Санкт-Петербургский Государственный университет

Математико-механический факультет

Кафедра системного программирования

Улучшение средств синхронизации в ОСРВ Embox

Курсовая работа студента 444 группы

Логиновой Виты Владимировны

Научный руководитель Козлов А. П.

аспирант кафедры системного программирования

Санкт-Петербург

2014

Оглавление

Введение

Многозадачность - свойство операционной системы выполнять несколько вычислительных задач одновременно. Многозадачность является неотъемлемой частью современной операционной системы. Грамотная реализация многозадачности может сильно ускорить программу, выполняющую несколько вычислительных задач одновременно за счет распределения ресурсов процессора между ними.

При разработки многозадачности программист сталкивается с различными сложностями [3], такими как:

- предотвращение ситуаций гонки, инверсии приоритета и так далее
- обработка аппаратных и программных прерываний
- поддержка защиты памяти
- поддержка надежности
- сохранение корректного состояния потока исполнения
- взаимодействие между потоками, использование общей памяти

В теории операционных систем разделяют четыре основных метода межпроцессного взаимодействия (IPC) [1]:

- обмен сообщениями
- разделяемая память
- синхронизация
- удаленные вызовы

В ОСРВ Embox многозадачность поддерживается с ранних версий. Кроме того, в ОС Embox реализован ряд средств IPC, например, различные примитивы синхронизации.

После того, как на ОСРВ Embox стало возможным запускать приложения, написанные с использованием библиотеки Qt, в которых средства синхронизации и механизм ожидания событий используются особо часто, стало ясно, что их реализация не достаточно эффективна.

Постановка задачи

Целью работы было исследовать существующую реализацию средств синхронизации и используемые ими средства в ОСРВ Embox. Далее мне предстояло найти и устранить причины, приводящие к низкой производительности при работе больших приложений.

Таким образом, в рамках данной курсовой работы передо мной были поставлены задачи:

- Исследовать различные методы IPC и их реализацию в ОСРВ Embox.
- Усовершенствовать существующую реализацию средств синхронизации в ядре ОС Embox и связанные с ними компоненты системы.
- Оценить полученную реализацию.

Обзор существующей реализации

Все существующие примитивы синхронизации в ОСРВ Embox реализованы с использованием общего интерфейса ожидания событий, основанного на очереди ждущих потоков. Таким образом, прежде, чем перейти непосредственно к обзору примитивов синхронизации, необходимо изучить машину состояний потока в ОСРВ Embox и работу очереди ждущих потоков.

Машина состояний потока

Машина состояний описывает все этапы существования потока в системе, начиная от создания, заканчивая завершением.

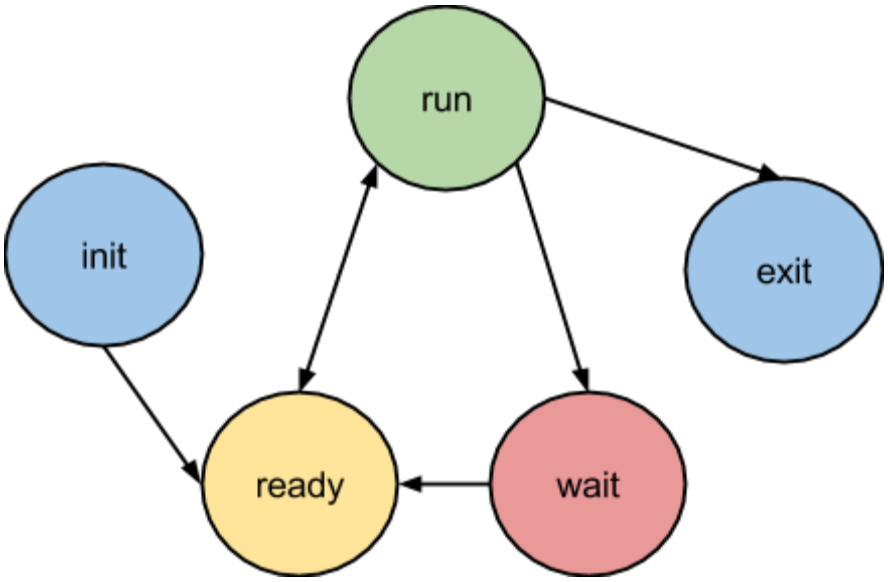

Рис. 1. Машина состояний потока в ОСРВ Embox.

