

Методические указания к лабораторной работе

«Обработчики прерываний»

Обработчики медленных прерываний делятся на две части: верхнюю (top) и нижнюю (bottom) половины (half).

В настоящее время нижние половины могут быть трех типов:

- Отложенные прерывания (softirq)
- Тасклеты (tasklets)
- Очереди работ (work queue).

Драйверы регистрируют обработчик аппаратного прерывания и разрешают определенную линию irq посредством функции:

<linux/interrupt.h>

Typedef irqreturn_t(*irq_handler_t)(int,void *);

Int request_irq(unsigned int irq, irq_handler_t handler, unsigned long flags, const char* name, void *dev);

Прототипы взяты из ядра 2.6.37.

Устаревший прототип. Как видно, из объявления handler убрана struct pt_regs.

*int request_irq(unsigned int irq, irqreturn_t(*handler)(int, void *, struct pt_regs *), unsigned long irqflags, const char *devname, void *dev_id);*

где: irq – номер прерывания, *handler – указатель на обработчик прерывания, irqflags – флаги, devname – ASCII текст, представляющий устройство, связанное с прерыванием, dev_id – используется прежде всего для разделения (shared) линии прерывания и *struct pt_regs ** - *этот параметр в настоящее время исключен.*

Флаги, определенные на прерываниях:

```
#define IRQF_SHARED          0x00000080 /*разрешает разделение irq
несколькими устройствами*/
#define IRQF_PROBE_SHARED    0x00000100 /*устанавливается
абонентами, если возможны проблемы при совместном
использовании irq*/
#define IRQF_TIMER           0x00000200 /*флаг, маскирующий данное
прерывание как прерывание от таймера*/
#define IRQF_PERCPU          0x00000400 /*прерывание, закрепленное за
определенным процессором*/
#define IRQF_NOBALANCING     0x00000800 /*флаг, запрещающий
использование данного прерывания для балансировки irq*/
#define IRQF_IRQPOLL         0x00001000 /*прерывание используется для
опроса*/.
```

```
#define IRQF_ONESHOT    0x00002000
#define IRQF_NO_SUSPEND 0x00004000
#define IRQF_FORCE_RESUME 0x00008000
#define IRQF_NO_THREAD 0x00010000
#define IRQF_EARLY_RESUME 0x00020000
#define IRQF_COND_SUSPEND 0x00040000
```

Флаги были изменены радикально после версии ядра 2.6.19.

extern void free_irq(unsigned int irq, void *dev);

Данные по указателю dev требуются для удаления только конкретного устройства. Указатель void позволяет передавать все, что требуется, например указатель на handler. В результате **free_irq()** освободит линию **irq** от указанного обработчика.

Тасклеты

Тасклеты — это механизм обработки нижних половин, построенный на основе механизма отложенных прерываний. Тасклеты представлены двумя типами отложенных прерываний: **HI_SOFTIRQ** и **TASKLET_SOFTIRQ**. Единственная разница между ними в том, что тасклеты типа **HI_SOFTIRQ** выполняются всегда раньше тасклетов типа **TASKLET_SOFTIRQ**. В ядре определена структура:

```
struct tasklet_struct
{
    struct tasklet_struct *next; /* указатель на следующий тасклет в
                                списке */
    unsigned long state; /* состояние тасклета */
    atomic_t count; /* счетчик ссылок */
    void (*func) (unsigned long); /* функция-обработчик тасклета */
    unsigned long data; /* аргумент функции-обработчика тасклета */
};
```

Тасклеты в отличие от **softirq** могут быть зарегистрированы как статически, так и динамически.

Статически тасклеты создаются с помощью двух макросов:

```
DECLARE_TASKLET(name, func, data)
```

```
DECLARE_TASKLET_DISABLED(name, func, data);
```

Оба макроса статически создают экземпляр структуры **struct tasklet_struct** с указанным именем (**name**).

Например.

```
DECLARE_TASKLET(my_tasklet, tasklet_handler, dev);
```

Эта строка эквивалентна следующему объявлению:

```
struct tasklet_struct my_tasklet = {NULL, 0, ATOMIC_INIT(0),
                                   tasklet_handler, dev};
```

В данном примере создается тасклет с именем **my_tasklet**, который разрешен для выполнения. Функция **tasklet_handler** будет обработчиком

этого тасклета. Значение параметра dev передается в функцию-обработчик при вызове данной функции.

