ОПТИМИЗАЦИЯ СИНХРОНИЗАЦИИ ПАРАЛЛЕЛЬНЫХ ПРОГРАММ ДЛЯ ВЫЧИСЛИТЕЛЬНЫХ СИСТЕМ С ОБЩЕЙ ПАМЯТЬЮ

Гайдай А.В.

Сибирский государственный университет телекоммуникаций и информатики, Новосибирск, Россия, E-mail: AVGaidai@ngs.ru

В данной работе затрагивается проблема производительности операции захвата мьютекса при возникновении конфликтной ситуации (contention): когда уже захваченным мьютексом пытаются завладеть другие потоки. Такие ситуации негативно сказываются на времени выполнения операции захвата. Реализованный программный пакет «mutex-optimizer» и его функциональные возможности, включают в себя средства профилирования и оптимизации многопоточных программ, благодаря которым достигается увеличение производительности при использовании мьютексов. Эффективность разработанного пакета исследовалось на синтетических тестах (microbenchmark).

Ключевые слова: мьютекс, конфликт, профилирование, оптимизация, многопоточность.

1. Введение

Многопоточность (multithreading) – одна из наиболее интересных и актуальных тем в области информационных технологий. Актуальность этой темы особенно велика, в связи с широким распространением многоядерных процессоров и их стремительном развитии.

В многопоточных приложениях имеется ряд сложностей, связанных с синхронизацией одновременного доступа потоков к одним и тем же участкам памяти. Для решения этих проблем существуют различные примитивы синхронизации, один из которых называется мьютексом.

Мьютекс [1, 2] (mutex, mutual exclusion – взаимное исключение) позволяет создавать в коде программы критические секции, выполнение которых возможно только одним потоком в каждый момент времени. Мьютексы могут находиться в одном из двух состояний – открытом или закрытом. При выполнении потоком операции захвата (mutex lock), мьютекс переводится в закрытое состояние – становится недоступен для захвата другими потоками. Когда поток освобождает мьютекс (mutex unlock), его состояние становится открытым – доступным для захвата.

Однако, при попытке захвата мьютекса может возникать ситуация, называемая конфликтом (contention) [3], при которой несколько потоков пытаются одновременно захватить мьютекс. Операция захвата мьютекса в случае возникновения конфликта требует больше времени, чем операция захвата без возникновения конфликтной ситуации за счёт негативных явлений процессорного кэша. Мьютекс является обычной переменной, которая кэшируется, используясь на разных ядрах. Таким образом, согласно, протоколу поддержания когерентности кэшей MESI [4, 5], при одновременном обращении нескольких потоков к одному мьютексу, фактически происходит одновременный доступ к одной и той же области памяти и, как следствие, используемые данные становятся невалидными для других ядер. Повышение производительности операции захвата мьютекса достигается за счёт уменьшения накладных расходов при работе с критическими секциями.

Стандартом реализации мьютексов при разработке параллельных программ является динамическая библиотека pthread в GNU libc, однако, реализация в текущей

версии glibc не учитывает динамически изменяющееся состояние конфликта при захвате мьютекса.

В данной работе реализован адаптивный алгоритм захвата мьютекса, который учитывает динамически изменяющиеся характеристики критических секций, используя результаты предварительного профилирования.

2. Адаптивный алгоритм захвата мьютекса

При конкуренции потоков в конечном итоге только один из них сможет захватить мьютекс, а остальные будут вынуждены либо продолжать пытаться захватить мьютекс в цикле (spinlock блокировка), либо, не тратить процессорное время и освободить вычислительные ресурсы для выполнения другого потока, перейдя в состояние ожидания [1, 2]. Обычно каждый конфликтующий поток сначала пытается захватить мьютекс с помощью spinlock'а и только в случае неудачи перестаёт расходовать временные ресурсы процессора (или ядра) погружаясь в «сон» на определённое время. Очевидно, что чем больше количество конфликтующих потоков, тем менее эффективно будет расходоваться процессорное время. Однако, если сократить количество попыток захвата мьютекса, перед тем как отправить тот или иной поток «спать», общее время простоя также сократиться. Таким образом для оптимизации процесса использования мьютексов необходима некоторая статистика возникновения возможных конфликтных ситуаций при работе с ним. Данная информация – это результат предварительного профилирования.

Реализованный алгоритм захвата мьютекса можно разделить на два этапа:

1) Профилирование.
2) Оптимизация.

