**Workqueues**

The workqueue is another concept for handling deferred functions. It is similar to tasklets with some differences. Workqueue functions run in the context of a kernel process, but tasklet functions run in the software interrupt context. This means that workqueue functions must not be atomic as tasklet functions. Tasklets always run on the processor from which they were originally submitted. Workqueues work in the same way, but only by default. The workqueue concept represented by the:

Очередь заданий является еще одной концепцией для обработки отложенных функций. Это похоже на тасклет с некоторыми отличиями. Функции рабочих очередей выполняются в контексте процесса ядра, но функции тасклетов выполняются в контексте программных прерываний. Это означает, что функции очереди задач не должны быть атомарными, как функции тасклета. Тасклеты всегда выполняются на процессоре, с которого они были отправлены. Рабочие очереди работают таким же образом, но только по умолчанию. Концепция рабочей очереди представлена:

struct worker\_pool {

spinlock\_t lock;

int cpu;

int node;

int id;

unsigned int flags;

struct list\_head worklist;

int nr\_workers;

...

...

...

structure that is defined in the [kernel/workqueue.c](https://github.com/torvalds/linux/blob/16f73eb02d7e1765ccab3d2018e0bd98eb93d973/kernel/workqueue.c) source code file in the Linux kernel. I will not write the source code of this structure here, because it has quite a lot of fields, but we will consider some of those fields.

In its most basic form, the work queue subsystem is an interface for creating kernel threads to handle work that is queued from elsewhere. All of these kernel threads are called -- worker threads. The work queue are maintained by the work\_struct that defined in the [include/linux/workqueue.h](https://github.com/torvalds/linux/blob/16f73eb02d7e1765ccab3d2018e0bd98eb93d973/include/linux/workqueue.h). Let's look on this structure:

структура, которая определена в файле исходного кода kernel/workqueue.c в ядре Linux. Я не буду писать здесь исходный код этой структуры, поскольку в ней достаточно много полей, но мы рассмотрим некоторые из этих полей.

В своей самой основной форме подсистема рабочей очереди представляет собой интерфейс для создания потоков ядра для обработки работы, находящейся в очереди из других источников. Все эти потоки ядра называются рабочими потоками. Рабочая очередь поддерживается work\_struct, определенной в include / linux / workqueue.h. Давайте посмотрим на эту структуру:

struct work\_struct {

atomic\_long\_t data;

struct list\_head entry;

work\_func\_t func;

#ifdef CONFIG\_LOCKDEP

struct lockdep\_map lockdep\_map;

#endif

};

Here are two things that we are interested: func -- the function that will be scheduled by the workqueue and the data - parameter of this function. The Linux kernel provides special per-cpu threads that are called kworker:

Вот две вещи, которые нас интересуют: func - функция, которая будет запланирована в рабочей очереди, и data - параметр этой функции. Ядро Linux предоставляет специальные потоки для каждого процессора, которые называются kworker:

systemd-cgls -k | grep kworker

├─ 5 [kworker/0:0H]

├─ 15 [kworker/1:0H]

├─ 20 [kworker/2:0H]

├─ 25 [kworker/3:0H]

├─ 30 [kworker/4:0H]

...

...

...

This process can be used to schedule the deferred functions of the workqueues (as ksoftirqd for softirqs). Besides this we can create new separate worker thread for a workqueue. The Linux kernel provides following macros for the creation of workqueue:

Этот процесс можно использовать для планирования отложенных функций рабочих очередей (например, ksoftirqd для softirqs). Помимо этого мы можем создать новый отдельный рабочий поток для рабочей очереди. Ядро Linux предоставляет следующие макросы для создания очереди задач:

#define DECLARE\_WORK(n,f) struct work\_struct n = \_\_WORK\_INITIALIZER(n,f)

#define DECLARE\_DELAYED\_WORK(n,f) struct delayed\_work n = \_\_DELAYED\_WORK\_INITIALIZER(n,f,0)

|  |
| --- |
|  |

#define DECLARE\_WORK(n, f) \

struct work\_struct n = \_\_WORK\_INITIALIZER(n, f)

for static creation. It takes two parameters: name of the workqueue and the workqueue function. For creation of workqueue in runtime, we can use the:

для статического создания. Он принимает два параметра: имя рабочей очереди и функцию рабочей очереди. Для создания очереди задач во время выполнения мы можем использовать:

