МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное бюджетное образовательное учреждение высшего образования

«МИРЭА - Российский технологический университет»

РТУ МИРЭА

Институт информационных технологий

Кафедра математического обеспечения и стандартизации информационных технологий

ОТЧЕТ

ПО ПРАКТИЧЕСКОЙ РАБОТЕ № 4

по дисциплине

«Тестирование и верификация программного обеспечения»

Выполнил студент группы *ИКБО-74-23*

Ермоленко В.М.

Принял Ильичев Г.П.

Практическая работа выполнена «20» ноября 2025 г. _____

«Зачтено» «__»_____ 2025 г. _____

Москва 2025

1. Цель и задачи практической работы.

Цель работы: ознакомиться с основными принципами и методами использования статических и динамических анализаторов кода для раннего выявления ошибок и потенциальных уязвимостей, что позволит повысить качество, безопасность и надёжность программного обеспечения.

Задачи работы:

1. Изучить теоретические основы статического и динамического анализа кода.
2. Ознакомиться с популярными инструментами статического анализа (например, ESLint, Pylint, Checkmarx, SonarQube, FindBugs, TSLint, Cppcheck) и динамического анализа (например, Valgrind, DynamoRIO, Java VisualVM, Burp Suite, OWASP ZAP).
3. Применить выбранные анализаторы к ранее разработанным учебным проектам на разных языках программирования.
4. Провести анализ исходного кода до и после внесения целенаправленных ошибок, оценить адекватность обнаружения дефектов.
5. Сформировать детальный отчёт с критическим анализом результатов, выводами о преимуществах и ограничениях каждого подхода.

1.2. Теоретический раздел

1.2.1. Статический анализ кода

Статический анализ — это метод анализа программного кода без его исполнения. Его основная задача — обнаружить потенциальные ошибки, нарушения стандартов кодирования, неэффективные или опасные конструкции и утечки ресурсов непосредственно в исходном коде.

Основные характеристики:

1. Позволяет выявить ошибки ещё до запуска программы, что снижает затраты на их исправление.
2. Результаты анализа оформляются в виде отчётов, включающих список найденных проблем, ссылки на документацию, описание потенциальных рисков и рекомендации по исправлению.

1.2.2. Динамический анализ кода

Динамический анализ представляет собой процесс изучения поведения программы во время её исполнения. Этот метод позволяет выявить ошибки, которые не обнаруживаются статическим анализом, например, утечки памяти, ошибки исполнения и проблемы с производительностью.

Основные характеристики:

1. Позволяет оценить, как программа работает в условиях, приближенных к боевым, выявляя проблемы, связанные с взаимодействием модулей и ресурсами системы.
2. Результаты динамического анализа оформляются в виде отчётов, включающих данные о профилировании, использовании памяти, времени выполнения и других аспектах работы программы**.**

2. Практическая часть

Для выполнения работы были использованы два учебных проекта:

- Проект на Python (main.py): Скрипт для обработки данных студентов, включающий математические операции и работу со списками.
- Проект на JavaScript (app.js): Веб-сервер на базе Node.js, обрабатывающий HTTP-запросы.

2.1. Статический анализ кода (Часть 1)

Используемые инструменты:

Python:

- Pylint: Комплексный анализатор качества кода (стиль, ошибки, стандарты).
- Flake8: Инструмент для проверки соответствия стилю PEP8.
- Mypy: Статический анализатор типов.

JavaScript:

- ESLint: Линтер для поиска проблемных шаблонов кода.
- JSHint: Инструмент для обнаружения ошибок и потенциальных проблем.
- Standard: Форматировщик и линтер с жесткими правилами стиля.

2.1.1. Анализ исходного кода

Рис. 1 – Исходный код приложения на python (main.py)

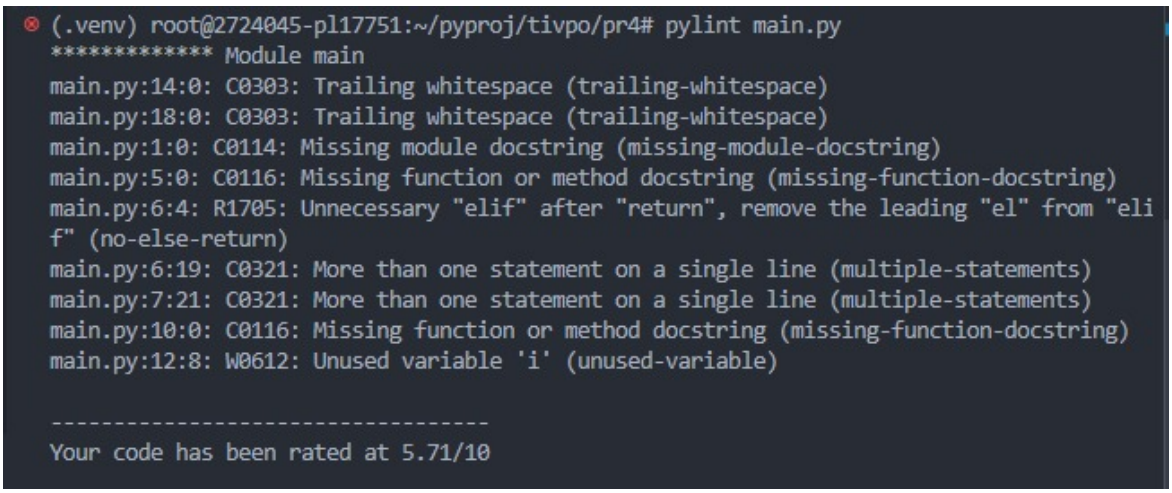


Рис. 2 – Результат работы pylint (оценка 5.71 / 10)

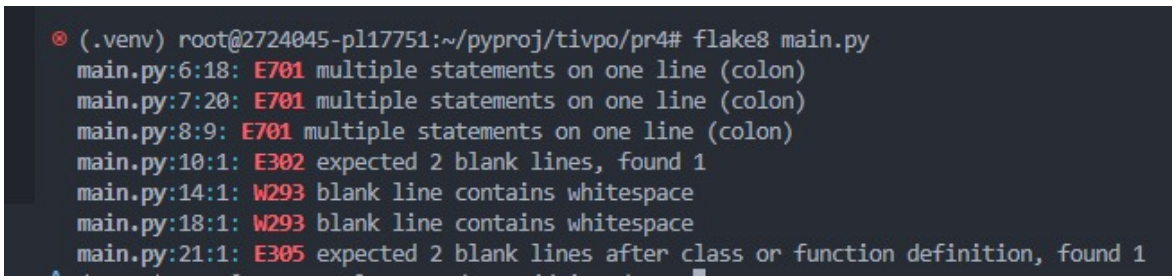


Рис. 3 – Результат работы flake8 (есть несоответствия PEP8)

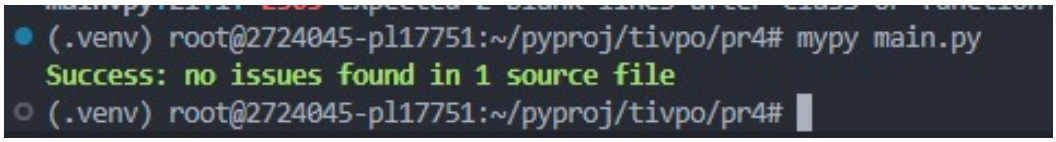


Рис. 4 – Результат работы mypy (успех)

Критических ошибок выявлено не было. Pylint оценил код на 5.71/10. Муру подтвердил корректность типов.

