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 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. 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(_work, _func, _onstack)
```

```
#define __INIT_WORK(work, func, onstack)
#define INIT_WORK(work, func)
```

```
#define INIT_WORK(_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 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 queue_work_on(int cpu, struct workqueue_struct *wq,
                          struct work_struct *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 queue_work_on(int cpu, struct workqueue_struct *wq, struct work_struct *work)
{
        bool ret = false;
        unsigned long flags;
        local_irq_save(flags);
        if (!test_and_set_bit(WORK_STRUCT_PENDING_BIT, work_data_bits(work))) {
                __queue_work(cpu, wq, work);
                ret = true;
        }
        local_irq_restore(flags);
        return ret;
}
EXPORT_SYMBOL(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 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 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 flush_work(struct work_struct *work);
extern bool cancel_work_sync(struct work_struct *work);

extern bool flush_delayed_work(struct delayed_work *dwork);
extern bool cancel_delayed_work(struct delayed_work *dwork);
extern bool cancel_delayed_work_sync(struct 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 schedule_work(struct work_struct *work)
{
        return queue_work(system_wq, work);
}
```