Длинные работы в workqueue

Модули ядра Linux
Практическая работа

Михаил Пожидаев

Институт прикладной математики и компьютерных наук ТГУ

2025 г.

Workqueue

Отложенное выполнение работы в контексте ядра

Механизм `workqueue` позволяет передать работу специальному потоку ядра.
Функция, выполняющая работу, будет вызвана позднее, после постановки работы в очередь.
Код вызывающего потока при этом может продолжить выполнение.
`workqueue` особенно полезна для операций, которые нельзя или не стоит выполнять в контексте прерывания.

Главная идея

Быстро зарегистрировать событие, а длительную обработку выполнить позже в контексте рабочего потока ядра.

Почему нельзя делать долгую работу сразу

Контекст выполнения

Некоторые функции ядра вызываются из контекста, в котором запрещено долго работать.
Обработчик аппаратного прерывания не может заснуть.
Обработчик таймера также должен завершаться быстро.
Функция загрузки модуля обычно выполняется в контексте процесса, но блокировать её на долгое время всё равно нежелательно.
Длительная работа в неподходящем контексте приводит к задержкам, предупреждениям ядра или зависанию системы.
`workqueue` переносит обработку в специальный поток ядра, который может планироваться и засыпать.

Работа в очереди не становится пользовательским процессом.

Она выполняется внутри ядра, но в контексте kernel worker thread.

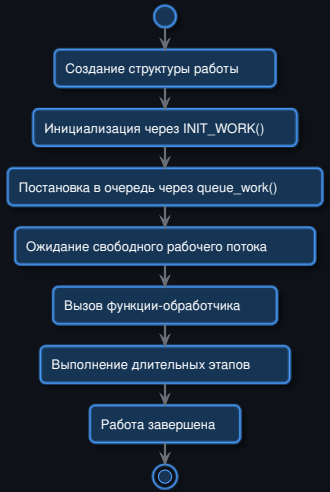
Жизненный цикл работы

От постановки до завершения

Работа проходит несколько состояний.
Сначала выделяется память под объект работы.
Затем объект инициализируется и ставится в очередь.
Позднее ядро вызывает функцию-обработчик.
После завершения обработчика объект можно освободить.

Жизненный цикл работы

От постановки до завершения



Структура work_struct

Описание одной работы

Каждая работа представляется структурой `struct work_struct`.
Структура содержит служебное состояние, используемое механизмом `workqueue`.
В ней также хранится функция, которая будет вызвана при выполнении работы.
`work_struct` не содержит автоматически произвольные данные задания.
Обычно её помещают внутрь собственной структуры.

```
struct lab_work {
    struct work_struct work;
    unsigned int id;
    unsigned int stages;
    unsigned int completed;
};
```

Инициализация работы

Макрос INIT_WORK()

Функция `INIT_WORK()` связывает структуру работы с обработчиком.
Инициализация должна завершиться до постановки работы в очередь.
Нельзя повторно инициализировать работающую или поставленную в очередь структуру.
Обработчик получает адрес поля `work`, а не адрес всей структуры `lab_work`.

```
static void lab_work_handler(struct work_struct *work);
static struct lab_work item;

INIT_WORK(&item.work, lab_work_handler);
```

Получение собственной структуры

Макрос container_of()

В обработчик передаётся указатель на поле `struct work_struct`.
Для получения окружающей структуры используется макрос `container_of()`.
Макрос учитывает смещение поля `work` внутри структуры.
Нельзя приводить указатель простым преобразованием типов, если поле не находится в начале структуры.

```
static void lab_work_handler(struct work_struct *work)
{
    struct lab_work *item;

    item = container_of(work, struct lab_work, work);

    pr_info("work %u started\n", item->id);
}
```

Обработчик работы

Функция, выполняющаяся в ядре

Обработчик вызывается рабочим потоком ядра.
Он выполняется асинхронно относительно функции, поставившей работу в очередь.
Обработчик может выполнять операции, допускающие засыпание.
В частности, допустимы `msleep()`, `mutex_lock()` и ожидание доступного ресурса.
Любая ошибка в обработчике может повредить всё ядро или привести к аварийному завершению системы.

```
static void lab_work_handler(struct work_struct *work)
{
    struct lab_work *item;

    item = container_of(work, struct lab_work, work);

    pr_info("work %u started\n", item->id);

    msleep(1000);

    pr_info("work %u finished\n", item->id);
}
```