Вторым объектом исследования стал веб-сервер, разработанный на платформе Node.js (файл app.js).

Для обеспечения объективности проверки использовались три независимых инструмента статического анализа:

- ESLint: Стандарт де-факто для JavaScript, настроенный на поиск проблемных паттернов.
- JSHint: Инструмент для обнаружения ошибок и потенциальных проблем в коде.
- Standard: Линтер с "нулевой конфигурацией", навязывающий строгий стиль кодирования (отсутствие точек с запятой, отступы 2 пробела).

Код был приведен к стилю Standard. При запуске всех трех анализаторов ошибок выявлено не было.

Рис. 1.1 – Исходный код приложения на js (app.js)

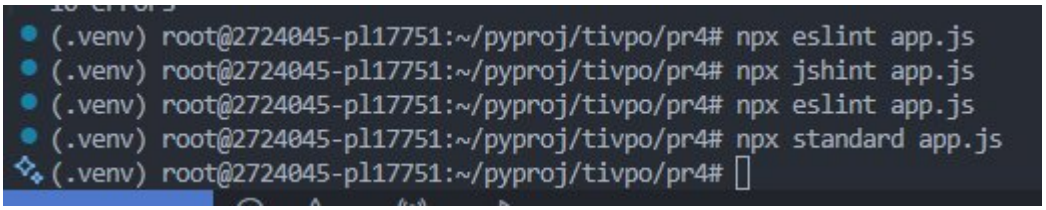


Рис. 2.1 – Результат работы анализаторов на js

2.1.2. Анализ с внесением дефектов

В каждый проект было внесено по 5 целенаправленных ошибок различного типа.

Список внесенных ошибок (Python):

- Синтаксис: Отсутствие двоеточия : в конструкции if.
- Стиль: Имя функции ProcessStudents (нарушение стандарта).
- Логика: Лишний импорт import os.
- Типы: Сложение строки и числа.
- Identation: Использование лишнего отступа перед return.

Рис. 5 – Измененный код приложения на python (main.py)

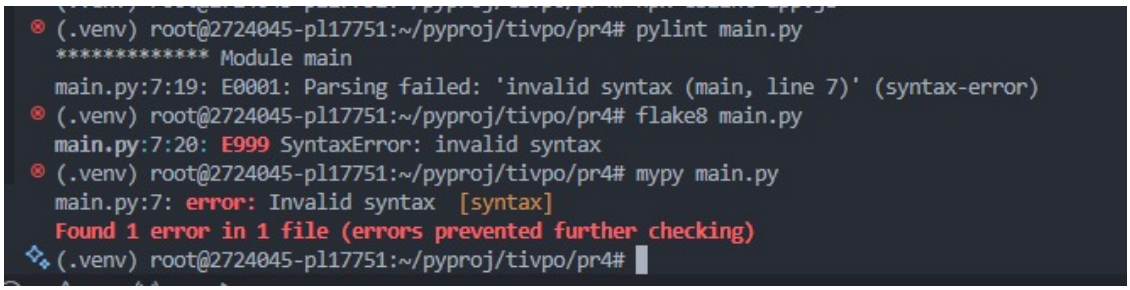


Рис. 6 – Результат работы анализаторов (все три упали)

При запуске анализаторов (Pylint, Flake8, Муру) на модифицированном коде было выявлено, что синтаксические ошибки являются блокирующими.

Таблица 1. Результаты обнаружения дефектов в main.py

Ошибка	**Тип**	**Результат анализа**
Отсутствие двоеточия в конструкции if	Синтаксис	ОБНАРУЖЕНА. Анализаторы выдали SyntaxError и аварийно завершили работу, не проверив остальные ошибки.
Имя функции Process_Students	Стиль	ПРОПУЩЕНА (анализ прерван из-за ошибки синтаксиса).
Лишний импорт import os	Логика	ПРОПУЩЕНА (анализ прерван из-за ошибки синтаксиса).
Сложение строки и числа	Типы	ПРОПУЩЕНА (анализ прерван из-за ошибки синтаксиса).
Использование лишнего отступа перед return	Identation	ПРОПУЩЕНА (анализ прерван из-за ошибки синтаксиса).

Исходя из этого, было решено провести отдельную проверку исключительно для логических и стилевых ошибок.

main.py > ProcessStudents

```python
import random
import time
import os

def calculate_grade(score):
    if score > 90: return 'A'
    elif score > 75: return 'B'
    else: return 'C'

def ProcessStudents() -> list:
    students = []
    results = []
    for i in range(5000):
        students.append(random.randint(50, 100))
        results.append("Grade: " + 5)

    for s in students:
        results.append(calculate_grade(s))

    return results

if __name__ == "__main__":
    start_time = time.time()
    print("Starting processing...")
    ProcessStudents()
    print(f"Finished in {time.time() - start_time} seconds")

```

Рис. 7 – Модифицированный main.py

```
(.venv) root@2724045-pl17751:~/pyproj/tivpo/pr4# mypy main.py
main.py:15: error: Unsupported operand types for + ("str" and "int")  [operator]
Found 1 error in 1 file (checked 1 source file)
(.venv) root@2724045-pl17751:~/pyproj/tivpo/pr4# pylint main.py
************* Module main
main.py:1:0: C0114: Missing module docstring (missing-module-docstring)
main.py:5:0: C0116: Missing function or method docstring (missing-function-docstring)
main.py:6:4: R1705: Unnecessary "elif" after "return", remove the leading "el" from "elif" (no-else-return)
main.py:6:19: C0321: More than one statement on a single line (multiple-statements)
main.py:7:21: C0321: More than one statement on a single line (multiple-statements)
main.py:10:0: C0116: Missing function or method docstring (missing-function-docstring)
main.py:10:0: C0103: Function name "ProcessStudents" doesn't conform to snake_case naming style (invalid-name)
main.py:13:8: W0612: Unused variable 'i' (unused-variable)
main.py:3:0: W0611: Unused import os (unused-import)

-------------------------------------------------------------------
Your code has been rated at 6.09/10 (previous run: 3.91/10, +2.17)

(.venv) root@2724045-pl17751:~/pyproj/tivpo/pr4# flake8 main.py
main.py:3:1: F401 'os' imported but unused
main.py:5:1: E302 expected 2 blank lines, found 1
main.py:6:18: E701 multiple statements on one line (colon)
main.py:7:20: E701 multiple statements on one line (colon)
main.py:8:9: E701 multiple statements on one line (colon)
main.py:10:1: E302 expected 2 blank lines, found 1
main.py:22:1: E305 expected 2 blank lines after class or function definition, found 1
(.venv) root@2724045-pl17751:~/pyproj/tivpo/pr4#
```

Рис. 8 – Результаты работы анализаторов

Таблица 2. Обновленные результаты обнаружения дефектов в main.py

Ошибка	**Тип**	**Результат анализа**
Отсутствие двоеточия в конструкции if	Синтаксис	ОБНАРУЖЕНА ВСЕМИ. Анализаторы выдали SyntaxError и аварийно завершили работу, не проверив остальные ошибки.
Имя функции Process_Students	Стиль	ОБНАРУЖЕНА (Pylint, Flake8), C0103: Function name "ProcessStudents" doesn't conform to snake_case naming style
Лишний импорт import os	Логика	ОБНАРУЖЕНА (Pylint, Flake8), W0611: Unused import os (unused-import)
Сложение строки и числа	Типы	ОБНАРУЖЕНА (MyPy), error: Unsupported operand types
Использование лишнего отступа перед return	Identation	ОБНАРУЖЕНА ВСЕМИ, IndentationError: unexpected indent.

