(v)` - this macro decreases by one the value of the `atomic_t` variable pointed by a call argument and returns the result of this action.

All of these functions are `inline` functions. There are also equivalents for these subprograms for the `atomic64_t` type. Their prototypes differ from the prototypes described above only by the fact that the word `atomic`, written in lowercase or uppercase, is followed by the number `64`.

Indivisible operations on single bits are performed on memory words indicated by a `void *` pointer using the following functions and macros:

`void set_bit(int nr, volatile void * addr)` - the function sets to 1 the value of a bit which position is given by its first call argument, in the bit word pointed by its second argument.

`void clear_bit(int nr, volatile void * addr)` - the function sets to 0 the value of a bit which position is given by its first call argument, in the bit word pointed by its second argument.

`void change_bit(int nr, volatile void * addr)` - the function changes to the opposite the value of a bit which position is given by its first call argument, in the bit word pointed by its second argument.

`int test_and_set_bit(int nr, volatile void * addr)` - the function sets to 1 the value of a bit which position is given by its first call argument, in the bit word pointed by its second argument and returns the previous value of this bit.

`int test_and_clear_bit(int nr, volatile void * addr)` - the function sets to 0 the value of a bit which position is given by its first call argument, in the bit word pointed by its second argument and returns the previous value of this bit.

`int test_and_change_bit(int nr, volatile void * addr)` - the function changes to the opposite the value of a bit which position is given by its first call argument, in the bit word pointed by its second argument and returns the previous value of this bit.

`test_bit(nr,addr)` - a macro that returns the bit value in the bit word pointed to by its second call argument. The position of the bit is given by its first argument.

The functions presented in this overview are `inline` functions. The implementation of subprograms performing atomic operations on bits depends on the hardware platform.

Listing 2 contains the source code of an example module in which two threads perform operations on a variable of type `atomic64_t`.

Listing 2: Example module presenting the use of indivisible operations

```
#include<linux/module.h>
#include<linux/kthread.h>
#include<linux/wait.h>
#include<linux/types.h>

enum thread_index {WAKING_THREAD, FIRST_THREAD, SECOND_THREAD};

static struct thread_structure
{
        struct task_struct *thread[3];
} threads;

static wait_queue_head_t wait_queue;
static bool condition;
static atomic64_t number = ATOMIC64_INIT(0);

static int first_thread(void *data)
{
        int counter = 0;
        DEFINE_WAIT(wait);
        for(;;) {
                pr_info("[first_thread] Number value: %ld\n",atomic64_read(&number));
                atomic64_inc(&number);
                if(counter%3) {
                        add_wait_queue(&wait_queue,&wait);
                        while(!condition) {
                                prepare_to_wait(&wait_queue,&wait,TASK_INTERRUPTIBLE);
                                if(kthread_should_stop())
                                        return 0;
```

```
                                printk(KERN_INFO "[first_thread]: awake\n");
                                schedule();
                        }
                        condition=false;
                        finish_wait(&wait_queue,&wait);
                }
                counter++;
        }
}

static int second_thread(void *data)
{
        int counter = 0;
        DEFINE_WAIT(wait);
        for(;;) {
                pr_info("[second_thread] Number value: %ld\n",atomic64_read(&number));
                atomic64_dec(&number);
                if(counter%7) {
                        add_wait_queue(&wait_queue,&wait);
                                while(!condition) {
                                        prepare_to_wait(&wait_queue,&wait,TASK_INTERRUPTIBLE);
                                        if(kthread_should_stop())
                                                return 0;
                                        printk(KERN_INFO "[second_thread]: awake\n");
                                        schedule();
                                }
                                condition=false;
                                finish_wait(&wait_queue,&wait);
                }
                counter++;
        }
}

static int waking_thread(void *data)
{
        for(;;) {
                if(kthread_should_stop())
                        return 0;
                set_current_state(TASK_INTERRUPTIBLE);
                if(schedule_timeout(1*HZ))
                        printk(KERN_INFO "Signal received!\n");
                condition=true;
                wake_up_all(&wait_queue);
        }

}

static int __init threads_init(void)
{
        init_waitqueue_head(&wait_queue);
        threads.thread[FIRST_THREAD] = kthread_run(first_thread,NULL,"first_thread");
        threads.thread[SECOND_THREAD] = kthread_run(second_thread,NULL,"second_thread");
        threads.thread[WAKING_THREAD] = kthread_run(waking_thread,NULL,"waking_thread");
        return 0;
}

static void __exit threads_exit(void)
{
        kthread_stop(threads.thread[WAKING_THREAD]);
        kthread_stop(threads.thread[SECOND_THREAD]);
```

```
90          kthread_stop(threads.thread[FIRST_THREAD]);
91  }
92
93  module_init(threads_init);
94  module_exit(threads_exit);
95
96  MODULE_LICENSE("GPL");
97  MODULE_DESCRIPTION("An example of using the kernel linux threads and the atomic64_t data type.");
98  MODULE_AUTHOR("Arkadiusz Chrobot <a.chrobot@tu.kielce.pl>");
```

