# Laboratory 6: „Tasklets and work queues” (one week)

Arkadiusz Chrobot, PhD
Karol Tomaszewski, PhD

April 29, 2024

# Contents



# Introduction

Tasklets and work queues are mechanisms classified as so-called „bottom halves", i.e. elements of the interrupt handling, in which developers place activities related to interrupt support, especially those that can be postponed and that are time-consuming. These mechanisms can also be used for purposes other than interrupt handling. Chapter 1 of this guide provides information on tasklets and how to use them. Chapter 2 contains a description of work queues. The last chapter of this document contains a list of tasks to be solved as part of the laboratory.

# 1. Tasklets

Tasklets are a mechanism that allows to postpone actions that must be performed in the context of an interrupt, but due to time constraints, they cannot be implemented in the interrupt handling procedure („upper half"). They do not necessarily have to be related to the interrupt handling, but they are most often used for this purpose. In multiprocessor computer systems, a given type of tasklet can be performed on one processor at a time. Because it operates in the context of an interrupt, it must not contain any actions that might require to suspended the execution. The advantage of tasklets is that they can be used in kernel modules. There are two types of tasklets available to programmers: regular and high priority tasklets. The latter are always executed before the first. Execution of the tasklet is a one-time process, i.e. to re-execute a tasklet it must be re-scheduled.

## 1.1. API description

The tasklet API is located in the `linux/interrupt.h` header file. Tasklets are represented in the system kernel by a variables of type `struct tasklet_struct`, which can be created dynamically (while the module is running) or statically (when the module is compiled). Actions that should be performed as part of the tasklet must be placed in a function whose prototype is as follows:

```
void func(unsigned long);
```

The following functions and macros have been defined to handle tasklets:

`DECLARE_TASKLET(name, func, data)` - a macro that is used to statically create and initiate tasklets. Its first argument is the name of the tasklet (the name of a struct `struct tasklet_struct` variable that will be created by this macro), the second is a pointer to a function that will be executed within the tasklet, and the third argument is an `unsigned long int` value that will be used as an input data for this function.

`DECLARE_TASKLET_DISABLED(name, func, data)` - a macro that works similarly to DECLARE_TASKLET and takes the same call arguments, but creates a tasklet that is disabled by default, i.e. after scheduling it will not be immediately executed until it is enabled.

`void tasklet_init(struct tasklet_struct *t, void (*func)(unsigned long), unsigned long data)` - a function that initializes a variable of type `sturct tasklet_struct`. Its first call argument is the address of this variable, the second argument is the address of the function that will be executed within the tasklet, and the third argument is an `unsigned long int` value that will be used as an input data for this function.

`void tasklet_schedule(struct tasklet_struct *t)` - an `inline` function that is used to schedule regular tasklets to be executed. As the call argument, it takes the address of a `struct tasklet_struct` variable that represents the tasklet.

`void tasklet_hi_schedule(struct tasklet_struct *t)` - an `inline` function that is used to schedule high priority tasklets. It takes the same call argument as `tasklet_schedule()`.

`void tasklet_disable(struct tasklet_struct *t)` - an `inline` function that blocks (disables) the scheduled tasklet for execution. As the call argument, it takes the address of a `struct tasklet_struct` variable that represents the tasklet. If the tasklet is currently being executed, the function suspends its operation until the tasklet is completed.

`void tasklet_disable_nosync(struct tasklet_struct *t)` - an `inline` function that blocks (disables) the scheduled tasklet for execution. It takes the same argument as `tasklet_disable()`, but unlike the latter it terminates, regardless of whether the tasklet-related function was in progress or not.

`void tasklet_enable(struct tasklet_struct *t)` - an `inline` function that unlocks (enables) the tasklet for execution. As the call argument, it takes the address of a `struct tasklet_struct` representing the tasklet.

`void tasklet_kill(struct tasklet_struct *t)` - a function that removes a scheduled tasklet from the queue. As the call argument, it takes the address of a `struct tasklet_struct` representing the tasklet.

## 1.2. Example

Listing 1 contains the source code of the module, that creates and schedules four tasklets for execution. Two of them are regular tasklets and the other two are high priority tasklets.

