Немного об отладке. Часть 2

[**https://habr.com/ru/articles/867482/**](https://habr.com/ru/articles/867482/)

**Средний**

**116 мин**

**4K**

[Отладка\*](https://habr.com/ru/hubs/debug/)[Системное программирование\*](https://habr.com/ru/hubs/system_programming/)[Программирование\*](https://habr.com/ru/hubs/programming/)[Linux\*](https://habr.com/ru/hubs/linux_dev/)

Это продолжение моей [прошлой статьи](https://habr.com/ru/articles/867478/). В этой части мы коснемся сред разработки, а потом будем постепенно деградировать спускаться ниже по абстракциям. Приятного чтения.

Содержание:

* [Среды разработки](https://habr.com/ru/articles/867482/#ide)
  + [VS Code](https://habr.com/ru/articles/867482/#ide-vscode)
  + [GDB/MI](https://habr.com/ru/articles/867482/#ide-gdb-mi)
  + [MIEngine](https://habr.com/ru/articles/867482/#ide-miengine)
  + [Code::Blocks](https://habr.com/ru/articles/867482/#ide-code-blocks)
  + [Eclipse](https://habr.com/ru/articles/867482/#ide-eclipse)
  + [Spyder](https://habr.com/ru/articles/867482/#ide-spyder)
* [Отладка на ОС](https://habr.com/ru/articles/867482/#os)
  + [Windows](https://habr.com/ru/articles/867482/#os-windows)
  + [KolibriOS](https://habr.com/ru/articles/867482/#os-kolibrios)
* [Архитектуры ЦП](https://habr.com/ru/articles/867482/#arch)
  + [x86](https://habr.com/ru/articles/867482/#arch-x86)
  + [AMD64](https://habr.com/ru/articles/867482/#arch-amd64)
  + [ARM](https://habr.com/ru/articles/867482/#arch-arm)
  + [Loongson](https://habr.com/ru/articles/867482/#arch-loongson)

Среды разработки

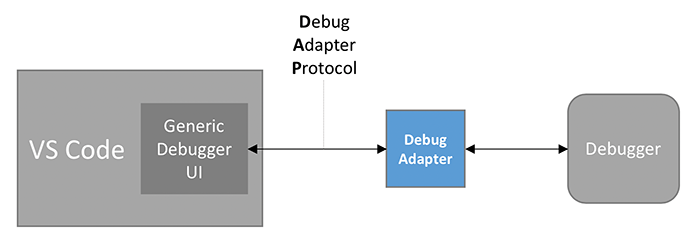
Теперь мы знаем о том как взаимодействуют отладчики с программой, но сегодня большая часть действий выполняется через IDE. Но как это происходит? Не может же IDE fork/exec'аться, а затем через stdin/stdout посылать команды *интерактивному* отладчику. Давайте разберемся как же устроен этот последний слой: IDE <-> отладчик.

VS Code

Начнем с Visual Studio Code. Чтобы понять, как реализована отладка в нем достаточно посмотреть на [страницу документации](https://code.visualstudio.com/api/extension-guides/debugger-extension) связанной с разработкой расширения отладки.

VS Code это браузер и в качестве рантайма использует NodeJS. Как отлаживать последний мы уже рассмотрели, но ведь в нем мы можем отлаживать любую программу (и даже не программу). Почему так?

Дело в том как устроен сам процесс отладки. Расширение, которое именуется отладчиком, на самом деле является прослойкой между IDE и настоящим отладчиком. А главная его задача - это обеспечение взаимодействия между IDE и отладчиком по понятному (в первую очередь IDE) протоколу. И протокол этот называется DAP - [Debug Adapter Protocol](https://microsoft.github.io/debug-adapter-protocol/overview" \t "_blank).



Архитектура отладчиков VS Code

DAP - это протокол, созданный Microsoft. Как понятно из названия, главная его задача - абстрагироваться от деталей языка и предоставить общий интерфейс взаимодействия с отладчиком (шаги, исследование переменных, точки останова и т.д.). Этот протокол не привязан к конкретному IDE, но описание самого протокола дано в виде TypeScript объектов: interface, типизация, комментарии. Предполагаю, что сделано это из-за того, что протокол в первую очередь был создан для VS Code, но сейчас им пользуются и другие IDE, например, [NeoVIM](https://github.com/mfussenegger/nvim-dap" \t "_blank). [На сайте](https://microsoft.github.io/debug-adapter-protocol/implementors/adapters/) можно найти более полный список известных адаптеров для различных IDE.

Для взаимодействия между клиентом и адаптером используется простой текстовый протокол. Он похож на HTTP: сообщение состоит из заголовка и содержимого, которые разделяются CRLF, и все это в ASCII. Заголовок состоит из полей (таких же, как в HTTP), но сейчас поддерживается только 1 - Content-Length. Содержимое же - это JSON объект сообщения.

Сообщений может быть 3 типа:

1. Request - запрос, клиент -> адаптер, синхронный
2. Response - ответ, адаптер -> клиент, синхронный
3. Event - событие, адаптер -> клиент, асинхронный

Для различия этих объектов используется поле type (принимает значения request, response, event). Также и у каждого типа есть свои обязательные поля. Например, для request - это command, название команды, которую мы запрашиваем. Ей еще передаются аргументы, но как понятно, они зависят от самой команды. А для response - флаг success, успешно или нет выполнилась команда.

Мы уже видели, что многие события в отладчике происходят асинхронно. Аналогично это относится и к сообщениям в DAP. Чтобы решить некоторые проблемы связанные с асинхронностью, каждое сообщение снабжается полем seq, числом монотонно инкрементирующимся с каждым сообщением. Благодаря этому такие сообщение можно идентифицировать и выстраивать их историю.

Но голая теория мало что даст. Получилось так, что я уже работал с этим протоколом задолго до написания этой секции. Мне необходимо было разработать расширение VS Code для взаимодействия с отладчиком - [PostgreSQL Hacker Helper](https://github.com/ashenBlade/postgres-dev-helper" \t "_blank). Если вкратце, то в PostgreSQL (ядре) существует своя система типов, основанная на C-style наследовании: первое поле хранит тип (тэг), а дальше структура приводится к типу соответствующему этому тэгу. В коде на макросах это реализуется довольно просто, так как название тэга (enum'а) составляется как T\_ + название структуры. Но в рантайме есть только числа, то есть мы не можем в окне выражений написать что-то по типу (substr(nodeTag(node)) \*)node, это приходилось делать отдельными шагами. Все что я хотел сделать - автоматизировать эту рутину. И спойлер - у меня получилось.

Если кого заинтересовала тема типов в PostgreSQL, то ранее я писал статьи по исходному коду PostgreSQL (вот [в этой](https://habr.com/ru/articles/723668/) краткое их описание). Также у меня есть [доклад](https://rutube.ru/video/58b85572a9a6154cc5a8a2493e6a2e16/) на тему разработки PostgreSQL - в ней я рассказываю о разработке исходного кода и отладке (конечно с упоминанием расширения).

Если рассмотреть ядро логики расширения, то оно состоит из 4 простых шагов:

1. Получаем переменную - (Node \*)variable
2. Получаем ее тэг - ((Node \*)variable)->type = T\_SampleStruct
3. Убираем префикс T\_ - T\_SampleStruct = struct SampleStruct
4. Приводим переменную к нужному типу - ((SampleStruct \*)variable)

Это простая логика, но дьявол кроется в деталях. Во-первых, как нам получить первые переменные (не поля уже известных переменных)? Это решается за счет 3 дополнительных вызовов: threads, stackTrace и scopes - получение информации о всех потоках, фреймы нужного потока и информация о scope'ах нужного фрейма соответственно. Последнее, scope - это какой-то именованный набор переменных. В моем случае, их было 2: locals (локальные переменные) и registers (регистры).

Во-вторых, как нам обращаться к ресурсам (указывать на конкретные переменные, потоки, фреймы и т.д.)? Для этого имеются специальные идентификаторы. У них всех имеется суффикс reference. Например, variablesReference - это ID набора переменных (см. ранее). И этот variablesReference краеугольный камень моей логики, так как благодаря ему я и получаю информацию о переменных и полях: он передается в scopes (ответе), а также каждая переменная ее имеет (в этом случае, полученные переменные считаются либо полями структуры, либо элементами массива).

Теперь переведем это все на язык DAP:

1. threads - находим нужный нам поток
2. stackTrace - находим нужный нам фрейм
3. scopes - получаем variablesReference области переменных
4. getVariables - получаем все переменные по variablesReference
5. evaluate - вычисляю значение тэга узла с помощью выражения ((Node \*)variable)->type и нахожу настоящий тип структуры
6. evaluate - получаю variablesReference уже приведенного типа ((RealType \*)variable)

Повторяем 4 шаг уже с новым variablesReference пока не будем получать пустой ответ. И да, как вы могли догадаться evaluate - это вызов, с помощью которого можно динамически вычислять выражения, и он также возвращает variablesReference.

И здесь стоит сказать о третьей детали - DAP описывает протокол, но он не формат возвращаемых данных. В моем случае, это означает что я жестко привязан к конкретному адаптеру так как моя логика завязана на парсинг его ответов. К примеру, чтобы понять, что getVariables вернул простую структуру я проверяю не только тип (поле type), но и значение (поле value), так как для переменных значение равно {...}, а вот для поля (с типом структуры, не указатель) он возвращает пустую строку. Это отладчик [C/C++](https://marketplace.visualstudio.com/items?itemName=ms-vscode.cpptools), но когда я попробовал использовать расширение [CodeLLDB](https://marketplace.visualstudio.com/items?itemName=vadimcn.vscode-lldb" \t "_blank), то к моему удивлению значения практически всех полей были отформатированы по другому и не были совместимы с моей логикой. В частности, структуры в поле значения форматировались полностью со всеми полями.

GDB/MI

Ранее я уже сказал, что использовал расширение C/C++. Это всего лишь прослойка (вообще это прослойка прослойки, но об этом позже) между настоящим отладчиком и VS Code. В качестве настоящего отладчика используется gdb. Если редактировали файл launch.json, то могли видеть такое поле как miMode. mi в данном случае - это сокращение Machine Interface.

[gdb mi](https://www.sourceware.org/gdb/current/onlinedocs/gdb.html/GDB_002fMI.html) (gdb Machine Interface) - текстовый интерфейс для взаимодействия с gdb. Его можно активировать запустив gdb с флагом --interpreter и предназначен для окружений, где отладчик является частью большой системы, такой как IDE. На данный момент, существует 4 версии этого протокола.

Сам протокол сильно похож на стандартный протокол/алгоритм отладчика. Имеем команды, ответы и уведомления. Ответ присылаются при отправке команды, а уведомления приходят асинхронно. Из интересного - совместимость с командами интерактивного режима. Она есть, но оговаривается, что поведение может быть непредсказуемым. Чтобы использовать их без проблем есть отдельная команда - -interperter-exec.

На странице документации есть примеры взаимодействия с gdb mi. Например, таким образом выставляется точка останова:

-break-insert main

^done,bkpt={number="1",type="breakpoint",disp="keep",enabled="y",addr="0x000000000000114c",func="main",file="main.c",fullname="/path/to/file/main.c",line="9",thread-groups=["i1"],times="0",original-location="main"}

Сразу можно заметить, что у названий команд в начале идет тире, а затем список их аргументов. Если смотрели внутрь launch.json, то теперь знаете, что за команды в нем имеются при создании (-enable-pretty-printing например).