Three threads are created in the module: one thread activated by the time and two threads responding to the activation event triggered by a time-activated thread. Compared to the previous module, the number of the latter type of threads has changed. The operations on a variable of type `atomic64_t` are performed within the functions called for those threads. The variable of this type itself is declared in line 15 of the module source code. In the thread function, named `first_thread()`, its value is read and placed in the kernel buffer in line 22, and then increased by one in line 23. In the second thread function, called `second_thread()`, the value of this variable is also read and placed in the kernel buffer (line 45), but in the next row it is decreased by one (line 46). To increase the intensity of operations performed on this variable, the thread executing the `first_thread()` function is set to wait every three iterations of the `for` loop, and the thread executing the `second_thread()` function is waiting every seventh iteration. This effect was achieved by declaring a local variable named `counter` in each function, increasing its value after each iteration of the `for` loop and putting the thread to sleep when its value is divisible by `3` or `7`, respectively (lines 24 and 47).

2.2. Mutexes

Mutexes in the Linux kernel are binary semaphores whose implementation is independent of the hardware platform on which they are used. They are usually used to protect the complex type resources (such as structures) against hazard situations, but they can also be used to protect simple types variables. The `linux/mutex.h` header file have to be included in the source code to use this synchronization mechanism in a module. It defines the `struct mutex` type used for declaring mutexes, and the following subprograms that operate on them:

`DEFINE_MUTEX(mutexname)` - a macro that defines and initiates as an unoccupied mutex, with a name that is given as its call argument.

`mutex_init(mutex)` - a macro that initiates the mutex pointed by its call argument as unoccupied.

`mutex_lock_interruptible(lock)` - the macro locks (acquires) the mutex pointed by the call argument. If the attempt is unsuccessful, the thread that uses this macro is put into a `TASK_INTERRUPTIBLE` (waiting) state.

`mutex_lock(lock)` - the macro locks (acquires) the mutex pointed by the call argument. If the attempt is unsuccessful, the thread that uses this macro is put into a `TASK_UNINTERRUPTIBLE` (waiting) state. This macro returns a non-zero value if the thread was activated for a reason other than the one it was waiting for.

`mutex_lock_killable(lock)` - the macro locks (acquires) the mutex pointed by the call argument. If the attempt is unsuccessful, the thread that uses this macro is put into a `TASK_KILLABLE` state. This macro returns a non-zero value if the thread was activated for a reason other than the one it was waiting for.

`int mutex_trylock(struct mutex *lock)` - the macro tries to lock the mutex pointed by the call argument. If the attempt is unsuccessful, the function returns zero, and otherwise it returns one. The function does not put the thread that called it into a waiting state.

`void mutex_unlock(struct mutex *lock)` - the macro releases the mutex pointed by the call argument.

`int mutex_is_locked(struct mutex *lock)` - the function returns one if the mutex pointed by the call argument is occupied by the thread, or it returns zero otherwise.

Listing 3 contains the source code of the module being a modification of the module in Listing 2, in which instead of atomic operations the mutexes were used on the `atomic64_t` variable to synchronize access to the variable type `int`.