При динамическом создании тасклета объявляется указатель на структуру `struct tasklet_struct *t` а затем для инициализации вызывается функция:

```
tasklet_init(t, tasklet_handler, dev);
```

Тасклеты должны быть зарегистрированы для выполнения. Тасклеты могут быть запланированы на выполнение функциями:

```
tasklet_schedule(struct tasklet_struct *t);
tasklet_hi_scheduler(struct tasklet_struct *t);
void tasklet_hi_schedule_first(struct tasklet_struct *t); /* вне очереди */
```

Эти функции очень похожи и отличие состоит в том, что одна функция использует отложенное прерывание с номером `TASKLET_SOFTIRQ`, а другая — с номером `HI_SOFTIRQ`.

Когда `tasklet` запланирован, ему выставляется состояние **`TASKLET_STATE_SCHED`**, и тон добавляется в очередь. Пока он находится в этом состоянии, запланировать его еще раз не получится, т.е. в этом случае просто ничего не произойдет. `Tasklet` не может находиться сразу в нескольких местах очереди на планирование, которая организуется через поле `next` структуры `tasklet_struct`.

После того, как тасклет был запланирован, он выполнится только один раз.

Пример объявления и планирования тасклета.

```
/* Declare a Tasklet (the Bottom-Half) */
void tasklet_function( unsigned long data );

DECLARE_TASKLET(tasklet_example,tasklet_function,tasklet_data);

...

/* Schedule the Bottom-Half */
tasklet_schedule( &tasklet_example );
```

Пример:

```
irqreturn_t irq_handler(int irq, void *dev, struct pt_regs *regs)
{
    if(irq==define_irq)
    {
        tasklet_schedule(&my_tasklet);
        return IRQ_HANDLED;// прерывание обработано
    }
    else return IRQ_NONE; // прерывание не обработано
```

```

}
int init_module(void)
{
    return request_irq(define_irq, /*номер irq*/
                       (irq_handler_t)irq_handler, /*наш обработчик*/
                       IRQF_SHARED,
                       "test_my_irq_handler", /*имя устройства*/
                       (void *) (irq_handler)); /* void* dev_id – идентификатор
устройства. Может быть NULL, но тогда его
невозможно будет идентифицировать для отключения с
помощью free_irq. Т.к. это указатель на void, может
указывать на что угодно. Обычно используется
указатель на структуру описывающую устройство. В
нашем случае используем указатель на обработчик.*/
    ....
}
....

```

Tasklet можно активировать и деактивировать функциями:

```

void tasklet_disable_nosync(struct tasklet_struct *t); /* деактивация */
tasklet_disable(struct tasklet_struct *t); /* с ожиданием завершения работы tasklet'а */
tasklet_enable(struct tasklet_struct *t); /* активация */

```

Если tasklet деактивирован, его по-прежнему можно добавить в очередь на планирование, но исполняться на процессоре он не будет до тех пор, пока не будет вновь активирован. Причем, если tasklet был деактивирован несколько раз, то он должен быть ровно столько же раз активирован, поле count в структуре как раз для этого.

tasklet_trylock() выставляет tasklet'у состояние **TASKLET_STATE_RUN** и тем самым блокирует tasklet, что предотвращает исполнение одного и того же tasklet'а на разных CPU.

tasklet_kill (struct tasklet_struct *t) – ждет завершения тасклета и удаляет таскет из очереди на выполнение только в контексте процесса.

tasklet_kill_immediate (struct tasklet_struct *t, unsigned int cpu) – удаляет таскет в любом случае.

Причем, убит он будет только после того, как tasklet исполнится, если он уже запланирован.