Но давайте посмотрим на другую строку. Как можно понять, это ответ. На [этой странице](https://www.sourceware.org/gdb/current/onlinedocs/gdb.html/GDB_002fMI-Output-Syntax.html#GDB_002fMI-Output-Syntax) приводится синтаксис вывода (output), но мне кажется приводить его здесь лишнее, поэтому просто опишу в общем и приведу примеры.

Вывод может быть 2 видов: результат (Result) и асинхронные (Async) события. На примере выше мы видели результат. Он начинается с циркумфлекса (^) и типа ответа - ^done. На этом же примере, тип - завершение синхронной операции. После идет результат операции. Опять же, здесь это [информация о точке останова](https://www.sourceware.org/gdb/current/onlinedocs/gdb.html/GDB_002fMI-Breakpoint-Information.html#GDB_002fMI-Breakpoint-Information). Кроме done есть еще 4 типа ответа, но они не такие частые: запуск/остановка отладчика и ошибка.

И последнее - асинхронные события. Они разделяются на 3 категории: уведомления выполнения (изменение в состоянии), статусные и общие уведомления. Как и у результата, в начале идет специальный символ идентифицирующих их тип. Например, мы продолжили выполнение и наткнулись на точку останова:

^running

\*running,thread-id="all"

(gdb)

=breakpoint-modified,bkpt={number="2",type="breakpoint",disp="keep",enabled="y",addr="0x0000555555555153",func="main",file="main.c",fullname="/path/to/file/main.c",line="10",thread-groups=["i1"],times="1",original-location="main.c:10"}

~"\n"

~"Breakpoint 2, main () at main.c:10\n"

~"10\t printf(\"%d\\n\", value);\n"

\*stopped,reason="breakpoint-hit",disp="keep",bkptno="2",frame={addr="0x0000555555555153",func="main",args=[],file="main.c",fullname="/path/to/file/main.c",line="10",arch="i386:x86-64"},thread-id="1",stopped-threads="all",core="3"

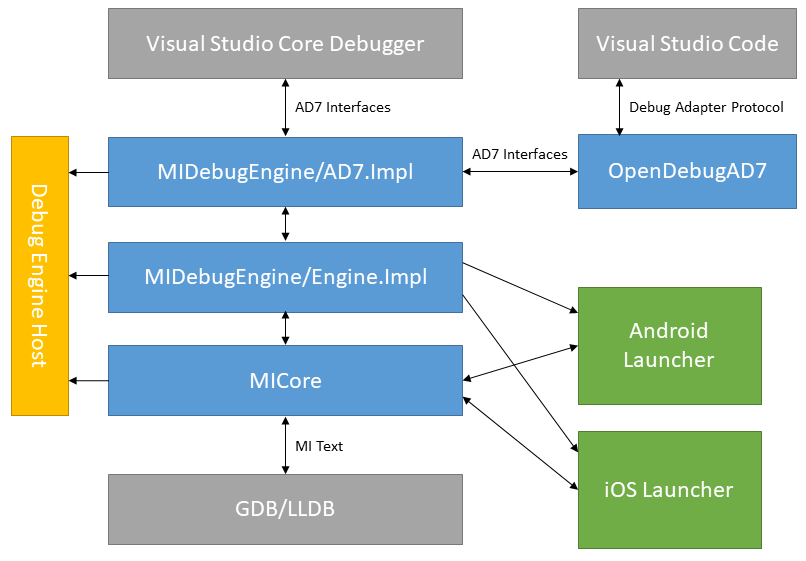
Здесь можно заметить все 2 типа этих уведомлений:

* \* - изменение состояния выполняемого процесса
* = - статусное уведомление

Также есть ~. Некоторые команды могут в ответе отправлять какой-либо текст. Если так, то для его отправки используется такой тип уведомления.

MIEngine

В VS Code я использую расширение C/C++ для разработки и мое расширение использует его как DAP адаптер. Но если посмотреть вглубь, то окажется, что C/C++ расширение не само реализует DAP. Она использует [MIEngine](https://github.com/Microsoft/MIEngine" \t "_blank) - движок отладки Visual Studio, который умеет "говорить" на языке MI (того самого как у gdb). Но это еще не все: последняя деталь - OpenDebugAD7, адаптер для DAP (его код в том же проекте, что и MIEngine). На этой схеме показано взаимодействие между этими частями:



Архитектура отладки C/C++ в VS Code

AD7 - Active Debugging 7, фреймворк для создания отладчиков в Visual Studio

Для большей наглядности будем рассматривать путь запроса от пользователя до MIEngine и обратно, то есть включая действия OpenDebugAD7 и отправляемые в gdb команды. И начнем с запуска отладки.

За запуск отладки отвечает либо launch, либо attach. Я рассмотрю первый вариант.

Отладка производится сессиями. Такую сессию отладки представляет класс AD7DebugSession. Он содержит в себе все методы, соответствующие запросам из DAP. Таким образом, за обработку launch запроса отвечает метод HandleLaunchRequestAsync. Но по факту, все что он делает - перенаправляет запросы другому объекту, движку MI. Это интерфейс IDebugEngineLaunch2, который реализует AD7Engine.

Запуск процесса для отладки

Как можем видеть по этой цепочке вызовов - мы запускаем отладчик, инициализируем его и запускаем программу. Но где же сами MI команды? Да, я их пропустил, но намеренно, чтобы сконцентрироваться на логике, а сейчас посмотрим на то, как их отправляют. Первый пример - это команды инициализации, те что были в методе Initialize. Они создаются в методе GetInitializeCommands.

GetInitializeCommands

Как можете заметить, во-первых, оперируем простыми, иногда с интерполированными параметрами, строками (это ведь текстовый протокол), и, во-вторых, больше количество кода используется для кроссплатформенности: речь не только об ОС (Windows, Cygwin, Linux ...), но и отладчике (gdb, lldb ...).

Теперь, посмотрим, как эти самые команды отправляются в сам отладчик. Для отправки данных на отладчик используется абстракция ITransport и зачем она нужна ставится понятно, если понять, что отладчик может находиться не на нашем ПК, а где-нибудь на сервере и взаимодействовать с ним нужно через сеть. Поэтому и реализаций этого транспорта несколько: pipe, tcp, shell, stream, local. Если мы хотим запустить локальный gdb, то будем использовать LocalTransport - он настраивает окружение, находим файл конфигурации (.gdbinit) и запускает сам gdb, а далее, для взаимодействия с ним используются абстракции C# - StreamReader и StreamWriter (можно сказать, простые дескрипторы stdin/stdout).

Вот мы выполнили свою работу - теперь нужно ответить клиенту DAP. Для реализации DAP используется собственный фреймворк и в самом его ядре находится класс DebugProtocol. Он содержит основную логику: последний seq, заголовок ответа, поток ввода/вывода (для взаимодействия), сериализация и т.д. Клиенту просто необходимо реализовать бизнес-логику, то есть взаимодействие с отладчиком. За ответ отвечает метод SendMessageCore - в нем находится практически вся необходимая инфраструктурная логика: увеличение seq, сериализация, выставление заголовка:

SendMessageCore

Подобная обобщенная машинерия используется и для вызова соответствующего обработчика запроса на основании переданной команды. Но располагается она в классе DebugAdapterBase:

Вызов соответствующего обработчика

Таким образом, можно сказать, что инфраструктурная часть реализуется с помощью паттерна Шаблонный метод.

Точки останова

Теперь перейдем к точкам останова. В DAP они разделяются на 3 вида: function (на входе в функцию), data (точки останова на данные, watchpoint) и обычные. Я рассмотрю самые обычные точки останова. За них отвечает метод HandleSetBreakpointsRequestAsync и запрос setBreakpoints в DAP.

Здесь адаптер выступает в роли кэша - он запоминает какие точки останова уже были выставлены и при необходимости добавляет или удаляет.

HandleSetBreakpointsRequestAsync

Ответственность за создание и управление (выставление/удаление) точкой останова лежит на разных сущностях. Создание - это движок (MIEngine), а управление - на самой точке останова.

За создание отвечает метод CreatePendingBreakpoint. По факту, что он делает - создает новый объект, выставляет необходимые поля и добавляет ее в свой список. Поэтому опустим реализацию и сконцентрируемся на управлении.

Управление точкой останова проблема самой точки останова. Перед тем как выполнить целевое действие она инициализирует и проверяет свое состояние: условие срабатывания, расположение и т.д. Когда приходит время выставить или удалить точку останова (отправить команду), то это делегируется классу MICommandFactory.

Из названия становится понятно, что это вспомогательный класс, который используется работы с командами MI. Так и есть, этот класс предоставляет программный интерфейс для команд MI протокола, а сам внутри собирает строки этих команд. Например, для выставления точки останова используется метод BreakInsert:

BreakInsert

Удаление происходит аналогично и метод для MI - BreakDelete:

BreakDelete

Шаги

Переходим к шагам. В DAP за них отвечают команды next, stepIn и stepOut - step over, step into и step out соответственно.

Обработка этих запросов происходит похожим образом как видели ранее - все они обрабатываются единственным методом, в который просто передается тип шага, - StepInternal. И вся его логика состоит в том, чтобы вызвать метод Step у объекта движка. Который в свою очередь просто вызовет нужный метод у MICommandfFactory. Никакой другой логики кроме валидации и маппинга особо нет (в сравнении с другим кодом).

Step XXX

Code::Blocks

[Code::Blocks](https://www.codeblocks.org/) - это кроссплатформенная IDE для C/C++/D/Fortran. Для визуальной части используется wxWidgets (что важно).

Отладка поддерживается IDE, но сама же функциональность реализована с помощью плагинов (отладчики реализуют интерфейс cbDebuggerPlugin). В частности, плагин debuggergdb - добавляет поддержку отладки с помощью GDB и CDB (Microsoft Console Debugger, для Windows).

Код визуальной части не тема статьи, но обойти не могу, поэтому вкратце. Имеется класс DebuggerManager и он тот самый менеджер - ничего не делает, а только дает доступ к другим. И под другими, в основном говорю об окнах. Таким окном, например, является cbBreakpointsDlg - окно точек останова. И вот эти самые окна содержат методы-обработчики кнопок, которые и запускают функциональность. Для того же окна точек останова есть метод AddBreakpoints, который как можно догадаться и вызывает логику добавления точки останова.

Для начала поговорим о главном - кто же отвечает за логику отладки. Уже заспойлерил - сами отладчики, а IDE просто с ними взаимодействует. Для взаимодействия с отладчиком используются методы класса cbDebuggerPlugin. В debuggergdb находится единственная реализация этого интерфейса - DebuggerGDB. И вот тут еще одна интересная деталь - это класс также является простым фасадом. Почему? Как минимум потому что поддерживается не только gdb, но и cdb, а интерфейс у них разный. Поэтому логика адресуется интерфейсу DebuggerDriver. И этих драйверов 2: GDB\_driver и CDB\_driver.

Так как же происходит взаимодействие с ними? Все просто - запускаем через консоль, а затем взаимодействуем с помощью stdin/stdout/stderr. Причем для запуска используется функция wxExecute (из wxWidgets, как можно догадаться). А строку запуска создает уже сам драйвер, так как он сам знает где лежит его исполняемый файл.

Запуск отладчика

Но самая интересная часть - это обработка ответа. Ранее мы использовали DAP или GDB/MI. Но здесь мы запускаем интерактивный клиент, он ориентирован на человека. Писать кастомные парсеры? Почти - использовать регулярные выражения. Да, если заглянуть в код gdb\_driver.cpp, то первое, что мы увидим, это регулярные выражения для парсинга ответов, например, результат после команды выставления точки останова. И эта функциональность (regex) также поставляется wxWidgets:

Регулярки парсинга ответа

Но на этом инфраструктура отладчика не закончена. Мы рассмотрели запуск, но что же в рантайме? Как выполняются запросы? Проблема здесь заключается в IDE, а точнее оконном приложении. Они должны быть однопоточными, но многие команды отладчика могут выполняться долго и просто заблокироваться на его ожидании означает смерть зависание всего приложения.

