Санкт-Петербургский государственный университет

Кафедра системного программирования

Группа 22.Б08-мм

Реализация GDB сервера для эмулятора целевых процессоров

Чернобровкина Юлия Владиславовна

Отчёт по учебной практике
в форме «Производственное задание»

Научный руководитель:
ст. преподаватель кафедры ИАС, Смирнов К. К.

Консультант:
Старший разработчик ПО I категории ООО «Софтком», Бабанов П. А.

Санкт-Петербург
2024

Оглавление

Введение

Важным этапом разработки программного обеспечения является тестирование и отладка. Чтобы облегчить жизнь программистам и сэкономить их время, были созданы инструменты для отладки — отладчики. Среди существующих на данное время лидером является GDB [4] — GNU Debugger — бесплатный кросс-платформенный отладчик проекта GNU, который позволяет отлаживать программы, написанные на множестве языков программирования, включая Си, C++, Free Pascal, FreeBASIC, Ada и Фортран, и работает на широком наборе целевых платформ.

Тем не менее, существуют системы, на которых запустить GDB не представляется возможным, в частности это встроенные системы, симуляторы и эмуляторы. Для них GDB предлагает средство по удаленной отладке. Разработчику достаточно создать приложение, реализующее RSP протокол (Remote Serial Protocol) — высокоуровневый протокол, который определяет взаимодействие GDB и удаленного сервера. Создав такой сервер, программист получает возможность отлаживать программу на своей системе и пользоваться всеми удобствами, предоставляемыми GDB. Практика написания своего удаленного GDB сервера достаточно распространена. Свои серверы имеют проекты эмуляторов QEMU [5], gem5 [1] и операционная система Zephyr.

Данная тема была предложена компанией ООО «Софтком». Одним из направлений деятельности компании является разработка эмулятора процессора MIPS64, и для удобства работы с ним необходимо создать сервер, который бы позволил отлаживать программы на эмуляторе удаленно с помощью GDB. Кроме того, одним из требований со стороны компании является реализация возможности прерывать работу эмулятора по запросу пользователя. Таким образом, данная работа посвящена созданию GDB сервера для эмулятора компании «Софтком».

1. Постановка задачи

Целью работы является создание GDB сервера для эмулятора целевых процессоров компании «Софтком».

В ходе работы в осеннем семестре были выполнены следующие задачи:

1. изучен протокол RSP;
2. изучен предоставляемый эмулятором интерфейс;
3. создан сервер, поддерживающий некоторую часть команд GDB.

В весеннем семетре были поставлены следующие **задачи**:

1. провести обзор существующих решений;
2. реализовать недостающую часть функциональности GDB;
3. обеспечить возможность прерывания работы эмулятора.

2. Обзор

В осеннем семестре был детально разобран RSP протокол, определяющий взаимодействие удаленного сервера и GDB, а также интерфейс эмулятора, с помощью которого происходит обработка команд GDB. В весеннем семестре был проведен обзор существующих на данный момент GDB серверов.

При выборе кандидатов на обзор учитывалось наличие открытого исходного кода, а также язык программирования C или C++, поскольку со стороны компании было ограничение на использование в разработке именно этих языков.

2.1. Обзор существующих решений

Реализация собственного GDB сервера — распространенная практика среди разработчиков симуляторов, эмуляторов и встроенных систем. Рассмотрим некоторые из существующих на данный момент решений.

2.1.1. QEMU GDBstub

QEMU — эмулятор разных целевых процессоров с открытым исходным кодом. Поддерживает удаленную отладку с помощью GDB по RSP протоколу. QEMU GDBstub позволяет выполнять стандартные для отладки действия, а именно [2]:

1. изучение регистров и памяти;
2. управление выполнением (continue, run, kill, quit);
3. работа с точками останова (breakpoint и watchpoint);
4. дамп памяти;
5. пошаговое выполнение (s, n, si, ni, finish);
6. вывод информации о стеке и фрейме;

7. печать и запись переменных (print var, set var);

8. работа с сигналами;

9. работа с потоками (thread, info thread).

