Санкт-Петербургский Государственный Университет

Математико-механический университет

Трассировки OCPB Embox

Курсовая работа студентки 345 группы

Крамар Алины Сергеевны

Научный руководитель А. В. Бондарев

аспирант кафедры системного программирования

Санкт-Петербург

2012

Содержание

Введение

Проект по созданию операционной системы реального времени Embox развивается уже несколько лет. Он накопил довольно большую базу кода (порядка 60 тысяч строк кода только на языке Си), большое количество реализаций тех или иных алгоритмов, включая не только классические описанные в литературе и используемые в других подобных системах, но и оригинальные (например, алгоритм планирования таймеров в системе и алгоритм распределения памяти). Проект применяется и в коммерческих проектах, и в исследовательских и обучающих, и даже энтузиастами различных стран. Естественно, что в проекте появилась необходимость в разработке инструмента для профилирования системы, подтверждаемая все большим количеством запросов от пользователей и разработчиков. Такой инструмент требуется для исследования тех или иных свойств системы, сбора статистики о выполняемых программах и алгоритмах, произведения сравнительных оценок различных алгоритмов и реализаций и т.д. Вполне разумным желанием является и то, что все измерения предпочтительно производить в режиме работы самой системы, а не на ее модели, поскольку трудно воспроизвести все возможные воздействия на систему, и зачастую лишь режим сбора статистики при работе программы может выявить ее реальные свойства.

Обзор методов профилирования систем

Профилирование (исследование) системы можно разделить на три типа, различающиеся по способу взаимодействия с системой:

- Теоретическое, при котором строится полная модель системы и возможных событий на нее.
- Экспериментальное - информация собирается на основе ряда внешних экспериментов

- Трассировки - в ход программы добавляется специальная информация на основе которой собираются данные о системе.

Самым очевидным средством узнать информацию о системе является теоретическое доказательство всех ее свойств. Подобный метод применяется для задач реального времени, но обычно это несложные системы, которые достаточно хорошо подаются описанию. Кроме того, описанию должны поддаваться и все допустимые внешние воздействия на систему. Таким образом, такой подход применяется как правило лишь для очень важных систем, а доказательство достаточно дорого и занимает существенное время.

При использовании метода на основе экспериментов система, сконфигурированная разными способами, рассматривается как черный ящик, и экспериментатор на основе внешних тестов проводит измерения и собирает информацию. Очевидным недостатком является большая трудоемкость, высокая доля субъективности и малая точность измерений. Также к этому методу можно отнести и мониторинг или опрос пользователей.

Третьим способом является метод трассировки. В этом случае в специально собранную версию программы добавляется дополнительный код, который может быть использован для сбора информации о ходе программы на этапе ее исполнения. Такой метод очень удобен на этапе разработки, отладки и профилировании программы, ведь в этом случае не нужно моделировать специальные события, как этого требует использование первых двух методов. К тому же этот метод позволяет получить очень детальную и объективную информацию о работе системы, что невозможно в первом случае.

Как и любой способ профилирования, метод трассировок имеет накладные расходы, основными являются место в памяти под дополнительную информацию, и время для исполнения служебного кода.

Служебную информацию можно вставлять в программу как на этапе компиляции и сборки программы, так и на этапе выполнения. При выставлении специальных меток на этапе выполнения может быть использован следующий метод: команда

по требуемому адресу заменяется командой вызова исключения или передачи управления на какую то процедуру; процессор, выполняя код данной команды, переходит по адресу функции обработки, и выполняет все необходимые действия по сбору информации. Данный метод часто используется для отладки программ, хотя может быть использовать и для трассировки. Обычно же для трассировки используется метод, когда дополнительная информация внедряется на том или ином этапе сборки программы. То есть она может быть внедрена на этапе компиляции или после сборки на специальном этапе, когда вызовы всех функций заменяются на вызов функции-обертки.

Постановка задачи

Разработать инструментарий для профилирования кода в ОСРВ Embox, позволяющий собирать информацию о ходе программы и ее свойствах. Необходимой информацией для сбора является количество посещений и время выполнения определяемых разработчиком участков кода. Необходимо также иметь средства управления разрабатываемым инструментарием в процессе работы системы. Также желательно предусмотреть интерфейс расширения возможностей инструментария и иметь возможность задавать условия трассировки.