Машину состояний потока можно представить в виде схемы, показанной на рис. 1. За то, в каком состоянии находится поток, отвечает целочисленное поле state в структуре потока. Комбинация флагов в значении поля state в зависимости от состояния:

- init - INIT
- run - ACTIVE & ONCPU
- ready - ACTIVE
- wait - ACTIVE & SLEEPING
- exit - EXITED

Состояние init поток получает при инициализации, поток остается в этом состоянии до

тех пор, пока его не запустят. На этом этапе у планировщика нет никаких данных о существовании потока.

После того, как поток запущен, он получает состояние ready и помещается в очередь runq. Runq - это очередь потоков на планирование. Из нее планировщик достает поток, чтобы предоставить ему ресурсы процессора. То, в каком порядке в ней располагаются потоки, зависит от заданных правил, так называемой стратегии планирования. В очереди находятся лишь потоки в состоянии ready, и исполняющегося на процессоре потока в runq нет, планировщик помещает его в очередь перед непосредственным перепланированием.

Потоку, который планировщик достает из runq, присваивается состояние run. В этом состоянии поток выполняет инструкции своей основной функции. В случае вытесняющего планирования, когда потоку выделяется квант времени, поток в любой момент может быть прерван и помещен обратно в runq, где он будет снова ждать своей очереди. Тогда поток снова будет в состоянии ready.

Когда потоку необходимо прервать свое исполнение, чтобы дождаться какого-то события, например, синхронизации с другим потоком, он использует либо механизм ожидания событий напрямую, либо какие-то средства синхронизации. В таком случае он переходит в состояние wait, вызывает перепланирование. Планировщик в очередь потоков его не кладет. Таким образом, его исполнение не будет возобновлено до тех пор, пока его не разбудят с помощью специальных функций. После пробуждения поток помещается в очередь runq и получает состояние ready.

Когда поток выполнит свою функцию, он переходит в состояние exit. Планировщик не кладет его в очередь и ресурсы процессора больше не предоставит. То, что происходит с потоком дальше, зависит от машины управления ресурсами. В коечном итоге экземпляр потока уничтожается, поток перестает существовать.

Можно заметить некоторую избыточность состояний. Например, состояние init не несет полезной информации: с точки зрения планировщика нет разницы, запускать поток в первый раз или будить после ожидания, так в обоих случаях он выполняет один набор основных действий: поменять состояние потока на ready, поместить поток в очередь runq. Впрочем, функции планировщика для запуска и пробуждения потока различаются без особой на то необходимости, что также избыточно.

От состояния exit тоже можно избавиться, так как оно информативно только в

контексте машины управления ресурсами.

В реализации машины состояний потока есть еще один существенный недостаток: при работе с ней не учитываются промежуточные состояния. Связанные с этим проблемы описаны ниже при рассмотрении примитивов синхронизации.

Механизм ожидания

Механизм ожидания событий - ключевое средство для взаимодействия между потоками. Он реализован на основе очереди спящих потоков. Архитектура механизма ожиданий представлена на рис. 2.

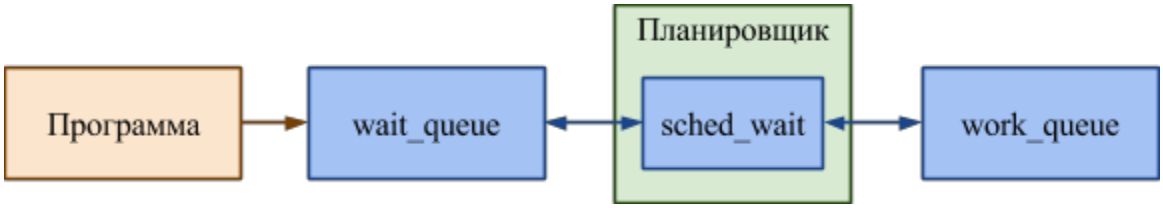

Рис. 2. Архитектура механизма ожидания событий.

Потоки, ожидающие одно и то же событие, например, разрешение на вход в критическую секцию, с помощью интерфейса wait_queue регистрируют себя в очереди и засыпают. Когда событие происходит, обработчик вызывает функцию пробуждения потоков из очереди.