Эту проблему решили просто - очередь команд/паттерн команда. Все действия, которые можно отправить отладчику представляются в виде класса DebuggerCmd (базовый класс). У него 2 главных метода: Action - выполнение действия и ParseOutput - парсинг ответа. А для их выполнения используется абстракция очереди команд. При добавлении новой команды она выполняется, но вот ответ парсится тогда, когда будет доступен. Ответ передается с помощью обработчика события OnGDBOutput (или OnGDBError).

Зная это, мы можем понять как происходит взаимодействие с отладчиком:

1. Форматируем строку команды как мы бы это сделали в gdb
2. Отправляем ее по stdin процессу
3. Ждем ответа от gdb
4. Парсим ответ

На этом можно было бы и закончить, так как это и есть ответ на основной вопрос - а по какому протоколу ведется взаимодействие. Но, чтобы закрыть гештальты, предоставлю команды добавления точки останова:

GdbCmd\_AddBreakpoint

Не надо забывать и о CDB, у него тоже есть аналогичная команда:

CdbCmd\_AddBreakpoint

Eclipse

[Eclipse](https://eclipseide.org/) - это кроссплатформенная IDE, разрабатываемая Eclipse Foundation. Изначально, разработана на Java для Java, но сегодня поддерживает много других ЯП.

Это очень модульная IDE и многая функциональность поставляется с помощью пакетов. Отладка не исключение. В пакете org.eclipse.debug.core содержатся интерфейсы, которые необходимо реализовать отладчику (например, IBreakpoint - это интерфейс точки останова), а также объекты бизнес-логики (например, BreakpointManager - класс, ответственный за управление точками останова).

Таким образом, пакет с отладчиком должен реализовать все необходимые интерфейсы. И тот самый org.eclipse.jdt.debug это делает ([репозиторий](https://github.com/eclipse-jdt/eclipse.jdt.debug)). Например, IJavaBreakpoint - это еще один интерфейс точки останова, но уже для Java, затем идет JavaBreakpoint - это абстрактный класс Java точки останова реализующий этот интерфейс, а вот затем все конкретные точки останова его реализуют: JavaLineBreakpoint, JavaClasPrepareBreakpoint...

Я буду рассказывать об отладке Java (и его плагине), хотя Eclipse поддерживает и другие языки.

В инфраструктуре eclipse, чтобы создать плагин, необходимо отнаследоваться от класса Plugin. Это делает класс JDIDebugPlugin и из названия становится понятно, что для отладки используется JDI, а также JDWP.

Реализация отладчика находится в 2 пакетах: самого плагина и com.sun.jdi. По факту, расширение - это удобная обертка над com.sun.jdi. Последний берет на себя инфраструктурные вопросы: запуск, подключение к JVM, отправка команд и т.д.

Взаимодействие с JVM реализуется с помощью класса com.sun.jdi.SocketListen. То есть для подключения к JVM используется сокет. Далее, мы получаем инстанс класса com.sun.jdi.VirtualMachine - прокси для взаимодействия с JVM.

Если говорить о самом расширении, то корнем всей иерархии является класс JDIDebugTarget, он представляет отлаживаемую JVM и все операции проводятся над ней. Когда мы хотим выполнить какую-либо операцию, то вначале выполняется наша логика проверка (расширение), а затем вызывается код com.sun.jdi. Причем каждый такой запрос представляется в виде объекта запроса (Request), а затем он отправляется по протоколу.

Но давайте посмотрим как это все реализуется в коде. Для начала рассмотрим как выставляются точки останова. Как уже сказал, классов точек останова несколько, но логика их выставления везде одинакова - создаем запрос (BreakpointRequest), а затем настраиваем его (выставляем нужные поля).

Стек вызовов примерно такой:

Выставление точки останова

Но здесь только код *создания* запроса. Где же отправка по JDWP? Она в самом классе запроса. Базовая реализация запроса - класс EventRequestImpl. Вот он то формирует и отправляет сетевые пакеты.

EventRequestImpl

Отправили запрос теперь нужно ждать ответа. Где его обработка? Вернемся назад, в тот момент, когда добавляли запрос в класс JDIDebugTarget. Мы добавили не только запрос, но и специальный класс Listener - это колбэк, который вызывается после выполнения запроса. Обработка же ответов возлагается на класс EventDispatcher - для него запускается отдельный поток, в котором обрабатываются ответы от JVM.

Обработка результата запроса

Таким образом, процесс отладки такой:

1. UI вызывает обработчик и он доходит до JDIDebugTarget
2. Находится объект, ответственный за логику (если это точка останова, то вызывается ее код)
3. Создается и регистрируется объект запроса
4. Запрос отправляется по протоколу JDWP
5. Регистрируется обработчик результата команды
6. EventDispatcher получает ответ от JVM
7. Вызывается обработчик запроса

Spyder

[Spyder](https://github.com/spyder-ide/spyder) - это научно-ориентированная IDE для Python. Соответственно, у нее есть интеграция с популярными Data Science библиотеками: NumPy, SciPy, Matplotlib, Pandas. Если когда-нибудь запускали PyCharm с Matplotlib, то справа высвечивались отформатированные таблицы и графики - здесь то же самое (бросив беглый взгляд, я разницы в этих таблицах/графиках не заметил).

Для запуска кода используется IPython (интерактивная оболочка для выполнения кода). Его логика выполнения отличается от ванильного Python, но не сильно.

Поддержка отладки также реализуется за счет плагинов (но IDE знает об отладке и для отладчика выделен отдельный тип плагина). Есть 2 плагина: ipythonconsole - интерактивная консоль IPython, и debugger - сама функциональность отладчика. Первый предоставляет интерфейс для запуска команд отладчика (с помощью pdb), а debugger - вызывает команды отладчика (step over, breakpoint и т.д.).

Поддерживается 4 команды: next (step over), step (step into), return (step out) и continue (продолжить выполнение). Но вот как они выполняются отличается от рассмотренных ранее вариантов. Если уже работали с IPython, то знаете, что она запускает интерактивную консоль и для управляющих команд используется префикс !. Так вот, плагин debugger просто отправляет эти команды в консоль как текст. Например, когда мы нажимаем кнопку step over, то !next отправляется в консоль и выполняется.

Выполнение команд

Но мы просто переложили ответственность за исполнение команд на другую сущность. Давайте рассмотрим то, как эти команды выполняет IPython. Для начала запуск.

Хоть за запуск отвечает ядро IPython, но при отладке мы можем добавить свои хуки - magic (деталь реализации IPython). Такие магические команды начинаются с %. Если IPython встречает такую команду, то вызывает ассоциированный с ним обработчик. Для отладки IDE регистрирует команду debugfile - магическую команду. При ее вызове управление передается Pdb - отладчику Python, о котором уже говорили. Но не конкретно ему - IPython и spyder создают свои подклассы, наследующиеся от Pdb (добавляют свою логику).

Таким образом, запуск кода под отладкой имеет следующий вид:

Запуск под отладкой

На самой вершине мы имеем класс SpyderCodeRunner. У него есть тот самый магический метод debugfile - отладка файла. Все уходит в общий метод \_exec\_code: парсит текст в байт-код и выполняет его (напомню, что в шаблоне).

В самом низу видим строку super(SpyderPdb, self).run(cmd, globals, locals). Кому же передает управление SpyderPdb? Bdb - непосредственно отладчику Python. Таким образом, мы начинаем отладку.

Но вопрос еще без ответа - как мы все-таки вызываем step XXX, continue и т.д.? Сейчас мы просто делегируем это консоли IPython. Ответ прост - инфраструктура отладчика.

Bdb - это базовый класс отладчика. Он содержит в себе, назовем это, кодовую часть. А вот Pdb - это уже *интерактивный* отладчик. Он умеет обрабатывать команды пользователя, введенные им интерактивно. Но это не значит, что набор этих команд ограничен. Посмотрим на то, как обрабатываются команды пользователя.

У класса Cmd (от которого наследуется Pdb, но это не особо важно) есть метод cmdloop - цикл обработки команд, которые читаются с помощью readline. Затем эта команда парсится, причем:

1. Команда, начинающаяся с ! интерпретируется оболочкой (например, !ls).
2. Имеется метод default, который должен возвращать обработчик по умолчанию для переданной команды. И главное - как же обработчики команд (next, step ...) находятся? Также, с помощью инфраструктуры, но уже самого Python: getattr(self, 'do\_' + cmd) - таким образом мы получаем нужный нам обработчик, например, do\_next - обработчик step over (getattr - это функция, возвращающая нам атрибут объекта по переданному названию, а cmd - это название самой команды).

Если выполнить некоторые команды, то можно заметить, что отправляющиеся отладчиком команды в IPython консоль имеют префикс !, то есть должны отправляться в оболочку, но этого не происходит. Почему? Из-за SpyderPdb - он переопределяет метод default таким образом, что этот знак убирается. В результате, это то же самое, что и отправить next самому Pdb.

Но хватит слов, давайте посмотрим на код:

Обработка команд

За точки останова отвечает класс BreakpointsManager. И обрабатываются они так же как и оговаривалось в секции про отладку Python - точки останова хранятся в поле самого Bdb. Мы пишем туда же:

Выставление точек останова

Отладка на ОС

Мы забрались слишком высоко - управляемые языки со своим рантаймом и IDE. Теперь пора вернуться опуститься и рассмотреть подробнее примеры каждого слоя. Начнем с операционных систем. Ранее мы рассмотрели только Linux - перейдем к Windows.

Windows

Отладка на Windows отличается как используемым API, так и инструментами (это ведь разные ОС).

Во-первых, API. На Linux мы использовали (швейцарский нож) ptrace. Но в Windows пошли путем небольших функций делающих простые вещи. Для подключения функциональности отладки необходимо подключить соответствующий заголовок <debugapi.h>. В нем заключается основная функциональность связанная с отладкой.

Цикл отладки

Если грубыми штрихами обрисовывать цикл отладки, то он похож на ранее увиденные:

1. Создаем отлаживаемый процесс: CreateProcess(NAME, DEBUG\_PROCESS).
2. Заходим в вечный цикл ожидания очередного события: WaitForDebugEvent(&event, TIMEOUT).
3. Определяем, что за событие произошло: switch (event.dwDebugEventCode).
4. Продолжаем выполнение и идем на шаг 2: ContinueDebugEvent(PID, TID, DBG\_CONTINUE).

Теперь рассмотрим каждый шаг. В начале мы создаем отлаживаемый процесс. Заметьте, что это выполняется функцией CreateProcess - в Linux мы отдельно создавали процесс и делали его отлаживаемым (PTRACE\_TRACEME). И причина в архитектуре.

На Linux все процессы создается дочерними и после fork имеют какой-либо контроль над выполнением. Нам это время нужно для ptrace(PTRACE\_TRACEME). Но в Windows все процессы равноправны и при запуске выполняется сразу готовый код (можно сказать сразу exec выполняем). Теперь становится понятно, зачем флаг DEBUG\_PROCESS добавлен - это можно назвать шорткатом для fork + ptrace(PTRACE\_TRACEME) + exec). Еще есть флаг DEBUG\_ONLY\_THIS\_PROCESS на случай, если запускаемый будет создавать новые процессы - указываем, если хотим отлаживать только непосредственно созданный нами.