**Listing 1:** Example module using two types tasklets

```
1  #include<linux/module.h>
2  #include<linux/interrupt.h>
3
4  static void normal_tasklet_handler(unsigned long int data)
5  {
6          pr_info("Hi! I'm a tasklet of a normal priority. My ID is: %lu\n",data);
7  }
8
9  static void privileged_tasklet_handler(unsigned long int data)
10 {
11         pr_info("Hi! I'm a tasklet of a high priority. My ID is: %lu\n", data);
12 }
13
14 static DECLARE_TASKLET(normal_tasklet_1,normal_tasklet_handler,0);
15 static DECLARE_TASKLET(normal_tasklet_2,normal_tasklet_handler,1);
16 static DECLARE_TASKLET(privileged_tasklet_1,privileged_tasklet_handler,0);
17 static DECLARE_TASKLET(privileged_tasklet_2,privileged_tasklet_handler,1);
18
19
20 static int __init tasklets_init(void)
21 {
22         tasklet_schedule(&normal_tasklet_1);
23         tasklet_schedule(&normal_tasklet_2);
```

```
24          tasklet_hi_schedule(&privileged_tasklet_1);
25          tasklet_hi_schedule(&privileged_tasklet_2);
26          return 0;
27  }
28
29  static void __exit tasklets_exit(void)
30  {
31          tasklet_kill(&normal_tasklet_1);
32          tasklet_kill(&normal_tasklet_2);
33          tasklet_kill(&privileged_tasklet_1);
34          tasklet_kill(&privileged_tasklet_2);
35  }
36
37  module_init(tasklets_init);
38  module_exit(tasklets_exit);
39  MODULE_LICENSE("GPL");
40  MODULE_AUTHOR("Arkadiusz Chrobot <a.chrobot@tu.kielce.pl>");
41  MODULE_DESCRIPTION("A module demonstrating the use of tasklets.");
42  MODULE_VERSION("1.0");
```

In line 2 of the module source code, the `linux/interrupt.h` header file containing tasklet API is included. Lines 4-7 contain the definition of the function executed within a regular tasklets, and lines 9-12 provide the definition of the function executed within high priority tasklets. They differ only in two elements: the name and message placed in the kernel buffer. Besides that, the code for these functions is the same. The tasklet's priority is not determined by the functions it performs, but by the way it has been scheduled for execution. Lines 14-17 contain a declaration of tasklets. The first two tasklets, named `normal_tasklet_1` and `normal_tasklet_2`, will execute the `normal_tasklet_handler()` function. To distinguish them, a different values will be given to the function through the third call argument of the `DECLARE_TASKLET` macro. The first tasklet will be marked with `0` and the second tasklet will be marked with `1`. The next two tasklets named `privileged_tasklet_1` and `privileged_tasklet_2` will execute the `privileged_tasklet_handler()` function and also for distinction the first will receive the value `0` and the second the value `1`. In the module constructor, at the beginning two regular tasklets are scheduled (lines 22 and 23), followed by two high priority tasklets (lines 24 and 25). Please note that after using the `dmesg` command in the console the messages from high priority tasklets will be the first to appear on the screen, even though they were scheduled later for execution than regular tasklets. The order of messages inside of these two tasklet groups matches the order in which they are scheduled. In the destructor, all tasklets are removed from the queues in which they were scheduled. This action is necessary in case the module is removed from the system kernel before the related tasklets can execute. If the destructor would not remove them, at the time of execution they would try to perform related functions that would no longer be available at that time, and this could destabilize the operating system. Executing the `tasklet_kill()` function for a tasklet that has already been performed is not an error and has no side effects. However, if the tasklet has not yet been executed, this function will safely remove it from the queue in which it is scheduled.

## 2. Work queues

Work queues, like tasklets are a mechanism that allows to delay the execution of interrupt handling activities, but they can also be used for other purposes. Unlike tasklets, work queues perform their operations in the context of a process, not an interrupt. This means that their execution can be suspended waiting for a specific event from other kernel subsystems. The context for work queues is provided by a specialized kernel thread called a *worker thread*. By default, there are as many worker threads associated with a single work queue as the number of processors on the computer. Simply said, the work queue mechanism can be described as a system of lists and threads related to them. In the lists that act as FIFO queues there are structures placed that contain pointers to the functions performing specific operations. Work threads remove structures from the list one by one and call the related functions. This action ends when the list is emptied. Each work is carried out only once. Than means after it is being scheduled and executed it must be scheduled again to repeat the execution. Work queues can do two types of jobs.

The first type is a work whose execution is delayed to an unspecified moment. For a simplicity we will call them *postponed works*. The second type is a work whose implementation has been postponed by a certain period of time, e.g. for one second. We will refer to them as *delayed works*. Please also note that **only modules whose code is available under the GPL may use the queue mechanism**.