Основная задача интерфейса wait_queue - подготовить необходимые структуры, а именно саму очередь и ссылочную структуру для потока в этой очереди. Задачи, связанные непосредственно с изменением состояния потока, пробуждением и так далее, выполняют функции планировщика sched_wait.

Функции sched_wait работают за счет особой очереди work, обработка которой происходит в прерывании, что откладывает пробуждение потоков и, соответственно, тормозит работу планировщика и системы в целом.

Кроме того, существующая архитектура выглядит нагроможденной, когда должна быть максимально простой. Возникает вопрос, насколько эффективна реализация, и можно ли сделать ее проще и продуктивней.

Примитивы синхронизации

При работе над данной курсовой работой я изучила различные методы

межпроцессного взаимодействия[1] (курсивом выделены средства, реализованные в ОСРВ Embox):

- *семафоры*
- *мьютексы*
- *условные переменные*
- мониторы
- передача сообщений
- *барьеры*

Из средств IPC при написании этой курсовой работы я в основном занималась примитивами синхронизации. Как говорилось выше, почти все примитивы синхронизации реализованы с помощью механизма ожиданий. Поэтому достаточно хорошо изучить один из них, чтобы составить общее представление обо всех. Рассмотрим основные моменты реализации мьютекса.

```
struct mutex {
	struct wait_queue wq;
	struct thread *holder;
	struct mutexattr attr;

	int lock_count;
};
```

Листинг 1. Структура мьютекса.

Структура мьютекса представлена в листинге 1. Как видно, структура содержит ссылку на поток, владеющий мьютексом (holder), и очередь wait_queue, которая состоит из потоков, желающих завладеть мьютексом. Работа с этой очередью осуществляется за счет интерфейса wait_queue.

Рассмотрим реализацию функций захвата мьютекса, представленных в листинге 2.

```
int mutex_lock(struct mutex *m) {
int ret = 0;
struct thread *current = thread_self();
```

[1] Все перечисленные методы IPC подробно описан в разделе “Взаимодействие процессов” во второй главе книги “Современные операционные системы” Э. Таненбаума. [1]

```
sched_lock();
{
	while ((ret = trylock_sched_locked(m, current)) != 0) {
		if (ret == -EAGAIN && (m->attr.type & MUTEX_ERRORCHECK)){
			goto out;
		}
		/* We have to wait for a mutex to be released. */
		priority_inherit(current, m);
		wait_queue_wait_locked(&m->wq, SCHED_TIMEOUT_INFINITE); /* Sleep
here... */
	}
}

out:
sched_unlock();
return ret;
}

static int trylock_sched_locked(struct mutex *m, struct thread *current) {
	if(mutex_static_inited(m)) {
		mutex_init(m);
	}

	if (m->holder == current) {
		if (m->attr.type & MUTEX_RECURSIVE){
			/* Nested locks. */
			m->lock_count++;
			return 0;
		}
		if (m->attr.type & MUTEX_ERRORCHECK){
			/* Nested locks. */
			return -EAGAIN;
		}
	}

	if (m->holder) {
```

```
            return -EBUSY;
        }

        m->lock_count = 1;
        m->holder = current;

        return 0;
}
```

Листинг 2. Функция захвата мьютекса. Некритичные для понимания проверки упущены.

Из кода видно, что основные инструкции вызываются при выключенном планировщике. Это сделано для предотвращения ситуации тупика, которая может возникнуть во время использования интерфейса wait_queue. В ОС Linux [2], [4] используется подобная схема, причем в их реализации отключения планировщика не требуется.

Представим себе, что мы разрешаем функциям работать при включенном планировщике. Кроме того, мы пренебрегаем различными проверками в коде, связанными с этим допущением. У нас есть два потока, которые хотят захватить один мьютекс. Один из неблагоприятных сценариев приведен на рис 3. Как видно из рисунка, освобождение мьютекса происходит до того, как второй поток успел уснуть. А значит, что его никто не разбудит, так как событие уже успеет обработаться в прерывании. То есть программа зависает.

