Правительство Российской Федерации
Федеральное государственное бюджетное образовательное учреждение
высшего профессионального образования
«Санкт-Петербургский государственный университет»

Кафедра Системного Программирования

Шашкова Елизавета Михайловна

Визуализация и поиск дефектов в многозадачных программах на языке Python в интегрированной среде разработки PyCharm

Дипломная работа

Допущена к защите.
Зав. кафедрой:
д. ф.-м. н., профессор Терехов А. Н.

Научный руководитель:
к. ф.-м. н. Булычев Д. Ю.

Рецензент:
Власовских А. С.

Санкт-Петербург
2015

SAINT-PETERSBURG STATE UNIVERSITY

Department of Software Engineering

Elizaveta Shashkova

Visualisation and searching for defects in multitask Python programs in PyCharm integrated development environment

Graduation Thesis

Admitted for defence.
Head of the chair:
professor Andrey Terekhov

Scientific supervisor:
Dmitri Boulytchev

Reviewer:
Andrey Vlasovskikh

Saint-Petersburg
2015

Оглавление

Введение

В настоящее время многозадачное программирование активно применяется в индустрии. Разработка и понимание многозадачных программ является сложной задачей и поэтому сопровождается большим количеством ошибок.

Язык Python[1] является высокоуровневым динамически типизированным языком программирования общего назначения, поддерживающим многопоточные вычисления. Python предоставляет интерфейс к работе с потоками операционной системы, который представлен в модуле `threading`[2]. Потоки в Python используются разработчиками для реализации пользовательских интерфейсов, неблокирующего ввода/вывода и одновременного исполнения задач.

Начиная с версии 3.4 в состав стандартной библиотеки языка Python вошёл пакет `asyncio`[3]. Этот пакет позволяет создавать многозадачные программы, параллельные по операциям ввода/вывода, с использованием одного стандартного потока Python. Он реализует собственный цикл обработки событий и был создан для стандартизации и налаживания взаимодействия между существующими фреймворками (такими как Twisted[4], Tornado[5] или ZeroMQ[6]).

Далее в этой работе под многозадачными программами будем понимать программы, использующие в процессе выполнения несколько потоков или несколько задач из пакета `asyncio`.

Продукт PyCharm IDE[7] является интегрированной средой разработки для языка Python. PyCharm разрабатывается на языке программирования Java на основе платформы с открытым кодом IntelliJ[8], которая создана компанией JetBrains. Среда разработки PyCharm помогает при написании, исполнении и отладке кода на языке Python. В PyCharm имеется отладчик, который позволяет в том числе отлаживать многопоточные программы.

Часто программисты испытывают трудности с пониманием хода выполнения многозадачных программ. Таким образом, оказывается вос-

[1] https://www.python.org/
[2] https://docs.python.org/3.4/library/threading.html
[3] https://docs.python.org/3/library/asyncio.html
[4] http://twistedmatrix.com/
[5] http://www.tornadoweb.org
[6] http://zeromq.org/bindings:python
[7] https://www.jetbrains.com/pycharm/
[8] http://www.jetbrains.org/display/IJOS/Home

требованным инструмент, который будет визуализировать исполнение многозадачных программ, сопоставлять события межзадачного взаимодействия с исходным кодом программы и анализировать свойства выполняемых трасс. Подобный инструмент необходим для среды разработки PyCharm для того, чтобы выполнять анализ программ, запущенных в режиме отладки.

В связи с этим в рамках данной дипломной работы сформулированы следующие задачи:

- Реализовать протоколирование событий в многозадачной программе на языке Python, запущенной в режиме отладки в PyCharm.
- Реализовать расширение к PyCharm (Java/Python), обладающее следующим набором возможностей:
  - визуализация исполнения многозадачной программы;
  - осуществление навигации по исходному коду и отображение стека вызовов в момент наступления события.
- На основе полученных журналов (logs) реализовать поиск следующих дефектов в многозадачной программе:
  - неиспользованных объектов синхронизации;
  - задач, не дождавшихся объединения;
  - взаимных блокировок (deadlocks).
- Провести тестирование разработанного инструмента.
- Провести анализ полученных результатов.

1. Анализ предметной области

В разделе проводится обзор инструментов для анализирования многозадачных программ. Описываются особенности языка Python с точки зрения многозадачности, а также устройство среды разработки PyCharm и устройство отладчика в её составе.

1.1. Инструменты для анализа выполнения многозадачных программ

С момента возникновения и распространения многозадачного программирования создавалось большое количество систем с целью проведения анализа многозадачных программ. Проведение такого анализа возможно как на основе статической, так и на основе динамической информации. Статический анализ работает с кодом программы без её запуска и анализирует все возможные варианты выполнения. При динамическом анализе проводится работа с информацией, полученной в результате запуска программы.

Основное назначение систем, анализирующих потоки с помощью динамического анализа, можно разделить на две основные группы:

1. Проведение оптимизации и поиск ошибок, связанных с многопоточным программированием.

2. Упрощение понимания архитектуры программы.

К первой группе можно отнести целый класс инструментов, называемых профилировщиками (profilers). Профилировщики позволяют исследовать поведение программы на основе информации, полученной во время исполнения. Обычно такой анализ проводится с целью нахождения участков программы, требующих проведения оптимизациии. Корректная работа с потоками может значительно увеличить производительность программы, поэтому анализ потоков, использующихся в программе, является неизменной составляющей многих профилировщиков. Однако ни в стандартных библиотеках[9] для профилирования языка Python cProfile, profile, hotspot, ни в одном из известных сторон-

[9]https://docs.python.org/3.4/library/profile.html

них инструментов, таких как kcachegrind[10], RunSnakeRun[11], gprof2dot[12], информация о потоках не анализируется.