Простой пример тасклета в контексте модуля ядра без обработчика прерывания:

```

#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/interrupt.h>

MODULE_LICENSE("GPL");

char my_tasklet_data[]="my_tasklet_function was called";

/* Bottom Half Function */
void my_tasklet_function( unsigned long data )
{
    printk( "%s\n", (char *)data );
    return;
}

DECLARE_TASKLET( my_tasklet, my_tasklet_function,
                (unsigned long) &my_tasklet_data );

int init_module( void )
{
    /* Schedule the Bottom Half */
    tasklet_schedule( &my_tasklet );

    return 0;
}

void cleanup_module( void )
{
    /* Stop the tasklet before we exit */
    tasklet_kill( &my_tasklet );

    return;
}

```

Задание:

- Написать загружаемый модуль ядра, в котором зарегистрировать обработчик аппаратного прерывания с флагом IRQF_SHARED.
- Инициализировать тасклет.
- В **обработчике прерывания (обязательно)** запланировать тасклет на выполнение.
- Вывести информацию о тасклете используя, или printk(), или seq_file interface - <linux/seq_file.h> (Jonathan Corber: <http://lwn.net//Articles//driver-porting/>).

Очереди работ

Основные понятия CMWQ (Concurrency Managed Workqueue)

Несколько объектов, связанных с очередью работ (*workqueue*), представлены в ядре соответствующими структурами:

- 1) Работа (work);
- 2) Очередь работ (workqueue) – коллекция work. Workqueue и work относятся как один-ко-многим;
- 3) Рабочий (worker). Worker соответствует потоку ядра worker_thread;
- 4) Пул рабочих потоков (worker_pool) это – набор рабочих (worker). Worker_pool и worker относятся как «один ко многим»;
- 5) Pwd (pool_workqueue) это – посредник, который отвечает за отношение workqueue и worker_pool: workqueue и pwd является отношением один-ко-многим, а pwd и worker_pool – отношение один-к-одному.

Очередь работ создается функцией (см. приложение 1):

int alloc_workqueue(char *name, unsigned int flags, int max_active);

- name - имя очереди (workqueue), но в отличие от старых реализаций потоков с этим именем не создается
- flags - флаги определяют как очередь работ будет выполняться
- max_active - ограничивает число задач (work) из некоторой очереди, которые могут выполняться на одном CPU.

Флаги

enum

```
{
    WQ_UNBOUND          =1<<1,/*not bound to any cpu*/
    WQ_FREEZABLE        =1<<2,/*freez during suspend*/
    WQ_MEM_RECLAIM      =1<<3,/*may be used for memory reclaim*/
    WQ_HIGHPRI          =1<<4,/*high priority*/
    WQ_CPU_INTENSIVE    =1<<5,/*cpu intensive workqueue*/
    WQ_SYSFS             =1<<6,/*visible in sysfs, see
                                wq_sysfs_register()*/
    WQ_POWER_EFFICIENT  =1<<7 ...
    WQ_MAX_ACTIVE       =512
    ...
}
```

- **WQ_UNBOUND**: По наличию этого флага очереди (workqueue) делятся на привязанные (normal) и непривязанные (unbound). В привязанных очередях work'и при добавлении привязываются к текущему CPU, то есть в таких очередях work'и исполняются на том ядре, которое его планирует (на котором выполнялся обработчик прерывания). В этом плане привязанные очереди напоминают tasklet'ы. Привязанные очереди работ исключают выбор планировщиком процессора для выпол-

нения рабочего потока, что имеет неприятный побочный эффект увеличения энергопотребления. Привязанные рабочие очереди, как правило, предпочтительнее из-за лучших показателей локализации кеша. Данный флаг отключает это поведение, позволяя отправлять заданные рабочие очереди на любой процессор. В непривязанных очередях work'и могут исполняться на любом ядре. Флаг предназначен для ситуаций, когда задачи могут выполняться в течение длительного времени, причем так долго, что лучше разрешить планировщику управлять своим местоположением. В настоящее время единственным пользователем является код обработки объектов в подсистеме FS-Cache.