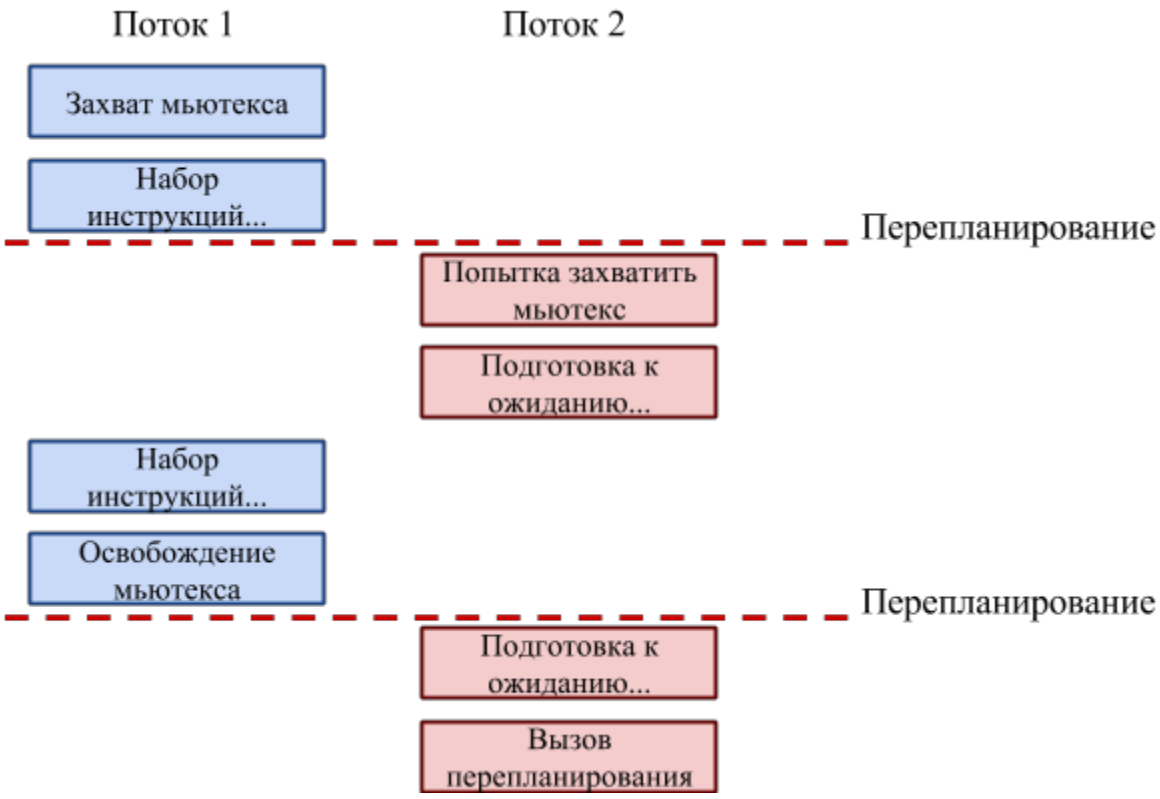

Рис. 3. Сценарий захвата мьютекса при включенном планировщике, который приводит к тупику.

Проблемы реализации

Из всего вышесказанного можно сделать вывод, что существующая реализация неэффективна. Выделим основные проблемы:

- Машина состояний избыточна.
- Промежуточные состояния в машине не используются.
- Архитектура механизма ожиданий громоздкая.
- Использование work замедляет пробуждение потоков.
- Безопасное использование механизма ожидания возможно только при выключенном планировщике.

Описание решения

Как говорилось выше, текущая реализация обладает рядом недостатков и узких мест, которые влекут за собой снижение производительности программы. Моя основная задача - избавиться от этих недостатков, то есть увеличить эффективность работы средств синхронизации. Для этого нужно внести изменения в различные подсистемы:

- Планировщик. В первую очередь изменения затронут машину состояния потока, которая тесно связна с работой планировщика.
- Механизм ожидания событий должен быть переработан так, чтобы его можно было использовать даже тогда, когда планировщик включен, а также упростить архитектуру.
- Изменения в механизме ожидания повлечет за собой изменения в библиотеке средств межпроцессного взаимодействия.

На рис. 4 представлено содержание работы, где красным цветом отмечены компоненты, в которые мне предстояло внести изменения.

Рис. 4. Содержание курсовой работы.

Переработка машины состояний

Как было отмечено выше, существующая машина состояний обладает избыточностью.