Также к первой группе можно отнести инструменты, работающие со снимками потоков (thread dumps). Например, для языка Java существуют такие инструменты как Thread Dump Analyzer[13] и IBM Thread and Monitor Dump Analyzer[14]. Они позволяют анализировать состояния потоков и объектов синхронизации в программе в момент снятия снимка. В обоих представленных инструментах имеется возможность проведения сравнения интересующих объектов, поиск взаимных блокировок и конфликтов доступа к ресурсам. В языке Python схожей функциональности можно добиться только с использованием стандартной функции `sys._current_frames`, с помощью которой можно сделать снимок текущих потоков. Инструментов, способных проанализировать полученный снимок и наглядно представить информацию из него, для языка Python не было обнаружено.

Требующийся в рамках данной работы инструмент можно отнести ко второй группе систем. Отчасти он помогает проводить оптимизацию и искать ошибки в многопоточных программах, однако его основной целью является предоставление информации об устройстве программы с точки зрения потоков.

Одним из наиболее известных инструментов из первой группы является система визуализации PARADE [8], целью создания которой было облегчение понимания программы и оптимизация производительности. Недостатком данной системы является то, она визуализирует работу потоков по отдельности, не отображая их на общей схеме, что затрудняет её восприятие. Позже на основе этой системы была разработана среда для визуализации потоков Pthreads с использованием среды POLKA [9], которая уже строила общую схему работы потоков в программе и соотносила объекты на схеме с исходным кодом программы. Исходя из опыта графической интерпретации параллельно исполняющихся программ авторы [1] реализовали собственную систему визуализации для языка OCCAM. В [4] авторы используют наиболее распространённые и привычные для программистов схемы и предлагают визуализировать многопоточные программы, основываясь на диаграмме последователь-

[10]http://kcachegrind.sourceforge.net/html/Home.html

[11]http://www.vrplumber.com/programming/runsnakerun/

[12]https://pypi.python.org/pypi/gprof2dot/

[13]https://java.net/projects/tda

[14]http://www.ibm.com/developerworks/java/jdk/tools/dumpanalyzer/

ности UML.

Подобные системы визуализации оказываются особенно востребованными в образовании. Инструментом, наиболее точно удовлетворяющим описанной функциональности, является ThreadMentor [5]. Он создавался для студентов с целью облегчения понимания поведения потоков и процессов синхронизации. Этот инструмент является отдельной программой, которая обменивается информацией с пользовательской программой, написанной на языке C/C++, посредством сообщений. ThreadMentor поддерживает создание, приостановку, объединение потоков и работу с примитивами синхронизации. Визуализация отображается в отдельном окне и позволяет приостанавливать исполнение программы, исследовать текущие состояния и историю работы потоков.

1.2. Платформа IntelliJ

Платформа IntelliJ является платформой с открытым исходным кодом, разрабатываемой компанией JetBrains. Данная платформа позволяет создавать интегрированные среды разработки и обладает набором компонент, который включает в себя виртуальную файловую систему, фреймворк (framework) с пользовательским интерфейсом, текстовый редактор, абстрактные синтаксические деревья, фреймворки для работы с кодом и отладчиком. На базе платформы IntelliJ построены несколько сред разработки, в том числе и PyCharm — интегрированная среда разработки для языка Python.

Платформа IntelliJ создана на языке Java версии 1.6 и имеет обширный пользовательский интерфейс, разрабатываемый с помощью программной библиотеки Swing, которая является частью стандартной библиотеки Java.

1.3. Язык Python

Язык Python является высокоуровневым динамически типизированным языком программирования общего назначения. В Python поддерживаются структурная, объектно-ориентированная, функциональная и императивная парадигмы программирования. В языке Python автоматическое управление памятью, полная интроспекция, механизм обработки исключений, поддержка многопоточных вычислений и высокоуровневые структуры данных. Отличительной чертой языка является

простота его синтаксиса, что улучшает читаемость кода и повышает производительность разработчика. Это, в свою очередь, приводит к широкому распространению и использованию языка.

Во многих приложениях существует потребность в использовании асинхронных операций ввода-вывода. Обычно для этих целей в Python используются потоки. Однако в действительности программы на Python, использующие несколько потоков, не выполняются параллельно. В канонической реализации языка Python, называемой CPython[15], существует мезханизм глобальной блокировки интерпретатора (GIL, Global Interpreter Lock), позволяющий только одному потоку выполнять байткод Python в единицу времени. Создатели языка утверждают, что это упрощает реализацию одновременного доступа к данным. Интерпретатор переключается между потоками, имитируя одновременное выполнение, хотя это и приводит к потере производительности [2]. Стоить отметить, что потенциально блокирующие или долго выполняющиеся операции, такие как операции ввода/вывода, обработка изображений и работа с многомерными массивами данных из пакета NumPy [16], производятся за пределами GIL. Несмотря на то, что операции ввода/вывода осуществляются за пределами GIL и не происходит замедления выполнения программы, использование потоков не удовлетворяет в полной мере потребности, имеющиеся при осуществлении таких операций.

Прежде всего, использование потоков влечёт накладные расходы в программе. Создание большого количества потоков на практике может значительно ухудшить производительность приложения и операционной системы в целом. Часто такая ситуация встречается при работе с сокетами. Несмотря на то, что операционная система накладывает ограничения на количество используемых сокетов, их количество может на 2 или 3 порядка превышать количество потоков, которые является целесообразным запускать в системе.

Ещё одним «бутылочным горлышком» в использовании потоков в данном контексте является процесс управления ими операционной системой. Поток может быть прерван операционной системой в любой момент, даже если он дожидается операций ввода/вывода. К тому же использование потоков требует обеспечения корректной работы с общими данными, использования объектов синхронизации и контроля за отсутствием состояния гонок в многопоточной программе.

[15]https://docs.python.org/2/glossary.htmlterm-cpython

[16]https://wiki.python.org/moin/NumPy

Проводить асинхронные операции ввода-вывода можно также используя системные вызовы select и poll, однако они не являются легко расширяемыми. В Python для этих целей можно использовать существующие фреймворки, такие как Twisted, Tornado или ZeroMQ. Некоторые библиотеки, написанные на языке Си, работающие с асинхронным вводом-выводом, имеют «обёртки» для языка Python, однако вынуждают к работе с определенным программным интерфейсом.