2.1. Этап профилирования

На этапе профилирования производится подсчёт конфликтных ситуаций, возникающих при захвате мьютекса. Вся информация записывается в заранее созданную хэш-таблицу, где ключом является адрес конкретного мьютекса, а значением – частота возникновения искомой ситуации для него. На основе полученных данных высчитывается коэффициент загруженности для каждого мьютекса по формуле 2.1.

$$K_i = \frac{C_i}{A}, \qquad \text{(Формула 2.1)}$$

где K_i – коэффициент загруженности для *i*-ого мьютекса; C_i – количество возникновений ситуаций конфликта для *i*-ого мьютекса; A – среднее значение возникающих конфликтных ситуаций для всех используемых мьютексов.

Результат работы профилировщика сохраняется в файл «stat.bin».

Листинг 2.1 – Фрагмент кода профилировщика (profiler.c)

```
#include "htable.h"

#define  OUTFILE    "stat.bin"

struct hash_elem *table;          // Hash table (100 elements)

void profiler_init () __attribute__((constructor));
void profiler_exit () __attribute__((destructor));
```

```
/*
 * Initializer profiler
 */
void profiler_init ()
{
    table = hash_table_create ();

    orig_pthread_mutex_lock = dlsym (RTLD_NEXT, "pthread_mutex_lock");
}

/*
 * Exit function of profiler
 */
void profiler_exit ()
{
    avg_contention /= mutex_count;
    info_print (table, avg_contention);
    write_result (OUTFILE, table, avg_contention);

    hash_table_free (table);
}
```

На листинге 2.2 приведён код синтетического теста для иллюстрации этапа профилирования.

Листинг 2.2 – Код синтетического теста

```
#include <pthread.h>
#define NUM_THREADS    2


pthread_mutex_t mutex;
pthread_barrier_t barrier;

int shared_obj;


void *foo (void *arg)
{
    pthread_barrier_wait (&barrier);
    for (int i = 0; i < 500000; ++i) {
        pthread_mutex_lock (&mutex);
        shared_obj = i;
        pthread_mutex_unlock (&mutex);
    }

    pthread_exit (NULL);
}

int main (int argc, char *argv[])
{
    pthread_mutex_init (&mutex, NULL);
    pthread_barrier_init (&barrier, NULL, NUM_THREADS);

    pthread_t threads[NUM_THREADS];

    for (int i = 0; i < NUM_THREADS; ++i)
        pthread_create (&threads[i], NULL, foo, NULL);

    for (int i = 0; i < NUM_THREADS; ++i)
        pthread_join (threads[i], NULL);
```

```
    pthread_mutex_destroy (&mutex);
    pthread_barrier_destroy (&barrier);

    return 0;
}
```

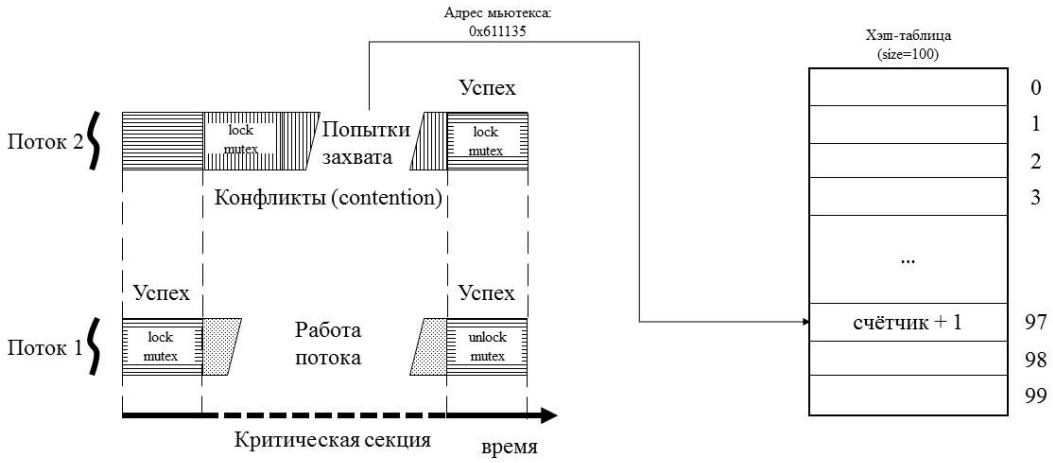

Рис. 2.1 – Схематичное представление этапа профилирования

2.2. Этап оптимизации

На данном этапе производится оптимизация «проблемных мьютексов», то есть таких, коэффициент загруженности для которых больше единицы. Таким образом при работе с загруженным мьютексам, в случае возникновения конфликтной ситуации, поток, конкурирующий за доступ, будет сразу же переходить в режим ожидания на указанное постоянное время.