Длительная работа

Засыпание вместо активного ожидания

Для имитации длительного этапа можно использовать `msleep()`.
Во время сна рабочий поток не занимает процессор.
Планировщик может передать процессор другому потоку.
Цикл с проверкой времени и вызовом `cpu_relax()` является активным ожиданием.
Такой цикл бесполезно расходует процессорное время.

```
static void lab_work_handler(struct work_struct *work)
{
    struct lab_work *item;
    unsigned int stage;

    item = container_of(work, struct lab_work, work);

    for (stage = 0; stage < item->stages; ++stage) {
        pr_info(
            "work %u: stage %u/%u\n",
            item->id,
            stage + 1,
            item->stages
        );

        msleep(1000);
        item->completed = stage + 1;
    }
}
```

Где можно засыпать

Контексты выполнения

Рабочая функция `workqueue` выполняется в контексте потока ядра.
В этом контексте допустимы операции, которые могут блокировать текущий поток.
В контексте аппаратного прерывания засыпать нельзя.
В нижней половине обработки прерываний также нужно учитывать ограничения контекста.
`spin_lock()` нельзя удерживать во время `msleep()`.

Запомни

Возможность засыпать определяется контекстом выполнения, а не тем, является ли функция частью модуля.

Постановка работы в очередь

Функция queue_work()

После инициализации работа ставится в очередь функцией `queue_work()`.
Функция возвращает `true`, если работа была поставлена в очередь.
Возвращаемое значение `false` обычно означает, что работа уже находится в очереди или выполняется.
Одна структура `work_struct` описывает один активный запуск работы.
Для нескольких независимых запусков нужны несколько структур или собственные объекты заданий.

```
bool queued;

queued = queue_work(queue, &item.work);
```

Повторная постановка

Одна структура не создаёт несколько работ

Следующие вызовы используют одну и ту же структуру.
Второй вызов не создаёт вторую копию работы.
Одна структура `work_struct` может находиться в очереди только один раз.
Если нужны пять независимых работ, необходимо создать пять объектов.
Или можно выделять объекты динамически при обработке команды `start N`.

```
queue_work(queue, &item.work);
queue_work(queue, &item.work);

struct lab_work items[5];
```

Общая и собственная очереди

Где выполняются работы

Ядро предоставляет системные очереди работ, например `system_wq`.
Для большинства простых случаев работу можно поставить в системную очередь.
Собственная очередь создаётся с помощью `alloc_workqueue()`.
Собственная очередь позволяет задать имя, флаги и максимальное число одновременно выполняемых работ.

```
struct workqueue_struct *queue;

queue = alloc_workqueue(
    "tsulab_queue",
    WQ_UNBOUND,
    2
);
```