Поэтому решено было отказаться от состояний init и exit. Теперь в Embox состояния потоков представляются тремя полями:

- active - запущен на процессоре.
- ready - находится под управлением планировщика, т.е. лежит в runq или запущен на процессоре.
- waiting - ожидает какого-то события.

Описание состояния потока характеризуется комбинациями букв A, W и R. Например, если поток исполняется в данный момент, то его состояние - это AR. При инициализации и завершении потоки имеют состояние W. При инициализации это происходит потому, что с точки зрения планировщика пробуждение и запуск - один и тот же процесс. Состояние W у завершившего свое исполнения потока не имеет особого значения в случае с обычными потоками. В целом его можно понимать так, что планировщик не знает о существовании этого потока и не может им управлять.

Рассмотрим самый простой случай работы потока, который не использует механизм ожидания. Эта ситуация представлена на Рис. 5. Другие случаи будут рассмотрены после описания переработанного механизма ожиданий.

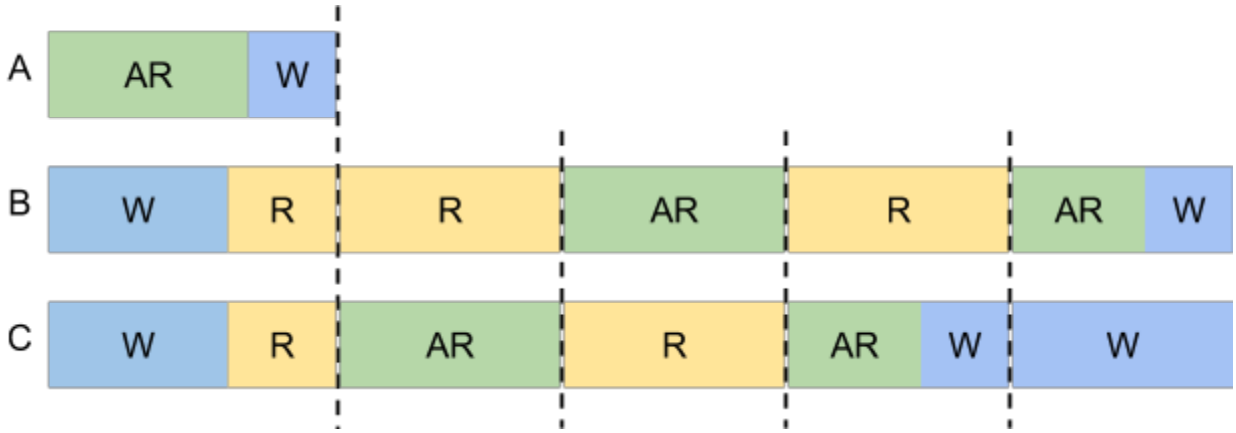

Рис. 5. Псевдопараллельное исполнение потоков, механизм ожидания событий не используется.

Вертикальными прерывистыми чертами показаны моменты перехода ресурсов процессора от одного потока к другому, то есть произошло перепланирование.

Активное состояние - это AR, а когда поток помещается в runq и ждет, когда планировщик передаст ему управление, пометка A снимается. В данном примере пометка W у потоков означает начальное или конечное состояние.

Переработка механизма ожидания событий

Первое, что было сделано для улучшения работы механизма ожиданий - это отказ от work. Новый механизм ожиданий содержит два интерфейса, первый из которых работает с одним конкретным потоком, а другой - с очередью потоков. Причем интерфейс с очередью полностью основан на интерфейсе без очереди, то есть является его оберткой. Новая архитектура представлена на рис. 6.