В результате для языка Python был разработан специальный фреймворк для асинхронной работы с операциями ввода-вывода, который, начиная с версии 3.4, есть в стандартной библиотеке в виде модуля `asyncio`. В основе этого фреймворка лежит понятие цикла обработки событий, который обеспечивает параллельное исполнение кода программы. Интерфейс из этого фреймворка используется для унификации и создания единого протокола для фреймворков, реализующих собственный цикл обработки событий [6].

Цикл обработки событий работает с задачами (tasks), попеременно переключаясь между ними и таким образом имитируя синхронное исполнение разных участков кода. По аналогии со стандартными потоками, в модуле `asyncio` были реализованы объекты синхронизации для контроля за доступом к данным, использование и не использование которых в свою очередь может привести к различным ошибкам, в том числе к взаимным блокировкам.

1.4. Отладчик для языка Python в PyCharm

Среда разработки PyCharm имеет в своём составе отладчик программ на языке Python. В CPython существует стандартный отладчик, называемый `pdb` и некоторые среды разработки используют его для отладки пользовательских программ. В среде PyCharm реализуется собственный отладчик.

Отладчик состоит из двух основных частей: трассировщик исполнения программы (написан на Python) и пользовательский интерфейс (написан на Java). Эти части взаимодействуют между собой через сокеты.

Трассировщик реализован на языке Python с использованием функции `sys.settrace(tracefunc)` из стандартной библиотеки[17]. Её единственным аргументом является функция трассировки `tracefunc`, об-

[17]https://docs.python.org/3.4/library/sys.html#sys.settrace

рабатывающая события в коде и принимающая, в свою очередь, три аргумента: frame, event и arg:

- `frame` — текущий фрейм стека;
- `event` — строка, описывающая событие, произошедшее в коде;
- `arg` — аргумент, зависящий от типа события.

Функция трассировки устанавливается каждый раз при заходе в очередной стековый фрейм и вызывается при выполнении очередной строки кода или при переходе между областями видимости.

В процессе трассировки данная функция обрабатывает полученную из фрейма информацию и через сокет отправляет её на сторону пользовательского интерфейса. Пользовательский интерфейс отладчика является частью интегрированной среды разработки и написан на Java. Полученная через сокет информация обрабатывается и представляется пользователю. В свою очередь, действия пользователя (установка точек останова, запуск/остановка сеанса отладки) сообщаются обратно на трассирующую сторону отладчика и меняют ход его работы.

Отладчик в среде PyCharm позволяет пользователю работать с многопоточными программами.

При запуске отладчика ему передаётся скрипт для запуска и дополнительные параметры. Сессия отладки начинается с создания экземпляра отладчика и запуска пользовательского кода с помощью стандартной функции `exec()`, которая осуществляет динамическое исполнение кода. Также в процессе запуска отладчика инициализируются вспомогательные компоненты: запускается поток для обработки команд, приходящих из интерфейса, и отправки сообщений о событиях, происходящих в отладчике; сохраняется информация о точках останова, поставленных пользователем в интерфейсе; устанавливается функция трассировки для текущего фрейма.

Далее при каждом заходе в новый стековый фрейм, если он подлежит отладке, а не является, например, одним из файлов самого отладчика, то создаётся экземпляр класса `PyDbFrame`. Этот класс осуществляет обработку событий, происходящих в программе, и содержит в себе основную логику работы отладчика.

В зависимости от текущей строки кода, выполняемой в отладчике, и команды, которую пользователь сообщил отладчику, определяется

дальнейшее поведение в текущем фрейме. Это может быть продолжение исполнения кода скрипта, а может быть остановка на текущей строке кода.

В случае остановки главный поток трассировщика отдельным сообщением передаёт интерфейсу информацию об осуществлении остановки и переходит в режим ожидания ответных команд от интерфейса, то есть от пользователя. Пользователь, в свою очередь, может возобновить выполнение кода, остановить отладочную сессию или выполнить одну из пошаговых отладочных команд (step commands). При каждом из этих действий соответствующее сообщение передаётся трассировщику и требуемое действие выполняется в коде.

В данной главе был проведён обзор существующих подходов и инструментов для графической интерпретации многозадачных программ. Затем с точки зрения многозадачности был рассмотрен язык Python и отладчик в среде разработки PyCharm.

2. Проектирование инструмента

Анализ, проведённый в предыдущей главе показал актуальность поставленной задачи. В данной главе будет описан процесс проектирования интерфейса инструмента, сформулированы требования и ограничения к его архитектуре. Также будут проанализированы дефекты, возникающие в многозадачных программах, и выбраны дефекты для автоматического поиска в разрабатываемом инструменте.

2.1. Пользовательский интерфейс

Визуализатор исполнения многозадачной программы должен представлять собой инструментальное окно в среде разработки PyCharm. Так как планируется построение визуализации двух типов многозадачности в языке Python, панель должна содержать две вкладки для каждого из типов.

Для пользователя представляют интерес следующие события в многозадачных программах:

- создание задач;
- объединение задач;
- создание объектов синхронизации;
- ожидание захвата объекта синхронизации;
- захват объекта синхронизации;
- освобождение объекта синхронизации.

Как видно из проведённого обзора существующих инструментов, почти всегда пользователю проще воспринимать параллельное исполнение задач в виде двумерного графа, на котором отображается состояние системы с течением времени. Структура такого графа схожа с диаграммой последовательности UML[18]. Её преимуществом является наглядность отображения состояния различных компонент с течением времени и последовательностей их взаимодействия.