Параметр max_active

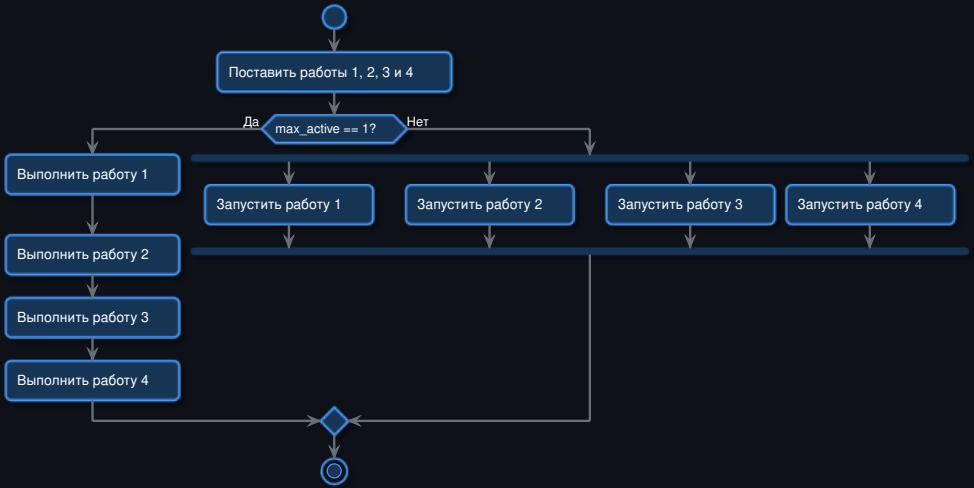
Ограничение параллельного выполнения

Если `max_active` равен единице, работы этой очереди выполняются последовательно.
Если значение больше единицы, несколько работ могут выполняться одновременно.
Параллельность не означает обязательный одновременный запуск.
Фактическое выполнение зависит от планировщика и доступных процессоров.
Такая очередь может одновременно выполнять до четырёх работ.
Общие данные между работами необходимо защищать от гонок.

```
queue = alloc_workqueue(
    "tsulab_queue",
    WQ_UNBOUND,
    4
);
```

Последовательная и параллельная очереди

Влияние max_ active



Память и время жизни

Главная опасность асинхронного выполнения

Обработчик может начать работу после того, как функция загрузки модуля уже завершилась.
Поэтому данные работы должны существовать до момента завершения обработчика.
Нельзя передавать в очередь адрес локальной переменной, которая будет уничтожена после выхода из функции.
Нельзя освобождать структуру работы сразу после `queue_work()`.

```
struct lab_work *item;

item = kmalloc(sizeof(*item), GFP_KERNEL);
if (!item)
    return -ENOMEM;

INIT_WORK(&item->work, lab_work_handler);
queue_work(queue, &item->work);
```

Освобождение объекта

Нельзя освобождать память слишком рано

Следующий код ошибочен.
Рабочий поток может обратиться к `item` после вызова `kfree()`.
Это приводит к использованию освобождённой памяти.
Последствия могут проявиться не сразу.
Обычно такая ошибка приводит к повреждению памяти ядра или к ошибке доступа.

```
queue_work(queue, &item->work);
kfree(item);
```

flush_workqueue()

Дождаться завершения всех работ

Функция `flush_workqueue()` дожидается выполнения работ из очереди.
Она не отменяет работы.
После возврата из `flush_workqueue()` работы, поставленные в очередь до начала ожидания, должны завершиться.
Эту функцию можно использовать перед освобождением данных, которыми пользуются обработчики.
Новые работы не должны добавляться параллельно с финальным освобождением ресурсов.

```
flush_workqueue(queue);
```

Отмена работы

cancel_work() и cancel_work_sync()

Функция `cancel_work()` пытается удалить работу из очереди.
Она не гарантирует, что уже выполняющийся обработчик завершился.
Функция `cancel_work_sync()` дожидается завершения обработчика, если он уже начал выполняться.
После `cancel_work_sync()` структуру работы можно освобождать, если на неё больше не ссылаются другие объекты.

Нельзя уничтожать очередь сразу после вызова `cancel_work()`**.**

```
cancel_work_sync(&item->work);
kfree(item);
```

Уничтожение очереди

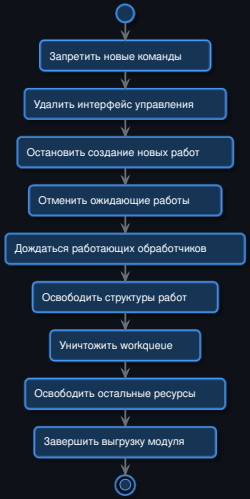
Правильный порядок выгрузки

При выгрузке сначала нужно прекратить появление новых работ.
Затем необходимо дождаться завершения или отменить уже поставленные работы.
После этого можно освободить структуры работ.
В конце очередь уничтожается через `destroy_workqueue()`.
Нельзя выгружать модуль, пока обработчик выполняет код, находящийся в этом модуле.

```
flush_workqueue(queue);
destroy_workqueue(queue);
```

Безопасная выгрузка модуля

Порядок освобождения ресурсов



Отмена длинной работы

Кооперативное завершение

Работу, которая уже выполняется, нельзя безопасно прервать в произвольной инструкции.
Для этого используется собственный признак отмены.
Обработчик периодически проверяет этот признак между этапами.
После установки признака отмены всё равно необходимо дождаться завершения обработчика.
Для этого используется `cancel_work_sync()`.

```
if (READ_ONCE(item->cancel_requested))
    return;
```

```
static void lab_work_handler(struct work_struct *work)
{
    struct lab_work *item;
    unsigned int stage;

    item = container_of(work, struct lab_work, work);

    for (stage = 0; stage < item->stages; ++stage) {
        if (READ_ONCE(item->cancel_requested)) {
            item->state = LAB_CANCELLED;
            return;
        }

        msleep(1000);
        item->completed = stage + 1;
    }

    item->state = LAB_DONE;
}
```