Для подключения к уже запущенному процессу используется функция DebugActiveProcess

На 2 шаге мы заходим в вечный цикл ожидания следующего события от отлаживаемого процесса. Это также похоже на Linux, только там мы не явно ожидали события для отладчика, а просто события дочернего процесса (с помощью waitpid).

На этом шаге тоже есть разница с Linux. В Windows вызывать WaitForDebugEvent может только поток, который вызвал CreateProcess. Но в Linux аналогичное требование отсутствует.

3 шаг - определение события. По моему мнению, в Windows это устроено проще: когда WaitForDebugEvent завершается, то он заполняет структуру DEBUG\_EVENT. В ней довольно много информации, для получения которой не нужно сильно заморачиваться.

В начале мы также с помощью switch/case определяем, что перед нами за событие. Их может быть 9: создание/завершение процесса/потока, загрузка/выгрузка DLL, OutputDebugString (дочерний процесс явно хочет передать отладчику какую-то строку), RIP\_EVENT (системная ошибка) и EXCEPTION\_DEBUG\_EVENT (возникло исключение). Для каждого события есть своя структура, которая и заполняется. А хранится она в объединении (union).

Когда мы обработали событие, отлаживаемый процесс необходимо продолжить. Это делается с помощью ContinueDebugEvent. Кажется это также на Linux - ptrace(PTRACE\_CONT), но не тут то было. Заметьте последний аргумент этой функции - DBG\_CONTINUE. Этот последний аргумент имеет смысл только если событие было типа EXCEPTION\_DEBUG\_EVENT. Этот флаг принимает 2 значения: DBG\_CONTINUE и DBG\_EXCEPTION\_NOT\_HANDLED. Разница между ними проста - когда отладчик получает исключение от отлаживаемого, то это может быть как фатальная ошибка, либо как сигнал для отладчика (спойлер, точки останова). Поэтому, когда мы продолжаем выполнение этот флаг может сказать ОС что исключение обработано (DBG\_CONTINUE), иначе обработка уже произойдет на стороне отлаживаемого процесса (DBG\_EXCEPTION\_NOT\_HANDLED). Это чуть-чуть похоже на Linux с точки зрения точки останова - если отладчика нет, то SIGTRAP приведет к core-dump'у.

Есть еще 3-ий флаг - DBG\_REPLY\_LATER. На сколько я понял, если указать этот флаг, то при возобновлении работы указанного потока это событие вернется, чтобы обработать заново.

Собирая все вместе, цикл работы отладчика можно представить следующим образом:

Цикл работы отладчика

Но это мой код. Давайте посмотрим на цикл отладки в lldb:

DebuggerThread::DebugLoop

Основная работа отладчика заключается в обработке этих событий и выполнении соответствующих действий при остановке (выставление точек останова, трейсинг и т.д.). Начнем с базы - точки останова.

Точки останова

Логика их обработки довольно проста. Для начала - как их обнаруживаем. Я уже заспойлерил, что за точки останова отвечает событие EXCEPTION\_DEBUG\_EVENT. Но вообще, это событие, как можно догадаться из названия, отвечает за исключения. Они могут быть разными, но определены. Чтобы понять какое исключение перед нами, необходимо использовать поле ExceptionCode у структуры его события.

Я насчитал 23 кода ошибки. Например, EXCEPTION\_INT\_DIVIDE\_BY\_ZERO - ошибка деления на 0. Но в рамках отладки нас будут интересовать 2 исключения:

* EXCEPTION\_BREAKPOINT - точка останова
* EXCEPTION\_SINGLE\_STEP - шаг инструкции

SEH

Логика выставления/удаления точек останова такая же как и в Linux, то есть ОС предоставляет нам возможность копаться в адресном пространстве отлаживаемого процесса, но что мы туда запишем - нам на откуп, мы сами должны знать инструкцию точки останова.

Для работы с адресным пространством Windows предоставляет 2 функции: ReadProcessMemory и WriteProcessMemory - чтение и запись соответственно. Это простые функции, которые читают непрерывный участок памяти (не векторизованное чтение).

Тут тоже можно заметить разницу ptrace(PTRACE\_PEEKDATA) позволяет читать с гранулярностью в 1 *слово* (8/4 байт в зависимости от битности). ReadProcessMemory и WriteProcessMemory - с гранулярностью 1 *байт*. Но чтение из файла памяти или process\_vm\_readv/process\_vm\_writev, которые рассмотрели в секции gdb, - в Linux уже позволяют по байту читать/писать.

Мы уже знаем как выставляются точки останова: запоминаем, что было по адресу и записываем туда 0xCC:

Выставление точки останова

Единственное замечание: использование функции FlushInstructionCache. Как можно догадаться, она используется для того, чтобы сбросить сделанные нами изменения в инструкциях и CPU их подхватил.

Теперь посмотрим, как это реализуется в lldb. Он кроссплатформенный и его платформно-зависимые части реализованы в виде подклассов других абстрактных классов. И оно понятно: у разных платформ разные системные вызовы.

Логика выставления точки останова находится в методе NativeProcessProtocol::EnableSoftwareBreakpoint.

EnableSoftwareBreakpoint

Можно выделить 3 платформно-зависимых места (функции): GetSoftwareBreakpointTrapOpcode, ReadMemory и WriteMemory. Первый - получение инструкции точки останова. Она зависит больше от CPU, чем от ОС, поэтому опустим и сконцентрируемся на 2 последних.

Реализация чтения/записи для Windows находится в классе ProcessDebugger методах ReadMemory/WriteMemory. Внутри это просто обертка над ReadProcessMemory. Но дополнительно мы делаем повторную попытку чтения, если произошла ошибка чтения из-за того, что указанный диапазон слишком большой (превышает допустимый), то пытаемся прочитать сколько сможем:

ReadMemory

Запись выглядит так же как и описали ранее. WriteProcessMemory + FlushInstructionCache:

WriteMemory

hardware vs software

Но это точки останова. Иногда необходимо выполнить только 1 инструкцию. В Windows это реализуется не с помощью отдельной функции, а с помощью выставления TF (Trap Flag) флага в регистре RFLAGS (для x86). Этот флаг 9-ый, то есть его шестнадцатеричное представление - 0x100.

Когда этот шаг сделан, то возникает то же исключение EXCEPTION\_DEBUG\_BREAK, но код исключения будет уже EXCEPTION\_SINGLE\_STEP.

Выставление TF

Если мы посмотрим на реализацию lldb, то увидим, что они также выставляют этот флаг:

TargetThreadWindows::DoResume

Отладочная информация

Теперь переходим к части поинтереснее - отладочные символы. Подход Windows - хранить отладочные символы CodeView в отдельном файле .pdb.

Для работы с отладочными символами необходимо подключить уже другой заголовок - <DbgHelp.h>. В нем определено множество вспомогательных функций для работы с символами отладки.

Перед началом работы с символами необходимо инициализировать состояние. Это делается с помощью функции SymInitialize. Эта библиотека часто на вход принимает дескрипторы процесса, потока и даже файла. Но откуда их взять?  
Кажется это сложно, но в Windows поняли, что возможно чаще всего библиотеку будут инициализировать в самом начале. Поэтому они сделали такое решение: когда процесс создается, то генерируется событие CREATE\_PROCESS\_DEBUG\_EVENT и в его структуре данных события уже хранятся нужные нам дескрипторы.

Когда библиотека инициализирована нужно ее настроить. Для этого используется функция SymSetOptions. На вход ей передается маска флагов состояния. Сейчас я передаю 1 флаг - SYMOPT\_LOAD\_LINES, загрузить информацию о строках исходного кода

Последним шагом, скармливаем ей наш модуль, из которого она должна прочитать символы. Это делается с помощью функции SymLoadModule.

Собирая все вместе получаем нечто подобное.

Инициализация при запуске

Как можете заметить, вся необходимая информация уже находится с данных события, что очень удобно. Можете заметить, что SymLoadModule возвращает какой-то base. Это базовый адрес модуля, но я его интерпретирую как дескриптор/идентификатор, так как его нужно передавать везде.

Далее можем приступать к работе. Для начала, рассмотрим как получается информация о строках исходного кода. И на удивление просто: нам нужно только 2 функции: SymEnumSourceFiles и SymEnumLines. Обе устроены одинаково: передаем идентификатор цели (процесс, дескриптор, файл...), контекст (значение, которое будет передаваться) и колбэк, который будет вызван для каждого элемента множества. Под множеством, как можно догадаться из названий функций, имеется ввиду множество файлов исходного кода и строк внутри них соответственно.

Все это можно описать так:

SymEnumSourceFiles

Можете заметить, что колбэк для SymEnumLines получает структуру SRCCODEINFO и она довольно полезная, потому что хранит в себе не только номер строки и файл, но еще и адрес этой инструкции (line->Address).

Выводиться будут все файлы, участвовавшие в компиляции конкретного файла. Даже заголовки (или как в моем случае, неизвестные .cpp файлы), поэтому стоит проверить название файла в колбэке.

Теперь, перейдем к более интересной теме - типам. Для них также есть отдельная функция, которая принимает колбэк - SymEnumTypes. Передаваемый колбэк получает на вход указатель на структуру SYMBOL\_INFO. В ней хранится информация связанная с символом.

Также, как как Windows сильно связан с PDB, то в этой структуре есть поле Tag. Она хранит тэг типа из *PDB* файла. С ее помощью (если PDB доступен) мы можем получить тип символа.

Но было бы все так просто. Я сказал, что в SYMBOL\_INFO хранится информация о символе, но символы бывают разные и, соответственно, данные могут быть разные. В этой структуре хранятся все необходимые данные для *начала* работы, а все остальное надо получить самим.

Получать дополнительную информацию о символе мы можем с помощью функции SymGetTypeInfo. Для ее использования мы передаем значение из перечисления IMAGEHLP\_SYMBOL\_TYPE\_INFO (команду) и адрес, в который будет записываться результат (тип зависит от команды). Всего есть 39 команд, но все описывать не буду.

Просто покажу пример использования. Мы будем получать все структуры (классы) и выводить все их поля вместе с типами. Для простоты, я буду выводить только int, char и float.

Определение типа структур

Теперь, давайте разберем, что здесь написано. Для удобства я добавил комментарии по частям. Вначале (1) мы проверяем, что переданный символ - это пользовательский тип (SymTagUDT). Тэг хранится в поле Tag, но также мы можем получить тэг сами (1.5).

После (2) мы получаем информацию о дочерних членах этого символа. В этой части необходимо вначале получить их общее количество, а затем аллоцировать необходимое место для структуры. Делается это с помощью 2 команд.  
В результате мы получим массив из TypeIndex дочерних членов (т.е. просто индексы, не структура SYMBOL\_INFO).

Далее (3) начинаем итерироваться по всем членам и обрабатывать каждый. Сперва (4), проверяем, что этот член - поле (SymTagData). Дело в том, что дочерними могут быть функции или базовые классы (я тестировал логику на C, поэтому такую проверку я мог и не делать).

Теперь (4), получаем информацию о типе этого поля. Делается это за 3 шага:

1. получаем индекс типа поля
2. получаем тэг этого тип и
3. получаем название этого типа.

Шаг 1 наверное понятен, поэтому рассмотрим остальные. Здесь типы хранятся примерно так же как и в DWARF, то есть для указателя, структуры пользовательской, базового (встроенного) типа, typedef'ов и т.д. есть свои тэги и обрабатывать их необходимо также в зависимости от тэга. Для этого мы его и получили. Я вывожу название только для базовых типов, поэтому на шаге 3 получаю индекс базового типа (Base Type). Они все определены и значения хранятся в перечислении BasicType (префикс bt у его значений).