## 2.1. API description

The `linux/workqueue.h` header file contains API related to work queue support. Work queues are represented by the `struct workqueue_struct` type. Individual postponed works are described by structures of type `struct work_struct`, while delayed works are described by structures of type `struct delayed_work`. Operations to be performed as part of any type of work must be included in the function with the following prototype:

```
void work_handler(struct work_struct *work)
```

The name of the function in the above prototype (`work_handler`) is an example and in the actual function may be different, more adequate to the work that this function is to perform. Please note the argument type of this function. It is the same for postponed and delayed works. This is possible because the `struct delayed_work` structure contains as one of its fields the `struct work_struct` structure. The following macros and functions have been defined to support work queues:

- `create_workqueue(name)` - a macro that creates a work queue by creating as many threads for it, as the number of processors (cores) on the computer. It returns a pointer to a structure of type `struct workqueue_struct`, and takes the name of the queue as a call argument. Also, each of the worker threads associated with such a queue will have that name.
- `create_singlethread_workqueue(name)` - this macro, similar to `create_workqueue` macro, creates a work queue, but with only one worker thread that is executed on the first processor in the computer. Using this macro is the kernel programmer's preferred way of creating work queues due to a fact that the kernel thread creation and handling is computationally expensive.
- `void destroy_workqueue(struct workqueue_struct *wq)` - a function that deletes a work queue created using one of the previously described macros. It returns no value, and takes the pointer to the queue being deleted as the call argument.
- `DECLARE_WORK(n, f)` - a macro that creates and initializes a structure of type `struct work_struct`. Its first call argument is the name of the structure (identifier), and the second argument is the pointer for the function containing the code to be executed as part of the postponed work.
- `DECLARE_DELAYED_WORK(n, f)` - a macro that creates and initializes a structure of type `struct delayed_work`. It has the same call arguments as `DECLARE_WORK`.
- `INIT_WORK(_work, _func)` - a macro that initializes a structure of type `struct work_struct`. It takes two call arguments. The first is the pointer for the structure to be initialized, and the second is the pointer for the function containing the code to be executed as part of the postponed work. The initialized structure can be created in a dynamic or static way.
- `INIT_DELAYED_WORK(_work, _func)` - this macro, similar to `INIT_WORK`, initializes the structure, but this time of type `struct delayed_work`.
- `bool queue_work(struct workqueue_struct *wq, struct work_struct *work)` - an `inline` function that schedules the postponed work into the specified work queue. It takes two call arguments: a pointer to a work queue and a pointer to a `struct work_struct` associated with this work. Returns `false` if the work was previously scheduled in this queue, or `true` if the work was successfully scheduled.
- `bool queue_delayed_work(struct workqueue_struct *wq, struct delayed_work *dwork, unsigned long delay)` - an inline function that schedules a delayed work to the indicated queue. It returns the same value as `queue_work()` and takes the same first call argument. The second argument is the pointer to a structure of type `delayed_work` with which this work is associated, and the third call argument is the delay value expressed in the number of a system clock cycles.
- `bool queue_work_on(int cpu, struct workqueue_struct *wq, struct work_struct *work)` - the function works like `queue_work()`, but as the first call argument it takes the number of the processor on which the delayed work should be executed by the work thread. It is useful in multiprocessor systems.

`bool queue_delayed_work_on(int cpu, struct workqueue_struct *wq, struct delayed_work *work, unsigned long delay)` - a function that works similarly to `queue_delayed_work()`, but as the first call argument it takes the number of the processor on which the delayed work should be executed by the work thread. It also is useful in multiprocessor systems.

`static inline bool mod_delayed_work(struct workqueue_struct *wq, struct delayed_work *dwork, unsigned long delay)` - an `inline` function that allows to change the delay of a scheduled delayed work. It takes the same call arguments as the `queue_delayed_work()` function, but the third call argument is used as the new delay value. If its value is `0`, then the work will be executed immediately. The function returns `false` if the work was not scheduled yet. Then `mod_delayed_work()` will act as the `queue_delayed_work()` besides the the returned value. If the work was scheduled and the delay was changed, the function returns `true`.

`bool mod_delayed_work_on(int cpu, struct workqueue_struct *wq, struct delayed_work *dwork, unsigned long delay)` - a function that works similar to `mod_delayed_work()`, but as the first call argument it takes the number of the processor on which the delayed work should be executed by the work thread.

