Санкт-Петербургский государственный университет
Математико-Механический факультет
Кафедра системного программирования

Разработка профайлера уровня ядра ОС Windows
Анализ блокировок и ожиданий

Курсовая работа студента 344 группы
Анисимова Константина Александровича

Научный руководитель…….....Баклановский М.В.
старший преподаватель
кафедры системного программирования

Санкт-Петербург, 2013

Оглавление:

Введение.

Оптимизация приложений является неотъемлемой частью разработки и поддержки современных высокоэффективных и отказоустойчивых приложений. Но для того чтобы что-то оптимизировать нужно знать, где именно нужно оптимизировать, То есть найти те самые 20% кода, которые, согласно принципу Парето, работают 80% времени.

Для этого нужны специальные программные средства, которые позволяют оценить производительность, выявить причины замедления и непосредственно указать на те части кода, которые необходимо переписать более рационально. Такие программы называются профайлерами. Они позволяют путем минимального вмешательства в процесс исполнения приложения, измерять и фиксировать различные параметры производительности, такие как:

- время исполнения программы или функции (в тактах или миллисекундах)
- частоту вызова различных функций
- количество кеш промахов и попаданий
- критические секции (блокировки)
- объем используемой памяти
- стек вызовов

Способы профилирования

Есть два принципиально разных подхода к профилированию программ. Это семплирование и инструментирование.

Семплирование – способ профилирования, суть которого заключается в том, что мы ни каким образом не вмешиваемся в логику исполнения программы. А информацию для последующего изучения получаем путем периодического анализа окружения, в котором профилируемая программа исполняется. Такой информацией может служить стек вызовов, количество выделенной памяти, и т.п. Плюсами такого подхода считаются легкость реализации и низкие накладные расходы

Инструментирование - способ профилирования, в котором мы изучаем состояние программы в определенные, нужные нам моменты времени. Это делается путем внедрения своего кода, который производит необходимые замеры, в исследуемую программу. То есть мы можем сами выбрать где и когда можно будет получить данные о состоянии анализируемой программы. Внедрение может происходить как статически – изменением исполняемого файла до его запуска или во время компиляции, так и

динамически – прямо во время исполнения программы. Таким образом достигается значительная гибкость и точность по сравнению с семплированием. Ценой за такие возможности является вмешательство в код исполняемой программы, и, как следствие, некоторое искажение полученных результатов.

Как может показаться на первый взгляд семплирование, по совокупности плюсов и минусов, наиболее выгодный способ профилирования ресурсоемких программ. Подавляющее большинство программных продуктов также использует семплирование. Но в 2012 году было показано[1], на примере ряда промышленных профайлеров, использующих метод семплирования, что в ряде не очень редких случаев, данный способ дает либо высокие накладные расходы, либо высокую погрешность измерений. Также для некоторых задач семплирование не применимо в принципе. В частности при анализе блокировок и ожиданий, нужно как можно точнее знать, когда произошел вызов функции, отвечающей за работу с примитивом синхронизации. А поскольку это является одной из основных целей данного проекта, то семплирование применять нельзя.

В операционной системе Windows есть встроенное средство для создания, обработки, хранения и буферизации событий различных типов, среди которых есть события, отвечающие за профилирование – Event Tracing Tools for Windows(ETW). Также возможно создавать свои типы событий, обработчики и т.п., но только в пользовательском режиме. Но на уровне ядра, возможности этой системы очень сильно ограничены. В частности, можно пользоваться только некоторыми заранее предопределенными типами событий, среди которых есть всего несколько, связанных с профилированием.

Существующие решения

Существует множество программных занимающихся анализом производительности. Самыми известными из таких продуктов являются:

-Intel VTune Amplifier XE

AMD CodeAnalyst Performance Analyzer

Windows Performance Analysis Toolkit(Microsoft xPerf).

Стоит отметить все анализаторы упомянутые выше нацелены на профилирование программ работающих с привилегиями обычного пользователя. Хотя некоторые из них имеют некоторые возможности для анализа программ, работающих на уровне ядра ОС.

Большинство из них использует ETW. Но необходимость профилирования может возникать и при разработке программ исполняющихся на уровне ядра операционной системы.

	xPerf	VTune
Семплирование (user mode)	+	+
Семплирование (kernel mode)	+	+
Инструментирование (user mode)	-	+
Инструментирование (kernel mode)	- (ETW)	-