В пользовательском интерфейсе последовательности схожих по структуре событий было решено представить в виде таблицы. Каждая строка

[18]http://www.ibm.com/developerworks/rational/library/3101.html

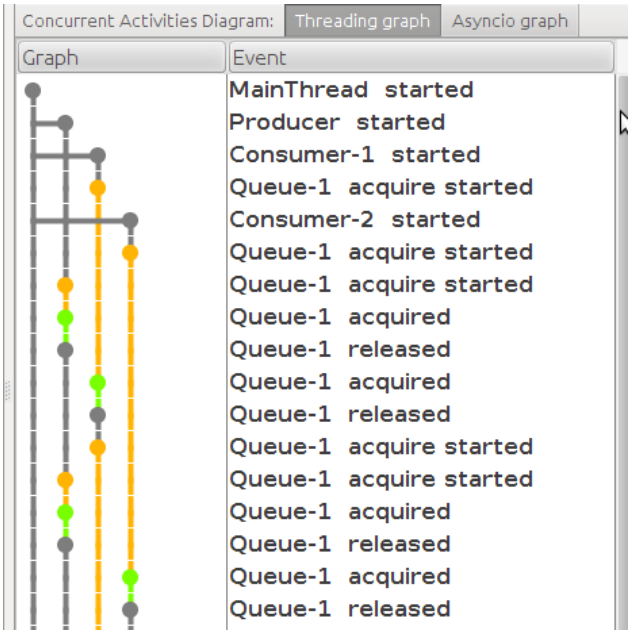

Рис. 1: Пример запуска программы.

соответствует событию, произошедшему в программе. Первый столбец содержит граф — визуализацию исполнения собранных событий. Помимо схематичного отображения события, требовалось представить более подробную информацию о нём, поэтому второй столбец в таблице содержит словесное описание события.

Помимо событий межзадачного взаимодействия в каждый момент времени работы программы важно понимать текущий статус выполняющихся задач, который было решено отображать различными цветами, чтобы не перегружать визуализацию избыточными надписями. Цвет задачи обозначает одно из четырёх его возможных состояний: не взаимодействует ни с какими объектами синхронизации, дожидается объекта синхронизации, работает с захваченным объектом синхронизации или находится в состоянии взаимной блокировки.

Реализация описанной визуализации представлена на Рис. 1. Из него видно, что изначально был создан главный поток MainThread, затем из этого потока были созданы потоки Producer, Consumer-1 и Consumer-2. Из графа видно, что после создания оба потока-получатели (Consumer-1 и Consumer-2) перешли в режим ожидания, когда в очереди появятся элементы. Затем с некоторой частотой поток Producer добавляет в неё элементы. Один из потоков-получателей выходит из режима ожидания, взаимодействует с очередью (достаёт из неё элементы), и затем снова переходит в режим ожидания.

Помимо факта наступления некоторого события очень важной яв-

ляется информация о месте в исходном коде, в котором это событие произошло. Реализовать такую функциональность позволяет интеграция с отладчиком, а среда разработки даёт возможность осуществлять навигация к месту в коде. Под навигацией здесь понимается действие, при котором файл открывается внутри среды разработки и выделяется строка в этом файле, в которой произошёл вызов.

Однако помимо вызова функции, связанной с работой задач, для пользователя представляет интерес состояние всего стека вызовов на момент наступления события. Поэтому при нажатии на строку таблицы появляется дополнительное окно, в котором отображается стек вызовов.

В процессе работы программы имеется необходимость проанализировать статистическую информацию о ходе выполнения программы: время работы каждого потока, суммарное время, проведённое каждым потоком в процессе ожидания объектов синхронизации и количество захватов объектов синхронизации. Данная статистика становится дополнительным инструментом в поиске дефектов многозадачной программы. Например, обнаружение задач, проводящих слишком много времени в ожидании объектов синхронизации, или задач, не использующих в ходе работы объекты синхронизации. По этой причине было принято решение расширить функциональность инструмента автоматическим сбором статистики, которая в виде таблицы показывается пользователю в отдельном окне.

2.2. Архитектура системы

Инструмент для визуализации и поиска дефектов должен реализовывать функциональность, описанную в предыдущем разделе. Помимо упомянутой функциональности к инструменту предъявляются следующие нефункциональные требования:

- Система должна поддерживать расширяемость по данным, то есть должна быть предусмотрена возможность добавления новых типов событий и новых параметров к обрабатываемым событиям. Например, добавление других объектов синхронизации, таких как семафоры и барьеры[19], или информации о данных, которые задачи добавляют в очередь или достают из неё.

[19]https://docs.python.org/3.4/library/threading.html

- Система должна поддерживать добавление других типов многозадачности, отличных от двух имеющихся. Например, процессная многозадачность, которая поддерживается в языке Python.
- Система должна поддерживать добавление других языков программирования.
- Сбор и обработка данных в системе должна осуществляться в инкрементально, по ходу выполнения программы.

В соответствии с постановкой задачи и с выводами, сделанными в процессе проектирования интерфейса, от инструмента требуется интеграция с существующей средой разработки. Платформа IntelliJ имеет гибкую систему точек расширения (extention points), позволяющую создавать встраиваемые модули.

Получить информацию о месте в исходном коде, в котором произошло событие межзадачного взаимодействия, можно из отладчика. Однако отладчик в PyCharm, в отличие от пользовательского интерфейса среды разработки, не предусматривает создания расширений, поэтому возникает необходимость частично изменить его код. Также встраивание в отладчик накладывает ограничения на архитектуру инструмента. Требуется разбиение архитектуры инструмента на две составные части: трассировщик, который перехватывает интересующие события на стороне отладчика, и инструментальное окно визуализации, которое является частью среды разработки.

Для построения визуализатора использован шаблон проектирования «Модель — Представление — Контроллер» (Model — View — Controller, MVC), при которой модель приложения, пользовательский интерфейс и взаимодействие с пользователем разделяются на три отдельных компонента таким образом, что модификация одного из них оказывает минимальное воздействие на остальные. На Рис. 2 изображена архитектура системы со встроенным визуализатором. Для реализации инструмента был реализован компонент `Logger` на стороне трассировщика и компоненты `Log manager`, `Graph manager` и `Window` на строне среды разработки. Компонента `Graph manager` соответствует компоненте Model в модели MVC, компонента `Window` — компоненте View, а компонента `Log manager` — компоненте Controller.