Listing 3: Example module presenting the use of mutexes

```
#include<linux/module.h>
#include<linux/kthread.h>
#include<linux/wait.h>
#include<linux/mutex.h>

enum thread_index {WAKING_THREAD, FIRST_THREAD, SECOND_THREAD};

static struct thread_structure
{
        struct task_struct *thread[3];
} threads;

static wait_queue_head_t wait_queue;
static bool condition;
static DEFINE_MUTEX(number_lock);
static int number;

static int first_thread(void *data)
{
        int counter = 0;
        DEFINE_WAIT(wait);
        for(;;) {
                mutex_lock(&number_lock);
                pr_info("[first_thread] Number value: %d\n",number);
                number++;
                mutex_unlock(&number_lock);
                if(counter%3) {
                        add_wait_queue(&wait_queue,&wait);
                        while(!condition) {
                                prepare_to_wait(&wait_queue,&wait,TASK_INTERRUPTIBLE);
                                if(kthread_should_stop())
                                        return 0;
                                printk(KERN_INFO "[first_thread]: awake\n");
                                schedule();
                        }
                        condition=false;
                        finish_wait(&wait_queue,&wait);
                }
                counter++;
        }
}

static int second_thread(void *data)
{
        int counter = 0;
        DEFINE_WAIT(wait);
        for(;;) {
                mutex_lock(&number_lock);
                pr_info("[second_thread] Number value: %d\n",number);
                number--;
                mutex_unlock(&number_lock);
                if(counter%7) {
                        add_wait_queue(&wait_queue,&wait);
                                while(!condition) {
```

```
55                                      prepare_to_wait(&wait_queue,&wait,TASK_INTERRUPTIBLE);
56                                      if(kthread_should_stop())
57                                              return 0;
58                                      printk(KERN_INFO "[second_thread]: awake\n");
59                                      schedule();
60                              }
61                              condition=false;
62                              finish_wait(&wait_queue,&wait);
63                      }
64                      counter++;
65              }
66      }
67
68      static int waking_thread(void *data)
69      {
70              for(;;) {
71                      if(kthread_should_stop())
72                              return 0;
73                      set_current_state(TASK_INTERRUPTIBLE);
74                      if(schedule_timeout(1*HZ))
75                              printk(KERN_INFO "Signal received!\n");
76                      condition=true;
77                      wake_up_all(&wait_queue);
78              }
79
80      }
81
82      static int __init threads_init(void)
83      {
84              init_waitqueue_head(&wait_queue);
85              threads.thread[FIRST_THREAD] = kthread_run(first_thread,NULL,"first_thread");
86              threads.thread[SECOND_THREAD] = kthread_run(second_thread,NULL,"second_thread");
87              threads.thread[WAKING_THREAD] = kthread_run(waking_thread,NULL,"waking_thread");
88              return 0;
89      }
90
91      static void __exit threads_exit(void)
92      {
93              kthread_stop(threads.thread[WAKING_THREAD]);
94              kthread_stop(threads.thread[SECOND_THREAD]);
95              kthread_stop(threads.thread[FIRST_THREAD]);
96      }
97
98      module_init(threads_init);
99      module_exit(threads_exit);
100
101     MODULE_LICENSE("GPL");
102     MODULE_DESCRIPTION("An example of using the kernel linux threads and a mutex.");
103     MODULE_AUTHOR("Arkadiusz Chrobot <a.chrobot@tu.kielce.pl>");
```

There are several differences in Listing 3 compared to the one described earlier. First of all, in line 15 a mutex named `number_lock` was defined, and in line 16 a variable named `number` of type `int` was declared, on which both threads activated by events will perform operations. In the thread that performs the `first_thread()` function, in the `for` loop in line 23 the mentioned mutex is acquired. If this operation is successful, then the thread puts the value of the variable `number` (line 24) in the kernel buffer, then increases the value of this variable by one (line 25), and then releases the mutex (line 26). If the mutex acquisition fails, the thread will have to wait with the execution of lines 24-26 until the second thread releases the mutex. The second thread, executing the `second_thread()` function, also acquires the mutex and places the value of the `number` variable in the kernel buffer, but then this value is decreased by one (line 50), and then the mutex is released. Same as in case of the first thread, if it

fails to acquire the mutex, it goes into a waiting state until mutex is released. The rest of the module code is similar to the one presented in Listing 2.

2.3. Completion variables

Completion variables are a simplified version of semaphores, used when synchronization is carried out according to the scenario in which the thread must wait for the result of another thread. The definition of completion variable is contained in the `struct completion` type defined in the `linux/completion.h` header file. The following subprograms that perform indivisible operations on such variables are also declared or defined in this header file:

`DECLARE_COMPLETION(work)` - a macro that declares and initializes a completion variable with the name given as its call argument

`void init_completion(struct completion *x)` - an inline function that initializes a completion variable pointed by the call argument.

`void wait_for_completion(struct completion *)` - function called by a thread waiting for a termination of another thread. It takes the pointer of the completion variable as the call argument. This function puts the calling thread in the `TASK_UNINTERRUPTIBLE` state.

`int wait_for_completion_interruptible(struct completion *x)` - function called by a thread waiting for a termination of another thread. It takes the pointer of the completion variable as the call argument. This function puts the calling thread in the `TASK_INTERRUPTIBLE` state. If the thread will be activated by an event other than the one it is waiting for, the function will return a non-zero value and zero otherwise.