Листинг 2.3 – Фрагмент кода оптимизатора (optimizer.c)

```
/*
 * Hooking function "pthread_mutex_lock"
 */
int pthread_mutex_lock (pthread_mutex_t *__mutex)
{
    int stat;
    if((stat = find_hash_elem((void*)__mutex, table))!=-1){
        if (table[stat].cont > avg_contention) {
            while((stat=pthread_mutex_trylock(__mutex))
                    == EBUSY) {
                usleep (10);
            }
            return stat;
        } else {
            return orig_pthread_mutex_lock (__mutex);
        }
    } else {
        return orig_pthread_mutex_lock (__mutex);
    }
}
```

3. Функциональная структура программного пакета «mutex-optimizer»

Для профилирования приложения используется методика подмены стандартных функций библиотеки Pthread. Таким образом «mutex-optimizer» включает в себя две динамические библиотеки: profiler.so — позволяет строить статистику возникновения конфликтных ситуаций; optimizer.so — позволяет оптимизировать процесс захвата мьютекса по результатам профилирования. Каждая библиотека используется в определённом режиме функционирования: профилирования или оптимизации.

Выбор режима выполняется при помощи переменной среды окружения LD_PRELOAD:

[name@host] $ LD_PRELOAD=profiler.so ./prog.out — режим профилирования;
[name@host] $ LD_PRELOAD=optimizer.so ./prog.out — режим оптимизации.

Программный пакет «mutex-optimizer» условно можно разделить на две части:
• Модуль профилирования;
• Модуль оптимизации.

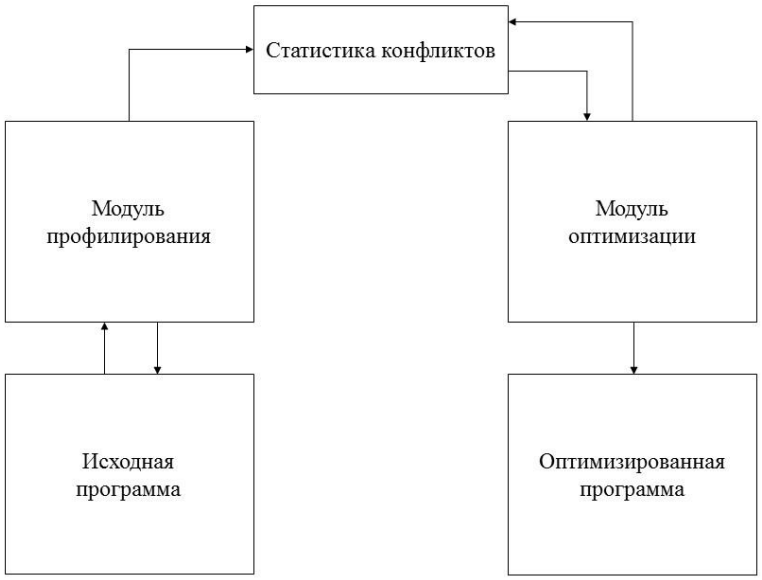

Рис. 3.1 – Структура функционирования программного пакета «mutex-optimizer»

3.1. Структура модуля профилирования

Конструктор, определяющий адреса всех оригинальных функций, которые в дальнейшем будут перехвачены. Также в конструкторе создаётся и инициализируется хэш-таблица, хранящая информацию о частоте возникновения конфликтов при работе с мьютексами.

Функция pthread_mutex_lock () подменяющая оригинал: если мьютекс используется впервые — информация о нём добавляется в таблицу; если мьютекс свободен и используется не в первый раз — выполняется оригинал функции; если мьютекс занят и, следовательно, используется не в первый раз — увеличивается счётчик конфликтных ситуаций для данного мьютекса.

Деструктор, записывающий статистику во внешний файл и очищающий память, выделенную под хэш-таблицу.

3.2. Структура модуля оптимизации

Конструктор, определяющий адреса всех оригинальных функций, которые в дальнейшем будут перехвачены.

Функция pthread_mutex_lock () подменяющая оригинал. В этой функции происходит сравнение показателей частоты возникновения конфликтов при захвате мьютекса для частного случая и для общего (средней показатель). Если показатель частного случая больше среднего, то уменьшается количество попыток захвата перед тем как отправить работающий поток «спать».

4. Результаты экспериментов

Эффективность разработанного пакета исследовалось на синтетических тестах (microbenchmark, листинг 2.2).