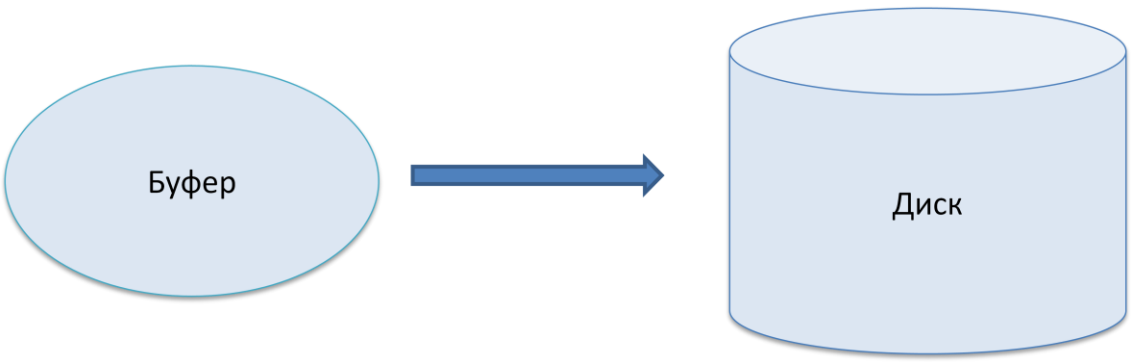
Как можно видеть из таблицы выше, ни один из существующих профайлеров не поддерживает в полной мере инструментирование в ядре Windows. А семплирование, как было сказано ранее, не подходит для достижения поставленной цели. Microsoft xPerf поддерживает события ETW, относящиеся к анализу производительности на уровне ядра, но эти события могут генерироваться только при вызове небольшого фиксированного набора функций ядра Windows. Этот набор определяется разработчиками ОС, и не может быть изменен из вне. Хотя в нем и есть функции работы с блокировками, но все равно это сильно ограничивает возможности, и не является инструментированием как таковым.

Описание и постановка проблемы

Поскольку не было найдено существующего решения для инструментирования, а также анализа блокировок и ожиданий на уровне ядра, был реализован[2] инструмент позволяющий проводить динамическое инструментирование практически любых функций программ уровня ядра, зная адрес загрузки программы и файл с информацией для отладки(pdb).

Так как скорость записи на диск слишком мала, а задержки велики, и при этом запись на диск как таковая требует немало системных ресурсов, что сильно влияет на исследуемую программу, и следовательно на достоверность полученных результатов. Поэтому для хранения собираемой информации используется буфер в оперативной памяти. Так как анализ блокировок имеет наибольший смысл на многопроцессорных системах, то нужна синхронизация для доступа к буферу. Для того чтобы избежать проблем с синхронизацией, каждому ядру процессоров, сопоставляется свой буфер.

Но в процессе эксплуатации на промышленных продуктах, были выявлены некоторые проблемы. Самой большой из них было быстрое заполнение внутреннего буфера. Информация обо всех события сохранялась в буфер, который после заполнения сбрасывался на диск. При размере буфера 1GB на процессорное ядро сброс на диск занимает значительное время. Анализ событий происходит уже после окончания работы программы.

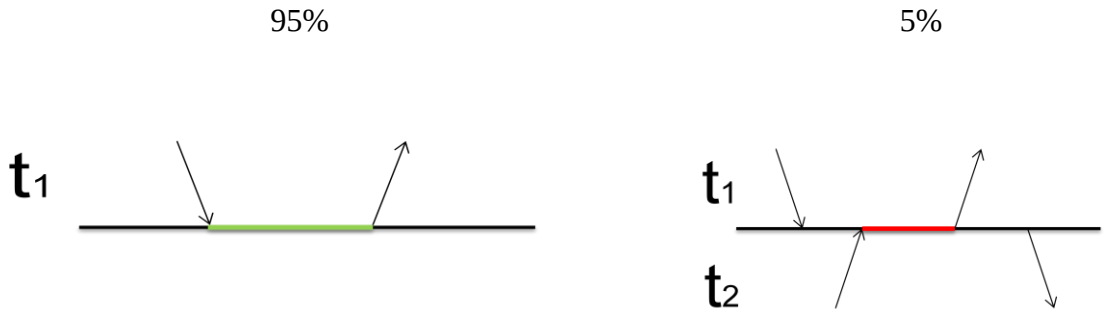


В многопоточных программах уровня ядра для получения более высокой производительности очень часто используется примитив синхронизации, называемый

SpinLock или «крутящаяся блокировка». То есть поток, который ждет освобождения блокировки, все время находится в активном состоянии, непрерывно опрашивая ее состояние. Таким образом не тратится время на переключение между разными процессами, но процессор во время ожидания работает практически в холостую. Поэтому использование этих блокировок выгодно, только если время работы с блокировкой очень мало. А это в свою очередь влечет очень частый вызов функций для работы с этими блокировками. Таким образом при инструментировании SpinLock'ов возникает огромное количество вызовов соответствующих функций. Тестирование на реальном промышленном проекте показало, что таких вызовом может происходить десятки миллионов в секунду на каждом из процессорных ядер. Это при информации о вызове в 64 байта выливается в гигабайт данных в секунду. Что ведет к заполнению буфера за секунду и следующему за ним минутному сбросу на диск. Таким образом эффективное время работы составляло меньше 1/60 от общего времени, что неприемлемо.