`int wait_for_completion_killable(struct completion *x)` - function called by a thread waiting for a termination of another thread. It takes the pointer of the completion variable as the call argument. This function puts the calling thread in the `TASK_KILLABLE` state. If the thread will be activated by an event other than the one it is waiting for, the function will return a non-zero value and zero otherwise.

`unsigned long wait_for_completion_timeout(struct completion *x, unsigned long timeout)` - this function works similar to `wait_for_completion()`, but takes an additional call argument that specifies time after which the wait will end if another thread does not finish its work. It returns `0` if the wait time elapses (timeout) or it returns the time remaining to wait, if another thread indicates that it has finished its work (before timeout). Both the waiting time, which is given as the call argument, and the time returned by the function is expressed as a number of system clock ticks.

`long wait_for_completion_interruptible_timeout(struct completion *x, unsigned long timeout)` - this function works similar to `wait_for_completion_interruptible()`, but takes an additional call argument that specifies time after which the wait will end if another thread does not finish its work. It returns `0` if the wait time elapses (timeout), a negative number if the thread is activated by an event other than the one it was waiting for, or it returns the time remaining to wait, if another thread indicates that it has finished its work (before timeout). Both the waiting time, which is given as the call argument, and the time returned by the function is expressed a number of system clock ticks.

`long wait_for_completion_killable_timeout(struct completion *x, unsigned long timeout)` - this function works similar to `wait_for_completion_killable()`, but takes an additional call argument that specifies time after which the wait will end if another thread does not finish its work. It returns `0` if the wait time elapses (timeout), a negative number if the thread is activated by an event other than the one it was waiting for, or it returns the time remaining to wait, if another thread indicates that it has finished its work (before timeout). Both the waiting time, which is given as the call argument, and the time returned by the function is expressed in system clock ticks.

`void complete(struct completion *)` - function called by the thread after finishing the work, at the end of which another thread was waiting. It takes the address of the completion variable as the call argument.

`void complete_all(struct completion *)` - a function called by the thread after finishing the work to notify other threads that were waiting for this completion. It takes the address of the completion variable as the call argument.

Listing 4 contains the source code of an example module in which threads use a completion variable to synchronize their work.

Listing 4: Example module presenting the use of completion variables

```
#include<linux/module.h>
#include<linux/kthread.h>
#include<linux/wait.h>
#include<linux/completion.h>

enum thread_index {WAKING_THREAD, WRITER_THREAD, READER_THREAD};

static struct thread_structure
{
        struct task_struct *thread[3];
} threads;

static wait_queue_head_t wait_queue;
static bool condition;
static DECLARE_COMPLETION(number_completion);
static int number;

static int writer_thread(void *data)
{
        DEFINE_WAIT(wait);
        for(;;) {
                number++;
                complete(&number_completion);
                add_wait_queue(&wait_queue,&wait);
                while(!condition) {
                        prepare_to_wait(&wait_queue,&wait,TASK_INTERRUPTIBLE);
                        if(kthread_should_stop())
                                return 0;
                        printk(KERN_INFO "[writer_thread]: awake\n");
                        schedule();
                }
                condition=false;
                finish_wait(&wait_queue,&wait);
        }
}

static int reader_thread(void *data)
{
        DEFINE_WAIT(wait);
        for(;;) {
                wait_for_completion(&number_completion);
                pr_info("[reader_thread] Number value: %d\n",number);
                if(kthread_should_stop())
                        return 0;
                schedule();
        }
}

static int waking_thread(void *data)
{
        for(;;) {
```

```
52                  if(kthread_should_stop())
53                          return 0;
54                  set_current_state(TASK_INTERRUPTIBLE);
55                  if(schedule_timeout(1*HZ))
56                          printk(KERN_INFO "Signal received!\n");
57                  condition=true;
58                  wake_up(&wait_queue);
59          }
60
61  }
62
63  static int __init threads_init(void)
64  {
65          init_waitqueue_head(&wait_queue);
66          threads.thread[READER_THREAD] = kthread_run(reader_thread,NULL,"reader_thread");
67          threads.thread[WRITER_THREAD] = kthread_run(writer_thread,NULL,"writer_thread");
68          threads.thread[WAKING_THREAD] = kthread_run(waking_thread,NULL,"waking_thread");
69          return 0;
70  }
71
72  static void __exit threads_exit(void)
73  {
74          kthread_stop(threads.thread[READER_THREAD]);
75          kthread_stop(threads.thread[WAKING_THREAD]);
76          kthread_stop(threads.thread[WRITER_THREAD]);
77  }
78
79  module_init(threads_init);
80  module_exit(threads_exit);
81
82  MODULE_LICENSE("GPL");
83  MODULE_DESCRIPTION("An example of using the kernel linux threads and a completion variable.");
84  MODULE_AUTHOR("Arkadiusz Chrobot <a.chrobot@tu.kielce.pl>");
```