Использование шаблона MVC позволило реализовать инкрементальную обработку данных. В ходе работы программы сообщения последовательно приходят от трассировщика и накапливаются в компоненте

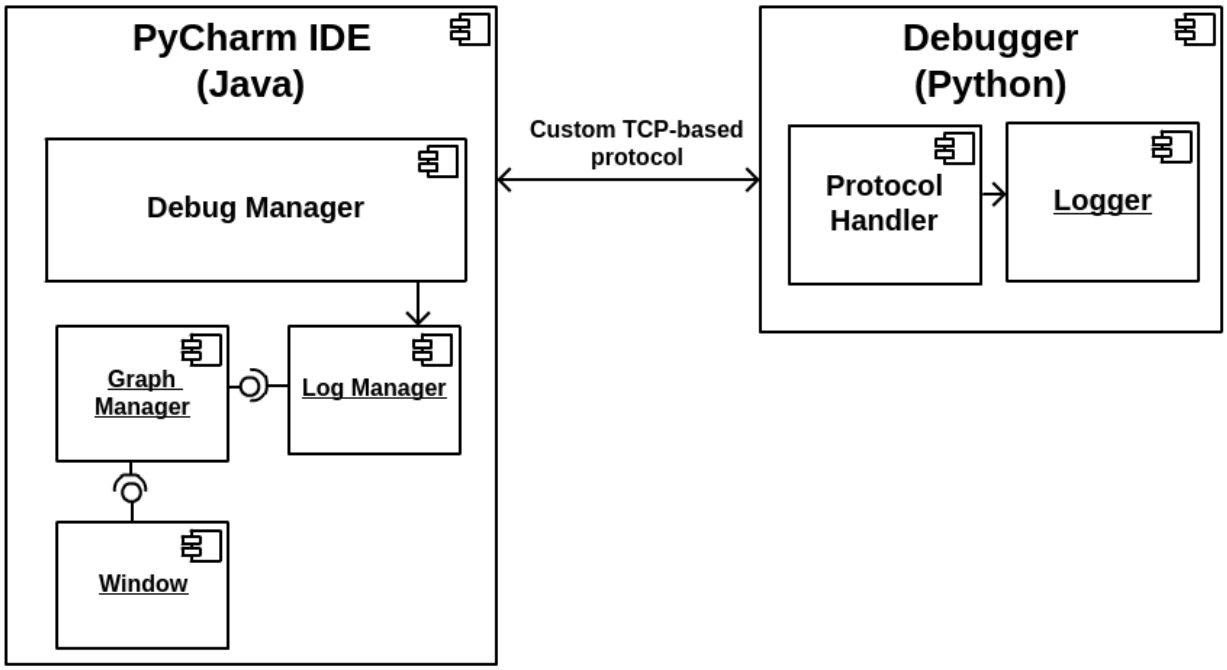

Рис. 2: Архитектура системы

`Log manager`. Далее информация о пришедшем событии через механизм оповещений сообщается компоненте `Graph manager` и затем передаётся компоненте `Window`, что приводит к обновлению вида.

Модель данных каждого события представляется структурой со следующими полями:

- `time` — время вызова функции;
- `thread id` — идентификатор потока, в котором произошло событие;
- `name` — имя потока;
- `type` — тип события (событие связано с задачей или с объектом синхронизации);
- `event` — название события;
- `file` — имя файла;
- `line` — номер строки;
- `frames` — стек фреймов в момент наступления события.

Использование шаблона MVC позволило сделать архитектуру расширяемой по данным. Например, добавление нового типа объекта синхронизации приведёт к обновлению компонент `Window` и `Graph manager`, однако оставит неизменным `Log manager`. Добавление новых полей к структуре событий также приведёт к изменению только двух компонент построения визуализации: будет изменён компонент `Graph manager`, так как изменится модель данных, и компонент `Window`, так как дополнительные данные должны быть отображены пользователю, чтобы добавление полей имело смысл. Поддержка другого языка программирования потребует только изменения компоненты `Log manager`.

В результате была спроектирована архитектура системы, которая удовлетворяет всем предъявленным к ней функциональным и нефункциональным требованиям.

2.3. Поиск дефектов

При создании многозадачных программ часто возникают ошибки. Наиболее распространёнными среди них являются состояние гонки (data race, одновременный доступ различных задач к разделяемым ресурсам) и состояние взаимной блокировки (deadlock, ожидание объекта синхронизации, который никогда не освободится).

Указанные проблемы часто можно разрешать автоматически при помощи статического или динамического подхода. При статическом анализе анализируется код программы и строится его модель. Преимущества статического анализа заключаются в том, что его проведение не замедляет работу программы и анализирует все пути её исполнения, независимо от того, выполнятся они или нет.

При динамическом подходе анализируется программа в ходе работы, поэтому анализируется единственный путь её исполнения. Преимуществами данного подхода является то, что анализу подвергаются только исполнявшиеся участки кода и реально произошедшие вызовы функций. Это ускоряет и упрощает проведение анализа, а также делает такой анализ точным, так как все обнаруженные ошибки являются истинными.

В соответствии с постановкой задачи поиск дефектов в настоящем инструменте основывается на динамически собранной информации и анализирует одну трассу выполнения программы. Поиск гонок посредством динамического анализа потребовал бы протоколирования и про-

ведения анализа большого объёма данных. Это бы привело к значительному увеличению времени работы программы, и не позволило бы удовлетворять требованию инкрементальной обработки данных, предъявленному к системе. В связи с этим в качестве дефектов многозадачной программы для автоматического обнаружения были выделены следующие проблемы:

- неиспользованные объекты синхронизации;
- задачи, не дождавшиеся объединения;
- взаимные блокировки (deadlocks).