Тесты запускались на кластере Jet, компилятор GCC версии 4.8.3, операционная система GNU/Linux (Fedora 20). Необходимые характеристики данного кластера можно увидеть в таблицах 6.6 и 6.7. Кластер Jet укомплектован 18 вычислительными узлами, управляющим узлом, вычислительной и сервисной сетями связи, а также системой бесперебойного электропитания.

Таблица 4.1 – Конфигурация вычислительного узла кластера Jet

Процессор	Intel® Xeon CPU E5420 (2.50 ГГц) x2
Количество ядер	8 x 2 = 16
Оперативная память	8 GB (4 x 2GB PC-5300)
Жесткий диск	SATAII 500GB (Seagate 500Gb Barracuda)

Таблица 4.2 – Конфигурация управляющего узла кластера Jet

Процессор	Intel® Xeon CPU E5420 (2.50 ГГц) x2
Количество ядер	8 x 2 = 16
Оперативная память	16 GB (8 x 2GB PC-5300)
Жесткий диск	3 x SATAII 500GB (Seagate 500Gb Barracuda)

При первом эксперименте входные данные были следующие: мьютекс – 1, количество итераций со входом в критическую секцию – 10 000 000, количество параллельных ветвей – 1, 2, 3, 4, 5, 6, 7, 8.

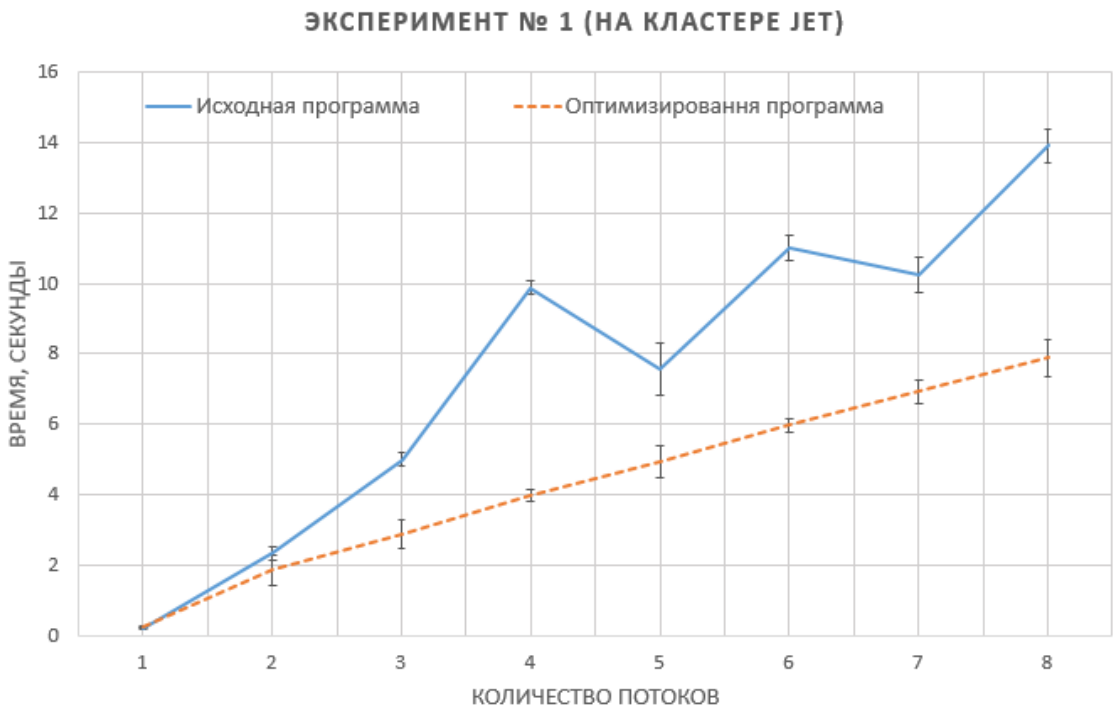

Рисунок 4.1 – График зависимости времени выполнения тестовой параллельной программы от количества параллельных ветвей в ней

Как видно из графика, при разных входных данных оптимизированная версия программы показывает лучшее время, за исключением работы при одном потоке, в этом случае конфликтов за доступ к мьютексу нет и, следовательно, возникают накладные расходы на инициализацию хэш-таблицы и освобождение памяти, выделенной под неё. Максимальное ускорение составило 2.57, среднее – 1.78, минимальное – 0.92. Нелинейный характер кривой, показывающий время выполнения неоптимизированной версии тестовой программы, объясняется особенностями работы планировщика.

При втором эксперименте входные данные были следующие: мьютекс – 1, количество итераций со входом в критическую секцию – 10 000 000, количество параллельных ветвей – 10, 20, 30, 40, 50, 60, 70, 80, 90, 100.