Выводы по части 1:

Использование комбинации инструментов (Pylint + Муру) позволило обнаружить 100% внесенных дефектов.

- Pylint и Flake8 эффективно находят ошибки стиля (PEP8) и синтаксиса.
- Муру обнаружил ошибку типов, которую стандартные линтеры пропустили бы (или заметили только при запуске).
- Эксперимент подтвердил, что грубые синтаксические ошибки (№1 и №5) имеют наивысший приоритет и должны быть исправлены первыми, так как они блокируют поиск остальных проблем.

В проект на js также было внесено 5 целенаправленных ошибок:

- Синтаксис: Удалена закрывающая фигурная скобка } в конце файла.
- Стиль: Использовано устаревшее объявление переменной через var.
- Логика: Опечатка в условии: присваивание = вместо сравнения ===.
- Область видимости: Использование переменной sum без объявления (создание неявной глобальной переменной).
- Недостижимый код: Наличие исполняемого кода после оператора return.

Рис. 3.1 – Модифицированный app.js

При запуске анализаторов на коде с полным набором ошибок было выявлено, что синтаксическая ошибка является блокирующей.

Инструменты ESLint и Standard выдали критическую ошибку Parsing error: Unexpected token и аварийно завершили проверку. JSHint обнаружил ошибку логики, но затем также остановился с ошибкой Unrecoverable syntax error. Синтаксическая ошибка (отсутствие скобки) заблокировала построение абстрактного синтаксического дерева (AST), из-за чего анализаторы не смогли обнаружить остальные 4 ошибки.

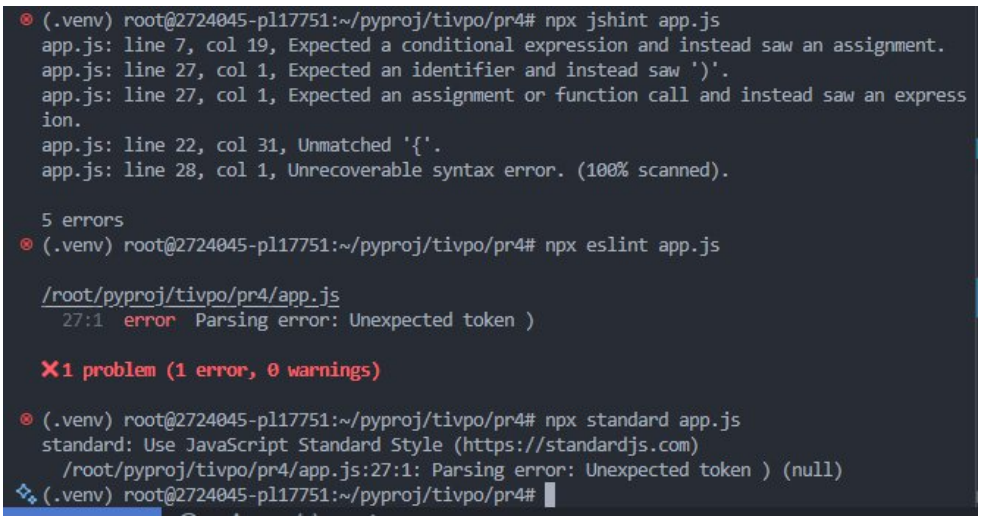


Рис. 4.1 – Результат запуска анализаторов на обновленном app.js

После исправления синтаксической ошибки, можно оценить, как анализаторы нашли остальные ошибки:

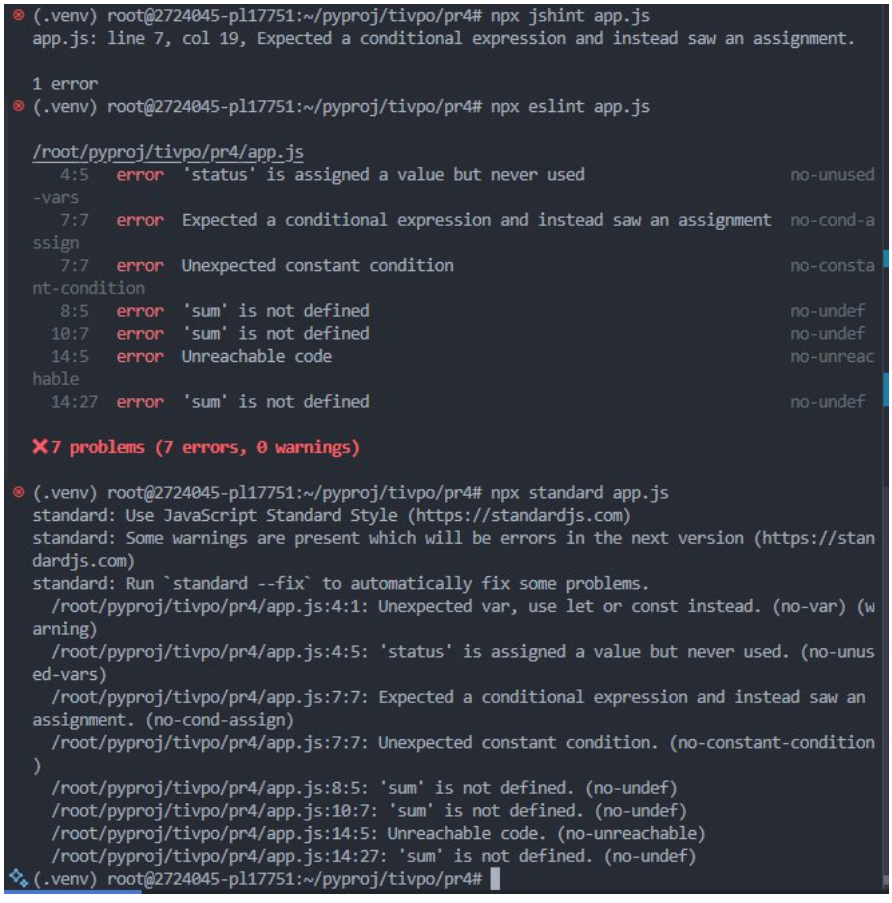


Рис. 5.1 – Результат запуска анализаторов на повторно обновленном app.js

Таблица 3. Итоговые результаты обнаружения дефектов в app.js

Ошибка	Тип	Примечание
Удалена	Синтаксис	ОБНАРУЖЕНА ВСЕМИ.

закрывающая скобка }		Критическая ошибка. ESLint, JSHint и Standard выдали «Parsing error» и остановили проверку.
Использование var	Стиль	ОБНАРУЖЕНА (Standard). Standard выдал «Unexpected var». Ошибка найдена только после исправления синтаксиса.
Присваивание в условии =	Логика	ОБНАРУЖЕНА ВСЕМИ. JSHint обнаружил даже при наличии синтаксических ошибок. ESLint и Standard нашли только после исправления скобки.
Переменная sum без объявления	Область видимости	ОБНАРУЖЕНА (ESLint и Standard). ESLint и Standard выдали «'sum' is not defined». Найдена только после исправления синтаксиса.
Код после return	Недостижимый код	ОБНАРУЖЕНА (ESLint). ESLint выдал «Unreachable code». Найдена только после исправления синтаксиса.