`bool cancel_work_sync(struct work_struct *work)` - this function cancels the execution of the scheduled postponed work. If the function associated with this work is in progress, then the `cancel_work_sync()` function will wait for its completion. As the call argument, it takes a pointer to a `struct work_struct` structure associated to the work to be cancelled. The function returns `true` if the work was previously scheduled and now successfully cancelled or `false` otherwise.

`bool cancel_delayed_work(struct delayed_work *dwork)` - this function cancels delayed work, but if the function associated with this work is in progress, it does not wait for it to finish. As call arguments it takes a pointer to a `struct delayed_work` structure representing the work to be cancelled, and returns `true` if the work was previously scheduled and now cancelled. The function returns `false` otherwise.

`bool cancel_delayed_work_sync(struct delayed_work *dwork)` - this function is a safer version of `cancel_delayed_work()`, because unlike its counterpart, if the work-related function is already in progress, it waits for it to finish.

`bool flush_work(struct work_struct *work)` - this function is waiting for the scheduled work to be completed. It returns `true` if the work was scheduled and completed or `false` if it was not scheduled. As the call arguments this function takes a pointer for the postponed work.

`bool flush_delayed_work(struct delayed_work *dwork)` - it is an equivalent to the `flush_work()` function for delayed works represented by the `struct delayed_work()` structure, but besides waiting for the completion of delayed work it forces work to be executed immediately.

`void flush_workqueue(struct workqueue_struct *wq)` - this function empties the selected queue, i.e. it forces the execution of all works in the queue that has been scheduled until it was called and waits for it to finish. It takes the pointer to the work queue as the call argument. Execution of this function is costly and, in the case of complex works, can even be considered dangerous. It is better to wait for the execution of individual works, or cancel them by using the previously described functions that are dedicated for this purpose.

## 2.2. Default work queue API description

The Linux kernel has a default work queue that is served by worker threads called `kworker`. Information about them can be obtained using the `ps aux` command, which, in addition to the name, will also provide the ID of the processor on which the particular thread is running. Creating new work queues is expensive, therefore it is recommended to use the default work queue unless there are significant reasons to do the opposite. The following macros and functions have been defined to support the default work queue:

`bool schedule_work(struct work_struct *work)` - an `inline` function that is used to schedule the postponed work into a default queue. As a call argument, it takes a pointer to a structure of type `struct work_struct` representing this work. The function returns `true` if the work was successfully scheduled or `false` if it was already scheduled previously.

`bool schedule_work_on(int cpu, struct work_struct *work)` - an `inline` function that, similarly to `schedule_work()` function, schedules postponed work to the default queue, indicating additionally on which processor it should be executed. This function is useful for multiprocessor computers. It takes two call arguments. The first is the processor number and the second is a pointer to a `struct work_struct` structure representing postponed work. The function returns the same values as `schedule_work()`.

`bool schedule_delayed_work(struct delayed_work *dwork, unsigned long delay` - an `inline` function that schedules delayed work to the default queue. It takes two call arguments. The first argument is a pointer to a structure of type `struct delayed_work` that represents the delayed work, and the second argument is the delay time expressed in the number of system clock cycles. The function returns `true` if the work has been successfully scheduled or `false` if the work was already scheduled previously.

`bool schedule_delayed_work_on(int cpu, struct delayed_work *dwork, unsigned long delay)` - this function works like `schedule_delayed_work()`, but its first call argument is the number of processor that should execute the delayed work.

`void flush_scheduled_work(void)` - a function that empties the default work queue. It forces the execution of all works in the default queue that has been scheduled until it was called and waits for it to finish. Its execution can be computationally expensive, similar to `flush_workqueue()`, and even considered dangerous in the case of complex works.

## 2.3. Example

Listing 2 contains the source code for the kernel module, that presents how to manage and use a work queue.