Кроме того, есть возможность отлаживать многоядерные машины. GDBstub проекта QEMU написан на языке программирования C[1]. Среди процессоров, эмулируемых QEMU, можно перечислить следующие: 80386, 80486, Pentium, Pentium Pro, AMD64 и другие x86-совместимые процессоры; ARM, MIPS, RISC-V, PowerPC, SPARC, SPARC64 и частично m68k [6].

2.1.2. Zephyr GDBstub

Zephyr — операционная система реального времени для встроенных систем с открытым исходным кодом [7]. Также имеет свой GDBstub, реализующий RSP протокол. Предоставляет следующие возможности [8]:

1. добавление и удаление точек останова;

2. пошаговое выполнение;

3. печать обратной трассировки;

4. чтение и запись регистров общего назначения;

5. чтение и запись в память.

GDBstub для операционной системы Zephyr написан на языке программирования C [2].

[1]Репозиторий с исходным кодом проекта QEMU: https://github.com/qemu/qemu/tree/master/gdbstub (дата обращения: 9 апреля 2024 г.)

[2]Репозиторий с исходным кодом проекта Zephyr: https://github.com/zephyrproject-rtos/zephyr/blob/main/subsys/debug/gdbstub.c (дата обращения: 9 апреля 2024 г.)

2.1.3. Embecosm riscv-GDBserver

Часть полного набора инструментов разработки для устройств с архитектурой RISC-V, созданного компанией Embecosm [3]. Сервер написан на языке программирования C++, код является открытым[3]. Функциональность сервера аналогична описанным выше решениям:

1. чтение памяти и запись в память;
2. установка и удаление точек останова;
3. работа с регистрами;
4. пошаговое выполнение программы.

2.2. Обзор используемых технологий

2.2.1. Экранирование спецсимволов

Протокол определяет, что в двоичном представлении данных символы «#», «$», «}», «*» всегда должны быть экранированы. Для этого перед экранируемым символом помещается управляющий символ «}», а к самому символу применяется оператор исключающего или с 0x20. Когда GDB посылает запрос «X addr, len: XXXX…» на запись двоичных данных, возникает необходимость их деэкранирования. Идея реализации деэкранирования была взята из проекта компании Embecosm.

Функция, осуществляющая деэкранирование, принимает указатель на буфер с двоичными данными и его длину. Создается два счетчика, один для просмотра исходного буфера, другой для записи деэкранированного буфера. При встрече управляющего символа «}» к следующему за ним символу применяется оператор исключающего или с 0x20, и таким образом восстанавливается исходный символ. Запись результата происходит в буфер, переданный в функцию изначально.

[3]Репозиторий с исходным кодом GDB сервера компании Embecosm: https://github.com/embecosm/riscv-gdbserver/blob/master/server/RspConnection.cpp (дата обращения: 9 апреля 2024 г.)

Algorithm 1 Деэкранирование спецсимволов в переданном буфере

Require: buf, len
  $i \leftarrow 0$
  $j \leftarrow 0$
  while $i \leq len$ **do**
    if $buf[i] =$ «}» **then**
      $i \leftarrow i + 1$
      $buf[j] = buf[i]$ XOR 0x20
    else
      $buf[j] = buf[i]$
    end if
    $i \leftarrow i + 1$
    $j \leftarrow j + 1$
  end while

2.3. Выводы

Все расмотренные выше серверы в целом предоставляют одинаковую функциональность. Набор команд, реализованный данными сереверами, позволяет пользователю:

1. читать и записывать регистры;
2. читать и записывать память;
3. пошагово выполнять программу;
4. работать с точками останова.

Поэтому сервер, реализованный в данной работе, также должен обеспечивать все перечисленные возможности.