- **WQ_FREEZEABLE**: работа будет заморожена, когда система будет приостановлена. Очевидно, что рабочие задания, которые могут запускать задачи как часть процесса приостановки / возобновления, не должны устанавливать этот флаг.
- **WQ_RESCUER**: код workqueue отвечает за гарантированное наличие потока для запуска worker'а в очереди. Он используется, например, в коде драйвера ATA, который всегда должен иметь возможность запускать свои процедуры завершения ввода-вывода.
- **WQ_HIGHPRI**: задания, представленные в такой workqueue, будут поставлены в начало очереди и будут выполняться (почти) немедленно. В отличие от обычных задач, высокоприоритетные задачи не ждут появления ЦП; они будут запущены сразу. Это означает, что несколько задач, отправляемых в очередь с высоким приоритетом, могут конкурировать друг с другом за процессор.
- **WQ_CPU_INTENSIVE**: имеет смысл только для привязанных очередей. Этот флаг — отказ от участия в дополнительной организации параллельного исполнения. Задачи в такой workqueue могут использовать много процессорного времени. Интенсивно использующие процессорное время worker'ы будут задерживаться.

Также может использоваться вызов `create_workqueue`:

```
#define create_workqueue(name) \
alloc_workqueue("%s", WQ_LEGACY | WQ_MEM_RECLAIM, 1, (name))
```

Например:

```
static int __init synthesizer_init(void)
{
    printk(KERN_INFO "Init synth.");
    // регистрация обработчика прерывания
    int res = request_irq(irq, irq_handler, IRQF_SHARED, synth.-
        name, &synth);
```

```

if (res == 0)
{
    printk(KERN_INFO "Keyboard irq handler was registered suc-
        cessfully.");
    // создание workqueue
    synth.wq = alloc_workqueue("sound_player", WQ_UNBOUND,
0);
    if (synth.wq)
        printk(KERN_INFO "Workqueue was allocated success-
fully");
    }
    else
    {
        free_irq(synth.keyboard_irq, &synth);
        printk(KERN_ERR "Workqueue allocation failed");
        return -ENOMEM;
    }
}

```

```

/*
* Очередь отложенных действий, связанная с процессором:
*/
struct cpu_workqueue_struct
{
    spinlock_t lock; /* Очередь для защиты данной структуры */
    long remove_sequence; /* последний добавленный элемент
(следующий для запуска) */
    long insert_sequence; /* следующий элемент для добавления */
    struct list_head worklist; /* список действий */
    wait_queue_head_t more_work;
    wait_queue_head_t work_done;
    struct workqueue_struct *wq; /* соответствующая структура
                                workqueue_struct */
    task_t *thread; /* соответствующий поток */
    int run_depth; /* глубина рекурсии функции run_workqueue() */
};

```

Заметим, что каждый *тип* рабочих потоков имеет одну, связанную с этим типом структуру `workqueue_struct`. Внутри этой структуры имеется по одному экземпляру структуры `cpu_workqueue_struct` для каждого рабочего потока и, следовательно, для каждого процессора в системе, так как существует только один рабочий поток каждого типа на каждом процессоре.

work item (или просто **work**) — это структура, описывающая функцию (например, обработчик нижней половины), которую надо запланировать. Её можно воспринимать как аналог структуры **tasklet**.

Для того, чтобы поместить задачу в очередь работ надо заполнить (инициализировать) 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
};
```

Структура **work_struct** представляет задачу (обработчик нижней половины) в очереди работ.

Поместить задачу в очередь работ можно во время компиляции (статически):

```
DECLARE_WORK( name, void (*func)(void *));
```

где: **name** — имя структуры **work_struct**, **func** — функция, которая вызывается из **workqueue** — обработчик нижней половины.

```
#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)
```

Если требуется задать структуру **work_struct** динамически, то необходимо использовать следующие два макроса:

```
INIT_WORK(struct work_struct *work, void (*func)(void), void *data);
```

```
#define INIT_WORK(_work, _func) \
    __INIT_WORK((_work), (_func), 0)
```

```
PREPARE_WORK(struct work_struct *work, void (*func)(void), void *data);
```

После того, как будет инициализирована структура для объекта **work**, следующим шагом будет помещение этой структуры в очередь работ. Это можно сделать несколькими способами. Во-первых, просто добавить работу (объект **work**) в очередь работ с помощью функции **queue_work** (которая назначает работу текущему процессору). Можно с помощью функции **queue_work_on** указать процессор, на котором будет выполняться обработчик.

```
int queue_work( struct workqueue_struct *wq, struct work_struct *work );
int queue_work_on( int cpu, struct workqueue_struct *wq, struct work_struct *work );
```

Две дополнительные функции обеспечивают те же функции для отложенной работы (в которой инкапсулирована структура `work_struct` и таймер, определяющий задержку).