The source code of this module, although based on others presented in this guide, is significantly different from the previous examples. The roles of the event-enabled threads have changed. Until now, these threads were equal in their functions. Now one of them has become a *writer* (producer) and the other has become a *reader* (consumer). This is reflected in the names of the enumerated type `thread_index` in line 6. However, the most important changes have occurred in thread functions, which are now called `wirter_thread()` and `reader_thread()`. Both use the completion variable that has been declared and initiated in row 15. Using this variable also requires enabling the `linux/completion.h` header file (line 4). The writer's thread increases the value of the `number` variable (line 22), which is shared by both threads, and signals the end of its work (line 23). Then it goes into the waiting state, just like the other threads described so far. The function executed by the reader thread is much simpler than other thread functions presented so far. In the `for` loop, it waits for the writer's thread to finish work (line 41), and then prints the value of the `number` variable, checks whether the thread under which it is being implemented should not end and calls the `schedule()` function. The rest of the module code is similar to other modules previously presented.

2.4. Sequential locks

Sequential locks are an example of synchronization mechanisms optimized for the problem of readers and writers in which writers have priority. These are variables that act as counters. The writer thread increases the value of this counter before and after modifying the resource. The reader thread reads the value of this variable before and after reading the state of the resource. If the reader receives two identical counter values, it means that the read operation has not been interlaced with the write operation. Only in this case the status of resource that has been read is correct. If these two counter values differ, the read operation has to be repeated. These locks are used in cases where the number of resource modifications

exceeds the number of readings.

Sequential locks are variables of the `seqlock_t` type, which, together with the subprograms that support these variables, is defined in the `linux/seqlock.h` header file. These subprograms include:

`DEFINE_SEQLOCK(x)` - a macro that initiates a sequential lock, with the name given as a call argument.

`unsigned read_seqbegin(const seqlock_t *sl)` - an `inline` function that is called by the reader before reading the shared resource. It takes the sequential lock address as the call argument and returns its value.

`unsigned read_seqretry(const seqlock_t *sl, unsigned start)` - an `inline` function that the reader calls after reading the shared resource. It takes the sequential lock address as the first argument, and the value returned by `read_seqbegin()` as the second argument. The function returns zero if the lock value read by it is equal to the value received by the second call argument. Otherwise, it returns a non-zero value.

`void write_seqlock(seqlock_t *sl)` - an `inline` function called by the writer before modifying the shared resource. It increases the value of the sequential lock by one, whose address is given as the call argument.

`void write_sequnlock(seqlock_t *sl)` - an inline function called by the writer after the modification of the shared resource. It increases the value of the sequential lock by one, whose address is given as the call argument.

Listing 5 shows the source code of an example module in which threads use the sequential lock mechanism.