Ни один из рассмотренных выше проектов не использовал в реализации GDB сервера методы многопоточного программирования. В этом аспекте данная работа отличается от расмотренных решений.

3. Описание решения

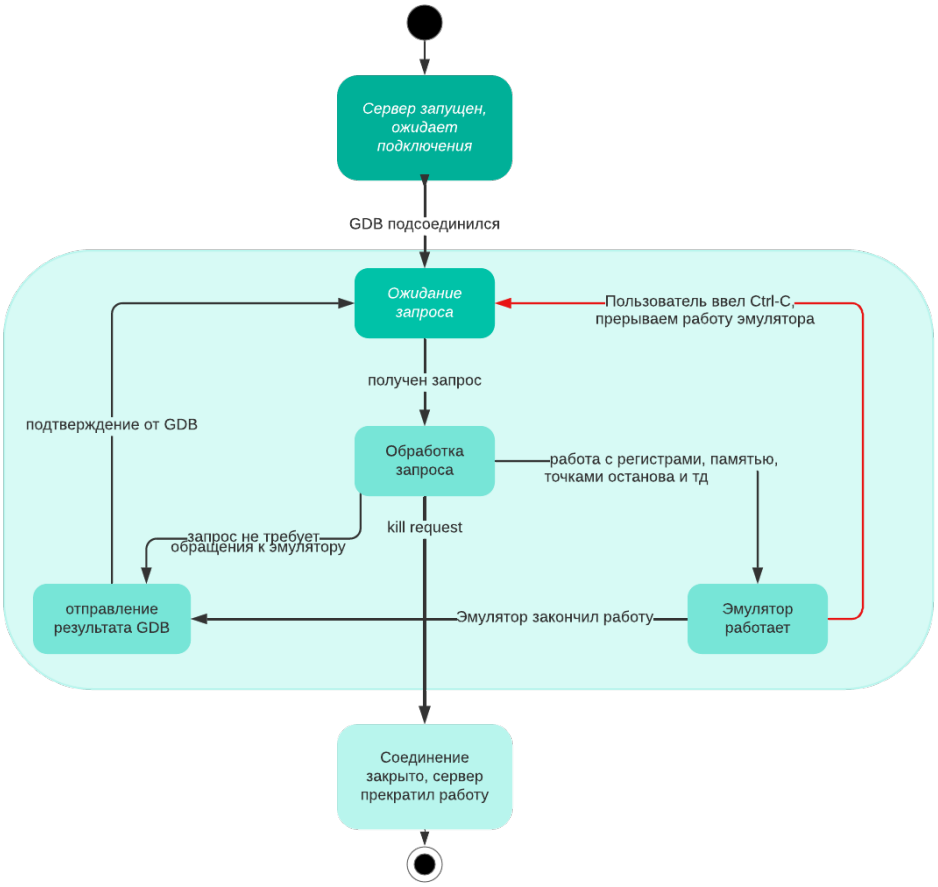

Рис. 1: диаграмма состояний сервера

3.1. Обработка команд GDB

3.1.1. Работа с памятью

Для управления памятью GDB использует команды «`m addr, len`», «`M addr, len:XXXX`» и «`X addr, len:XXXX`», где `addr` — это адрес блока памяти в эмуляторе, `len` — размер блока в байтах, `XXXX` — сами данные в шестнадцатеричном (команда «M») или в двоичном виде (команда «X»).

Интерфейс эмулятора предоставляет функции для чтения и записи блока данных, представленного структурой `data_block` (в ней хранятся размер данных и буффер для хранения). Соответственно, методы сервера должны передавать запрос эмулятору и ответ обратно GDB,

корректно преобразовывая данные в соответствии с требуемым форматом, а также обрабатывать при наличии возникшие ошибки.

За чтение памяти отвечает метод `ReadMem()`. Из пакета считывается адрес и размер блока, затем вызывается функция из интерфейса эмулятора `uemu_dsp()`, которая заполняет структуру `data_block`. Блок памяти поэлементно считывается и преобразуется в строку, которая отправляется GDB. Для преобразований хранящихся в памяти значений в строку используется вспомогательная функция `HexToChar()`.