#define INIT\_WORK(\_work, \_func) \

\_\_INIT\_WORK((\_work), (\_func), 0)

|  |  |
| --- | --- |
| #define | [**\_\_INIT\_WORK**](https://docs.huihoo.com/doxygen/linux/kernel/3.7/linux_2workqueue_8h.html#a42f798c797cec8987b7a78a1edbe8bc1)(\_work, \_func, \_onstack) |
| #define \_\_INIT\_WORK(work, func, onstack)  #define INIT\_WORK(work, func) | |
| #define | [**INIT\_WORK**](https://docs.huihoo.com/doxygen/linux/kernel/3.7/linux_2workqueue_8h.html#a6f796da8e8c0f48366eb8cc967687635)(\_work, \_func) |

#define \_\_INIT\_WORK(\_work, \_func, \_onstack)

do {

\_\_init\_work((\_work), \_onstack);

(\_work)->data = (atomic\_long\_t) WORK\_DATA\_INIT();

INIT\_LIST\_HEAD(&(\_work)->entry);

(\_work)->func = (\_func);

} while (0)

macro that takes work\_struct structure that has to be created and the function to be scheduled in this workqueue. After a work was created with the one of these macros, we need to put it to the workqueue. We can do it with the help of the queue\_work or the queue\_delayed\_work functions:

макрос, который принимает структуру work\_struct, которая должна быть создана, и функцию, которая должна быть запланирована в этой рабочей очереди. После того, как работа с одним из этих макросов была создана, мы должны поместить ее в очередь задач. Мы можем сделать это с помощью функций queue\_work или queue\_delayed\_work:

static inline bool queue\_work(struct workqueue\_struct \*wq,

struct work\_struct \*work)

{

return queue\_work\_on(WORK\_CPU\_UNBOUND, wq, work);

}

The queue\_work function just calls the queue\_work\_on function that queues work on specific processor. Note that in our case we pass the WORK\_CPU\_UNBOUND to the queue\_work\_on function. It is a part of the enum that is defined in the [include/linux/workqueue.h](https://github.com/torvalds/linux/blob/16f73eb02d7e1765ccab3d2018e0bd98eb93d973/include/linux/workqueue.h) and represents workqueue which are not bound to any specific processor. The queue\_work\_on function tests and set the WORK\_STRUCT\_PENDING\_BIT bit of the given work and executes the \_\_queue\_work function with the workqueue for the given processor and given work:

Функция queue\_work просто вызывает функцию queue\_work\_on, которая работает в очереди на конкретном процессоре. Обратите внимание, что в нашем случае мы передаем WORK\_CPU\_UNBOUND в функцию queue\_work\_on. Это часть enum, которая определена в include / linux / workqueue.h и представляет рабочую очередь, которая не связана с каким-либо конкретным процессором. Функция queue\_work\_on проверяет и устанавливает бит WORK\_STRUCT\_PENDING\_BIT для данной работы и выполняет функцию \_\_queue\_work с рабочей очередью для данного процессора и данной работы:

extern [**bool**](https://elixir.bootlin.com/linux/latest/ident/bool) [**queue\_work\_on**](https://elixir.bootlin.com/linux/latest/ident/queue_work_on)(int cpu, struct [**workqueue\_struct**](https://elixir.bootlin.com/linux/latest/ident/workqueue_struct) \***[wq](https://elixir.bootlin.com/linux/latest/ident/wq)**,

struct [**work\_struct**](https://elixir.bootlin.com/linux/latest/ident/work_struct) \*[**work**](https://elixir.bootlin.com/linux/latest/ident/work));

*/\*\**

*\* queue\_work\_on - queue work on specific cpu*

*\* @cpu: CPU number to execute work on*

*\* @wq: workqueue to use*

*\* @work: work to queue*

*\**

*\* We queue the work to a specific CPU, the caller must ensure it*

*\* can't go away.*

*\**

*\* Return: %false if @work was already on a queue, %true otherwise.*

*\*/*

*/\*\**

\* queue\_work\_on - очередь работы на конкретном процессоре

  \* @cpu: номер процессора для выполнения работы

  \* @wq: рабочая очередь для использования

  \* @work: работа в очереди

  \*

  \* Мы ставим работу в очередь на конкретный процессор, вызывающий должен убедиться в этом

  \* не может уйти.

  \*

  \* Return:% false, если @work уже был в очереди, в противном случае - % true.