```
int queue_delayed_work( struct workqueue_struct *wq,  
                       struct delayed_work *dwork, unsigned long delay );
```

```
int queue_delayed_work_on( int cpu, struct workqueue_struct *wq,  
                           struct delayed_work *dwork, unsigned long delay );
```

Кроме того, можно использовать глобальное ядро - глобальную очередь работ с четырьмя функциями, которые работают с этой очередью работ. Эти функции имитируют предыдущие функции, за исключением лишь того, что вам не нужно определять структуру очереди работ.

```
int schedule_work( struct work_struct *work );  
int schedule_work_on( int cpu, struct work_struct *work );
```

```
int scheduled_delayed_work( struct delayed_work *dwork, unsigned long  
delay );  
int scheduled_delayed_work_on(  
    int cpu, struct delayed_work *dwork, unsigned long delay );
```

Есть также целый ряд вспомогательных функций, которые можно использовать, чтобы принудительно завершить (flush) или отменить работу из очереди работ. Для того, чтобы принудительно завершить конкретный элемент `work` и заблокировать прочую обработку прежде, чем работа будет закончена, вы можете использовать функцию `flush_work`. Все работы в данной очереди работ могут быть принудительно завершены с помощью функции `flush_workqueue`. В обоих случаях вызывающий блок блокируется до тех пор, пока операция не будет завершена. Для того, чтобы принудительно завершить глобальную очередь работ ядра, вызовите функцию `flush_scheduled_work`.

```
int flush_work( struct work_struct *work );  
int flush_workqueue( struct workqueue_struct *wq );  
void flush_scheduled_work( void );
```

Можно отменить работу, если она еще не выполнена обработчиком. Обращение к функции `cancel_work_sync` завершит работу в очереди, либо возникнет блокировка до тех пор, пока не будет завершен обратный вызов (если работа уже выполняется обработчиком). Если работа отложена, вы можете использовать вызов функции `cancel_delayed_work_sync`.

```
int cancel_work_sync( struct work_struct *work );
int cancel_delayed_work_sync( struct delayed_work *dwork );
```

Наконец, можно выяснить приостановлен ли элемент work (еще не обработан обработчиком) с помощью обращения к функции `work_pending` или `delayed_work_pending`.

```
work_pending( work );
```

```
/**
 * work_pending - Find out whether a work item is currently pending
 * @work: The work item in question
 */
#define work_pending(work) \
    test_bit(WORK_STRUCT_PENDING_BIT, work_data_bits(work))
```

```
delayed_work_pending( work );
```

```
/**
 * delayed_work_pending - Find out whether a delayable work item is currently
 * pending
 * @w: The work item in question
 */
#define delayed_work_pending(w) \
    work_pending(&(w)->work)
```

Пример, в котором создаются две работы для одной очереди работ.

```
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/workqueue.h>
MODULE_LICENSE("GPL");
static struct workqueue_struct *my_wq; //очередь работ
typedef struct
{
    struct work_struct my_work;
    int x;
} my_work_t;
my_work_t *work1, *work2;
static void my_wq_function(struct work_struct *work) // вызываемая функция
{
    my_work_t *my_work = (my_work_t *)work;
    printk("my_work.x %d\n", my_work->x);
    kfree((void*)work);
    return;
}
int init_module(void)
{
    int ret;
```

```

my_wq = create_workqueue("my_queue");//создание очереди работ
if(my_wq)
{
    Work1 = (my_work_t *)kmalloc(sizeof( my_work_t),GFP_KERNEL);
    if (work1)
    {
        /* задача (item 1)*/
        INIT_WORK((struct work_struct)work, my_wq_function);
        work1->x = 1;
        ret = queue_work(my_wq, (struct work_struct *)work1);
    }

    work2 = (my_work_t *)kmalloc(sizeof( my_work_t),GFP_KERNEL);
    if (work2)
    {
        /* задача (item 2)*/
        INIT_WORK((struct work_struct)work, my_wq_function);
        work->x = 1;
        ret = queue_work(my_wq, (struct work_struct *)work2);
    }
}
return 0;
}
...

```

Задание:

- Написать загружаемый модуль ядра, в котором регистрируется обработчик аппаратного прерывания с флагом IRQF_SHARED.
- Инициализировать очередь работ.
- В обработчике прерывания запланировать очередь работ на выполнение.
- Вывести информацию об очереди работ используя, или printk(), или seq_file interface - <linux/seq_file.h> (Jonathan Corber: <http://lwn.net//Articles//driver-porting/>).

Приложение 1