Реализация

В своей работе я использовала метод трассировки на основе информации, внедренной на этапе компиляции программы. Это обусловлено прежде всего тем, что такой способ достаточно прост в реализации, позволяет пользователю хорошо настроить параметры собираемой информации о системе, вносит относительно небольшую и, главное, детерминированную погрешность в ход выполнения программы.

Рассмотрим данный механизм подробнее.

Подсчет количества вызовов

Первое и самое простое, что приходит в голову, когда речь заходит о трассировке и исследовании программы, - это поиск узких мест, критически важных, но долго выполняющийся участков кода. Очевидно, что если блок кода, пусть и не

оптимальный, занимает много времени, но при этом вызывается один раз при инициализации, то его нужно оптимизировать в последнюю очередь. Поэтому, первой, но очень важной информацией, которая может сильно помочь при оптимизации, является банальный подсчет количества вызовов того или иного участка кода.

Самый простой способ реализации такого механизма - это завести переменную и инкрементировать ее каждый раз при входе в подозрительный участок.
Ясно, что при росте числа исследуемых точек эффективно управлять и анализировать значения подобных переменных становится затруднительно. Чтобы упростить жизнь разработчику, нужно предоставить возможность как минимум не задумываться об именовании переменных и реализовать простой способ идентифицировать точку. Таким образом, моим первым шагом явилось добавление к каждой точке метаданных, содержащих положение этой точки в исходном коде программы. Прежде всего, это имя файла и номер строки, которые можно представить следующей структурой:

```
struct location {
     const char *file;
     const int line;
};
```

Важно отметить, что инициализировать подобную структуру можно статически, используя следующий макрос:

```
#define LOCATION_INIT { \
          .file = __FILE__, \
          .line = __LINE__, \
     }
```

Теперь добавим возможность сохранения не только имени файла и номера строки, но и названия функции, в которой была определена данная метка. В итоге получаем следующую структуру:

```
struct location_func {
     struct location at;
     const char *func;
};
```

Аналогично первой, эта структура также инициализируется статически:

```
#define LOCATION_FUNC_INIT { \
          .at = LOCATION_INIT, \
          .func = __func__,    \
     }
```

Возможность статической инициализации данных структур важна, поскольку это уменьшает количество операций, которые необходимо совершить во время исполнения, тем самым производя меньшее влияние на исследуемое приложение.

Кроме положения в коде, с точкой также можно ассоциировать произвольное имя, представляемое строковым литералом и определяемое разработчиком при вставке метки в программу.

Вернемся к основной функциональности, которую предоставляют точки трассировки - подсчету количества посещений потоком исполнения данной метки. Для этого достаточно добавить в структуру, представляющую точку, беззнаковый счетчик. И наконец, точка может быть активной и неактивной, что позволяет включать или отключать подсчет посещений во время работы программы. Таким образом, вся структура определяется следующим образом:

```
struct trace_point {
     struct location_func location;
     const char *name;
     bool active;
     unsigned int count;
};
```

И наконец, остается лишь инициализировать данную структуру в точке кода, куда разработчик добавил метку трассировки:

```
trace_point("name");
```

Данное выражение использует макрос `trace_point()`, определяемый следующим образом:

```
#define trace_point(_name) \
     do {                                                \
          static struct trace_point __trace_point = { \
               .location = LOCATION_FUNC_INIT,          \
               .name = "" _name,                        \
          };                                            \
          if (__trace_point.active)                     \
               __trace_point.count++;                   \
     while(0)
```

Поля структуры инициализируются статически (поле `count` инициализируется значением по умолчанию - нулем). Во время выполнения этого участка кода счетчик инкрементируется.

Следующей задачей, которую необходимо решить, - это доступ к этой структуре, ведь область видимости переменной `__trace_point` ограничена блоком `do … while`. Одно из решений - сохранить указатели на все точки, использованные в системе, в специальном массиве. Однако поскольку точки могут быть определены в любом месте исходного кода систем, и их число заранее неизвестно, построить такой массив стандартными средствами языка Си весьма затруднительно (если вообще возможно).