Вроде просто, но не совсем. Кто хочет использовать DbgHelp, то вам несколько советов:

1. Многие перечисления и другая информация о PDB файле (то же самое перечисление SymTagEnum) по умолчанию заголовком DbgHelp.h не отдается. Для этого необходимо перед его включением объявить макрос \_NO\_CVCONST\_H. Это тогда, когда у вас нет заголовочного файла cvconst.h. Его можно просто [скачать](https://github.com/microsoft/microsoft-pdb/tree/master/include) с репозитория Microsoft на GitHub (что я и сделал)
2. У каждого тэга есть свои определенные команды (TI\_GET\_XXX), которые он поддерживает. Если нет, то возникнет ошибка Incorrect function. Я искал, но не нашел список поддерживаемых команд (скорее всего они в документации PDB). Зато нашел [проект TypeInfoDump](https://www.debuginfo.com/tools/typeinfodump.html), в котором есть примеры использования этих команд для различных тэгов. Во время написания кода для меня он стал документацией.

Стек вызовов

Честно говоря, эту часть я писать изначально не планировал, но в любой статье посвященной отладчику на Windows она есть, поэтому исключением не стану. Напомню, что стек вызовов - это пройтись по всем фреймам, начиная с текущего, и вывести информацию о нем.

Для развертки стека в Windows имеется отдельная функция StackWalk. Это рабочая лошадка всего - мы просто вызываем ее в цикле, пока не наткнемся на адрес возврата 0. Это означает, что мы дошли до конца.

Эта функция работает со структурой STACKFRAME. В самом начале мы заполняем ее поля (обычно текущим фреймом с помощью GetThreadContext), а далее заходим в цикл и считываем данные очередного фрейма.

Для примера я вывожу файл и номер строки, на которой этот фрейм находится в данный момент. Делаю это с помощью DbgHelp.

Собирая все вместе мы получаем:

StackWalk

Но и тут не без замечаний. Для доступа к функции SymGetLineFromAddr необходима библиотека DbgHelp. И здесь стоит вернуться к моменту ее инициализации. Изначально я передавал еще 1 флаг - SYMOPT\_DEFERRED\_LOADS, для ленивой загрузки информации, но когда я начал тестировать код, то *всегда* возникала ошибка The specified module could not be found. Она исчезла после удаления этого флага. Причину я не искал.

Но все же эта ошибка говорит о том, что указанный модуль найти не удалось и это вполне нормальная (если так можно сказать) ситуация. Например, если вверх по стеку находится не наш код и доступа к отладочным символам у нас нет. Для тестирования я вызывал функцию DebugBreak (генерирует исключение для отладки) - она была первой в стеке и для нее не было отладочной информации, поэтому для нее возникала эта ошибка.

KolibriOS

Я думал, какую бы еще ОС рассмотреть. Но рынок делится между UNIX-подобными ОС и Windows. Linux рассмотрели, Window тоже, а остальные похожи на Linux. Поэтому принял решение рассмотреть что-нибудь необычное. Остановился на [KolibriOS](https://kolibrios.org/ru/" \t "_blank).

Для начала введение - что это такое. KolibriOS это форк (2004 год) другой ОС MenuetOS. Ее ядро написано полностью на ассемблере FASM, но прикладные программы (браузер, редакторы, плеер и т.д.), которые поставляются вместе, уже на C.

Часто говорят, что она помещается на 1 дискете. Это правда, но частично - это подготовленный образ для дискеты. Загрузочный образ для диска (.iso) весит уже чуть более 100 Мб. Чтобы добиться такого, флоппи образ был сильно урез в некоторых частях, но даже с учетом этого при первой загрузке я разницы в UI не заметил: то же окно и те же программы (даже игры).

Теперь перейдем к основной теме - отладке.

Как могли заметить, чтобы программу, выполняющуюся на платформе X, можно было отладить, необходимо, чтобы эта платформа X предоставляла нужные инструменты. В случае ОС это системные вызовы. KolibriOS их тоже имеет - 80 штук, причем у некоторых системных вызовов есть подкатегории. В KolibriOS они называются функции и подфункции, соответственно. Их вызов выполняется, как и во многих других ОС - выставляем в регистр eax номер системного вызова (другие регистры заполняем при необходимости, например в ebx - номер подфункции) и вызываем инструкцию int 0x40, которая нужна для системного вызова (на Linux используется же int 0x80).

Для отладки имеется собственный системный вызов под [номером 69](https://wiki.kolibrios.org/wiki/SysFn69/ru). Она так и называется - Отладка. Также имеется 10 подфункций (по номерам):

1. Указание буфера для сохранения отладочных сообщений
2. Получение регистров
3. Сохранение регистров
4. Отсоединение от отлаживаемого процесса
5. Приостановка потока
6. Возобновление потока
7. Чтение из памяти процесса
8. Запись в память процесса
9. Завершение отлаживаемого потока
10. Установить/снять *аппаратную* точку останова

Эти вызовы можно применить только для отлаживаемого процесса. Чтобы начать отлаживать процесс, необходимо создать его с определенным флагом. Для создания новых процессов используется системный вызов 70, подфункция 7 - Запуск программы (далее, такие пары буду указывать как 70,7). Ей передается указатель на особую структуру, в которой необходимо выставить бит 0 в поле флагов. Пока это единственный поддерживаемый флаг.

В документации на wiki.kolibrios.org используется именование функция-подфункция. Но чтобы не было конфликтов с обычными функциями, я буду называть их системными вызовами.

В качестве подопытного мы будем рассматривать koldbg - интерактивный (оконный) отладчик для KolibriOS. Он кстати тоже написан полностью на ассемблере, поэтому примеры кода будут интересными. Кроме него есть еще mtdbg, но у первого больше функциональности.

Макрос mcall

Архитектура

Для начала немного архитектуры. В koldbg она событийно-ориентированная.

Перед началом работы выставляются флаги событий, которые мы отслеживаем, системный вызов 40 - установить маску для ожидаемых событий. Отслеживается 4 события: нажатие на клавиши (клавиатуры), нажатие на кнопку (окна), событие отладки и перерисовка окна (да, оконная подсистема внутри ядра).

Далее, вызывается бизнес-логика и по ее окончании вызывается функция WaitEvent - ожидание другого события и запуск для него обработчика. Это системный вызов 10 - ожидать события.

Событийная часть

В ассемблере могли заметить обработчик нажатия клавиши - Key. Внутри он вызывает другую функцию DoCommandLine. Из названия становится ясно, что она обрабатывает командную строку. В общих чертах, отладчик хранит буфер текущей команды и как только обработчик Key получает Enter, то происходит процесс выполнения команды. Для поиска команды используется функция FindCmd. После ее нахождения в регистре esi будет храниться адрес нужного обработчика. Функции всех обработчиков имеют вид OnXXX, где XXX - это название команды.

Запуск процесса под отладкой

Теперь, мы знаем достаточно, чтобы начать исследовать отладчик.

В начале, рассмотрим процесс запуска отлаживаемого процесса. Для запуска программы используется команда load. На вход она получает путь к программе и аргументы. Здесь ничего необычного.

За load отвечает функция OnLoad. Внутри она просто обертка над системным вызовом 70,7. Этот системный вызов просто создает и запускает новый процесс. Он принимает указатель на структуру с информацией о целевом процессе. В нем имеется поле флагов, в котором необходимо выставить единственный флаг - сигнал о том, что процесс необходимо отлаживать.

Запуск процесса

Точки останова

Теперь, перейдем к более интересной теме - точки останова. За них отвечает команда bp и соответствующий обработчик OnBp. Внутри нее происходит тривиальная логика: вычисление расположения, нахождение существующей точки останова, ее выставление и показ сообщения.

Точка останова выставляется так же как и на всех других платформах - запись инструкции 0xCC. Эта логика находится в функции EnableBreakPoint. Для ее выставления необходимо взаимодействовать с памятью отлаживаемого процесса - это реализуется с помощью системных вызовов 69,6 (прочитать из памяти отлаживаемого процесса) и 69,7 (записать в память отлаживаемого процесса):

EnableBreakPoint

Когда мы достигаем точки останова (программной), то вызывается обработчик события исключения. Эту часть уже могли видеть, но продублирую еще раз:

Обработчик события исключения

Когда мы получаем исключение, то проверяем, что код исключения, который генерирует int3. А затем проверяем, что под нами находится либо:

* 0xCC - это наш int3 либо
* 0xCD03 - это уже развернутая версия int 3 (напомню, что 3 - это аргумент для int, а сама инструкция int имеет код 0xCD)

Потом мы просто проверим, что это наша точка останова (мы ее выставили).

Немного жаргонизма

Шаги

Теперь приступим к шагам. Мы рассматриваем koldbg, это ассемблерный отладчик, то есть здесь нет информации о строках исходного кода, только ассемблер, поэтому всего у нас будет 2 типа шагов:

1. Step
2. Proceed

Первое - это всем известное step into. То есть мы просто выполняем текущую инструкцию. Шаг выполняется с помощью команды step (или s) и за его обработку отвечает функция OnStep.

Реализовано это с помощью выставления TF флага в регистрах. Но дальше уже становится интереснее, потому что происходит исследование текущей инструкции. В частности, мы проверяем сейчас мы выполняем инструкции:

* int XXX
* syscall
* sysenter

Если мы встретили хотя бы один из них, то TF флаг убирается, и после этих инструкций выставляется временная точки останова (one-shot breakpoint).

OnStep

Теперь подойдем к Proceed. Если не догадались, то это step over. Шаг выполняется с помощью команды proceed (или p).

Я выделил 3 категории инструкций, которые необходимо обрабатывать по особенному:

1. Операции со строками: movs (копирование), cmps (сравнение), stos (сохранение), lods (чтение), scas (поиск)
2. Циклы: loopnz, loopz, loop
3. Вызов функции: call

Если мы встречаем одну из таких, то просто вставляем временную точку останова после этой инструкции. В противном случае, выполняем step (буквально вызываем его обработчик).

OnProceed

Также, вы могли заметить, что в начале этой функции находится вызов другой функции IsPrefix. Причина в том, что в инструкциях x86 каждая инструкция может иметь префикс - 1 байт описывающий команду или уточняющий ее семантику. Например, можно выделить такие примеры префиксов:

* Указание размера операнда (32/16 битные регистры)
* Указание размера адреса (64/32 битный адрес)
* Повторить (rep, repnz)
* Заблокировать кэш линию для атомарной операции (lock)

Они просто информационные, поэтому их стоит пропустить. Но здесь кроется разгадка того почему для строковых инструкций и цикла мы ставим точку останова вместо того, чтобы просто выполнить ее (step). Все дело в префиксе rep/repnz - повторение инструкции пока регистр RCX не станет равным 0. Сама инструкция (строковая или цикл) выполняет только 1 действие, но из-за rep она будет постоянно выполнять эту инструкцию раз за разом.

А если посмотрим на функцию IsPrefix, то поймем, что она не возвращает, что за тип префикса перед нами. Поэтому все что нам остается - думать о худшем и поставить точку останова.

IsPrefix

Ну и последний вопрос - как мы понимаем, что остановились из-за шага? Все очень просто - в обработчике исключения мы перепрыгиваем через обработчики точки останова и передаем управление пользователю. А флаг TF очищается всегда, даже если не выставлен.