```

/*
 * The externally visible workqueue. It relays the issued work items to
 * the appropriate worker_pool through its pool_workqueues.
 */
struct workqueue_struct {
    struct list_head pwqs;           /* WR: all pwqs of this wq */
    struct list_head list;          /* PR: list of all workqueues */

    struct mutex                    mutex;           /* protects this wq */
    int                             work_color;      /* WQ: current work color */
    int                             flush_color;     /* WQ: current flush color */

```

```

    atomic_t          nr_pwqs_to_flush; /* flush in progress */
    struct wq_flusher  *first_flusher; /* WQ: first flusher */
    struct list_head flusher_queue; /* WQ: flush waiters */
    struct list_head flusher_overflow; /* WQ: flush overflow list */

    struct list_head maydays; /* MD: pwqs requesting rescue */
    struct worker      *rescuer; /* MD: rescue worker */

    int                nr_drainers; /* WQ: drain in progress */
    int                saved_max_active; /* WQ: saved pwq max_active */

    struct workqueue_attrs *unbound_attrs; /* PW: only for unbound wqs */
    struct pool_workqueue *dfl_pwq; /* PW: only for unbound wqs */

#ifdef CONFIG_SYSFS
    struct wq_device *wq_dev; /* I: for sysfs interface */
#endif
#ifdef CONFIG_LOCKDEP
    char                *lock_name;
    struct lock_class_key key;
    struct lockdep_map  lockdep_map;
#endif

    char                name[WQ_NAME_LEN]; /* I: workqueue name */

    /*
     * Destruction of workqueue_struct is RCU protected to allow walking
     * the workqueues list without grabbing wq_pool_mutex.
     * This is used to dump all workqueues from sysrq.
     */
    struct rcu_head      rcu;

    /* hot fields used during command issue, aligned to cacheline */
    unsigned int         flags ____cacheline_aligned; /* WQ: WQ_* flags */
    struct pool_workqueue __percpu *cpu_pwqs; /* I: per-cpu pwqs */
    struct pool_workqueue __rcu *numa_pwq_tbl[]; /* PWR: unbound pwqs indexed
by node */
};

struct work_struct {
    atomic_long_t data;
    struct list_head entry;
    work_func_t func;
#ifdef CONFIG_LOCKDEP
    struct lockdep_map lockdep_map;
#endif
};

#ifdef CONFIG_LOCKDEP
#define __INIT_WORK(_work, _func, _onstack)
do {
    static struct lock_class_key __key;

    __init_work((_work), _onstack);
    (_work)->data = (atomic_long_t) WORK_DATA_INIT(); \
    lockdep_init_map(&(_work)->lockdep_map, "(work_completion)"#_work,
&__key, 0); \
    INIT_LIST_HEAD(&(_work)->entry); \
    (_work)->func = (_func); \
} while (0)
#else
#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)
#endif

#define INIT_WORK(_work, _func) \
    __INIT_WORK((_work), (_func), 0)

#define INIT_WORK_ONSTACK(_work, _func) \
    __INIT_WORK((_work), (_func), 1)

#define __INIT_DELAYED_WORK(_work, _func, _tflags) \
do { \
    INIT_WORK(&(_work)->work, (_func)); \
    __init_timer(&(_work)->timer, \
        delayed_work_timer_fn, \
        (_tflags) | TIMER_IRQSAFE); \
} while (0)

#define __INIT_DELAYED_WORK_ONSTACK(_work, _func, _tflags) \
do { \
    INIT_WORK_ONSTACK(&(_work)->work, (_func)); \
    __init_timer_on_stack(&(_work)->timer, \
        delayed_work_timer_fn, \
        (_tflags) | TIMER_IRQSAFE); \
} while (0)

