Санкт-Петербургский государственный университет

Программная инженерия

Поляков Александр Романович

Разработка системы для отладки ядра операционной системы

Выпускная квалификационная работа

Научный руководитель:
ст. преп. Я. А. Кириленко

Рецензент:
старший менеджер группы программных инструментов разработки
ООО "Синопсис СПб" А. С. Колесов

Санкт-Петербург
2019

SAINT-PETERSBURG STATE UNIVERSITY

Software Engineering

Alexander Polyakov

Development of a system to debug operating system kernel

Graduation Thesis

Scientific supervisor:
senior lecturer Iakov Kirilenko

Reviewer:
manager II at Synopsys SPb Anton Kolesov

Saint Petersburg
2019

Оглавление

Введение **4**

1. Постановка задачи **6**

2. Обзор **7**

2.1. Обзор предметной области 7

2.1.1. Отладка прикладных программ 7

2.1.2. Отладка приложений, исполняющихся "на железе" 7

2.1.3. Особенности отладки операционной системы . . . 8

2.2. Обзор существующих решений 9

2.2.1. Отладочный вывод 9

2.2.2. Модификация исходного кода ядра для добавления возможности трассировки его состояния . . . 9

2.2.3. Использование отладчика 10

2.2.4. Выводы 11

3. Архитектура и особенности реализации **13**

3.1. Обёртка над типами данных операционной системы . . . 14

3.2. Модуль кастомизации 15

3.3. Модуль целевой архитектуры 16

3.4. Модуль операционной системы 16

3.5. Модуль взаимодействия с отладчиком 17

3.6. Модуль трассировки и журналирования 17

4. Апробация **19**

5. Заключение **21**

Список литературы **22**

Введение

Дать точное определение операционной системы (ОС) [19] довольно трудно, все зависит от конкретной ОС, её архитектуры и задач. Можно сказать, что *ОС* — это программное обеспечение (ПО), предоставляющее прикладным программистам (и прикладным программам) вполне понятный абстрактный набор ресурсов, который можно считать API (Application Programming Interface) операционной системы, взамен неупорядоченного набора аппаратного обеспечения и управляет этими ресурсами. В современном мире ОС используются повсеместно: смартфоны, встраиваемые системы, настольные компьютеры. Практически каждый человек так или иначе взаимодействует с ними. Столь широкое распространение ОС делает задачу обеспечения качества необычайно важной в их процессе разработки. Данная задача появилась вместе с самими ОС, и на данный момент существует целое множество подходов к её решению [9]: ведение документации, тестирование, использование различных решателей (SMT Solvers [3], Horn Clause Solvers [8]) и другие.

Ядро является наиболее критичной частью ОС. В нем сосредоточена базовая функциональность, оно имеет доступ к любым ресурсами ОС и приложений, запускаемых на ней. Эти обстоятельства делают ядро особенно чувствительным к наличию уязвимостей в его программном коде, объем которого сильно варьируется: от нескольких тысяч строк в операционных системах реального времени, например, FreeRTOS, до десятков миллионов в ядрах многопользовательских систем, таких как GNU/Linux. На практике, такое большое и сложное программное обеспечение не обходится без ошибок, что подтверждает актуальность данной проблемы.

Одним из подходов к обеспечению качества ПО является *отладка* — процесс локализации и устранения ошибок и уязвимостей в исходном коде программы. Основными технологиями отладки являются использование отладочного вывода, трассировка (журналирование), использование отладчиков [5].

Современные ОС в большинстве своем оперируют схожими понятия-

ями: процесс, поток, смена контекста, состояние регистров, файловый дескриптор. Этот факт является основанием предположения о возможности создания универсального решения для отладки целого множества ОС, запускаемых на различных целевых архитектурах. Можно выделить два основных требования, которым должна удовлетворять такая система:

- расширяемость — система должна иметь возможность добавления поддержки новых ОС и целевых архитектур;
- лёгкость интегрирования в процесс разработки конкретной ОС.