**Listing 2:** Example module using work queue

```
#include<linux/module.h>
#include<linux/workqueue.h>

static struct workqueue_struct *queue;

static void normal_work_handler(struct work_struct *work)
{
        pr_info("Hi! I'm handler of normal work!\n");
}

static void delayed_work_handler(struct work_struct *work)
{
        pr_info("Hi! I'm handler of delayed work!\n");
}

static DECLARE_WORK(normal_work, normal_work_handler);
static DECLARE_DELAYED_WORK(delayed_work, delayed_work_handler);

static int __init workqueue_module_init(void)
{
        queue = create_singlethread_workqueue("works");
        if(IS_ERR(queue)) {
                pr_alert("[workqueue_module] Error creating a work queue: %ld\n",PTR_ERR(queue));
                return -ENOMEM;
        }

        if(!queue_work(queue,&normal_work))
                pr_info("The normal work was already queued!\n");
        if(!queue_delayed_work(queue,&delayed_work,10*HZ))
                pr_info("The delayed work was already queued!\n");
```

```
31
32          return 0;
33  }
34
35  static void __exit workqueue_module_exit(void)
36  {
37          if(!IS_ERR(queue)) {
38                  if(cancel_work_sync(&normal_work))
39                          pr_info("The normal work has not been done yet!\n");
40                  if(cancel_delayed_work_sync(&delayed_work))
41                          pr_info("The delayed work has not been done yet!\n");
42                  destroy_workqueue(queue);
43          }
44  }
45
46  module_init(workqueue_module_init);
47  module_exit(workqueue_module_exit);
48  MODULE_LICENSE("GPL");
49  MODULE_AUTHOR("Arkadiusz Chrobot <a.chrobot@tu.kielce.pl>");
50  MODULE_DESCRIPTION("A module demonstrating the use of work queues.");
51  MODULE_VERSION("1.0");
```

In line 2 the header file `linux/workqueue.h` is included in the module. Line 4 contains the declaration of the work queue pointer. Lines 6-9 define the `normal_work_handler()` function, which contains the code to be executed as part of the postponed work. In case of this function it consists of placing the appropriate message in the kernel buffer. Similarly, lines 11-14 define the `delayed_work_handler()` function containing the code to be executed within the delayed work. Like its counterpart for postponed work, it places the appropriate message in the kernel buffer. On line 16, a structure of type `struct work_struct` named `normal_work` is declared and initialized. As a result of initialization, it is associated with the `normal_work_handler()` function. In line 17, a structure of type `struct delayed_work` named `delayed_work` is declared and initialized. As a result of this initialization, it is associated with the `delayed_work_handler()` function.

In the module constructor, a work queue called „works" is created, that will be served by a single worker thread (line 21). Its address is stored in the `queue` pointer. If this queue could not be created, then the module constructor will put the appropriate message in the kernel buffer and terminate with an error (lines 22-25). In line 27, the postponed work is scheduled into the created work queue. If it was previously scheduled, the constructor will put an information about this in the kernel buffer (line 28). Similar actions are performed in lines 29 and 30 of the module source code, but this time they concern delayed work. The delay in this case is 10 seconds.

In the destructor, in line 38, the postponed work is cancelled. If this cancellation is successful, it will mean that the work was previously scheduled, but not yet completed, and on the line 39 the destructor will put a message in the kernel buffer informing about it. Similar operations are performed for delayed work in lines 40 and 41 of the module source code. In line 42, the work queue is deleted. The module can perform this last action in a safe manner, because the execution of instructions from the lines described earlier guarantees that the work queue is empty when it is deleted.

The module's execution is convenient to track using the `dmesg -w -d` command. Please note the messages that appear in the kernel buffer when the module is removed before 10 seconds from loading it and after that time.

## Exercises

1. `[3 points]` Demonstrate the functionality of the `DECLARE_TASKLET_DISABLE` macro and the `tasklet_enable()`, `tasklet_disable()` and `tasklet_disable_nosync()` functions.

2. `[5 points]` Create an automatically repetitive tasklet, i.e. a tasklet that after execution will schedule itself again for next execution.

3. [7 points] Using the knowledge about lists, threads and synchronization mechanisms create a simplified implementation of work queues mechanism i.e. without support for delayed works and with only one worker thread.

4. [3 points] Check whether it is possible to use the work queue mechanism in the module, which is not available under the GPL license.

5. [5 points] Create automatically repetitive delayed work, i.e. one that after execution will schedule itself again for next execution.

6. [7 points] Write a module that will use tasklets and create a file in the `/proc` directory containing statistics on how many tasklets with what priority has been created, scheduled and executed within the module, or is still waiting execution. Structures representing tasklets should be created and deleted automatically using the slab allocator.

7. [3 points] Rewrite the module from Listing 2 so that it uses the default work queue.

8. [5 points] Rewrite the module from Listing 2 so that it uses the `alloc_workqueue()` function, instead of the `create_singlethread_workqueue` macro. The function has been described in the lecture. Create an unbound work queue.