Тем не менее, в ОСРВ Embox предусмотрен механизм для определения таких массивов, называемый spread arrays. Этот механизм использует расширение компилятора GCC, позволяющее занести определенные данные в специальную секцию объектного файла, отличную от секции, назначаемую по умолчанию (“.bss”, “.data”, “.rodata” или “.text”). Spread arrays скрывает детали взаимодействия с компоновщиком, предоставляя лишь макрос для определения нового массива неизвестной длины и макрос для статического добавления в него нового

элемента, причем добавление может происходить в любом месте системы. После компоновки всех объектных файлов элементы оказываются рядом друг с другом, а имя, которым был назван массив при определении, указывает на первый элемент. Во время исполнения такой массив ничем не отличается от “обычных”.

Определим `NULL`-терминированный массив указателей на точки трассировки:

```
ARRAY_SPREAD_DEF_TERMINATED(
          typeof(struct trace_point *),
          __trace_points_array, NULL);
```

И модифицируем макрос `trace_point()` таким образом, чтобы указатель на каждую вновь определенную точку заносился в этот массив:

```
#define trace_point(_name) \
     do {                                                \
          static struct trace_point __trace_point = { \
               .location = LOCATION_FUNC_INIT,         \
               .name = "" _name,                       \
          };                                           \
          ARRAY_SPREAD_ADD(__trace_points_array,       \
                    &__trace_point);                   \
          __trace_point.count++;                       \
     while(0)
```

Теперь после компиляции и компоновки приложения в массиве `__trace_points_array` оказываются указатели на все точки трассировки. Этот массив используется для доступа ко всем точкам, в частности для поиска точки по номеру (индексу в массиве), имени или месту определения, включения/отключения счетчика определенной точки и снятия показаний счетчика.

Интерфейс для работы с точками трассировки описан в приложении.

Измерение времени исполнения участка кода

После выявления наиболее часто исполняемых точек кода, нужно выяснить сколько же времени реально исполняется тот или иной блок.

Для этого достаточно сохранить текущее время при входе в блок и при выходе вычислить разницу. Естественно, для более понятного представления

результатов разработчику хочется иметь не только одно значение измеренного интервала, но и максимальное, минимальное и среднее значение времени выполнения данного участка кода. Таким образом, структура немного усложняется, поскольку теперь в ней нужно хранить, как минимум, промежуточное время.

```
struct trace_block {
        unsigned int count;
        bool active;
        struct {
                struct timespec begin, end, last, min, max, avg;
        } time;
};
```

Структура `timespec` определена в стандарте POSIX и позволяет хранить время с точностью до наносекунд; такое представление времени используется прежде всего в пользовательских приложениях. В ядре существуют свои структуры представления времени, и интерфейс для его получения.

В отчете я опишу только работу с трассировками пользовательских приложений, поскольку ядерные трассировки по сути отличаются только функциями работы с подсистемой времени.

Реализация соответствующих макросов для объявления блоков трассировки во многом похожа на реализацию описанных ранее точек. Так, например, аналогичным образом используется механизм spread arrays.

Функции входа и выхода достаточно примитивны (для простоты обработка ошибок опущена, и приведена только функция входа):

```
void trace_block_enter(struct trace_block *tb) {
        if (tb->active)
                clock_gettime(CLOCK_UPTIME, &tb->time.begin);
}
```

Структура `timespec` содержит два поля - количество секунд и наносекунд,

поэтому разница между двумя значениями времени вычисляется достаточно очевидно - вычитаются значения наносекунд, и в случае переполнения производится перенос. Аналогичным образом производятся сравнения для определения минимального и максимального времени. Вычисление среднего значение производится инкрементально, по следующей формуле:

$$T_{avg} \mathrel{+}= (T - T_{avg}) / (count + 1);$$

Интерфейс для работы с блоками трассировки также описан в приложении.

Управление метками

Еще одним очень удобным механизмом, используемым для трассировки является задание условий, по которым возникает то или иное событие, например, когда число вхождений превышает какое-либо значение, или же время выполнения блока больше заданного.