Наличие такого решения позволит сократить время разработки систем отладки для новых ОС, использующих вышеперечисленные общие понятия, а также собрать в одном месте лучшие практики существующих подходов и инструментов.

1. Постановка задачи

Целью данной работы является разработка расширяемой системы для отладки ядра операционной системы.

Для достижения данной цели были поставлены следующие задачи.

- Выполнить обзор существующих решений.
- Разработать архитектуру системы.
- Реализовать требуемую систему.
- Провести апробацию реализованной системы:
  - добавить поддержку выбранной целевой архитектуры;
  - добавить поддержку выбранной ОС;
  - провести тестовую отладочную сессию.

2. Обзор

В данной секции описываются особенности предметной области, проводится обзор существующих решений.

2.1. Обзор предметной области

Широкое распространение и критичность операционных систем привели к появлению множества различных подходов к их отладке [5]. Основными являются:

- использование отладочного вывода;
- использование отладчика;
- модификация исходного кода ядра для добавления возможности трассировки его состояния.

Отладочный вывод и модификация исходного кода не нуждаются в дополнительном раскрытии, в то же время использование отладчика имеет ряд особенностей в случае отладки ОС.

2.1.1. Отладка прикладных программ

Типичным сценарием использования отладчика является отладка прикладных приложений, запускаемых внутри какой-либо ОС. В таком случае отладчик может использовать её API для получения контроля над отлаживаемым процессом. В качестве примера можно рассмотреть системный вызов UNIX-подобных операционных систем — ptrace. Он позволяет отладчику подключаться к целевому процессу, управлять его состоянием (изменять содержимое памяти, пошагово исполнять код и многое другое).

2.1.2. Отладка приложений, исполняющихся ”на железе”

В условиях отсутствия ОС, отладчику необходим механизм, способный заменить её API.

Печатные платы могут иметь интерфейс, обычно называемый Debug Port или Test Access Port, позволяющий внешнему устройству напрямую управлять состоянием процессора. Промышленный стандарт IEEE Standard Test Access Port and Boundary-Scan Architecture [15], иногда называемый JTAG по названию рабочей группы, предложившей его, определяет использование вышеупомянутого интерфейса.

Необходимую для отладчика функциональность реализуют адаптеры отладки. Они могут быть оформлены как отдельные модули, которые называют аппаратныными отладчиками, либо как сочетание программного отладчика, например, openocd [16], и TTL-микросхемы (TTL — transistor-transistor logic), транслирующей команды отладчика в последовательность JTAG-комманд.

Особенностью такого способа отладки является то, что количество потоков, видимых отладчиком, ограничено сверху количеством ядер процессора.

2.1.3. Особенности отладки операционной системы

Отладка ОС в отладчике является пограничным случаем: с одной стороны, ОС есть приложение, исполняющееся "на железе", а значит, что для её отладки справедливы утверждения секции 2.1.2, с другой стороны, полезная информация, такая как списки потоков (коих может быть сильно больше количества ядер процессора) и открытых файловых дескрипторов, сосредоточена в ядре и доступна только ему и клиентам, использующим определённый API изнутри ОС. Из этого можно сделать вывод о том, что отладчик ОС, находясь вне ядра, должен иметь представление об её внутреннем устройстве, уметь находить участки памяти, соответствующие интересующим пользователя запросам, анализировать и представлять их в человеко-читаемом виде.

2.2. Обзор существующих решений

2.2.1. Отладочный вывод

Данный подход является наиболее простым методом отладки, что является его основным достоинством. Недостатки такого решения:

- некоторые операционные системы могут не иметь средств печати;
- при добавлении/удалении команды отладочного вывода необходимо заново компилировать исходный код ядра;
- снижение производительности системы;
- ограниченность подхода как способа отладки:
  - разработчик может наблюдать лишь за тем, что было добавлено в отладочный вывод;
  - отсутствие возможности воздействия на отлаживаемое приложение (запись в память по указанному адресу).

Примером решения данного типа является PRINTK [13] — функция отладочного вывода ядра LINUX.

