```
* kernel/workqueue.c - generic async execution with shared worker pool
 *

/*
 * 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;  /* I: 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
        struct lockdep_map      lockdep_map;
#endif
        char                            name[WQ_NAME_LEN]; /* I: workqueue name */

        /*
         * Destruction of workqueue_struct is sched-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
};

typedef void (*work_func_t)(struct work_struct *work);
```

```
/*
 * initialize all of a work item in one go
 *
 * NOTE! No point in using "atomic_long_set()": using a direct
 * assignment of the work data initializer allows the compiler
 * to generate better code.
 */
#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, &__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)


/*
 * Workqueue flags and constants.  For details, please refer to
 * Documentation/workqueue.txt.
 */
enum {
        WQ_UNBOUND              = 1 << 1, /* not bound to any cpu */
        WQ_FREEZABLE            = 1 << 2, /* freeze 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() */

        /*
         * Per-cpu workqueues are generally preferred because they tend to
         * show better performance thanks to cache locality.  Per-cpu
         * workqueues exclude the scheduler from choosing the CPU to
         * execute the worker threads, which has an unfortunate side effect
         * of increasing power consumption.
         *
         * The scheduler considers a CPU idle if it doesn't have any task
         * to execute and tries to keep idle cores idle to conserve power;
         * however, for example, a per-cpu work item scheduled from an
         * interrupt handler on an idle CPU will force the scheduler to
         * excute the work item on that CPU breaking the idleness, which in
         * turn may lead to more scheduling choices which are sub-optimal
         * in terms of power consumption.
         *
         * Workqueues marked with WQ_POWER_EFFICIENT are per-cpu by default
         * but become unbound if workqueue.power_efficient kernel param is
```

```
                * specified.  Per-cpu workqueues which are identified to
                * contribute significantly to power-consumption are identified and
                * marked with this flag and enabling the power_efficient mode
                * leads to noticeable power saving at the cost of small
                * performance disadvantage.
                *
                * http://thread.gmane.org/gmane.linux.kernel/1480396
                */
```
* Рабочие очереди для каждого процессора, как правило, предпочтительнее, поскольку они, как правило,
* демонстрируют более высокую производительность благодаря локальности кэша. Для каждого процессора
* рабочие очереди исключают возможность выбора планировщиком процессора для
* выполнение рабочих потоков, что имеет неприятный побочный эффект
* увеличения энергопотребления.
*
* Планировщик считает, что процессор простаивает, если у него нет какой-либо задачи
* для выполнения и пытается поддерживать простаивающие ядра в режиме ожидания для экономии энергии;
* однако, например, рабочий элемент для каждого процессора, запланированный из
* обработчика прерывания на неработающем процессоре заставит планировщик
* исключить рабочий элемент на этом процессоре, нарушая бездействие, которое наоборот может привести к большему количеству вариантов планирования, которые являются неоптимальными с точки зрения энергопотребления.
*
* Рабочие очереди, помеченные символом WQ_POWER_EFFICIENT, по умолчанию предназначены для каждого процессора.
* но становится несвязанным, если рабочая очередь.параметр ядра power_efficient равен
* указано. Рабочие очереди для каждого процессора, которые идентифицируются для
* значительно увеличивают энергопотребление, идентифицируются и
* помечаются этим флагом и включают режим power_efficient.
* приводит к заметной экономии электроэнергии за счет небольших
* недостаток производительности.

```
        WQ_POWER_EFFICIENT      = 1 << 7,

        __WQ_DRAINING           = 1 << 16, /* internal: workqueue is draining */
        __WQ_ORDERED            = 1 << 17, /* internal: workqueue is ordered */
        __WQ_LEGACY             = 1 << 18, /* internal: create*_workqueue() */

        WQ_MAX_ACTIVE           = 512,       /* I like 512, better ideas? */
        WQ_MAX_UNBOUND_PER_CPU  = 4,         /* 4 * #cpus for unbound wq */
        WQ_DFL_ACTIVE           = WQ_MAX_ACTIVE / 2,
};

/* unbound wq's aren't per-cpu, scale max_active according to #cpus */
#define WQ_UNBOUND_MAX_ACTIVE \
        max_t(int, WQ_MAX_ACTIVE, num_possible_cpus() * WQ_MAX_UNBOUND_PER_CPU)

/*
 * System-wide workqueues which are always present.
 *
 * system_wq is the one used by schedule[_delayed]_work[_on]().
 * Multi-CPU multi-threaded.  There are users which expect relatively
 * short queue flush time.  Don't queue works which can run for too
 * long.
 *
 * system_highpri_wq is similar to system_wq but for work items which
 * require WQ_HIGHPRI.
 *
 * system_long_wq is similar to system_wq but may host long running
 * works.  Queue flushing might take relatively long.
 *
 * system_unbound_wq is unbound workqueue.  Workers are not bound to
 * any specific CPU, not concurrency managed, and all queued works are
 * executed immediately as long as max_active limit is not reached and
```

 * *resources are available.*
 *
 * *system_freezable_wq is equivalent to system_wq except that it's*
 * *freezable.*
 *
 * **_power_efficient_wq are inclined towards saving power and converted*
 * *into WQ_UNBOUND variants if 'wq_power_efficient' is enabled; otherwise,*
 * *they are same as their non-power-efficient counterparts - e.g.*
 * *system_power_efficient_wq is identical to system_wq if*
 * *'wq_power_efficient' is disabled. See WQ_POWER_EFFICIENT for more info.*
 */

* Общесистемные рабочие очереди, которые всегда присутствуют.

* system_wq - это тот, который используется schedule[_delayed]_work[_on]().

* Многопроцессорная многопоточность. Есть пользователи, которые ожидают относительно

* короткое время промывки очереди *queue flush*. Не ставьте в очередь работы, которые могут выполняться слишком долго длинный.

* system_highpri_wq аналогичен system_wq, но для рабочих элементов, которые

* требуется WQ_HIGHPRI.

* system_long_wq похож на system_wq, но может содержать длительное время

* работает. Очистка очереди может занять относительно много времени.

* system_unbound_wq - это несвязанная рабочая очередь. Работники не обязаны

* любой конкретный процессор, не управляемый параллелизмом, и все работы в очереди выполняются

* выполняется немедленно до тех пор, пока не будет достигнут предел max_active и

* ресурсы доступны.

* system_freezable_wq эквивалентен system_wq, за исключением того, что он

* возможность замораживания.

* *_power_efficient_wq склонны к экономии энергии и преобразованы

* в варианты WQ_UNBOUND, если включен параметр 'wq_power_efficient'; в противном случае

* они такие же, как и их неэнергосберегающие аналоги - например

* system_power_efficient_wq идентичен system_wq, если

* 'wq_power_efficient' отключен. Дополнительную информацию см. в разделе WQ_POWER_EFFICIENT.