2.2. Динамический анализ кода (Часть 2)

Динамический анализ производился во время выполнения программы для оценки потребления ресурсов (CPU, RAM) и стабильности.

Используемые инструменты:

Python:

- cProfile: Встроенный профилировщик времени выполнения.
- memory_profiler: Инструмент мониторинга использования памяти.

JavaScript:

- Node Profiler (--prof): Анализ загрузки V8.
- Clinic.js (Doctor): Комплексная диагностика производительности Node.js.

Для использования memory_profiler на python, код был дополнен декораторами @profile.

main.py > ...

```python
import random
import time
from memory_profiler import profile


def calculate_grade(score):
    if score > 90:
        return "A"
    elif score > 75:
        return "B"
    else:
        return "C"


@profile
def process_students() -> list:
    students = []
    results = []
    for _ in range(5000):
        students.append(random.randint(50, 100))

    for s in students:
        results.append(calculate_grade(s))

    return results


if __name__ == "__main__":
    start_time = time.time()
    print("Starting processing...")
    process_students()
    print(f"Finished in {time.time() - start_time} seconds")

```

Рис. 9 – Код main.py с поддержкой профайлера

2.2.1. Анализ нормальной работы

Для оценки потребления ресурсов в штатном режиме работы были использованы профилировщики memory_profiler (для оперативной памяти) и cProfile (для процессорного времени).

```
(.venv) root@2724045-pl17751:~/pyproj/tivpo/pr4# python main.py
Starting processing...
Filename: /root/pyproj/tivpo/pr4/main.py

Line #    Mem usage    Increment  Occurrences   Line Contents
=============================================================
    15     21.8 MiB     21.8 MiB           1   @profile
    16                                         def process_students() -> list:
    17     21.8 MiB      0.0 MiB           1       students = []
    18     21.8 MiB      0.0 MiB           1       results = []
    19     21.8 MiB      0.0 MiB        5001       for _ in range(5000):
    20     21.8 MiB      0.0 MiB        5000           students.append(random.randint(50, 100))
    21
    22     22.1 MiB      0.0 MiB        5001       for s in students:
    23     22.1 MiB      0.3 MiB        5000           results.append(calculate_grade(s))
    24
    25     22.1 MiB      0.0 MiB           1       return results


Finished in 1.1900877952575684 seconds
```

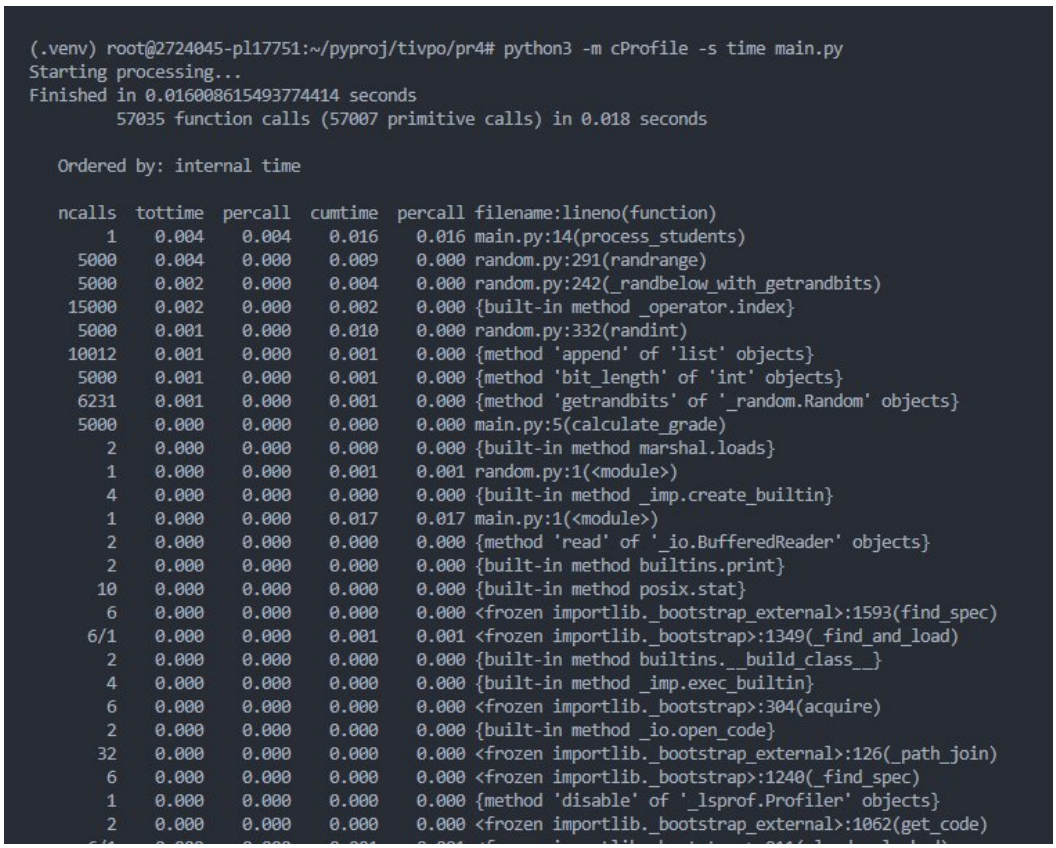
Рис. 10 – Результат работы memory_profiler

Анализ потребления памяти:

На рис. 10 представлен построчный отчет инструмента memory_profiler. Базовое потребление памяти интерпретатором и скриптом составляет порядка 21.8 MiB.

В ходе выполнения функции process_students и заполнения списков данными (строки 19-23) зафиксирован незначительный прирост памяти (+0.3 MiB).

Вывод: Управление памятью в исходном коде корректное, утечки отсутствуют, ресурсы освобождаются штатно.



```
(.venv) root@2724045-pl17751:~/pyproj/tivpo/pr4# python3 -m cProfile -s time main.py
Starting processing...
Finished in 0.016008615493774414 seconds
         57035 function calls (57007 primitive calls) in 0.018 seconds

   Ordered by: internal time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.004    0.004    0.016    0.016 main.py:14(process_students)
     5000    0.004    0.000    0.009    0.000 random.py:291(randrange)
     5000    0.002    0.000    0.004    0.000 random.py:242(_randbelow_with_getrandbits)
    15000    0.002    0.000    0.002    0.000 {built-in method _operator.index}
     5000    0.001    0.000    0.010    0.000 random.py:332(randint)
    10012    0.001    0.000    0.001    0.000 {method 'append' of 'list' objects}
     5000    0.001    0.000    0.001    0.000 {method 'bit_length' of 'int' objects}
     6231    0.001    0.000    0.001    0.000 {method 'getrandbits' of '_random.Random' objects}
     5000    0.000    0.000    0.000    0.000 main.py:5(calculate_grade)
        2    0.000    0.000    0.000    0.000 {built-in method marshal.loads}
        1    0.000    0.000    0.001    0.001 random.py:1(<module>)
        4    0.000    0.000    0.000    0.000 {built-in method _imp.create_builtin}
        1    0.000    0.000    0.017    0.017 main.py:1(<module>)
        2    0.000    0.000    0.000    0.000 {method 'read' of '_io.BufferedReader' objects}
        2    0.000    0.000    0.000    0.000 {built-in method builtins.print}
       10    0.000    0.000    0.000    0.000 {built-in method posix.stat}
        6    0.000    0.000    0.000    0.000 <frozen importlib._bootstrap_external>:1593(find_spec)
      6/1    0.000    0.000    0.001    0.001 <frozen importlib._bootstrap>:1349(_find_and_load)
        2    0.000    0.000    0.000    0.000 {built-in method builtins.__build_class__}
        4    0.000    0.000    0.000    0.000 {built-in method _imp.exec_builtin}
        6    0.000    0.000    0.000    0.000 <frozen importlib._bootstrap>:304(acquire)
        2    0.000    0.000    0.000    0.000 {built-in method _io.open_code}
       32    0.000    0.000    0.000    0.000 <frozen importlib._bootstrap_external>:126(_path_join)
        6    0.000    0.000    0.000    0.000 <frozen importlib._bootstrap>:1240(_find_spec)
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
        2    0.000    0.000    0.000    0.000 <frozen importlib._bootstrap_external>:1062(get_code)
```