#define INIT_DELAYED_WORK(_work, _func) \
    __INIT_DELAYED_WORK(_work, _func, 0)

#define INIT_DELAYED_WORK_ONSTACK(_work, _func) \
    __INIT_DELAYED_WORK_ONSTACK(_work, _func, 0)

#define INIT_DEFERRABLE_WORK(_work, _func) \
    __INIT_DELAYED_WORK(_work, _func, TIMER_DEFERRABLE)

#define INIT_DEFERRABLE_WORK_ONSTACK(_work, _func) \
    __INIT_DELAYED_WORK_ONSTACK(_work, _func, TIMER_DEFERRABLE)

#define INIT_RCU_WORK(_work, _func) \
    INIT_WORK(&(_work)->work, (_func))

#define INIT_RCU_WORK_ONSTACK(_work, _func) \
    INIT_WORK_ONSTACK(&(_work)->work, (_func))

```

```

/* Возвращает:
 * указатель на выделенный workqueue при успешном выполнении, %NULL при сбое.
 */
struct workqueue_struct * alloc_workqueue( const char *fmt,
                                           unsigned int flags,
                                           int max_active, ...);

```

```

#define create_workqueue(name) \
    alloc_workqueue("%s", WQ_LEGACY | WQ_MEM_RECLAIM, 1, (name))

```

```

extern void destroy_workqueue(struct workqueue_struct *wq);

```

```

extern void flush_workqueue(struct workqueue_struct *wq);

```

```

/* 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.
    */
static inline bool schedule_work(struct work_struct *work)
{
    return queue_work(system_wq, work);
}

/* After waiting for a given time this puts a job in the kernel-global
* workqueue.
*/
static inline bool schedule_delayed_work(struct delayed_work *dwork,
                                         unsigned long delay)
{
    return queue_delayed_work(system_wq, dwork, delay);
}

/* After waiting for a given time this puts a job in the kernel-global
* workqueue on the specified CPU.
*/
static inline bool schedule_delayed_work_on(int cpu, struct delayed_work *dwork,
                                             unsigned long delay)
{
    return queue_delayed_work_on(cpu, system_wq, dwork, delay);
}

/* In most situations flushing the entire workqueue is overkill; you merely
* need to know that a particular work item isn't queued and isn't running.
* In such cases you should use cancel_delayed_work_sync() or
* cancel_work_sync() instead.
*/
static inline void flush_scheduled_work(void)
{
    flush_workqueue(system_wq);
}

/* struct worker is defined in workqueue_internal.h */

struct worker_pool {
    raw_spinlock_t    lock;           /* the pool lock */
    int               cpu;           /* I: the associated cpu */
    int               node;          /* I: the associated node ID */
    int               id;            /* I: pool ID */
    unsigned int      flags;         /* X: flags */

    unsigned long     watchdog_ts;   /* L: watchdog timestamp */

    /*
    * The counter is incremented in a process context on the associated CPU
    * w/ preemption disabled, and decremented or reset in the same context
    * but w/ pool->lock held. The readers grab pool->lock and are
    * guaranteed to see if the counter reached zero.
    */
    int               nr_running;

    struct list_head worklist;      /* L: list of pending works */

    int               nr_workers;    /* L: total number of workers */
    int               nr_idle;       /* L: currently idle workers */

    struct list_head idle_list;     /* L: list of idle workers */
    struct timer_list idle_timer;   /* L: worker idle timeout */
    struct timer_list mayday_timer; /* L: SOS timer for workers */

    /* a workers is either on busy_hash or idle_list, or the manager */

```

```

DECLARE_HASHTABLE(busy_hash, BUSY_WORKER_HASH_ORDER);
/* L: hash of busy workers */

struct worker      *manager;      /* L: purely informational */
struct list_head workers; /* A: attached workers */
struct completion *detach_completion; /* all workers detached */

struct ida          worker_ida;    /* worker IDs for task name */

struct workqueue_attrs *attrs;      /* I: worker attributes */
struct hlist_node hash_node; /* PL: unbound_pool_hash node */
int refcnt; /* PL: refcnt for unbound pools */
*/

/*
 * Destruction of pool is RCU protected to allow dereferences
 * from get_work_pool().
 */
struct rcu_head rcu;
};

```