2.2.2. Модификация исходного кода ядра для добавления возможности трассировки его состояния

Некоторые операционные системы имеют встроенный в ядро модуль трассировки, который используется различными утилитами. Примерами таких решений являются FTRACE [14] — фреймворк для трассировки различных параметров ядра LINUX, KTRACE [6] — утилита операционных систем BSD UNIX и MACOS для трассировки взаимодействия ядра и пользовательских приложений.

Решения такого типа имеют схожие преимущества: доступность ресурсов ядра для модуля трассировки; и недостатки:

- ограниченность подхода как способа отладки:

  – разработчик может наблюдать только за тем, что было заранее включено в трассировку;
  – отсутствие возможности воздействия на отлаживаемое приложение (запись в память по указанному адресу);

- снижение производительности системы в общем случае (исключение — LTTNG [4]).

2.2.3. Использование отладчика

Отладчики имеют ряд преимуществ по сравнению с трассировщиками и отладочным выводом.

- Отладчик позволяет производить над отлаживаемой ОС все действия, привычные для отладки пользовательских приложений, например, установка точек останова, пошаговое исполнение, запись в память.
- ОС в большинстве своем оперируют схожими понятиями, что позволяет одному отладчику поддерживать множество операционных систем.

Сравним существующие решения по базовым критериям: открытость исходного кода, расширяемость множества поддерживаемых операционных систем и целевых архитектур.

	Открытость кода	**Расширяемость мн-ва ОС**	**Расширяемость мн-ва архитектур**
ARM DS-5	—	+	—
WinDbg	—	—	+
GDB	+	+/—	+
pyocd	+	+	—
XNU lldbmacros	+	—	+

Таблица 1: Базовые параметры существующих решений

Таблица 1 показывает, что на данный момент не существует отладчика, предоставляющего универсальное решение рассматриваемой задачи.

Дальшейнее рассмотрение ARM DS-5 [1] и WinDbg [20] не имеет смысла ввиду их проприетарности.

Отладчик GDB [7] ближе всех подобрался к решению исходной проблемы, однако, на сегодняшний день, полноценно поддерживает только ядро Linux, остальным ОС доступна лишь малая часть функциональности. Добавление поддержки новой ОС требует внесения изменений в исходный код отладчика, что порождает нежелательные зависимости между проектами.

Наиболее перспективным и продуманным решением автор считает xnu lldbmacros [21]. Оно разработано на базе отладчика LLDB [10], переиспользует его функциональность, поддерживает множество целевых архитектур, имеет высокий уровень абстракции. Проект спроектирован для отладки XNU — ядра операционной системы Darwin.

2.2.4. Выводы

В результате анализа существующих решений в области отладки ядра ОС было принято решение использовать отладчик в качестве основы реализуемой системы. Так, базовым отладчиком для реализуемой системы был выбран проект *LLDB* — высокопроизводительный консольный отладчик, широко использующий существующие библиотеки проекта LLVM [12]. Среди его преимуществ можно отметить проработанный API [11], доступный из языков C++ и Python, и возможность использования пользовательсих плагинов. К системе были выдвинуты следующие требования:

- переиспользование функциональности отладчика;
- исключение зависимостей между отладчиком и ОС;
- высокий уровень абстракции;
- расширяемость множества ОС и целевых архитектур;

- лёгкость интегрирования в процесс разработки ОС.

3. Архитектура и особенности реализации

С учетом анализа существующих решений и выявленных требований, была разработана архитектура системы, представленная на рис. 1.