Архитектуры ЦП

Теперь перейдем к самому низу - процессорам. К этому моменту уже становится понятно, что от процессора нам необходимо по сути 2 вещи: знать инструкцию точки останова и возможность выполнить только 1 инструкцию. Дополнительно, можно указать возможность поставить аппаратную точку останова.

Начнем с x86, так как все примеры выше относились к нему, поэтому начать будет просто.

x86

Также известен под: i386, 8086, IA-32

Это 32 бит, 64 рассматриваю отдельно

В начале немного теории.

Процессор выполняет инструкции последовательно (есть пайплайны, но они должны быть прозрачными для пользователя). Но что же делать, если нам пришло уведомление от кого-то и надо его проверить?

Здесь в игру вступают прерывания (interrupts) и исключения (exceptions):

* interrupt - это прерывание, генерируемое внешним устройством (например, сетевая карта)
* exception - это сигнал, который генерируется самим процессором из-за внештатной (или не очень) ситуации (например, деление на 0)

Когда ОС загружается одна из первых вещей, которые она делает - инициализирует IDT, Interrupt Descriptor Table. Можно сказать, что это массив обработчиков таких прерываний/исключений. Все исключения имеют свой номер и когда он случается, то запускается обработчик из IDT. Можно сказать, что это массив, но не указателей, а специальной записи, которая хранит информацию о расположении этого обработчика.

Подобный механизм для обработки асинхронных событий есть практически в каждом процессоре. Теперь перейдем к функциональности.

Точки останова

x86 - это CISC архитектура, то есть размеры ее инструкций переменные. Чтобы мы могли безопасно вставлять инструкции точки останова, ее размер должен быть равен размеру наименьшей адресуемой ячейки. В данном случае это 1 байт - 0xCC. Вообще, это сокращенная форма общей 2 байтной инструкции int 3 (0xCD03). Когда процессор выполняет эту инструкцию, то генерируется исключение Breakpoint Exception (#BP).

Но эти 2 формы - не одно и то же. Согласно [документации](https://www.felixcloutier.com/x86/intn:into:int3:int1), разница проявляется при использовании VM8086 режима (virtual-8086 mode). Цитирую из [документации](https://moss.cs.iit.edu/cs450/etc/325383-sdm-vol-2abcd.pdf):

An interrupt generated by the INTO, INT3, or INT1 instruction differs from one generated by INT n in the following ways:

* The normal IOPL checks do not occur in virtual-8086 mode. The interrupt is taken (without fault) with any IOPL value.
* The interrupt redirection enabled by the virtual-8086 mode extensions (VME) does not occur. The interrupt is always handled by a protected-mode handler.

(These features do not pertain to CD03, the “normal” 2-byte opcode for INT 3. Intel and Microsoft assemblers will not generate the CD03 opcode from any mnemonic, but this opcode can be created by direct numeric code definition or by self-modifying code.)

Подытоживая, INT3, INT1 и INTO обходят различные проверки IOPL (I/O Priviledge Level) и сразу же отправляются обработчику прерывания. Но к CD03 (полный INT3) это не относится.

Посмотрим как это исключение обрабатывается в ядре Linux:

Обработка BP

Можете заметить, что мы зарегистрировали обработчик для int 3 (по коду исключения 3), а затем превратили его в сигнал SIGTRAP, который отправили процессу.

Если вы внимательно читали код, то могли заметить, что в KolibriOS обрабатывается не #BP, а *#GP* - General Protection Exception (код исключения 0x0D). Еще раз приведу этот участок проверки:

Обработка GP вместо BP

Но точка останова генерирует #BP (0x03). Почему так? Ответ мы найдем, если почитаем документацию немного дальше:

Each of the INT n, INTO, and INT3 instructions generates a general-protection exception (#GP) if the **CPL** is greater than the **DPL** value in the selected gate descriptor in the IDT.

Здесь речь идет о каких-то CPL и DPL - Current Protection Level и Descriptor Protection Level, соответственно. Если говорить коротко, то когда выполняем int XXX, то за его обработку отвечает функция, которая хранится в IDT (Interrupt Description Table). Грубо говоря, это массив из 8 байтных (для 32-битной системы) дескрипторов. За обработку прерывания XXX (аргумент для int) отвечает обработчик под индексом XXX в IDT.

Но безопасность превыше всего, поэтому, чтобы ограничить доступ непривилегированным процессам, каждый такой дескриптор имеет "уровень допуска" - этот самый DPL. Он задается как 2 битное число и соответствует своему [кольцу защиты](https://wiki.osdev.org/Security#Rings). Их может быть 4 (от 0 до 3), но на практике используется 0 (kernel-space) и 3 (user-space).

А теперь, следите за руками. Согласно документации, все int XXX должны проверять свой IOPL (I/O Priviledge Level) - CPL *должен быть не больше* DPL (то есть у вызывающего процесса привилегий для вызова не меньше, что логично), но если это не так, то будет вызвано исключение #GP и int3 тому не исключение.

Где задается этот уровень привилегий для обработчика прерывания? В том самом дескрипторе - элементе IDT.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Offset** | **P** | **DPL** | **0** | **Gate Type** | **Reserved** | **Segment Selector** | **Offset** |
| 63-48 | 47 | **46-45** | 44 | 43-40 | 39-32 | 31-16 | 15-0 |

Жирным я выделил поле, которое за этот DPL отвечает. Оно как раз занимает 2 бита.

Чтобы все положить на свои места посмотрим как регистрируются обработчики IDT в ядре KolibriOS:

Регистрация IDT

Можно заметить, что для создания этой таблицы используется шаблонный код - в каждом элементе IDT различаются только адреса функции обработчика. Но все остальное у них одинаково. Давайте рассмотрим эту одинаковую часть, а именно вот эту инструкцию - mov eax, (10001110b shl 24) + os\_code.

Адрес обработчика занимает 32 бита, но разбит на 2 16-битные части. В начале мы копируем первую часть, т.е. 16 бит, и уже после нее выполняем интересующую нас инструкцию. Эта инструкция копирует 32 бита в текущий дескриптор (который заполняем на этой итерации). Заполнение происходит с 16-ого бита, поэтому мы заполняем биты с 16 по 48. Вычислим чему будет равно (10001110b shl 24) + os\_code - это 32 битное число. Если мы наложим ее на маску этого дескриптора, то получим, что битовое число 10001110 это биты с 40 по 47, то есть отвечают за поля P, DPL и Gate Type. А что же в DPL у нас записано? Правильно - 00, то есть все обработчики в IDT имеют право выполняться только в кольце ядра. А согласно документации, если CPL > DPL (3 > 0, в случае отладчика), то будет генерироваться #GP. Вот и все!

Шаги

Теперь, посмотрим на шаги. Реализовать их можно 2 способами.

Про первый мы уже слышали: выставление флага TF (Trap Flag) в EFLAGS регистре. Если он выставлен, то после каждой выполненной инструкции процессор будет генерировать исключение #DB (0x01).

Второй способ - это инструкция int1 (у нее также есть своя 1-байтовая короткая версия - 0xF1). Она также посылает исключение #DB. Но разница между ними есть - работа с регистрами. Согласитесь, если мы получили #DB, то в общем случае не ясно из-за чего - выставленный флаг или инструкция. Поэтому, было решено добавить специальный флаг BS в регистр DR6 (Debug-Status): этот флаг выставляется, если #DB посылается из-за выставленного флага TF.

Я не разработчик железа, но про int1 (osdev или документация intel) пишут, что он подходит для отладки железа по причине того, что он обходит проверку на DPL и может сразу отправить #DB (т.е. не получим #GP):

In contrast, the INT1 instruction can deliver a #DB even if the CPL is greater than the DPL of descriptor 1 in the IDT. (This behavior supports the use of INT1 by hardware vendors performing hardware debug)

Таким образом, в обычной программе нам достаточно просто выставить флаг TF, чтобы запустить single-stepping режим (что в Windows руками и делали).

Давайте, посмотрим как ptrace реализует этот режим:

ptrace(PTRACE\_SINGLESTEP)

Как и ожидалось, ptrace просто выставляет флаг TF в EFLAGS.

Аппаратные точки останова

Последнее - точки останова. Как вы могли догадаться, речь пойдет об аппаратных точках останова.

В x86 поддерживается 4 аппаратных точки останова. Напомню, что эти точки останова срабатывают при попытке работы с какой-либо ячейкой памяти. Регистры DR0, DR1, DR2 и DR3 хранят эти адреса.

Также, в регистре DR7 (Debug Control) хранится описание этих точек останова. Для каждой свой набор характеристик (далее n - это номер точки останова, 0-3):

* Локальная или глобальная (первая, очищается при переключении задачи, вторая - сохраняется) - Ln или Gn
* Условия запуска (чтение, запись, выполнение) - Bn
* Размер адреса (накладывается маска, чтобы определить подходит ли адрес) - LENn

Все эти точки останова вызывают #DB, но это исключение генерируется в различное время: чтение/запись после, а выполнение - перед выполнением целевой инструкции. В документации делают разделение на классы исключений: первое - это Trap (генерируется после выполнения инструкции), второе - Fault (перед выполнением).

На самом деле, Fault возникает *после* выполнения, просто состояние восстанавливается каким было до выполнения.

Чтобы понять, что #DB возник из-за аппаратной точки останова используются флаги B0 - B3 регистра DR6. Они выставляются перед выполнением обработчика. Это позволяет нам определить, какая из них сработала.

Давайте посмотрим как они (аппаратные точки останова) используются. Но делать это будет на примере gdb, так как это функциональность предоставляемая процессам.

В gdb аппаратные точки останова называются watchpoint (или data breakpoint). Для ее установки используется команда watch.

Сама логика ее выставления находится в функции x86\_dr\_insert\_watchpoint:

x86\_dr\_insert\_watchpoint

Можем заметить 4 вещи:

1. В начале, мы проверяем маску длины адреса. Проверяются 3 маски, а также еще 4-ая, но с флагом (в следующем разделе объяснение)
2. gdb хранит копию (mirror) настоящих регистров отладки, работает с ними, а после обновляет настоящие
3. Для обновления DR регистров используется ptrace(POKEUSER)
4. Все аппаратные точки останова выставляются локальными (X86\_DR\_LOCAL\_ENABLE (state, i);), хотя поддержка для глобальных имеется
5. gdb не поддерживает watchpoint с доступом чтения (hw\_read)

По последнему пункту отдельно. В [документации gdb](https://www.sourceware.org/insight/onlinedocs/gdbint_3.html) нашел следующее:

x86 processors support setting watchpoints on I/O reads or writes. However, since no target supports this (as of March 2001), and since enum target\_hw\_bp\_type doesn't even have an enumeration for I/O watchpoints, this feature is not yet available to GDB running on x86.

То есть, физически мы можем выставить эти флаги, но работать они не будут. Поэтому в gdb просто не стали это делать.

AMD64

Только что мы рассмотрели x86. Но у него есть 64-битный брат - amd64 (x64, x86\_64, x86-64).

По факту, это расширение x86, поэтому сильных различий в ней нет, но некоторые есть, их и перечислю. Изменения я брал из [документации](https://parallel.ru/sites/default/files/ftp/computers/amd/24593.pdf).

Во-первых, регистры. В amd64 они были расширены до 64 бит и регистры отладки тоже - DR0 - DR3 теперь занимают 64 бита. Если вспомнить, то длина адреса определяется полем LENn в DR7. Но поддержка 64-бит оказалась проста - в таблице соответствия значения к длине оставалось одно неиспользуемое - 10. Вот его и стали использовать для определения 64-битного адреса.