В методе записи данных `WriteMem()` структура `data_block` заполняется переданными данными, при этом для преобразований используется функция `CharToHex()` в сочетании с побайтовыми сдвигами. Заполненная структура передается в функцию `uemu_dsp` и записывается в память эмулятора.

В методе `WriteMemBin()` кроме описанных выше шагов добавляется деэкранирование буфера функцией `Unescape()`, так как при передаче двоичных данных GDB экранирует специальные символы.

3.1.2. Работа с регистрами

При работе с регистрами GDB может присылать команды «`g`» и «`G XX…`» — чтение и запись регистров общего назначения, «`p num`» и «`P num = XX…`» — чтение и запись одиночных регистров, где `num` — номер регистра.

Данные о всех регистрах эмулятора хранятся в структуре `reg_str`, которая и передается в функцию-интерфейс эмулятора `uemu_dsp()`.

Для чтения всех регистров из обновленной вызовом `uemu_dsp()` структуры `reg_str` поэлементно считываются регистры, преобразуются в строку с помощью функции `ValToHex()` и отправляются GDB. Для чтения одиночного регистра вместо просмотра всех регистров достаточно считать один под нужным номером.

3.1.3. Выполнение инструкций

Эмулятор умеет выполнять единичную инструкцию либо запускать выполнение до встречи точки останова. Обработка команд исполнения инструкций от GDB сводится к вызову нужной функции из интерфейса эмулятора и обновлении состояния эмулятора.

Перед тем как отправить запрос на исполнение команд, GDB опрашивает сервер о доступных действиях. Этими действиями могут быть шаг и продолжение. В связи с этим сервер хранит текущее состояние эмулятора, которое обновляется после завершения команды исполнения инструкций. В соответствии с этим состоянием сервер отправляет GDB перечень доступных команд.

Обработка команд исполнения инструкций представлена методами `StepInstruction()` и `Continue()`.

3.1.4. Вставка и удаление точек останова

Эмулятор на данный момент поддерживает один вид точек останова: безусловный останов по адресу. Методы удаления и вставки точек останова `InsertBp()` и `RemoveBp()` считывают адрес, переданный GDB, и передают эмулятору этот адрес в параметре вызова соответствующей функции.

3.1.5. Обработка ошибок

Функции интерфейса эмулятора возвращают значение, являющееся результатом работы эмулятора. В частности, эмулятор может сигнализировать о произошедшей ошибке. Протокол определяет, что информацию о произошедшей ошибке следует передать GDB. Метод `SendErr()` сопоставляет код ошибки, переданный эмулятором, подходящему сообщению и передает его GDB.

3.2. Независимый слушатель соединения

Одним из требований к серверу является возможность прерывать последовательное исполнение команд эмулятора извне. Эта функциональность необходима в случае, когда пользователь захотел прервать выполнение текущей команды и нажал сочетание клавиш `Ctrl-C`. Протокол не обязывает сервер перехватывать этот сигнал и каким-либо образом его обрабатывать, то есть обработка прерывания со стороны пользователя остается полностью на усмотрении разработчика сервера.

Чтобы иметь возможность прервать эмулятор, работа сервера была разделена на два потока: один занимается обработкой обычных команд, посылаемых GDB, второй поток просматривает сокет GDB на наличие сигнала прерывания со стороны пользователя. При наличии сигнала происходит уничтожение потока, выполняющего обработку текущей команды.

Для работы с потоками используется интерфейс `pthread`, так как он предоставляет больше возможностей для взаимодейтсвия с потоками, чем `std::thread`. Более подробно: с помощью интерфейса `pthread` можно принудительно завершить работу дочернего потока без ущерба для родительского, в `std::thread` же такой функциональности нет.