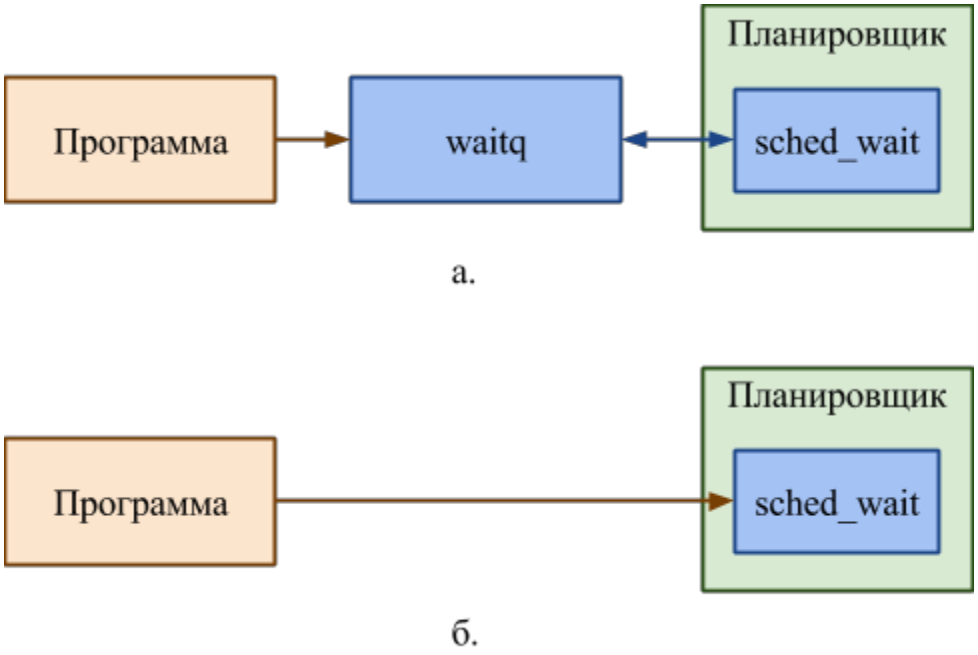

Рис. 6. Переработанная архитектура механизма ожидания событий: а) с использованием очереди, б) без использования очереди.

Основная проблема, которую решает новая реализация - это использование функций при включенном планировщике. Это достигается за счет учета промежуточного состояния в машине состояний потока. Когда поток засыпает, он из активного состояния AR переходит в состояние ожидания W. Рассмотрим шаги этого процесса:

1. Поток выполнил ряд инструкций перед тем, как уснуть. Сейчас его состояние AR.
2. Поток выполняет подготовку, используя функцию waitq_prepare() или sched_wait_prepare() в зависимости от выбранного интерфейса. Обе функции переводят поток в состояние ARW, первая функция еще и добавляет поток в очередь waitq.

3. Поток вызывает перепланирование с помощью функции ожидания sched_wait() или sched_wait_timeout(). Планировщик видит, что поток готов ко сну, то есть у него есть пометка W. Планировщик не кладет поток в очередь потоков на планирование, снимает пометки A и R.
4. Другой поток будит спящий, используя функции waitq_wakeup(), waitq_wakeup_all() или sched_wakeup().
5. Первый поток проснулся, и перед тем, как продолжить работу, он вызывает функцию waitq_cleanup() или sched_wait_cleanup(), чтобы вернуть потоку корректное состояние, а в случае с очередью, удалить поток из нее.

Этот сценарий описан на рис. 7. Темно-зеленым цветом на рисунке показан вызов перепланирования.

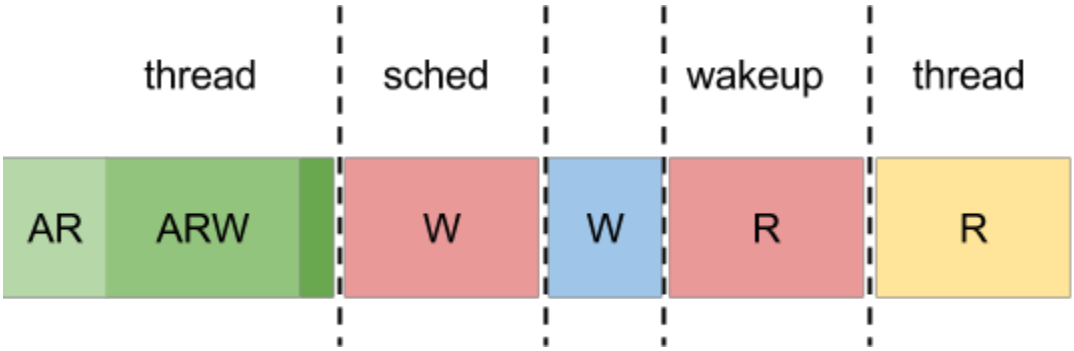

Рис. 7. Псевдопараллельное исполнение потоков, поток переходит в состояние ожидания, а после пробуждается.