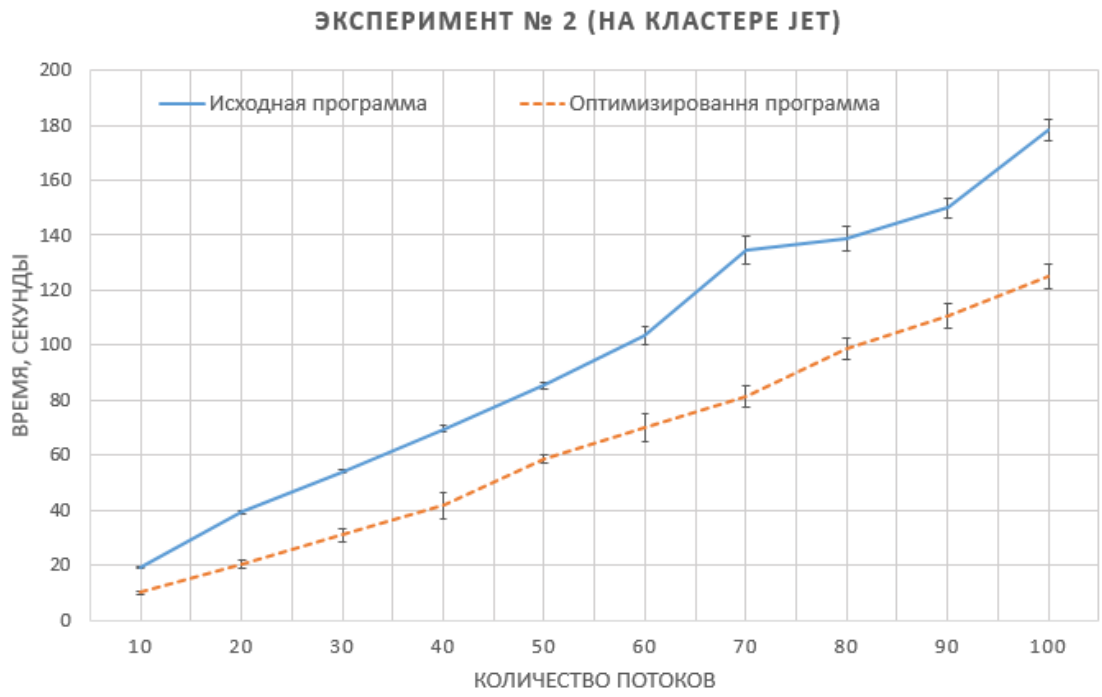

Рисунок 4.2 – График зависимости времени выполнения тестовой параллельной программы от количества параллельных ветвей в ней

Как видно из графика, при разных входных данных оптимизированная версия программы показывает лучшее время. Максимальное ускорение составило 1.84, среднее – 1.34, минимальное – 1.50.

Для измерений были построены доверительные интервалы [6], которые присутствуют на всех графиках, построенных в данной исследовательской работе. Коэффициент Стьюдента равен 8.610.

5. Заключение

В ходе данной работы был реализован программный пакет «mutex-optimizer». Эффективность пакета была экспериментально подтверждена с помощью синтетических тестов в которых были созданы оптимальные условия (большое количество возникающих в короткий период времени конфликтных ситуаций) для демонстрации положительных результатов. Проведён анализ характера получившихся графиков, рассчитано минимальное, среднее и максимальное ускорение для разных конфигураций системы и входных данных. Изучены основы оптимизации синхронизации параллельных программ для вычислительных систем с общей памятью. Освоена методика профилирования работы мьютексов в пользовательском пространстве операционной системы.

Литература

1 Эндрюс Г.Р. Основы многопоточного, параллельного и распределённого программирования. Пер. с англ. – М.: Вильямс // 2003. – 512 с.

2 Бовет. Д., Чезати М. Ядро Linux, 3-е изд.: Пер. с англ. // Спб. БХВ-Петербург, 2007. – 1104 с.

3 Herlihy M., Shavit N. The Art of Multiprocessor Programming, Revised Reprint // Elsevier – 2012.

4 John L. Hennessy; David A. Patterson (16 September 2011). Computer Architecture: A Quantitative Approach. Elsevier. // Retrieved 25 March 2012.

5 Drepper U. Memory part 2: CPU caches. // Retrieved from the internet on 10 November 2010. – С. 1-53.

6 Курносов М.Г., Пазников А.А. Основы теории функционирования распределенных вычислительных систем. // Новосибирск: Автограф, 2015. – 52 с.