В x86 я обмолвился, что в при выставлении DR регистров проверялась длина для 3 значений и еще 4 при специальном флаге. По сути это флаг 64-битного режима.

Также, добавлено еще 3 отладочных регистра:

1. DebugCtlMSR
2. LastBranchx
3. LastExceptionx

MSR - Model-specific register

DebugCtlMSR по сути добавляет 2 фичи:

1. Отслеживание производительности
2. Останов на ветвлении

Первое, отслеживание производительности, нам не особо интересно. Поэтому сфокусируемся на 2 - останов на ветвлении.

Раньше, мы могли остановиться либо после текущей инструкции, либо на определенной точке останова. Но благодаря флагу BTF (Branch Single Step) этого регистра мы можем выполнить инструкцию до следующей инструкции ветвления: jmp, loop, call, int, syscall и т.д.

Этот флаг вспомогательный к TF (если BTF = 0, то поведение старое, если 1, то останавливаемся на ветвлении).

Кроме того, есть другой флаг - LBR (Last Branch Record). Если и он выставлен, то после срабатывания BTF нам в отдельном регистре становится доступна информация ветвления (старый/новый RIP).

ARM

Также известен как: aarch64/aarch32.

Когда обсуждают отладку на уровне процессора, то чаще всего я слышал про x86. Поэтому сейчас настало время поговорить и про другие архитектуры. Начнем с ARM.

ARM - это не 1 архитектура, а их семейство. Существует несколько наборов инструкций (разделение по длине команд):

* A32 - команды 32 бита
* T32 - команды 16 и 32 бита
* A64 - команды 32 бита

A64 появилась позже остальных (с выходом ARMV8) и ее отличие в поддержке 64-битной разрядности. В контексте ARM имеется такое понятие как Execution State и их может быть 2: AArch64 и AArch32. Говоря грубо, это разрядность: первое - 64 бита (A64), вторая - 32 бита (A32, T32). Причем, в процессе работы возможен переход между ними.

Я буду рассматривать AArch64 и, соответственно, A64. В качестве документации используют [этот документ](https://github.com/kn-gloryo/armv8a/blob/master/docs/ARM%20Architecture%20Reference%20Manual%20-%20ARMv8%2C%20for%20ARMv8-A%20architecture%20profile.pdf).

Точки останова

Точка останова реализуется с помощью инструкции BRK. При ее выполнении генерируется исключение Breakpoint Instruction. Но в отличие от x86 это не просто константная инструкция - она принимает аргумент, 2 байтное число (константа). Эта константа затем сохраняется в регистре ESR\_ELn (n - уровень исключения).

Уровни исключений

Таким образом, чтобы поставить точку останова, нам необходимо подставить на место этой константы какое-нибудь число. Но в общем-то это не обязательно.

В указанном документе байты точки останова не указаны, зато они есть [здесь](https://student.cs.uwaterloo.ca/~cs452/docs/rpi4b/ISA_A64_xml_v88A-2021-12_OPT.pdf). Хотя зачем напрягаться, лучше посмотреть что там в gdb.

Выставление точки останова в gdb

Можно заметить, что gdb использует 0 для этой константы.

В AArch32 используется другая ассемблерная инструкция - BKPT. Но при этом в A32 используется 32 битная инструкция с 2 байтной константой, а в T32 уже инструкция короче - 16 бит, а константа - 1 байт.

Чтобы определить, что за тип исключения был, в регистр ESR\_ELn записывается код исключения. Для BRK он будет равен 0x3C. И если посмотрим в код Linux, то увидим его использование:

Обработка BKPT в Linux

Шаги

Реализация шагов аналогична x86. В регистре PSTATE имеется специальный флаг SS (Single Step). Если он выставлен, то после выполнения инструкции будет послан специальный сигнал.

Ранее, я уже показывал код из lldb для продолжения работы на Windows. Там, мы рассматривали TF флаг процессора. Но также внизу был код для ARM. Продублирую этот участок кода:

TragetThreadWindows::DoResume

Комментарий говорит сам за себя - выставляем флаг SS (бит) в регистре PSTATE. Ну и чтобы наверняка - посмотрим как в Linux реализуется ptrace(PTRACE\_SINGLESTEP):

ptrace(PTRACE\_SINGLESTEP)

На этом можно было и остановиться, если бы не gdb. В gdb имеется функциональность выполнять software шаги, то есть тогда когда не доступны шаги через hardware (те самые флаги регистра). В таких случаях, шаги реализуются через точки останова.

И вот тут следует знать об одной особенности этой архитектуры - расширение LSE, Large System Extensions. Грубо говоря, эта фича добавляет поддержку транзакций - набор инструкций, которые должны быть выполнены атомарно.

В gdb решено, что если мы решили сделать шаг, когда находимся перед такой последовательностью, то шаг - это вся транзакция. А для реализации этого, ищется конец этой последовательности. Реализовано это в функции aarch64\_software\_single\_step:

aarch64\_software\_single\_step

Аппаратные точки останова

ARM часто используется как микропроцессор на всяких платах, поэтому запустить на нем Линукс, накатить gdb и начать отладку как обсуждали ранее получится не всегда. Но разработчики не глупые, поэтому этот недостаток компенсировали множеством дополнительных регистров.

Для отладки существует несколько вспомогательных отладочных регистров:

* DBGAUTHSTATUS\_EL1 (Debug Authentication Status) - информация об интерфейсе аутентификации (зависит от реализации)
* DBGCLAIMSET\_EL1 (Debug Claim Tag Set) - выставление специальных битов тэга CLAIM (используется для взаимодействия отладчика и машины)
* DBGCLAIMCLR\_EL1 (Debug Claim Tag Clear) - очистка тэга CLAIM
* DBGDTR\_EL0 (Debug Data Transfer) - передача 64-битных слов между отладчиком и машиной (в обе стороны)
* DBGDTRRX\_EL0 (Debug Data Transfer) - передача 32-битных слов от отладчика к машине
* DBGDTRTX\_EL0 (Debug Data Transfer) - передача 32-битных слов от машины к отладчику
* DBGPRCR\_EL1 (Debug Power Control) - запрос на выключение (его эмуляция)
* DBGVCR32\_EL2 (Debug Vector Catch) - возможность доступа к регистру DBGVCR (он есть в AArch32, но не в AArch64)
* DSPSSR\_EL0 (Debug Saved Program Status) - хранит состояние при входе в режим отладки
* DLR\_EL0 (Debug Link) - адрес возврата для продолжения работы из режима отладки

Эти регистры необходимы при взаимодействии с внешним отладчиком, то есть когда мы отлаживаем железо. Вообще, чтобы отлаживать железо на ARM, необходимо войти в состояние отладки (Debug State). Это можно сделать с помощью инструкции HTL. Но про отладку железа я говорить не буду, а эти регистры просто упомянул. Сконцентрируемся мы на 4 других.

Эти регистры отвечают за точки останова. Можно сказать, что это 2 группы с 2 шаблонными регистрами: первый хранит какие-то данные, а второй эти данные интерпретирует (регистр данных и регистр контроля назовем их). И сразу спойлер: каждого регистра 16 штук и в названии имеется число <n> - это его номер (от 0 до 15), хотя в документации говорится, что их может быть от 2 до 16.

Первая группа - точки останова. Когда мы отлаживаем железо, то поставить программную точку останова не представляется возможным, поэтому единственный вариант в данном случае - поддержка со стороны самого железа. И это реализуется с помощью 2 регистров:

1. DBGBVR<n>\_EL1 (Debug Breakpoint Value) - данные
2. DBGBCR<n>\_EL1 (Debug Breakponit Control) - контроль

В регистре контроля хранятся разные поля, но главное - это BT (Breakpoint Type). Оно определяет тип точки останова и оно же определяет то, как интерпретировать регистр данных. Этот поле занимает 4 бита и точек останова 16 типов соответственно (строго говоря, их 8, так как 1 бит используется как флаг).

Далее идет группа уже для точек наблюдения (watchpoint). Это те самые аппаратные точки останова, о которых говорили в x86. Они нужны для отслеживания изменений в адресах данных:

1. DBGWVR<n>\_EL1 (Debug Watchpoint Value) - данные
2. DBGWCR<n>\_EL1 (Debug Watchpoint Control) - контроль

Здесь уже регистр контроля не определяет интерпретацию регистра данных. Но зато в нем также есть поле условий активации - LSC (Load Store Control). В отличие от x86 доступны только 3 события: запись (store), чтение (load) или запись/чтение (load/store).

Адрес в регистре данных тоже должен быть выровнен по 8 байтам. Но как же быть, если я хочу, например, отслеживать изменения на адресе 9 (8 + 1)? Для этого имеется поле BAS (Byte Address Select). Это битовая маска, в которой можно указать смещение отслеживаемого байта. Например, маска 0b00000010 говорит о том, что необходимо отслеживать адрес DBGWVR<n>\_EL1 + 1, поэтому записав в этот регистр адрес 4 и указанную маску. Так как это битовая маска, то, например, выставив все биты в 1 мы будем отслеживать все изменения в слове.

Посмотрим, как это реализуется в gdb:

Выставление аппаратной точки останова

Если мы попадаем на breakponit, то генерируется Breakpoint exception, а если на watchpoint - то Watchpoint exception. То есть в итоге, у нас 3 различных исключения (программная точка останова - Breakpoint instruction exception).

Отладка железа

Ранее я уже обронил слово о внешней отладке, но что такое внятно не объяснил. ARM часто используется на встраиваемых системах. На них ОС не накатишь и по SSH не подключишься. Для их отладки необходимо использовать внешние средства и отладка с помощью них называется внешней отладкой (external debug). По аналогии, отладка привычными нам средствами называется уже self-hosted debug.

Для того, чтобы начать внешнюю отладку на ARM, необходимо войти в состояние отладки (debug state). Когда мы входим в это состояние, то контроль передается внешнему отладчику и выполнение останавливается. А войти в это состояние можно, когда возникают, так называемые, Halting debug exceptions. Самое простое - это инструкция HLT, при ее выполнении машина переходит в режим отладки. Также, это может быть Breakpoint или Watchpoint exception (логично, что при их срабатывании необходимо остановиться и передать управление). Но чтобы external не конфликтовал self-hosted отладкой существует флаг HDE (Halting debug enable) в регистре EDSCR (external debug status and control).

Также, на сколько я понял, поведение инструкции точки останова (BRK) в debug state не определено:

The following events never generate entry to Debug state:

* Breakpoint Instruction exceptions. This means that these events can do one of the following:
* They can generate a debug exception.
* They can be ignored.

Пока мы находимся в режиме отладки мы можем выполнять отладочные действия: выполнять какие-то инструкции (в документации явно прописано как различные инструкции влияют на состояние), читать значения регистров или памяти и так далее.

Чтобы выйти из debug state, необходимо отправить Restart request на специальный триггер.

Но на главный вопрос пока не ответили - так как же происходит общение отладчика и процесса? Для взаимодействия между отладчиком и процессом используется DCC - Debug Communication Channel. Это *логический* канал взаимодействия, а его физическая реализация состоит из совокупности регистров и флагов:

* DBGDTRRX и DBGDTRTX - это флаги передачи данных (упоминал ранее). DBG - debug, DTR - data transfer register, RX - receive, TX - transmit.
* EDITR (External Debug Instruction Transfer Register) - регистр для передачи инструкций, которые необходимо выполнить.
* EDSCR (External Debug Status and Control Register) - регистр с флагами контроля. Например, RXO - флаг уведомления об ошибке потока передачи.