Решение проблемы

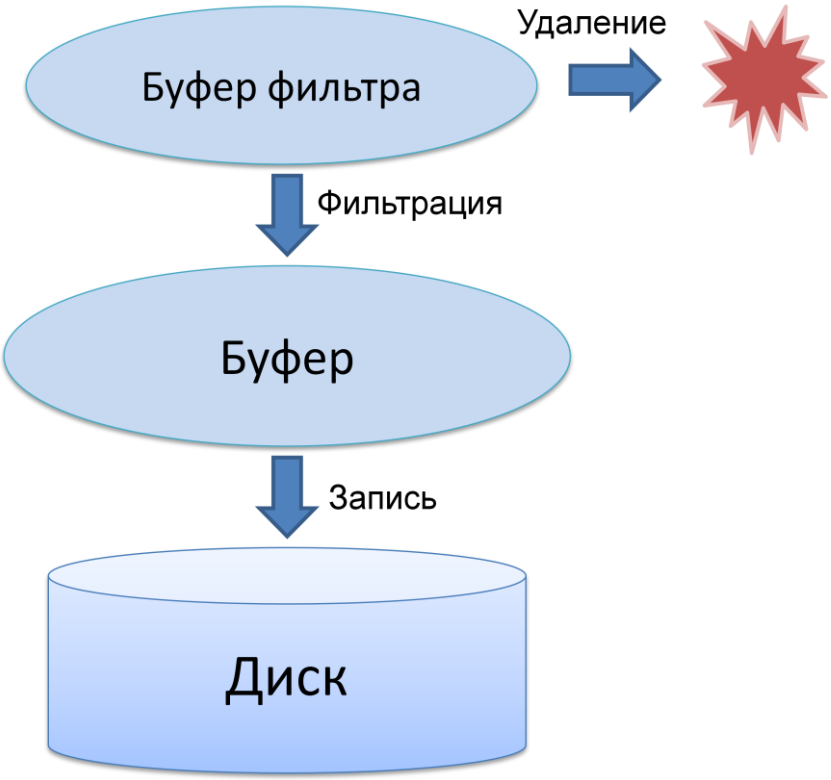
При анализе поставленной проблемы было выявлено, что при анализе блокировок большую часть событий составляют события, в которых не происходит никакого ожидания блокировки. То есть когда ровно один процесс хочет захватить или уже захватил блокировку. О таких событиях важно знать только их количество, так как оптимизировать синхронизацию в такой ситуации не имеет смысла. Более важными событиями являются те, во время которых происходило ожидание, то есть процессор работал вхолостую. Это именно те места, на которые мы должны указать разработчику профилируемой программы для последующей оптимизации. Практика показала, что в среднем всего около 5 % составляют ценные данные



Пример ненужных и ценных данных.
Захват и освобождение блокировки изображены стрелочками

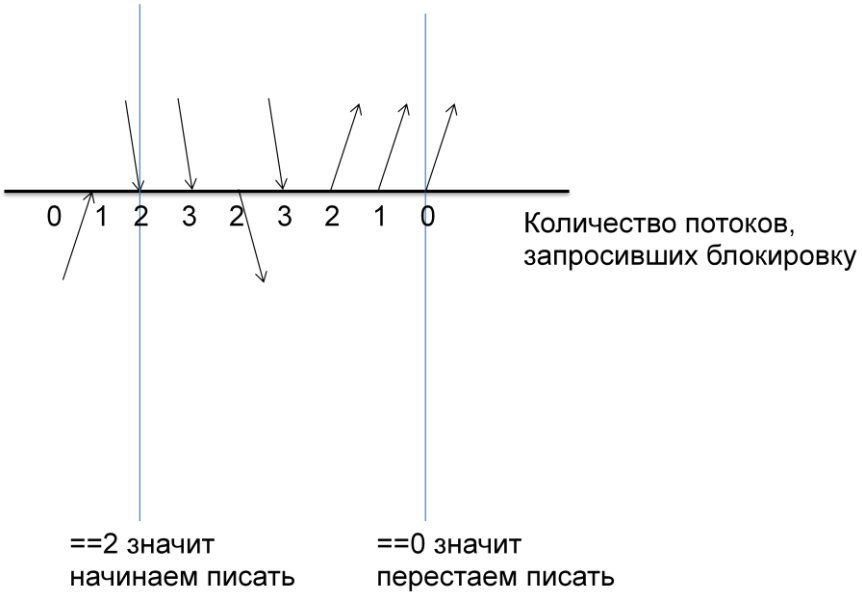
Если мы сможем в процессе сбора информации, «на лету» обнаруживать (фильтровать) события ненужные для последующего анализа, то можно существенно уменьшить скорость заполнения буфера. Это может увеличить время полезной работы до 20 раз и больше. Но анализ на лету занимает некоторое время, которое может сказываться на точности результатов. Таким образом наивысшим приоритетом при фильтрации является скорость работы.