Синхронизация состояния

Счётчики и список работ

Состояние работ обычно хранится в списке структур, защищённом mutex.
Команды управления могут одновременно добавлять, отменять и просматривать работы.
В это же время рабочие потоки изменяют состояние своих объектов.
Все операции со списком должны быть согласованы.
Mutex можно использовать в контексте работы, поскольку рабочий поток может заснуть.
Атомарность отдельного счётчика не защищает связанный с ним список.

```
static DEFINE_MUTEX(items_lock);
static LIST_HEAD(items);
```

Состояния работы

Не достаточно одного счётчика

Удобно явно хранить состояние каждой работы.
Состояние должно изменяться в одном согласованном месте.
Нельзя определять выполнение работы только по значению счётчика завершённых этапов.
Работа может быть отменена до начала выполнения или остановлена между этапами.

```
enum lab_work_state {
    LAB_CREATED,
    LAB_QUEUED,
    LAB_RUNNING,
    LAB_DONE,
    LAB_CANCELLED
};
```

Интерфейс /proc

Управление модулем из пользовательского пространства

Файл в `/proc` может использоваться как простой интерфейс управления модулем.
Запись строки в файл передаёт команду модулю.
Чтение файла возвращает текущее состояние.
Для современных ядер используется структура `struct proc_ops`.

```
proc_entry = proc_create(
    "tsulab",
    0666,
    NULL,
    &proc_ops
);
```

Команды управления

Простота интерфейса важнее гибкости

Команды можно передавать в виде коротких строк.
Команда `start N` должна создать и поставить в очередь `N` новых работ.
Команда `status` должна показать количество созданных, ожидающих, выполняющихся, завершённых и отменённых работ.
Команда `cancel` должна отменить работы, которые ещё не начали выполняться.
Работа, которая уже выполняется, должна остановиться в безопасной точке.

```
start 5
status
cancel
```

Копирование данных из пользовательского пространства

Функция copy_from_user()

Обработчик записи в `/proc` получает указатель на память пользовательского процесса.
Нельзя разыменовывать такой указатель непосредственно в ядре.
Данные сначала копируются во внутренний буфер.
Размер входной строки нужно ограничивать до копирования.

```
char command[64];
size_t length;

length = min(size, sizeof(command) - 1);

if (copy_from_user(command, buffer, length))
    return -EFAULT;

command[length] = '\0';
```

После копирования строки в память ядра её можно разобрать с помощью `sscanf()`, `sysfs_streq()` или собственного простого анализатора.
Нельзя доверять числу, переданному пользователем.
Нужно проверить переполнение.
Нужно ограничить максимально допустимое количество работ.
Нужно обрабатывать ошибки выделения памяти.
Команда управления не должна освобождать объект, обработчик которого ещё выполняется.

Чтение через seq_file

Построение содержимого /proc

Интерфейс `seq_file` предназначен для формирования последовательного вывода.
Он корректно работает при чтении файла частями.
Для небольшого отчёта удобно использовать `single_open()`.
Данные для отчёта нужно читать согласованно с их изменением.
Нельзя обходить изменяемый список без блокировки или другого механизма синхронизации.

```
static int lab_show(
    struct seq_file *file,
    void *unused)
{
    seq_printf(file, "queued: %u\n", queued_count);
    seq_printf(file, "running: %u\n", running_count);
    seq_printf(file, "done: %u\n", done_count);

    return 0;
}
```

Эксперимент с max_active

Как увидеть параллельность

Сначала создаётся очередь с `max_active == 1`.
Затем в неё ставится несколько работ с одинаковой продолжительностью.
Работы должны выполняться последовательно.
Общее время выполнения будет близко к сумме длительностей работ.
После этого очередь создаётся с `max_active == 2` или `max_active == 4`.
При наличии свободных процессоров общее время может уменьшиться.

```
max_active = 1:

work 1:  |---------|
work 2:            |---------|
work 3:                      |---------|

max_active = 3:

work 1:  |---------|
work 2:  |---------|
work 3:  |---------|
```

Эксперимент с max_active

Интерпретация результатов

Результаты зависят от загрузки системы и не являются строгой гарантией времени выполнения.
Параллельное выполнение может уменьшить общее время обработки.
Одновременный запуск зависит от планировщика и доступных процессоров.