Две первые разновидности дефектов не требуют дополнительного анализа журналов и становятся видны пользователю после построения визуализации исполнения программы. Если задача не дождалась объединения, это означает, что линия на графе, соответствующая этой задаче, не завершается поворотом к задаче, которая её породила, а продолжается вниз до окончания работы программы. Если объект синхронизации был создан, но не захватывался ни одной из задач, это также будет видно из графа.

2.4. Поиск взаимных блокировок

Наибольший интерес представляет поиск взаимных блокировок, возникающих в ходе работы программы, потому что, в отличие от остальных дефектов, требует проведения дополнительного анализа программы. На основе информации, собираемой в инструменте, можно применить алгоритм, предложенный в [7] и основанный на графе задач и ресурсов.

Согласно [3] поиск взаимных блокировок требует представления текущего состояния динамической системы в виде графа. Граф задач и ресурсов (TRG, Task-Resource Graph) — это двудольный граф, вершины которого разбиты на два непересекающихся подмножества: множество задач $T = \{T_1, T_2, ...T_n\}$ и множество ресурсов $R = \{R_1, R_2, ...R_k\}$. Рёбра в графе ресурсов и задач отображают захват или ожидание ресурсов. Ожидание ресурса обозначается ребром ожидания (T_i, R_j), которое идёт от задачи T_i к ресурсу R_j. Захват ресурса обозначается ребром захвата (R_k, T_l), которое идёт от ресурса R_k к задаче T_l и означает, что ресурс R_k был захвачен задачей T_l.

Алгоритм 1 Обнаружение взаимных блокировок

```
1: visited ← start_resource
2: for task ∈ acquired(start_resource) do
3:     push(task)
4: end for
5: while task ← pop() do
6:     if task = start_task then
7:         return true
8:     end if
9:     resource ← wait(task)
10:    if resource ∉ visited then
11:        push(acquired(resource))
12:    end if
13: end while
14: return false
```

На Рис. 3 изображён граф задач и ресурсов, на котором изображены две задачи T_1 и T_2 и два ресурса R_1 и R_2. Как видно из схемы, задача T_1 захватила ресурс R_1 и ожидает ресурс R_2. В то же время задача T_2 захватила ресурс R_2 и ожидает ресурс R_1. Путь $T_1 \to R_2 \to T_2 \to R_1 \to T_1$ является циклом, а значит, задачи T_1 и T_2 находятся в состоянии взаимной блокировки. В графе задач и ресурсов наличие такого цикла является необходимым и достаточным условием для наличия взаимной блокировки.

Алгоритм поиска циклов в графе задач и ресурсов описан в Алгоритме 1. Алгоритм запускается при попытке задачи $start\_task$ захватить ресурс $start\_resource$. В данном алгоритме используется функция $acquired(resource)$, которая возвращает список задач, захвативших ресурс $resource$, и функция $wait(task)$, которая возвращает ресурс, ожидаемый задачей $task$. Операции $push()$ и $pop()$ — операции работы со структурой данных очередью.

Если взаимная блокировка была найдена, алгоритм вернёт $true$. В противном случае он вернёт $false$. Стоит отметить, что при каждом обращении к ресурсу в ходе работы программы для каждого ресурса поддерживается список захвативших его задач, а для каждой задачи хранится информация об ожидаемом ею ресурсе. Таким образом, функции $acquired()$ и $wait()$ возвращают уже хранящуюся информацию и не требуют дополнительного обхода графа.

В ходе работы алгоритма не будет сделано больше операций, чем

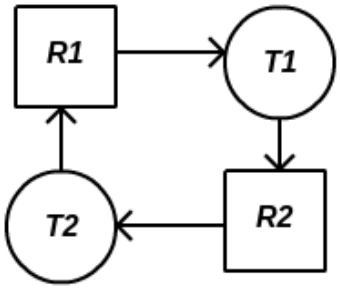

Рис. 3: Граф задач и ресурсов для двух задач и двух ресурсов.

количество рёбер в графе. Пусть R — количество ресурсов в графе, а T — количество задач. Каждый ресурс не может быть захвачен больше, чем T раз, а каждая задача в нашей модели может ожидать только один ресурс. Таким образом, сложность работы алгоритма равняется $O(R \cdot T + T) \sim O(R \cdot T)$.

Если взаимная блокировка была обнаружена, потоки, участвующие в блокировке, отображаются на графе красным цветом.

В данной главе был описан интерфейс инструмента для визуализации и анализа многозадачных программ. Затем были рассмотрены требования к архитектуре и ограничения, которые на неё накладываются. После этого были перечислены дефекты, которые могут присутствовать в многозадачных программах, и были выбраны дефекты, которые могут быть найдены при динамическом анализе программы без значительного увеличения времени её работы.

3. Особенности реализации

Требуемый инструмент был разработан в соответствии с архитектурой, описанной в предыдущем разделе. Реализация инструмента имела свои особенности, которые описываются в данной главе.

3.1. Протоколирование событий

Протоколирование событий в отладчике происходит на стороне трассировщика. Для протоколирования событий как в случае с потоками, так и в случае с модулем `asyncio` при инициализации отладчика осуществляется динамическое изменение кода библиотек. Для представляющих интерес классов и функций из этих модулей создаются «обёртки», которые вызывают оригинальные функции. В остальном работа отладчика остаётся прежней.

Теперь при переходе в новую область видимости производится проверка о переходе в одну из «обёрток». Если переход был осуществлён в одну из функций-«обёрток», значит была вызвана функция, подлежащая трассировке. В этом случае в интерфейс отладчика отправляется сообщение, содержащее информацию о произошедшем событии.

Как модуль `threading`, так и модуль `asyncio` имеют свои особенности устройства и поэтому получение информации о событиях осуществляется с помощью различных функций-«обёрток».

3.2. Стандартные потоки в языке Python

Работа с потоками в Python осуществляется с помощью функций из модуля `threading`. Потоки представляет класс `Thread`. Для работы с потоками можно как создавать экземпляры данного класса, так и наследоваться от него, перегружая метод `run`, который всегда начинает выполняться при запуске нового потока. Для протоколирования потоков были выбраны следующие функции.