Оба потока обращаются к сокету клиента, записывая или читая из него, из-за чего может возникнуть нежелательное состояние гонки. Чтобы не допустить состояние гонки была реализована синхронизация потоков с помощью взаимного исключения на критических секциях. Для блокировки и разблокировки мьютекса, связанного с сокетом клиента, используются функции `pthread_mutex_lock()` и `pthread_mutex_unlock()`. Проверяющий поток захватывает мьютекс перед тем как проверить сокет клиента, и освобождает после проверки. Обрабатывающий поток захватывает мьютекс перед отправкой сообщения GDB и освобождает после получения следующей команды. Таким образом обработка команды от GDB с обращением к эмулятору и проверка сокета на наличие сигнала о прерывании происходят параллельно, а обращение к сокету клиента синхронизировано.

Все действия, связанные с потоками, реализованы в методе `ThreadsHandler()`. Поток создается функцией `pthread_create()`, в качестве аргумента передается указатель на метод `RequestLoop()`, который обрабатывает команды GDB до получения сигнала о разрыве соединения. Родительский поток, когда мьютекс свободен, проверяет сокет GDB с помощью метода `TryGetChar()`. Чтобы избежать блокирующего чтения из сокета используется функция `select()` с таймаутом.

3.3. Тестирование

Для тестирования была использована библиотека Google C++ Testing Framework, а также функции из интерфейса `pthread`.

Тестирование вспомогательных функций проверяет корректность результата функций в зависимости от входных данных, а также выбрасывание функцией ошибки при некорректном аргументе.

Поскольку большинство методов сервера получают и отправляют данные через TCP соединение, был создан фиктивный клиент, который позволяет отправлять команды серверу и получать ответ от сервера. Тестирование этих методов происходит с помощью fixtures: в инициализирующем методе `SetUp()` в отдельном потоке запускается сервер и устанавливается соединение с фиктивным клиентом. За освобождение ресурсов отвечает метод `TearDown()`: соединение прерывается (отправляется kill-request) и поток сервера завершает работу.

4. Заключение

В ходе весенней учебной практики были выполнены следующие задачи:

1. проведен обзор существующих решений;
2. реализована недостающая часть функциональности GDB;
3. обеспечена возможность прерывания работы эмулятора.

Работа велась в репозитории компании и содержит разработки компании. Открытая часть кода была перенесена в отдельный репозиторий, доступный по ссылке: https://github.com/Chernobrovkina-Yulya/GDB_server

Список литературы

[1] Bruce Bobby R. — Debugging Simulated Code, 2023. — URL: https://www.gem5.org/documentation/general_docs/debugging_and_testing/debugging/debugging_simulated_code (дата обращения: 13 декабря 2023 г.).

[2] Debugging Guest Applications with QEMU and GDB. — URL: https://xilinx-wiki.atlassian.net/wiki/spaces/A/pages/821624963/Debugging+Guest+Applications+with+QEMU+and+GDB#DebuggingGuestApplicationswithQEMUandGDB-GDBCommands (дата обращения: 9 апреля 2024 г.).

[3] Embecosm. — URL: https://www.embecosm.com/ (дата обращения: 9 апреля 2024 г.).

[4] Features/gdbstub. — URL: https://www.ubuntupit.com/best-linux-debuggers-for-modern-software-engineers/ (дата обращения: 8 апреля 2024 г.).

[5] Features/gdbstub. — URL: https://wiki.qemu.org/Features/gdbstub (дата обращения: 13 декабря 2023 г.).

[6] QEMU. — URL: https://ru.wikipedia.org/wiki/QEMU (дата обращения: 9 апреля 2024 г.).

[7] Zephyr. — URL: https://docs.zephyrproject.org/latest/introduction/index.html (дата обращения: 9 апреля 2024 г.).

[8] Zephyr GDB stub. — URL: https://docs.zephyrproject.org/latest/services/debugging/gdbstub.html (дата обращения: 9 апреля 2024 г.).