Но конечно же это самый низкоуровневый интерфейс. Им пользуются уже более высокоуровневые, предоставляющие интерфейс удобнее, чем сырые регистры. Таковым является, например, [JTAG](https://habr.com/ru/articles/190012/).

PowerPC

Также известен как: ppc, ppc64, power isa, PowerPC64

PowerPC - это RISC архитектура, созданная совместно Motorolla, IBM и Apple (AIM). Она основана на другой, более ранней POWER, использовавшейся на RS/6000, и проектировалась с учетом совместимости. Работать может как в 32, так и в 64 битном режиме. Для 64 битного режима используется название PowerPC64, его мы и будем рассматривать.

Еще одной особенностью является то, что это гарвардская архитектура, то есть для инструкций и данных используется разная память.

PowerPC определяет 3 уровня ISA:

1. [User instruction set architecture](https://cr.yp.to/2005-590/powerpc.pdf) (UISA)
2. [Virtual environment architecture](https://www.ece.lsu.edu/ee4720/doc/ppc_isa_book2.pdf) (VEA)
3. [Operating envionment architecture](https://wiki.raptorcs.com/w/images/7/72/PPC_Vers201_Book3_public.pdf) (OEA)

Хотел бы я сказать, что это аналогия колец защиты x86, но это не так. IUSA определяет набор основных инструкций, которые можно использовать для написать программ, VEA (грубо говоря) дополняет это все кэшами, виртуальной памятью, атомарностью и т.д., а OEA - уже больше для системных программистов, так как там описывается модель исключений и прерываний, привилегированные инструкции.

У PowerPC 3 уровня привилегий, причем для определения уровня необходимо использовать 2 флага из регистра MSR (Machine State Register):

1. PR (Problem State)
2. HV (Hypervisor State)

|  |  |  |
| --- | --- | --- |
| **PR/HV** | **0** | **1** |
| 0 | Privileged | Hypervisor |
| 1 | User | User |

Уровни привилегий выстроены следующим образом, с увеличением привилегий:

1. User
2. Privileged
3. Hypervisor

Как можно догадаться, User - режим пользователя, а вот остальное - это уже привилегированный режим. Причем Hypervisor можно отнести к уровню самой ОС.

Когда возникает исключение, то оно обрабатывается соответствующим обработчиком. При этом, уровень привилегий повышается: PR всегда сбрасывается (то есть переход в привилегированный режим), а HV выставляется в зависимости от окружения.

Теперь, приступим к отладке.

Программные точки останова

Программные точки останова реализуются с помощью инструкции td - Trap Double Word. Она занимает 4 байта (это RISC) и при ее срабатывании генерируется прерывание Program interrupt.

Ее особенность заключается в том, что она условная, то есть существует возможность добавить условие ее срабатывания. Ее сигнатура td TO, RA, RB. RA и RB - это номера регистров общего назначения (просто их числа), которые сравниваются, а TO - это битовая маска условия срабатывания (меньше, равно, больше и т.д.). Из примера документации: td 0x2, 3, 4 - если регистр 3 (R3) больше (0x2) регистра 4 (R4), то будет сгенерирован Program interrupt.

Также существует и другая версия этой инструкции - tdi, Trap Double Word Immediate. Отличие от td в том, что вместо регистра RB используется константное число (16 битное).

Для 32 битного режима используются инструкции tw и twi (Word вместо Double Word).

Но Program interrupt может срабатывать и из-за других причин, например, при выполнении недопустимой инструкции (Illegal instruction). Для того, чтобы их различать, используется регистр MSR (Machine Status Register). Диапазон 4-48 записан как зарезервированный, но на самом деле он используется для записи дополнительной информации при возникновении исключений. Для Trap исключения выставляется бит 46.

Посмотрим как в Linux обрабатывается это исключение:

Program Interrupt

Шаги

Реализация шагов аналогична предыдущим архитектурам. В регистре MSR имеется флаг SE - Single Step Trace Enable (бит 53). Если он выставлен, то при выполнении следующей инструкции будет сгенерирован Trace interrupt.

Trace interrupt

Также существует флаг BE - Branch Trace Enable. Он занимает бит 54 (рядом с SE). Если он выставлен, то Trap interrupt будет сгенерирован при выполнении очередной инструкции ветвления.

Аппаратные точки останова

Наконец, аппаратные точки останова.

Во-первых, они не обязательны и в реализации могут отсутствовать. Во-вторых, для инструкций и данных используется 2 разных регистра. Причина этого - гарвардская архитектура, инструкции и данные хранятся отдельно. В-третьих, поддерживается только 1 такая точка останова.

Для данных используется регистр DABR - Data Address Breakpoint Register. Для PowerPC64 он имеет размер 64 бита. В нем имеется 3 флага и поле адреса. Адрес - 60 бит, то есть записываемые адреса необходимо выравнивать, а остальное - это флаги условий активации (чтение/запись).

Здесь тоже есть отличие от предыдущих архитектур. Если ранее мы самостоятельно (в коде отладчика) выставляли эти регистры, то сейчас это делается с помощью вызова ptrace. Еще одна причина почему ptrace - это швейцарский нож, так потому что его функциональность может меняться в зависимости от машины. Если мы работаем на PowerPC и имеется расширение HWDEBUG (которое и добавляет аппаратные точки останова), то нам становятся доступны еще 3 запроса и один из них PPC\_PTRACE\_SETHWDEBUG - команда для выставления аппаратной точки останова.

Соответственно, этот код располагается уже в самом ядре.

PPC\_PTRACE\_SETHWDEBUG

Но это точка останова для данных. Для инструкций используется другой регистр - IABR (Instruction Address Breakpoint Register). В ней по аналогии хранится адрес инструкции, такой же выровненный.

Loongson

Также известна как: loongarch, godson, LA32/LA64

Последним в нашем списке рассмотрю архитектуру [Loongson](https://loongson.github.io/LoongArch-Documentation/LoongArch-Vol1-EN.pdf" \t "_blank). Эта архитектура была создана в начале 2000-х, но уже имеет богатую историю - несколько поколений и семейств.

LoongArch - это RISC архитектура. Она состоит базовой части (Loongson Base) и множества (необязательных) расширений, например, LBT - Loongson Binary Translation (расширение для бинарной трансляции x86/ARM/MIPS).

На поверхности эта архитектура похожа на остальные: поддержка 32 и 64 битного режимов, все инструкции размером 32 бит, 3 уровня привилегий (причем уровень 0 - это уровень ядра, а 3 - пользователя), вектор прерываний и т.д.

Модель исключений тоже понятна - у каждого исключения свой номер и его обрабатывает соответствующая функция, хранящаяся в таблице.

Для отладки используется регистр DBG. В нем хранится информация, специфичная для режима отладки. В частности, DST - флаг нахождения в режиме отладки. Он выставляется автоматически при срабатывании Debug exception. Также, есть несколько флагов говорящих о том, какое исключение сработало: несколько отдельных флагов для возможных исключений отладки, а также поле с кодом исключения.

Программная точка останова

За программную точку останова отвечает инструкция break #imm. Ей на вход передается число, которое потом можно передать обработчику. А исключение, которое возникает - Breakpoint exception.

Здесь также уже знакомая ситуация. Но вот что интересно, так это код обработки в ядре. Ранее мы видели, что отладчики передают 0 в качестве заглушки. Но вот в ядре Linux оно используется для передачи некоторой информации, а конкретно используется для механизмов [kprobe](https://habr.com/ru/companies/metrotek/articles/261003/).

Обработка точки останова

Аппаратная точка останова

После программных рассмотрим аппаратные точки останова. Согласно документации их максимальное количество 14. Также для breakpoint'ов и watchpoint'ов используются разные регистры (n - номер регистра):

* watchpoint - MWPNCFG (M - memory)
* breakpoint - FWPnCFG (F - fetch)

Выставление точек останова

Шаг

Наконец-то что-то интересное - шаги. А точнее *их отсутствие*. Да, архитектура Loongson не поддерживает возможность выполнения только 1 инструкции. Но это не значит, что поддержки этого совсем нет.

Для обхода этого ограничения в Linux используются аппаратные точки останова. Она ставится на *текущую* инструкцию (на которую указывает PC). Таким образом, когда выполнение продолжится, будет мгновенно сгенерировано исключение. Последний вопрос - как обнаружить что мы запросили шаг и это не настоящая точка останова. Это сделано просто - в структуре потока хранится адрес инструкции для которой запросили single step.

Обработка шага

В предыдущей секции я не показал обработчик исключения, возникающего при срабатывании точки останова. Покажу здесь, так как заметная ее часть отводится этому шагу:

Обработка точки останова

Заключение

Вот и подошло к концу наше путешествие по отладчикам. Подытоживая, отладчик - это слоеный пирог с протекающей начинкой абстракцией. Нельзя так Просто создать отладчик для всех: разные ЦП, особенности ОС, детали реализации рантаймов, всякие UB, различные форматы отладочных символов и т.д. Мы обязаны знать для какой системы пишем этот отладчик, чтобы знать не только как правильно изменять его состояние, но и какие баги в нем содержатся.

Есть также множество вопросов, которые я не покрыл. Например: загружаемые библиотеки, многопоточные процессы, отладка железа и ОС реального времени, функциональность отладки для разных ЯП (например, точки останова при исключении в C++).

Здесь без особого вывода - каждый сделает его сам.

Ссылки:

* [How debuggers work](https://eli.thegreenplace.net/2011/01/23/how-debuggers-work-part-1) - серия статей, с которой начал изучение
* [Writing a Linux Debugger](https://blog.tartanllama.xyz/writing-a-linux-debugger-setup/) - статьи по написанию отладчика в Linux. Отсюда взяли идею реализации step over добавлением точек останова на все строки
* [Learning about debuggers](https://werat.dev/blog/learning-about-debuggers/) - сборник статей по отладчикам
* [Dumbugger](https://github.com/ashenBlade/dumbugger) - моя реализация отладчика

**Теги:**

* [отладка](https://habr.com/ru/search/?target_type=posts&order=relevance&q=%5b%D0%BE%D1%82%D0%BB%D0%B0%D0%B4%D0%BA%D0%B0%5d)
* [отладчик](https://habr.com/ru/search/?target_type=posts&order=relevance&q=%5b%D0%BE%D1%82%D0%BB%D0%B0%D0%B4%D1%87%D0%B8%D0%BA%5d)
* [debug](https://habr.com/ru/search/?target_type=posts&order=relevance&q=%5bdebug%5d)
* [debugger](https://habr.com/ru/search/?target_type=posts&order=relevance&q=%5bdebugger%5d)
* [ide](https://habr.com/ru/search/?target_type=posts&order=relevance&q=%5bide%5d)
* [x86](https://habr.com/ru/search/?target_type=posts&order=relevance&q=%5bx86%5d)
* [точки останова](https://habr.com/ru/search/?target_type=posts&order=relevance&q=%5b%D1%82%D0%BE%D1%87%D0%BA%D0%B8+%D0%BE%D1%81%D1%82%D0%B0%D0%BD%D0%BE%D0%B2%D0%B0%5d)
* [прерывания](https://habr.com/ru/search/?target_type=posts&order=relevance&q=%5b%D0%BF%D1%80%D0%B5%D1%80%D1%8B%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F%5d)

**Хабы:**

* [Отладка](https://habr.com/ru/hubs/debug/)
* [Системное программирование](https://habr.com/ru/hubs/system_programming/)
* [Программирование](https://habr.com/ru/hubs/programming/)
* [Linux](https://habr.com/ru/hubs/linux_dev/)

Если эта публикация вас вдохновила и вы хотите поддержать автора — не стесняйтесь нажать на кнопку