Рис. 11 – Результат работы cProfile

Анализ времени выполнения:

На рис. 11 представлен статистический отчет cProfile.

Общее время выполнения скрипта составило 0.016 секунды, что свидетельствует о высокой производительности.

Сортировка по времени (tottime) показывает, что основную нагрузку создают функции генерации случайных чисел (random.py) и операции со списками (append).

Вывод: Узких мест в производительности не выявлено.

Для анализа поведения *второй* программы во время выполнения использовались:

- Node.js Profiler (--prof): Встроенный в движок V8 профилировщик процессора.
- Clinic.js Doctor: Инструмент диагностики, визуализирующий состояние Event Loop, CPU и памяти.

Перед внесением ошибок был проведен контрольный запуск исправного приложения для фиксации базовых показателей производительности.

Clinic.js Doctor: Результат диагностики Clinic.js показал, что при штатной нагрузке задержек и утечек обнаружено не было.

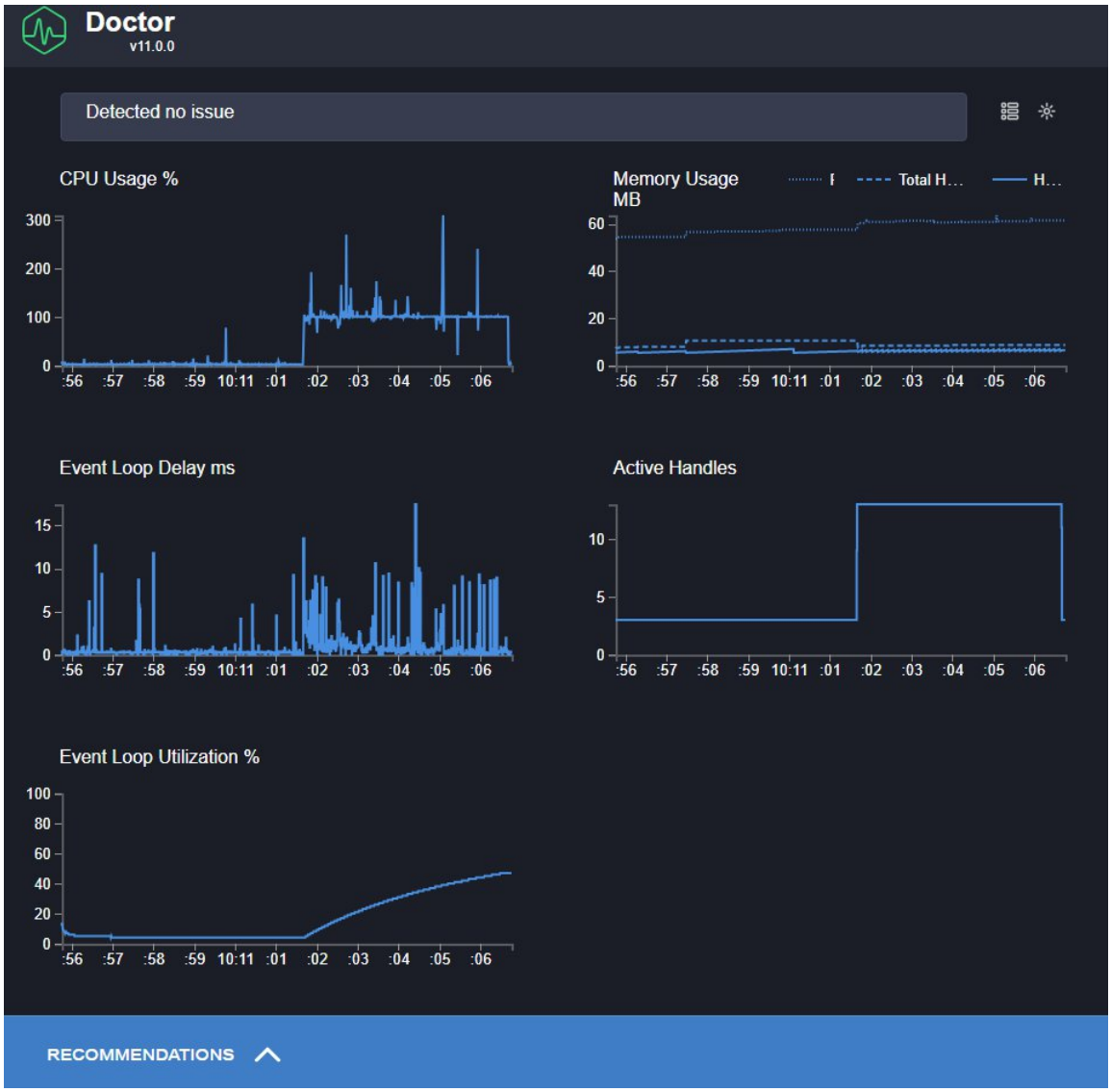
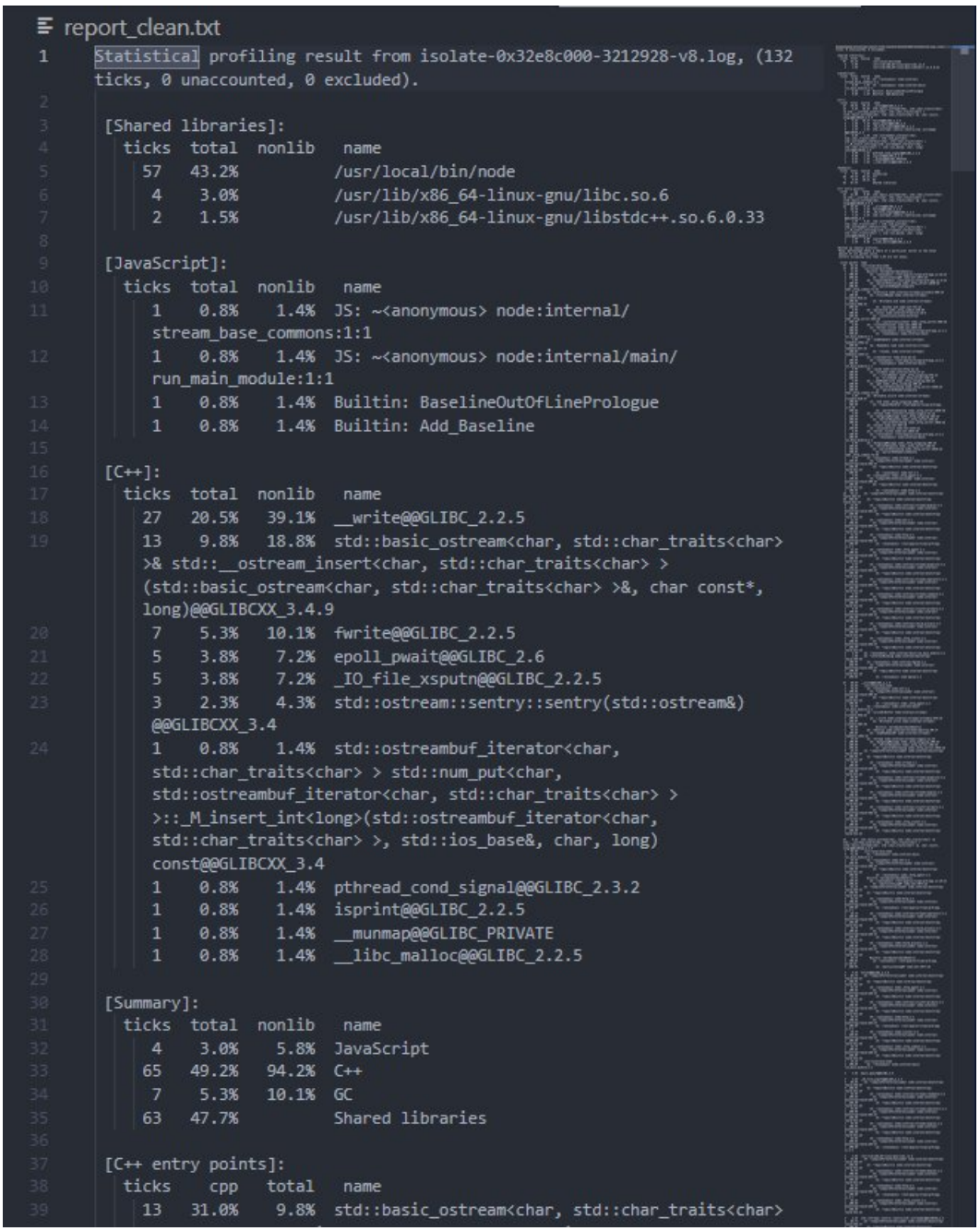