При этом все вышеописанное происходит при включенном планировщике. В прошлой реализации это могло привести к зависанию программы. Рассмотрим, что произойдет теперь, если прерывание повлечет за собой ожидаемое событие, теперь уже без подробного описания работы функций:

1. Поток подготовился ко сну, его состояние - ARW.
2. Произошло прерывание, которое вызвало ожидаемое событие.
3. Поток будят в прерывании, его состояние - AR.
4. Вызывается перепланирование. Планировщик видит, что поток активен и готов к исполнению. Он поступает с ним, как при обычном перепланировании по таймеру, кладет в очередь готовых к исполнению потоков.

Эта ситуация представлена на рис. 8. Видно, что поток продолжает нормальное

исполнение. Таким образом, новый механизм ожиданий лишен недостатков прошлой реализации.

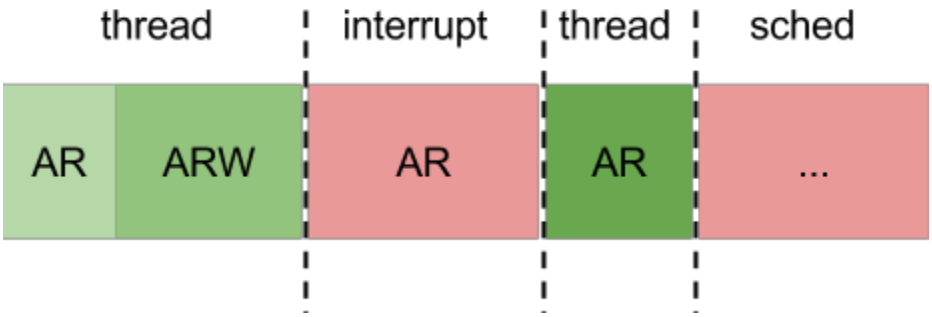

Рис. 8. Псевдопараллельное исполнение потоков, поток переходит в промежуточное состояние, после чего происходит событие.

Пример использования нового интерфейса приведен в листинге ?, где можно увидеть, как нужно использовать интерфейс waitq.

```
#define WAITQ_WAIT_TIMEOUT(wq, cond_expr, timeout) \
   ((cond_expr) ? 0 : ({                                                \
           struct waitq_link wql;                                       \
           clock_t __wait_timeout = timeout == SCHED_TIMEOUT_INFINITE ? \
                   SCHED_TIMEOUT_INFINITE : ms2jiffies(timeout);        \
           int __wait_ret = 0;                                          \
           waitq_link_init(&wql);                                       \
                                                                        \
           threadsig_lock();                                            \
           do {                                                         \
                   waitq_wait_prepare(wq, &wql);                        \
                                                                        \
                   if (cond_expr)                                       \
                       break;                                           \
                                                                        \
                   __wait_ret = sched_wait_timeout(__wait_timeout,      \
                                                   &__wait_timeout);    \
           } while (!__wait_ret);                                       \
                                                                        \
           waitq_wait_cleanup(wq, &wql);                                \
           __wait_ret;                                                  \
```

```
        }))
```

Листинг 3. Макрос ожидания с таймаутом, использующий очередь спящих потоков.

Переработка примитивов синхронизации

Переработка примитивов синхронизации заключалась в основном в том, чтобы переделать функции под использование нового интерфейса. Главное, теперь нет необходимости отключать планировщик. Рассмотрим в качестве примера новую версию функции захвата мьютекса, которая представлена в листинге 4.

```
int mutex_lock(struct mutex *m) {
    struct thread *current = thread_self();
    int errcheck;
    int ret, wait_ret;

    assert(m);

    errcheck = (m->attr.type & MUTEX_ERRORCHECK);

    wait_ret = WAITQ_WAIT(&m->wq, ({
       int done;

       ret = mutex_trylock(m);
       done = (ret == 0) || (errcheck && ret == -EAGAIN);
       if (!done)
               priority_inherit(current, m);
       done;

    }));

    if (!wait_ret)
       ret = wait_ret;

    return ret;
}
```

Листинг 4. Переработанная функция захвата мьютекса.