Listing 5: Example module presenting the use of sequential locks

```
#include<linux/module.h>
#include<linux/kthread.h>
#include<linux/wait.h>
#include<linux/seqlock.h>

enum thread_index {WAKING_THREAD, WRITER_THREAD, FIRST_READER_THREAD, SECOND_READER_THREAD};

static struct thread_structure
{
        struct task_struct *thread[4];
} threads;

static wait_queue_head_t wait_queue;
static bool condition;
static int number;
static DEFINE_SEQLOCK(number_lock);
static const int first_reader_number = 1, second_thread_number = 2;

static int reader_thread(void *data)
{
        DEFINE_WAIT(wait);
        unsigned long int seqlock_value = 0;
        int local_number = 0;
        for(;;) {
                do {
                                seqlock_value = read_seqbegin(&number_lock);
                                local_number = number;
                } while(read_seqretry(&number_lock, seqlock_value));
                pr_info("[reader_number: %d] Value of \"number\" variable: %d\n",
                                                        *(int *)data,local_number);
                add_wait_queue(&wait_queue,&wait);
```

```
                while(!condition) {
                        prepare_to_wait(&wait_queue,&wait,TASK_INTERRUPTIBLE);
                        if(kthread_should_stop())
                                return 0;
                        pr_info("[reader_thread %d]: awake\n", *(int *)data);
                        schedule();
                }
                condition=false;
                finish_wait(&wait_queue,&wait);
        }
}

static int writer_thread(void *data)
{
        for(;;) {
                write_seqlock(&number_lock);
                number++;
                write_sequnlock(&number_lock);
                if(kthread_should_stop())
                        return 0;
                set_current_state(TASK_INTERRUPTIBLE);
                if(schedule_timeout(HZ>>2))
                        pr_info("Signal received!\n");
                pr_info("[writer_thread] awake!\n");
        }
}

static int waking_thread(void *data)
{
        for(;;) {
                if(kthread_should_stop())
                        return 0;
                set_current_state(TASK_INTERRUPTIBLE);
                if(schedule_timeout(1*HZ))
                        pr_info("Signal received!\n");
                condition=true;
                wake_up_all(&wait_queue);
        }

}

static int __init threads_init(void)
{
        init_waitqueue_head(&wait_queue);
        threads.thread[WRITER_THREAD] = kthread_run(writer_thread,NULL,"writer_thread");
        threads.thread[FIRST_READER_THREAD] =
                kthread_run(reader_thread,(void *)&first_reader_number,"first_reader_thread");
        threads.thread[SECOND_READER_THREAD] =
                kthread_run(reader_thread,(void *)&second_thread_number,"second_reader_thread");
        threads.thread[WAKING_THREAD] = kthread_run(waking_thread,NULL,"waking_thread");
        return 0;
}

static void __exit threads_exit(void)
{
        kthread_stop(threads.thread[WAKING_THREAD]);
        kthread_stop(threads.thread[WRITER_THREAD]);
        kthread_stop(threads.thread[FIRST_READER_THREAD]);
        kthread_stop(threads.thread[SECOND_READER_THREAD]);
}
```

```
92
93  module_init(threads_init);
94  module_exit(threads_exit);
95
96  MODULE_LICENSE("GPL");
97  MODULE_DESCRIPTION("An example of using the kernel linux threads and a seqential lock.");
98  MODULE_AUTHOR("Arkadiusz Chrobot <a.chrobot@tu.kielce.pl>");
```

The structure of the source code for this module is similar the source code of the module from Listing 4, but it contains several significant changes. The first is to include the header file to support sequential locks (line 4). This module will create not one but two reader threads, which requires additional element in the descriptor table in the `thread_structure` structure (line 10), and changing the enumerated type `thread_index` (line 6). The variable to which access will be protected by the sequential lock is, as before, the variable `number` (line 15). The sequential lock is declared and initiated in line 16. In line 17, two constant variables with values `1` and `2` are defined. Readers threads will perform the same function (`read_thread()`), therefore, the addresses of those constants will be used as call arguments to distinguish those threads (lines 78 and 80). The mentioned function has two new local variables `seqlock_value` of type `unsigned long int` and `local_number` of type `int`. The first will be used to store the sequence lock value read by the `read_seqbegin()` function, and the second will be used to store the value read from the `number` variable. Lines 25-28 perform a `do…while()` loop in which the initial value of the sequential lock and the value of `number` variable are read and stored. Then the sequence lock value is read again and compared to its initial value in line 28, using the `read_seqretry()` function. The result of this comparison is also a stop-condition for the mentioned `do…while` loop. If it is different from zero, then the reading will be repeated - the loop will make another iteration, and if not then its execution will end. After finishing this loop, the function puts a message in the kernel buffer containing the value of the `number` variable and the thread identifier given to the function by the `data` argument. The writer thread executes the `writer_thread()` function. It is similar the one in Listing 4, but this time it puts the thread to sleep for a specific period of time, more precisely for $\frac{1}{4}$ of a second. Thus, the writer's thread is four times more active than both readers threads. The writer calls both of the functions described above in the `for` loop (lines 47 and 49) and increases the value of the `number` variable by one (line 48).

2.5. RCU

The name RCU comes from the first letters of the *Read-Copy-Update* words. It solves the problem of synchronizing readers and writers access to a shared resource, in which readers have a higher priority. Several conditions must be met for the RCU to be used:

1. the protected resource must be accessible to threads only through a pointer,
2. reader threads in the critical section cannot go into waiting state, i.e. their execution cannot be suspended,
3. writing of the protected resource should be sporadic and readings should be more frequent.

The RCU mechanism allows effective synchronization of access to a shared resource, at the expense of a certain overhead of operating memory. Please note that if there is more than one writer thread, their access to the resource is not synchronized by this mechanism and another additional mechanism should be used to provide such synchronization.

The principle of the RCU mechanism is that the reader thread acquires a pointer to the resource, reads it and signals the end of the reading. To modify a resource the writer thread creates a copy of the resource, saves it and then publishes the pointer to that new resource copy. The original resource is deleted only after all readers have finished reading it. The next read operations will apply to the latest copy.