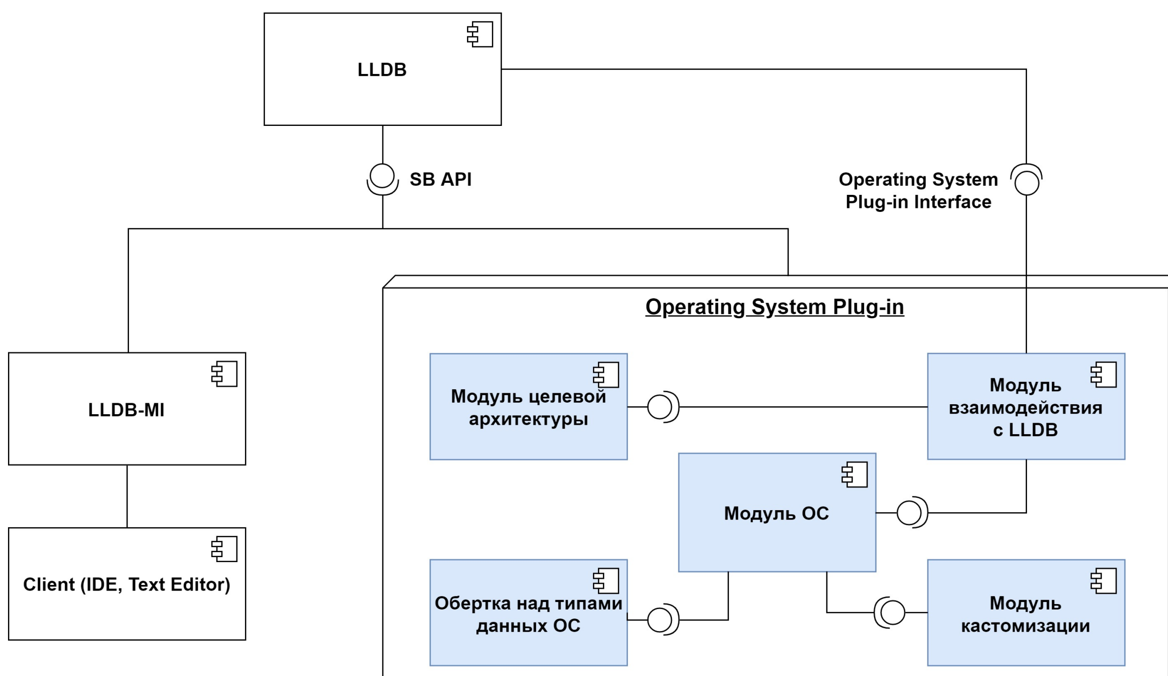

Рис. 1: Архитектура системы. Разработанные компоненты выделены цветом.

Система представляет из себя плагин к отладчику LLDB. Такой подход позволяет исключить зависимости между проектами отладчика и ОС, поскольку плагин, являясь прослойкой между ними, может взять на себя обязанности по их разрешению, а также переиспользовать всю функциональность LLDB, используя SB (Scripting Bridge) API [11].

Для реалзиации системы был выбран язык программирования Python. Такое решение облегчает сопровождение проекта, а также позволяет хранить модули системы в различных репозиториях, например, модуль ОС может храниться в репозитории самой ОС, развиваясь параллельно с ней.

Принимая во внимание модульность архитектуры, система обладает следующими возможностями расширения:

- горизонтальное развитие:
  - добавление поддержки новых целевых архитектур;
  - добавление поддержки новых ОС;
- вертикальное развитие:
  - поддержка новой функциональности, добавляющейся со временем в отладчик LLDB;
  - добавление поддержки новых сущностей, используемых в разработке ОС, например мьютексов.

3.1. Обёртка над типами данных операционной системы

Отладчик LLDB позволяет пользователю находить данные, соответствующие элементам таблицы символов [2], и проводить различные манипуляции над ними с помощью класса SBValue, доступного в рамках SB API. Однако, использование этого класса на практике сопровождается длинными последовательностями вызовов API, что крайне затрудняет читабельность исходного кода. Данный модуль решает эту проблему, а именно позволяет пользователям работать с их данными как с объектами языка Python. Листинг 1 демонстрирует пример типа данных, описанного на языке C, а также способы взаимодействия с ним на языке Python (без данного модуля и с ним). Среди особенностей модуля стоит отметить следующие:

- поддержка стандартных методов объектов языка Python;
- переиспользование предоставленной компилятором информации о типах данных для корректной работы с ними в языке Python;
- разыменование указателей;
- обращение к полям объекта;

- обращение к элементу массива по индексу;
- итерирование по объекту.

```
# struct Point {
#    int x;
#    int y;
# };
# Global variable defined somewhere in the debugging program
point = FindGlobalVariable(target, 'point')
# Without module
a = point.GetChildMemberWithName('x').GetValueAsSigned() + 1
# With module
a = point.x + 1
```

Листинг 1: Взаимодействие с пользовательскими данными

Модуль работает со всеми языками программирования, поддерживаемыми отладчиком LLDB: C, C++, Objective-C, Objective-C++.