Рис. 6.1 – Результат запуска Clinic.js Doctor

Node.js Profiler: Отчет Node Profiler показал, что в штатном режиме нагрузка на JavaScript минимальна – всего 3%. Это доказывает, что код практически не потребляет ресурсы процессора в нормальном режиме.



report_clean.txt

```
Statistical profiling result from isolate-0x32e8c000-3212928-v8.log, (132
ticks, 0 unaccounted, 0 excluded).

 [Shared libraries]:
   ticks  total  nonlib   name
     57   43.2%          /usr/local/bin/node
      4    3.0%          /usr/lib/x86_64-linux-gnu/libc.so.6
      2    1.5%          /usr/lib/x86_64-linux-gnu/libstdc++.so.6.0.33

 [JavaScript]:
   ticks  total  nonlib   name
      1    0.8%    1.4%  JS: ~<anonymous> node:internal/
     stream_base_commons:1:1
      1    0.8%    1.4%  JS: ~<anonymous> node:internal/main/
     run_main_module:1:1
      1    0.8%    1.4%  Builtin: BaselineOutOfLinePrologue
      1    0.8%    1.4%  Builtin: Add_Baseline

 [C++]:
   ticks  total  nonlib   name
     27   20.5%   39.1%  __write@@GLIBC_2.2.5
     13    9.8%   18.8%  std::basic_ostream<char, std::char_traits<char>
     >& std::__ostream_insert<char, std::char_traits<char> >
     (std::basic_ostream<char, std::char_traits<char> >&, char const*,
     long)@@GLIBCXX_3.4.9
      7    5.3%   10.1%  fwrite@@GLIBC_2.2.5
      5    3.8%    7.2%  epoll_pwait@@GLIBC_2.6
      5    3.8%    7.2%  _IO_file_xsputn@@GLIBC_2.2.5
      3    2.3%    4.3%  std::ostream::sentry::sentry(std::ostream&)
     @@GLIBCXX_3.4
      1    0.8%    1.4%  std::ostreambuf_iterator<char,
     std::char_traits<char> > std::num_put<char,
     std::ostreambuf_iterator<char, std::char_traits<char> >
     >::_M_insert_int<long>(std::ostreambuf_iterator<char,
     std::char_traits<char> >, std::ios_base&, char, long)
     const@@GLIBCXX_3.4
      1    0.8%    1.4%  pthread_cond_signal@@GLIBC_2.3.2
      1    0.8%    1.4%  isprint@@GLIBC_2.2.5
      1    0.8%    1.4%  __munmap@@GLIBC_PRIVATE
      1    0.8%    1.4%  __libc_malloc@@GLIBC_2.2.5

 [Summary]:
   ticks  total  nonlib   name
      4    3.0%    5.8%  JavaScript
     65   49.2%   94.2%  C++
      7    5.3%   10.1%  GC
     63   47.7%          Shared libraries

 [C++ entry points]:
   ticks    cpp   total   name
     13   31.0%    9.8%  std::basic_ostream<char, std::char_traits<char>
```

Рис. 7.1 – Отчет Node Profiler

При нормальной нагрузке процесс находится преимущественно в состоянии ожидания (Idle) или выполняет внутренние операции C++, так что нагрузка на JavaScript-код минимальна.

3.2.2 Анализ с Runtime-ошибками

В каждый проект было внесено по три ошибки, которые невозможно обнаружить статическим анализом, так как синтаксически код остается верным. Список ошибок в python-проекте (main.py):

- Утечка памяти: Внутри цикла обработки данных добавлен список leak_list, который бесконтрольно заполняется крупными строками без очистки.
- Проблема производительности: Добавлена искусственная задержка time.sleep(5) при обработке.
- Ошибка логики: Добавлено условие, при котором происходит деление на ноль, если случайное число меньше 76.