The subprograms responsible for handling the RCU mechanism are declared or defined in the `linux/rcupdate.h` header file. These include, among the others:

`void rcu_read_lock(void)` - an `inline` function that is called by the reader before starting the critical section (reading the resource).

`void rcu_read_unlock(void)` - an `inline` function that is called by the reader after the critical section (reading the resource).

`rcu_dereference(p)` - a macro that allows the reader to access a shared resource by dereferencing the main pointer to the resource that is passed to him by the argument.

`rcu_assign_pointer(p, v)` - a macro that allows the writer to publish the pointer to the new version (copy) of the modified resource. Its first call argument is the main pointer to the resource, and its second argument is a copy of the resource.

`void synchronize_rcu(void)` - a function that suspends the writer until all readers have finished using the previous version of the shared resource.

`void call_rcu(struct rcu_head *head, rcu_callback_t func)` - a function used by the writer to register a (callback) function that will be called after readers have finished using the current version of the shared resource. This function can delete current version of the resource. Most often, this solution is used when the protected resource is a structure. This structure must contain a field of type `struct rcu_head`. The function that will be called after the readers stop using the resource should have one call argument which is a pointer to a `struct rcu_head` and should not return any value. It can obtain a pointer to the structure that should be released from the `struct rcu_head` pointer by using the `container_of` macro[3]. The `call_rcu` function takes two call arguments: a pointer to a `struct rcu_head` structure and a pointer to a function to be called after readers have finished using the shared resource.

Listing 6 contains the source code of the module in which threads use the RCU mechanism.

Listing 6: Example module presenting the use of RCU

```
#include<linux/module.h>
#include<linux/kthread.h>
#include<linux/wait.h>
#include<linux/slab.h>
#include<linux/rcupdate.h>

enum thread_index {WAKING_THREAD, WRITER_THREAD, FIRST_READER_THREAD, SECOND_READER_THREAD};

static struct thread_structure
{
        struct task_struct *thread[4];
} threads;

static wait_queue_head_t wait_queue;
static bool condition;
static int *number_pointer;
static const int first_reader_number = 1, second_thread_number = 2;

static int reader_thread(void *data)
{
        int *local_number_pointer = NULL;
        for(;;) {
                rcu_read_lock();
                local_number_pointer = rcu_dereference(number_pointer);
                if(local_number_pointer)
                        pr_info("[reader_number: %d] Value of \"number\" variable: %d\n",
                                                        *(int *)data,*local_number_pointer);
                rcu_read_unlock();
                if(kthread_should_stop())
                        return 0;
```

[3]This macro has similar call arguments to `list_entry`, described in the kernel data structures guide.

```
                set_current_state(TASK_INTERRUPTIBLE);
                if(schedule_timeout(HZ>>2))
                        pr_info("Signal received!\n");
        }
}

static int writer_thread(void *data)
{
        int *local_number_pointer = NULL;
        int number = 0;
        DEFINE_WAIT(wait);
        for(;;) {
                void *old_pointer = NULL;
                local_number_pointer = kmalloc(sizeof(int),GFP_KERNEL);
                if(IS_ERR(local_number_pointer)) {
                        pr_alert("Error allocating memory: %ld\n",PTR_ERR(local_number_pointer));
                        return 0;
                }
                *local_number_pointer = number++;
                old_pointer = number_pointer;
                rcu_assign_pointer(number_pointer,local_number_pointer);
                synchronize_rcu();
                if(old_pointer)
                        kfree(old_pointer);
                add_wait_queue(&wait_queue,&wait);
                while(!condition) {
                        prepare_to_wait(&wait_queue,&wait,TASK_INTERRUPTIBLE);
                        if(kthread_should_stop())
                                return 0;
                        pr_info("[writer_thread]: awake\n");
                        schedule();
                }
                condition=false;
                finish_wait(&wait_queue,&wait);
        }
}

static int waking_thread(void *data)
{
        for(;;) {
                if(kthread_should_stop())
                        return 0;
                set_current_state(TASK_INTERRUPTIBLE);
                if(schedule_timeout(HZ))
                        pr_info("Signal received!\n");
                condition=true;
                wake_up(&wait_queue);
        }

}

static int __init threads_init(void)
{
        init_waitqueue_head(&wait_queue);
        threads.thread[WRITER_THREAD] = kthread_run(writer_thread,NULL,"writer_thread");
        threads.thread[WAKING_THREAD] = kthread_run(waking_thread,NULL,"waking_thread");
        threads.thread[FIRST_READER_THREAD] =
               kthread_run(reader_thread,(void *)&first_reader_number,"first_reader_thread");
        threads.thread[SECOND_READER_THREAD] =
               kthread_run(reader_thread,(void *)&second_thread_number,"second_reader_thread");
```

```
91          return 0;
92  }
93
94  static void __exit threads_exit(void)
95  {
96          kthread_stop(threads.thread[WAKING_THREAD]);
97          kthread_stop(threads.thread[WRITER_THREAD]);
98          kthread_stop(threads.thread[FIRST_READER_THREAD]);
99          kthread_stop(threads.thread[SECOND_READER_THREAD]);
100 }
101
102 module_init(threads_init);
103 module_exit(threads_exit);
104
105 MODULE_LICENSE("GPL");
106 MODULE_DESCRIPTION("An example of using the kernel linux threads and an RCU mechanism.");
107 MODULE_AUTHOR("Arkadiusz Chrobot <a.chrobot@tu.kielce.pl>");
```