- `start` — функция, которая запускает выполнение потока;
- `join` — функция ожидания и последующего присоединения дочернего потока, когда он завершит работу;
- `_stop` — функция завершения работы потока.

Основными объектами синхронизации в многопоточных программах являются объекты класса `Lock` и `RLock`. Каждый экземпляр этих классов имеет состояние «захвачен» и «свободен». Оба этих класса обладают следующими методами.

- `acquire` — захват объекта синхронизации текущим потоком, состояние объекта синхронизации меняется на состояние «захвачен»,
- `release` — освобождение объекта синхронизации текущим потоком, состояние объекта синхронизации меняется на состояние «свободен».

Отличие классов `Lock` и `RLock` заключается в том, что объекты класса `RLock` могут быть захвачены одним и тем же потоком несколько раз. В этом случае, объект переходит в состояние «свободен» только если был освобождён этим потоком столько же раз, сколько и захвачен. Экземпляры класса `Lock` поддерживают только одноразовые захват и освобождение.

При динамическом изменении кода также требуется записывать в том числе начало и окончание таких вызовов. Например, в случае с методом `acquire` требуется распознавать начало и окончание ожидания объекта синхронизации, потому что часто между этими событиями может пройти значительный промежуток времени. Эта задача была решена также посредством динамического изменения кода, а именно, перед и после вызова интересующих функций производился вызов дополнительных функций.

Также для обмена данными между потоками часто используются объекты класса `Queue`. Для построения визуализации представляли интерес следующие методы.

- `put` — добавить объект в очередь;
- `get` — изъять объект из очереди.

Объекты класса `Queue` в данной реализации рассматривались аналогично объектам класса `Lock`. То есть при вызове функции `put` или `get` считалось, что поток начал ожидать объект синхронизации. Когда вызов заканчивался, считалось, что доступ к объекту был получен и поток продолжает свою работу. Такой подход позволил отображать захват экземпляров класса `Lock` и класса `Queue` единообразно на графике визуализации.

3.3. Асинхронные задачи в языке Python

Основным понятием в модуле `asyncio` является цикл обработки событий (event loop). Каждая отдельная задача является экземпляром класса `Task`. Он попеременно переключается между задачами и таким образом с точки зрения пользователя создаётся впечатление параллельного выполнения задач.

Между модулем `asyncio` и потоками можно установить соответствие. Создание каждой новой задачи можно рассматривать как создание нового потока, а вызов функции `cancel()` (отмену задачи) рассматривать как остановку потока. Помимо этого, пакет `asyncio` предоставляет объекты синхронизации, аналогичные объектам синхронизации в стандартных потоках. В нём имеется класс `Lock`, способный находиться в двух состояниях «захвачен» и «свободен», и также обладающий методами `acquire()` и `release()`, которые означают захват и освобождение данного объекта синхронизации текущей задачей. По аналогии с модулем `threading` в пакете `asyncio` существует класс `Queue` для безопасного обмена данными между одновременно исполняющимися задачами.

Для протоколирования в модуле `asyncio` были выбраны функции, перечисленные в предыдущем разделе и реализующие схожую функциональность.

Отличительной особенностью пакета `asyncio` является то, что все функции для работы с задачами в нём являются функциями отложенного выполнения, и поэтому добавление «обёрток», как в случае с модулем `threading` не достаточно для решения поставленной задачи. Поэтому помимо отслеживания вызовов функций, производился анализ внутреннего состояния объекта, доступного из фрейма, и исходя из этого, собирается информация о работе задач с объектами синхронизации.

4. Тестирование и апробация

Тестирование реализованного инструмента проводилось с помощью автоматизированных функциональных тестов и реальных проектов с открытым исходным кодом. Также был проведён эксперимент с участием разработчиков на языке Python.

При проведении функционального тестирования проверялась корректность модели графа при построении визуализации. Для модулей `threading` и `asyncio` проверялась корректность создания задач, завершения задач, работы с различными комбинациями объектов синхронизации и поиск взаимных блокировок.

4.1. Влияние на производительность

В процессе тестирования инструмента исследовано влияние процесса построения визуализации на время работы программы. В Таблице 1 представлены результаты измерения времени работы четырёх программ на языке Python в различных режимах запуска в среде разработки PyCharm. Первая программа single работает с использованием одного (главного) потока. Вторая и третья программы — threads10 и threads100 создают в процессе своей работы соответственно 10 и 100 потоков, взаимодействующих через общие объекты синхронизации. Программа oscar — реальное веб-приложение, написанное с использованием веб-фреймворка Oscar. При проведении измерений в нём создавалось 12 потоков и 105 различных объектов синхронизации.

В первой колонке показаны результаты измерений при запуске программ в обычном режиме, во второй — при запуске в режиме отладки, в третьей — при запуске в режиме отладки с построением визуализации. Как видно из таблицы, построение визуализации замедляет время работы программы только при создании большого количества (около 100) потоков и их активной работы с объектами синхронизации. Стоит отметить, что запуск программы в режиме отладки уже значительно (почти в два раза) замедляет время выполнения программы, поэтому замедление, возникающее при построении визуализации, можно считать приемлемым.

Аналогичные измерения измерения были проведены для программ, использующих несколько задач из модуля `asyncio` и представлены в таблице 2. Здесь asyncio10 и asyncio100 — программы, использующие соответственно 10 и 100 задач из модуля `asyncio`, а aiohttp — веб-сервер,

Программа	Время работы, с	Время работы в режиме отладки, с	Время работы с построением визуализации, с
single	0.02 ± 0.01	0.04 ± 0.01	0.04 ± 0.01
threads10	0.02 ± 0.01	0.04 ± 0.02	0.05 ± 0.03
threads100	0.32 ± 0.05	0.52 ± 0.04	0.60 ± 0.05
oscar	6.5 ± 0.4	13.1 ± 0.4	14.5 ± 0.5

Таблица 1: Время работы многопоточных программ в различных режимах в PyCharm

Программа	Время работы, с	Время работы в режиме отладки, с	Время работы с построением визуализации, с
asyncio10	0.012 ± 0.001	0.023 ± 0.001	0.024 ± 0.001
asyncio100	0.117 ± 0.008	0.237 ± 0.012	0.265 ± 0.009
aiohttp	0.028 ± 0.004	0.045 ± 0.003	0.057 ± 0.004

Таблица 2: Время работы многозадачных программ в различных режимах в PyCharm

использующий библиотеку `aiohttp`, в процессе работы которого создавалось 5 задач.