Рис. 12 – Обновленный код main.py

```
(.venv) root@2724045-pl17751:~/pyproj/tivpo/pr4# python main.py
Traceback (most recent call last):
  File "/root/pyproj/tivpo/pr4/main.py", line 32, in <module>
    process_students()
  File "/root/pyproj/tivpo/pr4/.venv/lib/python3.12/site-packages/memory_profiler.py", line 1188, in wrapper
    val = prof(func)(*args, **kwargs)
          ^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "/root/pyproj/tivpo/pr4/.venv/lib/python3.12/site-packages/memory_profiler.py", line 761, in f
    return func(*args, **kwds)
           ^^^^^^^^^^^^^^^^^^^
  File "/root/pyproj/tivpo/pr4/main.py", line 26, in process_students
    results.append(calculate_grade(s))
                   ^^^^^^^^^^^^^^^^^^
  File "/root/pyproj/tivpo/pr4/main.py", line 9, in calculate_grade
    else: return score / 0 * "C"
                 ~~~~~~~~^~~
ZeroDivisionError: division by zero
(.venv) root@2724045-pl17751:~/pyproj/tivpo/pr4#
```

Рис. 13 – Результат вызова memory_profiler

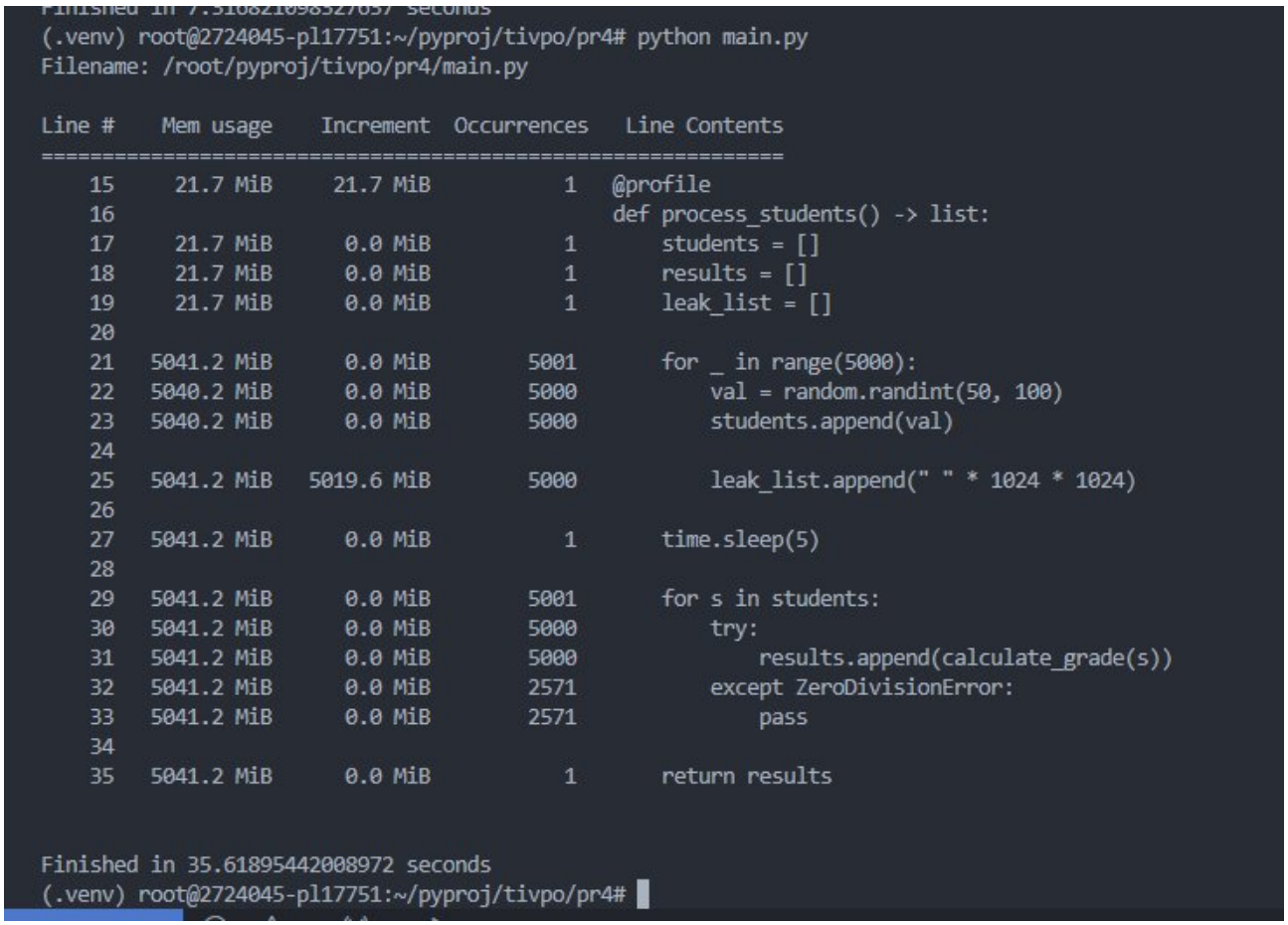
Программа аварийно завершилась с ошибкой ZeroDivisionError, и таблица памяти не была сформирована. Поэтому в код была добавлена минимальная обработка исключений (Рис. 14).

main.py > process_students

```python
import random
import time
from memory_profiler import profile


def calculate_grade(score):
    if score > 90:
        return "A"
    elif score > 75:
        return "B"
    else:
        return score / 0 * "C"


@profile
def process_students() -> list:
    students = []
    results = []
    leak_list = []

    for _ in range(5000):
        val = random.randint(50, 100)
        students.append(val)
        leak_list.append(" " * 1024 * 1024)

    time.sleep(5)

    for s in students:
        try:
            results.append(calculate_grade(s))
        except ZeroDivisionError:
            pass

    return results


if __name__ == "__main__":
    start_time = time.time()
    process_students()
    print(f"Finished in {time.time() - start_time} seconds")
```

Рис. 14 – Код main.py с обработкой ошибок

После этого изменения код был запущен повторно и таблица была отрисована.



```
Finished in 7.516821698327657 seconds
(.venv) root@2724045-pl17751:~/pyproj/tivpo/pr4# python main.py
Filename: /root/pyproj/tivpo/pr4/main.py

Line #    Mem usage    Increment  Occurrences   Line Contents
=============================================================
    15     21.7 MiB     21.7 MiB           1   @profile
    16                                         def process_students() -> list:
    17     21.7 MiB      0.0 MiB           1       students = []
    18     21.7 MiB      0.0 MiB           1       results = []
    19     21.7 MiB      0.0 MiB           1       leak_list = []
    20
    21   5041.2 MiB      0.0 MiB        5001       for _ in range(5000):
    22   5040.2 MiB      0.0 MiB        5000           val = random.randint(50, 100)
    23   5040.2 MiB      0.0 MiB        5000           students.append(val)
    24
    25   5041.2 MiB   5019.6 MiB        5000           leak_list.append(" " * 1024 * 1024)
    26
    27   5041.2 MiB      0.0 MiB           1       time.sleep(5)
    28
    29   5041.2 MiB      0.0 MiB        5001       for s in students:
    30   5041.2 MiB      0.0 MiB        5000           try:
    31   5041.2 MiB      0.0 MiB        5000               results.append(calculate_grade(s))
    32   5041.2 MiB      0.0 MiB        2571           except ZeroDivisionError:
    33   5041.2 MiB      0.0 MiB        2571               pass
    34
    35   5041.2 MiB      0.0 MiB           1       return results


Finished in 35.61895442008972 seconds
(.venv) root@2724045-pl17751:~/pyproj/tivpo/pr4#
```