The source code for this module is the modification of a source code from Listing 5. This time the writer's thread is activated every second, and the readers' threads every $\frac{1}{4}$ of second. Besides the `linux/rcupdate.h` header file (line 5), the module also includes the `linux/slab.h` header file (line 6), because the resource shared by threads will be a variable of type `int`, to which memory will be allocated dynamically using `kmalloc()` function and released with `kfree()` function. Line 16 defines the main pointer to the shared resource. The `writer_thread()` function performed by readers is defined in lines 19-35 of the module source code. Line 21 defines a local pointer in this function, to which the address of the shared resource will be stored. Before this happens, however, the function calls `rcu_read_lock()` to signal to the writer that it is reading the resource. Then it acquires a pointer to this resource (row 24) and checks if it is not an empty pointer. If this condition is true, it puts the read resource value in the kernel buffer, along with the thread identifier that was given to it by the `data` argument (lines 26 and 27), and then calls `rcu_read_unlock()` to inform the writer that it has finished reading the resource. The `writer_thread()` function performed within the writer's thread is more complicated. In line 40 the `number` variable is declared whose values will be assigned to subsequent copies of the shared resource. Line 39 defines a local pointer to the copy of the shared resource that has been created. Inside the `for` loop, in line 43, a pointer is declared in which the address of the previous copy of the shared resource will be stored. In line 44, the writer's thread creates a new copy of the shared resource, allocating memory for it with `kmalloc()`. If this allocation succeeds, this resource is given the value of variable `number`, which is then increased by one in this variable (line 49). In line 50 the address of the copy of the resource is stored in the `old_pointer` pointer, which is currently available to readers. It is copied from the main pointer. Please note that unlike readers, the writer does not have to obtain this address in a special way. It simply copies it using the assignment operator. In line 51, the writer publishes the address of a new copy of the resource by rewriting it from the `local_number_pointer` pointer to the `number_pointer` pointer (main shared resource pointer) by calling the `rcu_assign_pointer()` function. The writer then calls the `synchronize_rcu()` function to wait for readers to stop using the previous copy of the shared resource. If this happens, it will release memory for this resource using the `old_pointer` pointer (line 54), making sure that it was not empty before (line 53). The rest of the function code and the rest of the module are similar to those presented earlier in this document.

Please, read the source codes of the modules and pay special attention to the order in which each thread starts and ends! It is very important. If threads are terminated in the wrong order, then the kernel may crash.

Exercises

1. `[3 points]` Demonstrate the execution of indivisible operations on bits.
2. `[5 points]` Solve the producer-consumer problem with the use of a list and a completion variable. Note that if a list is used, then the producer does not need to be suspended.

3. [7 points] Modify the module in Listing 6 so that it uses the `call_rcu()` function to delete the previous version of the shared resource.

4. [3 points] Change the source code of the module from Listing 1 so that it uses the function `kthread_create()` instead of `kthread_run()`.

5. [5 points] Solve the producer-consumer problem with the use of the FIFO queue, atomic operations on `atomic_t` variable and a mutex.

6. [7 points] Solve the producer-consumer problem using a 10-element array, mutex and completion variables.

7. [3 points] Change the module in Listing 4 so that the reader thread uses the `wait_for_completion()` function instead of the `wait_for_completion_timeout()` function and puts into buffer an information on how many system clock cycles it waited.

8. [5 points] Change the module from Listing 4 so that the reader thread indicates to the writer thread that he has already finished reading the variable.

9. [7 points] Modify the module source code from Listing 6 so that two writer threads will run in it.