\*/

[**bool**](https://elixir.bootlin.com/linux/latest/ident/bool) [**queue\_work\_on**](https://elixir.bootlin.com/linux/latest/ident/queue_work_on)(int cpu, struct [**workqueue\_struct**](https://elixir.bootlin.com/linux/latest/ident/workqueue_struct) \***[wq](https://elixir.bootlin.com/linux/latest/ident/wq)**, struct [**work\_struct**](https://elixir.bootlin.com/linux/latest/ident/work_struct) \*[**work**](https://elixir.bootlin.com/linux/latest/ident/work))

{

[**bool**](https://elixir.bootlin.com/linux/latest/ident/bool) ret = [**false**](https://elixir.bootlin.com/linux/latest/ident/false);

unsigned long flags;

[**local\_irq\_save**](https://elixir.bootlin.com/linux/latest/ident/local_irq_save)(flags);

if (!**[test\_and\_set\_bit](https://elixir.bootlin.com/linux/latest/ident/test_and_set_bit)**([**WORK\_STRUCT\_PENDING\_BIT**](https://elixir.bootlin.com/linux/latest/ident/WORK_STRUCT_PENDING_BIT), [**work\_data\_bits**](https://elixir.bootlin.com/linux/latest/ident/work_data_bits)([**work**](https://elixir.bootlin.com/linux/latest/ident/work)))) {

[**\_\_queue\_work**](https://elixir.bootlin.com/linux/latest/ident/__queue_work)(cpu, [**wq**](https://elixir.bootlin.com/linux/latest/ident/wq), [**work**](https://elixir.bootlin.com/linux/latest/ident/work));

ret = [**true**](https://elixir.bootlin.com/linux/latest/ident/true);

}

[**local\_irq\_restore**](https://elixir.bootlin.com/linux/latest/ident/local_irq_restore)(flags);

[**return**](https://elixir.bootlin.com/linux/latest/ident/return) ret;

}

[**EXPORT\_SYMBOL**](https://elixir.bootlin.com/linux/latest/ident/EXPORT_SYMBOL)(**[queue\_work\_on](https://elixir.bootlin.com/linux/latest/ident/queue_work_on)**);

bool queue\_work\_on(int cpu, struct workqueue\_struct \*wq,

struct work\_struct \*work)

{

bool ret = false;

...

if (!test\_and\_set\_bit(WORK\_STRUCT\_PENDING\_BIT, work\_data\_bits(work))) {

\_\_queue\_work(cpu, wq, work);

ret = true;

}

...

return ret;

}

The \_\_queue\_work function gets the work pool. Yes, the work pool not workqueue. Actually, all works are not placed in the workqueue, but to the work pool that is represented by the worker\_pool structure in the Linux kernel. As you can see above, the workqueue\_struct structure has the pwqs field which is list of worker\_pools. When we create a workqueue, it stands out for each processor the pool\_workqueue. Each pool\_workqueue associated with worker\_pool, which is allocated on the same processor and corresponds to the type of priority queue. Through them workqueue interacts with worker\_pool. So in the \_\_queue\_work function we set the cpu to the current processor with the raw\_smp\_processor\_id (you can find information about this macro in the fourth [part](https://0xax.gitbooks.io/linux-insides/content/Initialization/linux-initialization-4.html) of the Linux kernel initialization process chapter), getting the pool\_workqueue for the given workqueue\_struct and insert the given work to the given workqueue:

Функция \_\_queue\_work получает рабочий пул. Да, рабочий пул не рабочий. На самом деле, все работы помещаются не в рабочую очередь, а в рабочий пул, который представлен структурой worker\_pool в ядре Linux. Как вы можете видеть выше, структура workqueue\_struct имеет поле pwqs, которое является списком worker\_pools. Когда мы создаем рабочую очередь, она выделяется для каждого процессора pool\_workqueue. Каждый pool\_workqueue связан с worker\_pool, который размещен на том же процессоре и соответствует типу очереди приоритетов. Через них workqueue взаимодействует с worker\_pool. Таким образом, в функции \_\_queue\_work мы устанавливаем процессор на текущий процессор с raw\_smp\_processor\_id (вы можете найти информацию об этом макросе в четвертой части главы процесса инициализации ядра Linux), получая pool\_workqueue для заданной workqueue\_struct и вставляя данную работу в заданная workqueue:

static void \_\_queue\_work(int cpu, struct workqueue\_struct \*wq,

struct work\_struct \*work)

{

...

...

...

if (req\_cpu == WORK\_CPU\_UNBOUND)

cpu = raw\_smp\_processor\_id();

if (!(wq->flags & WQ\_UNBOUND))

pwq = per\_cpu\_ptr(wq->cpu\_pwqs, cpu);

else

pwq = unbound\_pwq\_by\_node(wq, cpu\_to\_node(cpu));

...