3.2. Модуль кастомизации

Крайне важной задачей является учитывание особенностей каждой ОС, поскольку только так можно создать качественный инструмент отладки, которым смогут пользоваться разработчики различных ОС.

Модуль кастомизации предлагает пользователям несколько декораторов функций языка Python, из которых мы рассмотрим два основных: 1. @lldb_type_summary; 2. @lldb_command. Первый декоратор в качестве аргумента принимает список типов, для которых декорируемая функция станет методом их представления в виде строки, что может быть полезно для, например, выделения особенно важных полей некоторых структур данных. Второй декоратор предназначен для добавления новой консольной команды отладчика, реализацией которой является декорируемая функция. Этот декоратор принимает два аргумента — название команды и её опции в определенном формате [17], и позволяет пользователям значительно расширить базовую функциональность плагина для конкретной ОС или целевой архитектуры.

3.3. Модуль целевой архитектуры

Благодаря значительному переиспользованию функциональности отладчика LLDB, задачи данного модуля довольно минималистичны:

- описание регистров целевой архитектуры;
- упаковка значений регистров в последовательность байтов для дальнейшей их обработки отладчиком.

Теоретически, можно избавиться от этого компонента плагина в будущем, поскольку данная функциональность уже присутствует в LLDB, однако, на сегодняшний день она недоступна из публичного API.

3.4. Модуль операционной системы

Отладчик LLDB имеет свое собственное внутреннее представление потока отлаживаемой программы, а также механизм, позволяющий динамически создавать такие потоки. Этот механизм используется данной системой для отображения потоков ОС во внутренние потоки отладчика, что позволяет переиспользовать всю его функциональность, предназначенную для манипуляции потоками отлаживаемого приложения.

Данный модуль имеет две основные задачи:

1. определение интерфейса взаимодействия плагина с ОС;
2. определение сущностей, которые плагин способен отобразить в некое внутреннее представление отладчика.

Для решения первой задачи был разработан интерфейс, посредством которого плагин может получать необходимую информацию об ОС, например список потоков. Для решения второй задачи был разработан абстрактный класс потока, роль которого заключается в единообразном доступе к информации о потоке: ID, адрес, имя, значения регистров. Таким образом, при добавлении новой ОС в плагин, разработчику необходимо реализовать вышеописанные интерфейс и абстрактный класс, что не представляет труда, учитывая наличие в системе модуля 3.1.

На данный момент, отладчик LLDB поддерживает только абстракцию потока, однако, в дальнейшем возможно добавление и других сущностей. Также стоит отметить, что, при необходимости, пользователь может легко расширить функциональность плагина под свои потребности, использовав модуль кастомизации 3.2.

3.5. Модуль взаимодействия с отладчиком

Данный модуль является прослойкой между отладчиком и частью плагина, специфичной для конкретной ОС. Его основными задачами являются:

- инциализация плагина;
- инкапсуляция от пользователя процесса взаимодействия с отладчиком;
- предобработка информации, необходимой отладчику.

Наличие такой прослойки в системе позволяет создавать различные механизмы, способные положительно сказаться на скорости работы плагина, например кэширование данных, а также на объеме переиспользованного кода: когда отладчик запрашивает у плагина информацию о значении регистров определенного потока, плагин может выполнить проверку и узнать, является ли данный поток активным (то есть находившимся на исполнении каким-либо ядром процессора в момент поступления запроса) или нет. Если поток окажется активным, плагин может получить значениях всех его регистров напрямую "из железа", воспользовавшись API отладчика.