Для достижения поставленных целей в архитектуре программы было сделано существенное изменение.



Поскольку при поступлении некоторого события без анализа предшествующих событий мы не можем сделать вывод том, надо ли сохранять его на диск или нет. Поэтому был добавлен промежуточный буфер, в котором хранятся необработанные события. Для обработки событий удобно разделять их на блоки. Назовем блоком событий минимальную последовательность событий, перед которой и после которой блокировка была свободна. Тогда блок который нам не нужно сохранять на диск это захват и следующее непосредственно за ним освобождение блокировки. Все остальное нужно сохранить на диск для последующей обработки.

Осталось понять, каким образом определять окончание блока событий. Для этого можно заметить, что после каждой блокировки идет ее освобождение. Таким образом, как только количество захватов блокировки станет равным количеству освобождений, то блок закончен. Данная схема является конечным автоматом, в котором по событию захвата блокировки переход осуществляется в следующее состояние, а по событию освобождения – в предыдущее состояние. Лучше всего реализовать этот алгоритм в виде переменной-счетчика, увеличивая и уменьшая ее на захват и освобождение соответственно. Счетчик в итоге будет отображать количество потоков, которые работают с этой блокировкой в данный момент. Таким образом, если счетчик достиг значения 2, то значит у нас второй поток будет некоторое время ждать, значит этот блок нужно сохранять на диск пока он не закончится. Пример работы алгоритма можно видеть на схеме ниже.



Есть некоторые ситуации, которые данный алгоритм не обрабатывает. Это если начало работы пришлась на середину некоторого блока, если один поток попытался захватить блокировку несколько раз и если один поток освободил блокировке несколько раз. При этом во всех случаях кроме последнего все события будут проходить фильтр и записываться на диск для дальнейшего анализа.

Во время реализации были решены некоторые сопутствующие проблемы. В частности до введения промежуточного буфера каждому ядру соответствовал свой буфер, что позволяло избежать проблем с синхронизацией между ядрами. В свою очередь промежуточный буфер должен быть общим для всех ядер. Очевидно, что использовать для синхронизации инструментируемые функции нельзя. Для решения возникшей проблемы пришлось сделать свою собственную реализацию SpinLock’a, не зависящую от операционной системы, с помощью атомарной инструкции процессора LOCK CMPXCHG.

Заключение

В итоге время непрерывной работы профайлера было увеличено с нескольких секунд до минут, что позволяет использовать этот инструмент гораздо более эффективно.

Были произведены замеры накладных расходов профайлера. Времена измерялись с помощью ассемблерной инструкции rdtsc, которая измеряет количество тактов процессора. Измерения производились на функциях, отвечающих за работу со SpinLock’ами в ядре Windows, следующим образом:

```
rdtsc;
KeAquireSpinLock();
KeReleaseSpinLock();
rdtsc;
```

Про измерении были получены следующие результаты. Брался средний результат за 100 запусков.

- Без профайлера 500 тактов
- Без фильтрации 1500 тактов
- С фильтрацией 2700 тактов

Можно увидеть, что накладные расходы увеличились всего 2 раза, при увеличении времени работы профайлера в среднем 10-20 раз.

Литература

[1] Исследование и тестирование семплирующего метода профайлинга на примере профилировщика производительности Intel VTute Amplifier XE 2011 /Одеров Р.С. Серко С.А. Курсовая работа, СПбГУ, Мат-Мех ф-т., кафедра системного программирования 2012 год

[2]Способы размещения своего кода в ядре ОС Microsoft Windows Server 2008 / Одеров Р.С., Тенсин Е.Д. – сборник трудов межвузовской научно практической конференции «Актуальные проблемы организации и технологии защиты информации». СБбНИУ ИТМО, СПб, 2011г. //стр. 100-102

3. Intel 64 and IA-32 Architectures Optimization Reference Manual / Intel Corporation. – June 2012

4. Intel 64 and IA-32 Architectures Software Developer Manual / Intel Corporation. – March 2012

5. Windows Internals (6th edition) / Mark E. Russinovich and David A. Solomon: Microsoft Press. – 2012 год

6. Верификация дизассемблера x86-64 / Тенсин Е.Д. – Конференция «СПИСОК-2012» СПбГУ, 2012 год

7. Event Tracing for Windows // http://msdn.microsoft.com/en-us/library/windows/desktop/bb968803(v=vs.85).aspx