Наблюдение за выполнением

Журнал ядра и измерение времени

Сообщения модуля можно просматривать командой `dmesg -w`.
Для каждой работы полезно выводить идентификатор, состояние и номер этапа.
Время работы можно измерять с помощью `ktime_get_ns()`.
Нельзя использовать пользовательские функции измерения времени внутри модуля.

```
pr_info(
    "tsulab: work=%u stage=%u/%u\n",
    item->id,
    stage + 1,
    item->stages
);
```

Отложенная работа

delayed_work

`delayed_work` позволяет поставить работу в очередь с задержкой.
Задержка задаётся в тиках ядра.
Для перевода миллисекунд в тики используется `msecs_to_jiffies()`.
Отложенная работа не занимает процессор во время ожидания задержки.

```
struct delayed_work delayed;

INIT_DELAYED_WORK(
    &item->delayed,
    delayed_handler
);

queue_delayed_work(
    queue,
    &item->delayed,
    delay
);
```

```
delay = msecs_to_jiffies(1000);
```

Типичные ошибки

Что проверять при отладке

Освобождение структуры сразу после `queue_work()` приводит к обращению к освобождённой памяти.
Повторная постановка одной структуры не создаёт несколько независимых работ.
Вызов `cancel_work()` не дожидается завершения уже начатого обработчика.
Уничтожение очереди до завершения работ может привести к использованию выгружаемого кода.
Вывод большого количества сообщений через `pr_info()` может переполнить журнал ядра.
Доступ к списку работ без блокировки приводит к гонкам.
Засыпание с удерживаемым `spinlock` может заблокировать другие процессоры.

Сборка модуля

Использование kbuild

Модуль ядра собирается для конкретной версии ядра.
Проверить версию запущенного ядра можно командой `uname -r`.
После сборки должен появиться файл `tsulab.ko`.

```
obj-m += tsulab.o

all:
make -C /lib/modules/$(shell uname -r)/build \
M=$(PWD) modules

clean:
make -C /lib/modules/$(shell uname -r)/build \
M=$(PWD) clean
```

Модуль можно загрузить командой `sudo insmod tsulab.ko`.
Проверить сообщения загрузки можно командой `dmesg | tail`.
Посмотреть содержимое интерфейса можно командой `cat /proc/tsulab`.
Отправить команду можно командой `echo ''start 5'' | sudo tee /proc/tsulab`.
Выгрузить модуль можно командой `sudo rmmod tsulab`.

```
sudo insmod tsulab.ko
dmesg | tail
cat /proc/tsulab
echo "start 5" | sudo tee /proc/tsulab
sudo rmmod tsulab
```

Проверка результата

Что нужно показать

Нужно показать, что функция загрузки модуля не ждёт завершения длительных работ.
Нужно показать несколько одновременно выполняющихся работ.
Нужно показать изменение состояния через `/proc/tsulab`.
Нужно выполнить отмену при наличии ожидающих и уже работающих работ.
Нужно выгрузить модуль во время выполнения работы.
После выгрузки не должно остаться активных обработчиков и объектов модуля.

Эксперимент считается успешным только после проверки журнала ядра и отсутствия предупреждений после выгрузки.

Вопросы для защиты

Что нужно понимать

Почему работу нельзя выполнять непосредственно в функции загрузки модуля?
В каком контексте выполняется обработчик `workqueue`?
Почему в обработчике допустим `msleep()`?
Чем отличаются `queue_work()`, `flush_workqueue()` и `cancel_work_sync()`?
Почему одна структура `work_struct` не может представлять несколько одновременных запусков?
Как обеспечить время жизни структуры работы?
Какие данные нужно защищать mutex-ом?
Чем последовательная очередь отличается от параллельной?
Почему активное ожидание является плохой имитацией длительной работы?
Что может произойти, если выгрузить модуль во время выполнения его обработчика?

Спасибо за внимание!

Домашняя страница:
https://marigostra.ru

Электронная почта:
msp@luwrain.org

Telegram:
@Marigostra