Разумеется, самым простым и в тоже время часто используемым условием является флаг активности точки или блока. В рамках данной курсовой работы я ограничилась лишь реализацией этого условия, в перспективе возможно добавление более сложных условных трассировок.

Пользовательский интерфейс

Для управления разработанной системой сбора статистики, были добавлены две команды пользователя, которые могут быть вызваны из командной строки оболочки Embox. Это команды:

- trace_points, работающая с обычными метками для подсчета вхождений
- trace_blocks, работающая с трассировками блоков для измерения времени

Обе команды поддерживают следующие опции командной строки:

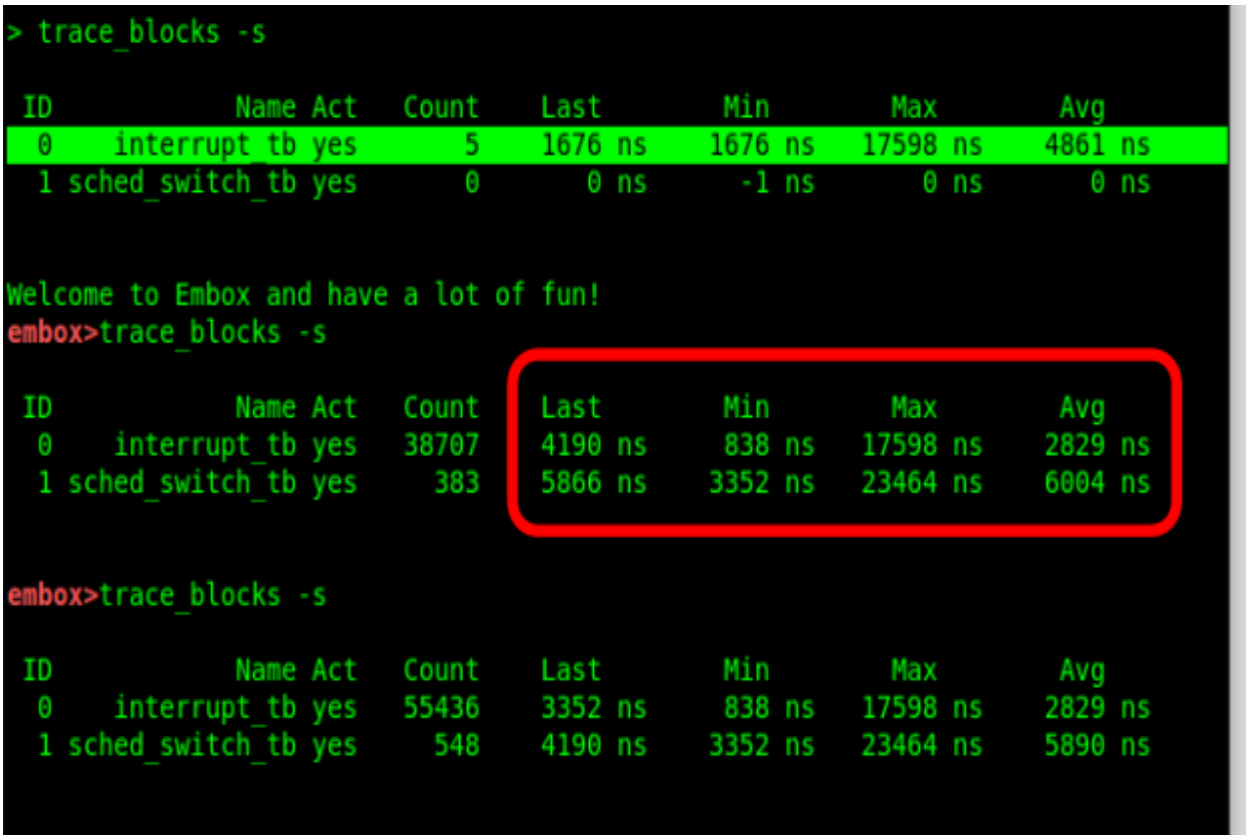
- `-s` - вывод информации по всем активным и неактивным точкам или блокам. Такая информация содержит номер, по которому можно однозначно идентифицировать точку/блок, имя, состояние флага

активности и количество посещений. Для блоков кроме этого выводятся временные показатели: длительность последнего посещения, а также максимального, минимального и среднего измерений.