Сравнение реализаций

Для сравнения реализаций был написан тест, в котором несколько потоков заходят в критическую область, огражденную мьютексом. В каждом тесте потоки инкрементируют глобальную переменную, таким образом к концу теста значение переменной должно быть равно числу потоков. Его результаты представлены на диаграмме 1.

Диаграмма 1. Тест с мьютексами на 20 потоков.

Также был проведен тест на отзывчивость, где было замерено время между освобождением мьютекса одним потоком и захватом его другим, его результаты представлены на диаграмме 2.

Замеры в обоих тестах были проведены с помощью инструмента профилирования, реализованного в ОСРВ Embox.

Замеры показали, что текущая реализация эффективней старой, а также относительная разница между средним и самым медленным результатами меньше, что говорит об устойчивости работы механизма ожиданий.

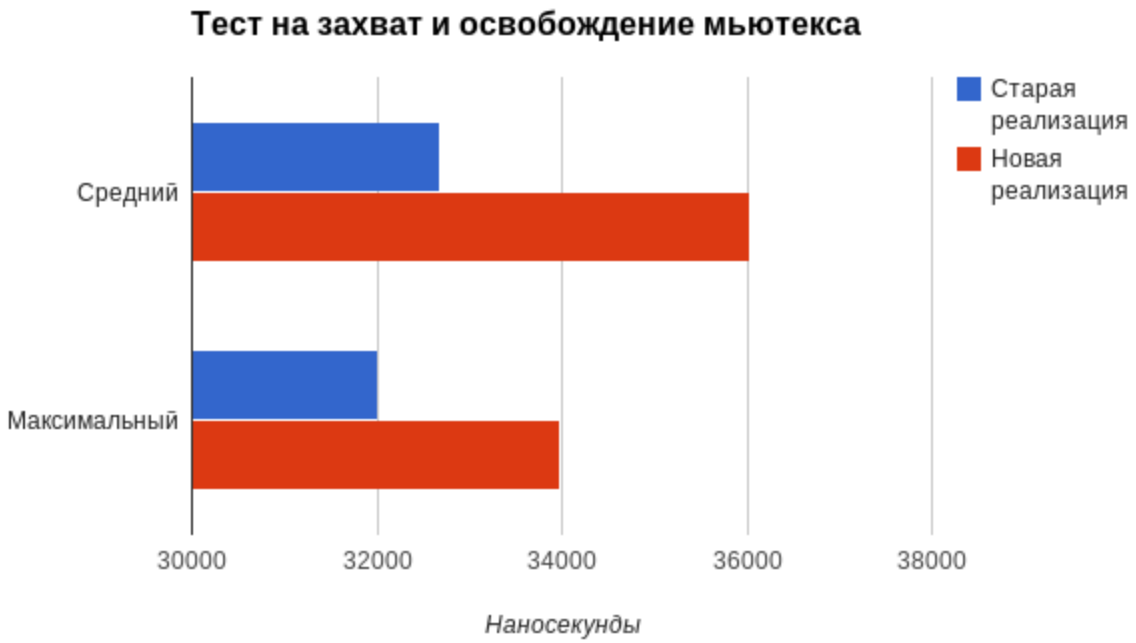

Диаграмма 2. Тест на захват и освобождение мьютекса

Заключение

В ходе работы были изучены различие средства межпроцессного взаимодействия и реализация средств синхронизации в ОСРВ Embox. Были обнаружены проблемные места в реализации, в том числе в связанных с ней таких компонент, как механизм ожидания событий и планировщик, которые приводили к низкой производительности при работе больших приложений.

В качестве решения проблемы была предложена и внедрена новая реализация машины состояний потоков, механизма ожидания событий, основных функций планировщика, а также были переработаны средства синхронизации, ранее реализованные в ОСРВ Embox.

В качестве доказательства того, что новая реализация более эффективна, был проведен сравнительный анализ на основе тестов с использованием средств профилирования.

Список литературы

[1] “Современные операционные системы” Э.Таненбаум, 3-е изд.
[2] The Linux Kernel Archives - URL: https://www.kernel.org/pub/
[3] “Архитектура компьютера и проектирование компьютерных систем” Паттерсон Д, 4-е изд.
[4] “The Linux Kernel” David A Rusling, Version 0.8-3 - URL: http://www.tldp.org/LDP/tlk/tlk.html