3.6. Модуль трассировки и журналирования

В связи с особенностями архитектуры, использовать отладчик для локализации и устранения ошибок в системе не представляется возможным. Для решения этой проблемы в систему был добавлен модуль,

предоставляющий пользователям возможности журналирования, а также специальный декоратор для функций — @traced. Этот декоратор сохраняет в лог историю вызовов функций вместе с переданными им параметрами. Для языка Python уже существуют сторонние библиотеки, предлагающие схожие возможности трассировки, однако, добавление в проект зависимостей от таких библиотек было крайне нежелательным, что и послужило основным поводом для реализации данной функциональности в рамках системы.

4. Апробация

Для апробации системы необходимо добавить в нее поддержку выбранных целевой архитектуры и ОС, а также провести тестовую отладочную сессию. Целевой архитектурой была выбрана архитектура ARC (Argonaut RISC Core) [18], разрабатываемая компанией Synopsys. Такой выбор обусловлен знакомством автора с данной архитектурой. В качестве тестовой ОС была выбрана операционная система реального времени Zephyr [22], обладающая открытым исходным кодом, небольшим объемом кодовой базы и простотой внутреннего устройства, что позволило автору работы в короткие сроки изучить ядро данной ОС. Так, в системе были реализованы модуль целевой архитектуры ARC и модуль ОС Zephyr.

Для проведения тестовой отладочной сессии была зафиксирована следующая конфигурация.

- Отлаживаемая программа — многопоточное (три потока) приложение из репозитория ОС Zephyr, демонстрирующее возможности синхронизации потоков.
- Версия отладчика LLDB, поддерживающая архитектуру ARC.
- Одноядерный режим ARC-процессора.

На листинге 2 показан фрагмент отладочной сессии, проведённой без использования реализованного плагина. Поток, видимый отладчиком и доступный пользователю, является представлением единственного ядра процессора. Листинг 3 демонстрирует фрагмент отладочной сессии с использованием плагина. В этом случае отладчику и пользователю становятся доступны все три потока отлаживаемого приложения, соответствующие внутренним структурам данных ОС Zephyr, над которыми пользователь может проводить различные манипуляции, например запрашивать стек вызовов функций конкретного потока. Таким образом, реализованная в рамках данной работы система положительно сказывается на информативности отладочной сессии, а значит и на скорости локализации и устранения ошибок в программном коде ОС.

```
(lldb) thread info all
thread #1: tid = 0x0001, 0x000004bc zephyr.elf`
helloLoop(my_name="threadB", my_sem=0x800022c4, other_sem=0x800022b4)
at main.c:46, stop reason = breakpoint 1.1
```

Листинг 2: Фрагмент отладочной сессии без использования плагина

```
(lldb) thread info all
thread #1: tid = 0x80000068, 0x000004bc zephyr.elf`
helloLoop(my_name="threadB", my_sem=0x800022c4, other_sem=0x800022b4)
at main.c:46, name = 'thread_b'

thread #2: tid = 0x80000000, 0x000004b2 zephyr.elf`
helloLoop [inlined] k_current_get at kernel.h:29,
name = 'thread_a'

thread #3: tid = 0x800000e8, 0x00000c10 zephyr.elf`
k_cpu_idle + 16, name = 'idle'
```

Листинг 3: Фрагмент отладочной сессии с использованием плагина

Результаты данной работы были внедрены в продукты компании Synopsys, разработчиками которой была отмечена гибкость и расширяемость системы.

5. Заключение

В рамках выпускной квалификационной работы были достигнуты следующие результаты.

- Выполнен обзор существующих решений в области отладки операционных систем:
  - использование отладочного вывода;
  - модификация исходного кода ядра с целью добавления возможности трассировки его состояния;
  - использование отладчика.
- Разработана архитектура системы, основанная на принципах простоты расширяемости и минимизации зависимостей между операционной системой и отладчиком.
- Реализована требуемая система, включая следующие компоненты:
  - обёртка над типами данных операционной системы;
  - модуль кастомизации;
  - модуль целевой архитектуры;
  - модуль операционной системы;
  - модуль взаимодействия с отладчиком;
  - модуль трассировки и журналирования.
- Проведена апробация системы:
  - добавлена поддержка целевой архитектуры ARC;
  - добавлена поддержка операционной системы Zephyr;
  - проведены тестовые отладочные сессии, демонстрирующие работоспособность реализованной системы.
- Результаты работы внедрены в разработки компании Synopsys.