- `-i ID` - вывод подробной информации о точке/блоке с номером ID.
- `-a ID` - включение подсчета и измерений;
- `-d ID` - отключение.

Результаты работы

На данном изображении представлен вывод пользовательской команды “trace_blocks”, запущенной на раннем этапе загрузки системы с помощью загрузочного скрипта, а также позднее вручную из оболочки системы. Красным выделены произведенные измерения: последнее, минимальное, максимальное и среднее.



Заключение

В результате данной работы были реализованы два механизма трассировки, в основе которых лежит внесение профилирующего кода в исследуемую систему на этапе компиляции приложения. Первый механизм позволяет выявить наиболее часто вызываемые участки кода, второй механизм позволяет оценить время, затрачиваемое процессором на выполнение данного участка программы.

Также были разработаны соответствующие команды пользователя, отвечающие за управления тем или иным механизмом во время исполнения программы. Данные команды позволяют не только выводить статистику по собранной информации, но и управлять трассировкой в режиме работы системы.

Исходный текст результатов моей работы можно получить по адресу https://embox.googlecode.com/svn/trunk/embox/

Мой ник для данного репозитория Kramar.Alina

Приложение

Интерфейс для работы с трассировками

Чтобы систему можно было подвергнуть трассировкам, был разработан интерфейс, который охватывает все возможные функции работы с метками. Этот интерфейс описывается в заголовочном файле profiler/tracing/trace.h.

Работа с trace point:

- `trace_point(char *)` - выставление анонимной метки в определенное место в программе. Разработчик не задает специальной переменной, для дальнейшего обращения с ней. Местоположением этой переменной является место, где был вызван этот метод.
- `TRACE_POINT_DEF(name)` - если же, в отличие от варианта выше, вам все-таки требуется переменная типа trace_point, то для ее объявление требуется использовать этот макрос. Т.о. локацией этой переменной является место объявления. Эта структура не будет собирать информацию,

пока не будет вызван метод `trace_point_set(&name)`.

- `trace_point_set(struct trace_point *)` - метод, необходимый для установления метки в конкретное место в системе. Используется только для именованных меток.
- t`race_point_get_value(struct trace_point *)` - метод, который по указателю на метку возвращает значения со счетчика этой метки.
- `trace_point_get_name(struct trace_point *)` - метод, возвращающий имя метки, которое задает пользователь.
- `trace_point_get_location_func(struct trace_point *)` - метод, возвращающий имя функции, в которой объявлена или выставлена метка в виде строки.

Интерфейс для работы с блоками трассировки

Работа с trace block:

- `TRACE_BLOCK_DEF(name)` - объявление блока.
- вхождение в блок (`trace_block_enter(struct trace_block *)`)
- покидание блока (`trace_block_leave(struct trace_block *)`)
- взятие разницы значений посещений начала и конца (`trace_block_diff(struct trace_block *)`)

Все эти функции могут быть использованы для прямой работы с трассировками.

Используемая литература

1. Tracing // Wikipedia, the free encyclopedia. URL. http://en.wikipedia.org/wiki/Tracing_(software) (Дата обращения: 01.11.11)
2. Mathieu Desnoyers. Using the Linux Kernel Tracepoints // URL http://www.mjmwired.net/kernel/Documentation/trace/tracepoints.txt (Дата обращения: 15.12.11)
3. The Linux Kernel Tracepoint API // URL http://kernel.org/doc/htmldocs/tracepoint.html (Дата обращения: 17.01.12)
4. Mike Holenderski, Martijn M.H.P. van den Heuvel, Reinder J. Bril and Johan J. Lukkien. Grasp: Tracing, Visualizing and Measuring the Behavior of Real-Time Systems
5. M. Desnoyers, M.R. Dagenais. The LTTng tracer: A low impact performance and behavior monitor for GNU/Linux // July 19th–22nd, 2006 Ottawa, Ontario. Canada