4.2. Упрощение понимания многозадачных программ

Разработанный инструмент был протестирован в ходе следующего эксперимента. В эксперименте участвовала группа опытных разработчиков на языке Python из шести человек. Все испытуемые пользовались инструментом визуализации впервые и предварительно не изучали его документацию. Эксперимент состоял из двух этапов.

На перовом этапе программисту давалась многозадачная программа на языке Python, состоящая из одного файла размером в сто строк и работающая с 12 потоками. Человек должен был оценить количество потоков, создающихся в программе, и описать ход её выполнения. Затем на втором этапе пользователь запускал инструмент визуализации, анализировал схема работы потоков, построенную автоматически и проверял правильность собственных предположений. Время прохождения обоих этапов измерялось.

Проведение эксперимента показало следующие результаты.

Время прохождения первого этапа было различным и находилось в

промежутке от 2,5 до 7 минут. На первом этапе, в процессе которого программисты понимали устройство многопоточной программы, почти все из них пользовались письменными принадлежностями и рисовали вспомогательную диаграмму последовательности, схожую по структуре с диаграммой, строящейся в визуализаторе.

Все программисты допустили ошибки при ответе на вопросы после первого этапа эксперимента. В большинстве случаев испытуемые правильно описывали общий ход выполнения программы, однако часто в описание не входили важные события из неё. Каждый испытуемый не замечал один или несколько объектов синхронизации, которые оказывали влияние на ход её выполнения. Также никто не смог правильно назвать точное количество потоков.

Инструмент визуализации пользователи видели впервые и тратили около двух минут на ознакомление с его интерфейсом без обращения к документации. Из построенной визуализации они находили события, которые были упущены из внимания при ознакомлении с кодом программы.

Также после проведения эксперимента испытуемые имели возможность высказать недостатки инструмента, замеченные ими во время работы. Чаще всего упоминались не всегда очевидная цветовая схема графа и непривычное вертикальное расположение временной прямой. Эти и другие недостатки пользовательского интерфейса планируется устранить в дальнейшем.

4.3. Анализ результатов эксперимента

Из полученных результатов можно сделать следующие выводы. Инструмент был освоен пользователями за короткий промежуток времени без обращения к документации. Из этого можно сделать вывод, что интерфейс инструмента является интуитивно понятным и дружественным пользователю, и начало работы с ним не требует длительной подготовки.

Использование реализованного инструмента визуализации ускоряет скорость понимания многозадачной программы. Эксперимент показал, что инструмент помогает программисту заметить все события многозадачности, проходящие в программе, часть из которых может быть пропущена при ознакомлении только с кодом программы.

Заключение

В рамках данной дипломной работы были достигнуты следующие результаты.

- Реализовано протоколирование событий в многозадачной программе на языке Python, запущенной в режиме отладки в среде разработки PyCharm: запуск и завершение задач; ожидание, захват и освобождение объектов синхронизации.
- Реализовано расширение к среде разработки PyCharm (Java/Python), обладающее следующим набором возможностей:
  - визуализация исполнения многозадачной программы;
  - осуществление навигации к исходному коду и отображение стека вызовов в момент наступления события.
- На основе полученных журналов (logs) реализован поиск дефектов в многозадачной программе:
  - неиспользованных объектов синхронизации;
  - задач, не дождавшихся объединения;
  - взаимных блокировок (deadlocks).
- Проведено тестирование разработанного инструмента.

В дальнейшем планируется расширение функциональности, упрощающее восприятие построенного графа и позволяющее проводить фильтрацию интересующей пользователя событий. Также планируется улучшение пользовательского интерфейса инструмента на основе отзывов пользователей. Разработанный инструмент показал свою применимость на практике и будет использован в среде разработки PyCharm для визуализации многозадачных программ.

Список литературы

[1] Cai W. Milne W. J. Turner S. J. Graphical views of the behavior of parallel programs. — J. Parallel Distrib. Comput. 18, 223–230, 1993.

[2] Global interpreter lock — Python Wiki. — https://wiki.python.org/moin/GlobalInterpreterLock.

[3] Holt R. C. Some Deadlock Properties on Computer Systems. — ACM Computing Surveys, Vol. 4, No. 3, pp. 179196,, September 1972.

[4] Kraemer E. Visualizing Concurrent Programs, in Software Visualization. — MIT Press, pp. 237–256, 1998.

[5] M. Bedy S. Carr X. Huang CK. Shene. A visualization system for multithreaded programming. ACM SIGCSE Bulletin. — Department of Computer Science, Michigan Technological University, 2000.

[6] PyCon 2013: Asynchronous I/O. — http://lwn.net/Articles/544522/.

[7] Shih C.-S., Stankovic J. Survey of Deadlock Detection in Distributed Concurrent Programming Environments and Its Application to Real-Time Systems and Ada. Technical Report UM-CS-1990-069. — University of Massachusetts, Amherst, MA, 1990.

[8] Stasko J. T. The PARADE Environment for Visualizing Parallel Program Executions: A Progress Report. Technical Report GIT-GVU-95-03. — College of Computing, Georgia Institute of Technology, 1995.

[9] Zhao Q. A., Stasko J. T. Visualizing the Execution of Threads-based Parallel Programs. Technical Report GIT-GVU-95-01. — College of Computing, George Institute of Technology, January 1995.