Список литературы

[1] ARM DS-5 OS awareness solution. — URL: https://developer.arm.com/products/software-development-tools/ds-5-development-studio (online; accessed: 21.04.2019).

[2] Aho A. V., Sethi R., Ullman J. D. Compilers principles, techniques, and tools. — Reading, MA : Addison-Wesley, 1986.

[3] Bjørner Nikolaj, de Moura Leonardo. Applications of SMT solvers to program verification. — 2014.

[4] Desnoyers Mathieu. Low-Impact Operating System Tracing. — 2009. — 01.

[5] Examensarbete VT 2013. Debugging methods applied on networking operating systems. — 2013.

[6] FreeBSD: ktrace. — URL: https://www.freebsd.org/cgi/man.cgi?query=ktrace&sektion=1&manpath=FreeBSD+4.7-RELEASE (online; accessed: 12.05.2019).

[7] GDB Homepage. — URL: https://www.gnu.org/software/gdb/ (online; accessed: 21.04.2019).

[8] Horn Clause Solvers for Program Verification / Nikolaj Bjørner, Arie Gurfinkel, Ken McMillan, Andrey Rybalchenko // Fields of Logic and Computation II: Essays Dedicated to Yuri Gurevich on the Occasion of His 75th Birthday / Ed. by Lev D. Beklemishev, Andreas Blass, Nachum Dershowitz et al. — Cham : Springer International Publishing, 2015. — P. 24–51. — ISBN: 978-3-319-23534-9. — URL: https://doi.org/10.1007/978-3-319-23534-9_2.

[9] Klein Gerwin. Operating system verification—An overview // Sadhana. — 2009. — 02. — Vol. 34. — P. 27–69.

[10] LLDB Homepage. — URL: https://lldb.llvm.org/ (online; accessed: 21.04.2019).

[11] LLDB SB API. — URL: https://lldb.llvm.org/python_reference/index.html (online; accessed: 19.05.2019).

[12] LLVM Homepage. — URL: https://llvm.org/ (online; accessed: 04.05.2019).

[13] Linux kernel API: printk. — URL: https://www.kernel.org/doc/html/v4.17/driver-api/basics.html?highlight=printk#c.printk (online; accessed: 12.05.2019).

[14] Linux kernel: ftrace. — URL: https://www.kernel.org/doc/Documentation/trace/ftrace.txt (online; accessed: 12.05.2019).

[15] Maunder Colin M., Tulloss Rodham. The Test Access Port and Boundary-Scan Architecture. — Los Alamitos, CA, USA : IEEE Computer Society Press, 1990. — ISBN: 0818690704.

[16] OpenOCD Homepage. — URL: http://openocd.org/ (online; accessed: 30.04.2019).

[17] Python getopt module's documentation. — URL: https://docs.python.org/3.7/library/getopt.html (online; accessed: 20.05.2019).

[18] Synopsys' ARC processors: general. — URL: https://www.synopsys.com/designware-ip/processor-solutions/arc-processors/general.html (online; accessed: 20.05.2019).

[19] Tanenbaum Andrew S., Bos Herbert. Modern Operating Systems. — 4th edition. — Upper Saddle River, NJ, USA : Prentice Hall Press, 2014. — ISBN: 013359162X, 9780133591620.

[20] Windows OS awareness solution. — URL: https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/index (online; accessed: 21.04.2019).

[21] XNU OS awareness solution. — URL: https://github.com/apple/darwin-xnu/tree/master/tools/lldbmacros (online; accessed: 21.04.2019).

[22] Zephyr Homepage. — URL: https://www.zephyrproject.org/ (online; accessed: 20.05.2019).