...

...

insert\_work(pwq, work, worklist, work\_flags);

As we can create works and workqueue, we need to know when they are executed. As I already wrote, all works are executed by the kernel thread. When this kernel thread is scheduled, it starts to execute works from the given workqueue. Each worker thread executes a loop inside the worker\_thread function. This thread makes many different things and part of these things are similar to what we saw before in this part. As it starts executing, it removes all work\_struct or works from its workqueue.

That's all.

Поскольку мы можем создавать работы и очереди работ, нам нужно знать, когда они выполняются. Как я уже писал, все работы выполняются потоком ядра. Когда этот поток ядра запланирован, он начинает выполнять работы из заданной очереди задач. Каждый рабочий поток выполняет цикл внутри функции worker\_thread. Этот поток делает много разных вещей, и часть этих вещей похожа на то, что мы видели раньше в этой части. Когда он начинает выполняться, он удаляет всю work\_struct или работает из своей очереди задач.

Это все.

**Conclusion**

It is the end of the ninth part of the [Interrupts and Interrupt Handling](https://0xax.gitbooks.io/linux-insides/content/Interrupts/index.html) chapter and we continued to dive into external hardware interrupts in this part. In the previous part we saw initialization of the IRQs and main irq\_desc structure. In this part we saw three concepts: the softirq, tasklet and workqueue that are used for the deferred functions.

The next part will be last part of the Interrupts and Interrupt Handling chapter and we will look on the real hardware driver and will try to learn how it works with the interrupts subsystem.

extern [**bool**](https://elixir.bootlin.com/linux/latest/ident/bool) [**flush\_work**](https://elixir.bootlin.com/linux/latest/ident/flush_work)(struct [**work\_struct**](https://elixir.bootlin.com/linux/latest/ident/work_struct) \*[**work**](https://elixir.bootlin.com/linux/latest/ident/work));

extern [**bool**](https://elixir.bootlin.com/linux/latest/ident/bool) [**cancel\_work\_sync**](https://elixir.bootlin.com/linux/latest/ident/cancel_work_sync)(struct [**work\_struct**](https://elixir.bootlin.com/linux/latest/ident/work_struct) \*[**work**](https://elixir.bootlin.com/linux/latest/ident/work));

extern [**bool**](https://elixir.bootlin.com/linux/latest/ident/bool) [**flush\_delayed\_work**](https://elixir.bootlin.com/linux/latest/ident/flush_delayed_work)(struct [**delayed\_work**](https://elixir.bootlin.com/linux/latest/ident/delayed_work) \*dwork);

extern [**bool**](https://elixir.bootlin.com/linux/latest/ident/bool) [**cancel\_delayed\_work**](https://elixir.bootlin.com/linux/latest/ident/cancel_delayed_work)(struct [**delayed\_work**](https://elixir.bootlin.com/linux/latest/ident/delayed_work) \*dwork);

extern [**bool**](https://elixir.bootlin.com/linux/latest/ident/bool) [**cancel\_delayed\_work\_sync**](https://elixir.bootlin.com/linux/latest/ident/cancel_delayed_work_sync)(struct [**delayed\_work**](https://elixir.bootlin.com/linux/latest/ident/delayed_work) \*dwork);

*/\*\**

*\* schedule\_work - put work task in global workqueue*

*\* @work: job to be done*

*\**

*\* Returns %false if @work was already on the kernel-global workqueue and*

*\* %true otherwise.*

*\**

*\* This puts a job in the kernel-global workqueue if it was not already*

*\* queued and leaves it in the same position on the kernel-global*

*\* workqueue otherwise.*

*\**

*\* Shares the same memory-ordering properties of queue\_work(), cf. the*

*\* DocBook header of queue\_work().*

*\*/*

static inline **[bool](https://elixir.bootlin.com/linux/latest/ident/bool)** **[schedule\_work](https://elixir.bootlin.com/linux/latest/ident/schedule_work)**(struct **[work\_struct](https://elixir.bootlin.com/linux/latest/ident/work_struct)** \***[work](https://elixir.bootlin.com/linux/latest/ident/work)**)

{

**[return](https://elixir.bootlin.com/linux/latest/ident/return)** **[queue\_work](https://elixir.bootlin.com/linux/latest/ident/queue_work)**(**[system\_wq](https://elixir.bootlin.com/linux/latest/ident/system_wq)**, [**work**](https://elixir.bootlin.com/linux/latest/ident/work));

}