Рис. 15 – Результат работы memory_profiler

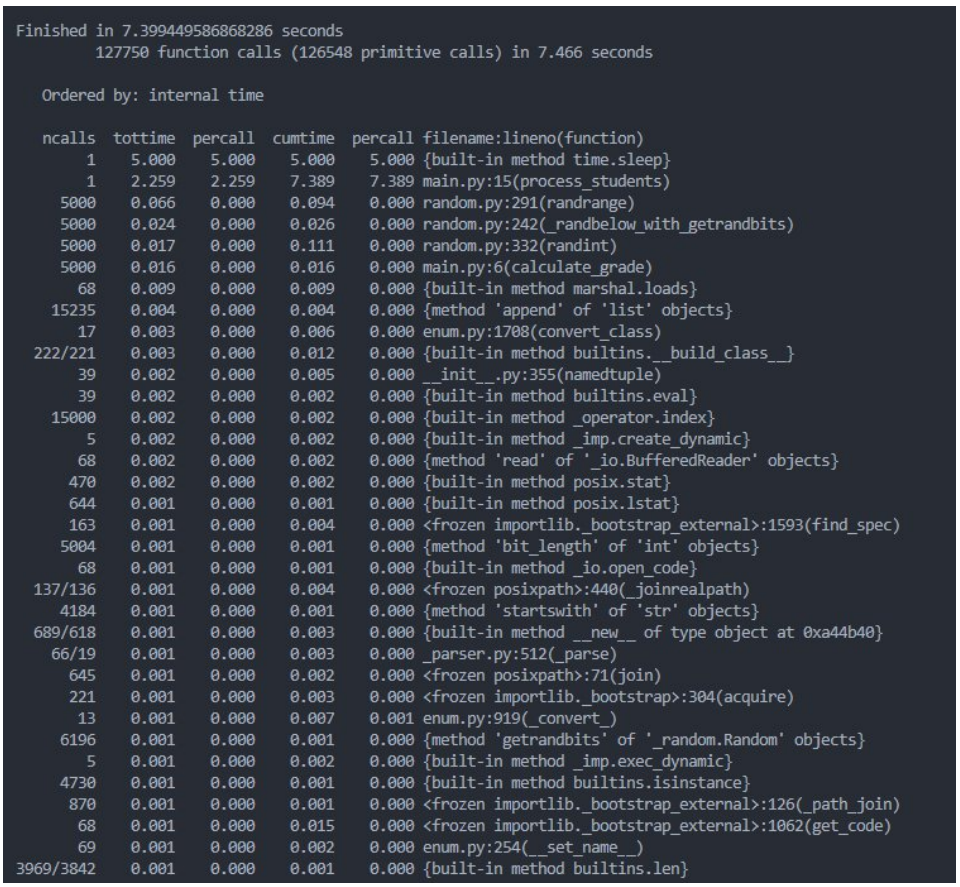
Анализ потребления памяти:

Инструмент зафиксировал критическую утечку памяти.

В начале работы скрипт потреблял всего 21.7 MiB. К концу выполнения цикла потребление выросло до 5041.2 MiB (более 5 ГБ).

Колонка Increment точно указала на строку №25 (leak_list.append), которая в одиночку увеличила потребление памяти на 5019.6 MiB.

Вывод: Приложение бесконтрольно расходует ресурсы, что в реальной среде (на сервере с ограниченной памятью) привело бы к падению процесса за считанные секунды.



```
Finished in 7.399449586868286 seconds
         127750 function calls (126548 primitive calls) in 7.466 seconds

   Ordered by: internal time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    5.000    5.000    5.000    5.000 {built-in method time.sleep}
        1    2.259    2.259    7.389    7.389 main.py:15(process_students)
     5000    0.066    0.000    0.094    0.000 random.py:291(randrange)
     5000    0.024    0.000    0.026    0.000 random.py:242(_randbelow_with_getrandbits)
     5000    0.017    0.000    0.111    0.000 random.py:332(randint)
     5000    0.016    0.000    0.016    0.000 main.py:6(calculate_grade)
       68    0.009    0.000    0.009    0.000 {built-in method marshal.loads}
    15235    0.004    0.000    0.004    0.000 {method 'append' of 'list' objects}
       17    0.003    0.000    0.006    0.000 enum.py:1708(convert_class)
  222/221    0.003    0.000    0.012    0.000 {built-in method builtins.__build_class__}
       39    0.002    0.000    0.005    0.000 __init__.py:355(namedtuple)
       39    0.002    0.000    0.002    0.000 {built-in method builtins.eval}
    15000    0.002    0.000    0.002    0.000 {built-in method _operator.index}
        5    0.002    0.000    0.002    0.000 {built-in method _imp.create_dynamic}
       68    0.002    0.000    0.002    0.000 {method 'read' of '_io.BufferedReader' objects}
      470    0.002    0.000    0.002    0.000 {built-in method posix.stat}
      644    0.001    0.000    0.001    0.000 {built-in method posix.lstat}
      163    0.001    0.000    0.004    0.000 <frozen importlib._bootstrap_external>:1593(find_spec)
     5004    0.001    0.000    0.001    0.000 {method 'bit_length' of 'int' objects}
       68    0.001    0.000    0.001    0.000 {built-in method _io.open_code}
  137/136    0.001    0.000    0.004    0.000 <frozen posixpath>:440(_joinrealpath)
     4184    0.001    0.000    0.001    0.000 {method 'startswith' of 'str' objects}
  689/618    0.001    0.000    0.003    0.000 {built-in method __new__ of type object at 0xa44b40}
    66/19    0.001    0.000    0.003    0.000 _parser.py:512(_parse)
      645    0.001    0.000    0.002    0.000 <frozen posixpath>:71(join)
      221    0.001    0.000    0.003    0.000 <frozen importlib._bootstrap>:304(acquire)
       13    0.001    0.000    0.007    0.001 enum.py:919(_convert_)
     6196    0.001    0.000    0.001    0.000 {method 'getrandbits' of '_random.Random' objects}
        5    0.001    0.000    0.002    0.000 {built-in method _imp.exec_dynamic}
     4730    0.001    0.000    0.001    0.000 {built-in method builtins.isinstance}
      870    0.001    0.000    0.001    0.000 <frozen importlib._bootstrap_external>:126(_path_join)
       68    0.001    0.000    0.015    0.000 <frozen importlib._bootstrap_external>:1062(get_code)
       69    0.001    0.000    0.002    0.000 enum.py:254(__set_name__)
3969/3842    0.001    0.000    0.001    0.000 {built-in method builtins.len}
```

Рис. 16 – Результат работы cProfile

Анализ времени выполнения:

Инструмент зафиксировал критическое замедление работы приложения.

Общее время выполнения скрипта составило 7.466 секунды (против 0.02 сек в базовой версии).

Сортировка по времени (tottime) показала причину: вызов {built-in method time.sleep} занял ровно 5 секунд.

Вывод: Профилировщик позволяет найти узкие места в коде. В данном случае искусственная задержка стала основным фактором деградации производительности, занимая около 67% всего времени работы программы.

Заключение по Части 2:

Динамический анализ позволил выявить критические проблемы эксплуатации: cProfile и замеры времени отклика позволили найти узкие места, замедляющие работу системы в сотни раз.

memory_profiler позволил визуализировать утечку памяти, которая привела бы к отказу сервера.

Эксперимент показал, что отсутствие обработки ошибок (try-except/try-catch) приводит к полному падению сервисов.

Во второй проект также были внесены 3 ошибки времени выполнения.
Список ошибок в JavaScript-проекте (app.js):

- Утечка памяти: Добавлен глобальный массив leak, который при каждом запросе заполняется крупными строковыми данными без последующей очистки.
- Проблема производительности: В обработчик запроса добавлена искусственная задержка через синхронный цикл while, блокирующая основной поток (Event Loop).
- Критический сбой: Добавлен маршрут /crash, при обращении к которому происходит попытка чтения свойства у null (TypeError), приводящая к падению сервера.

Рис. 8.1 – Модифицированный app.js

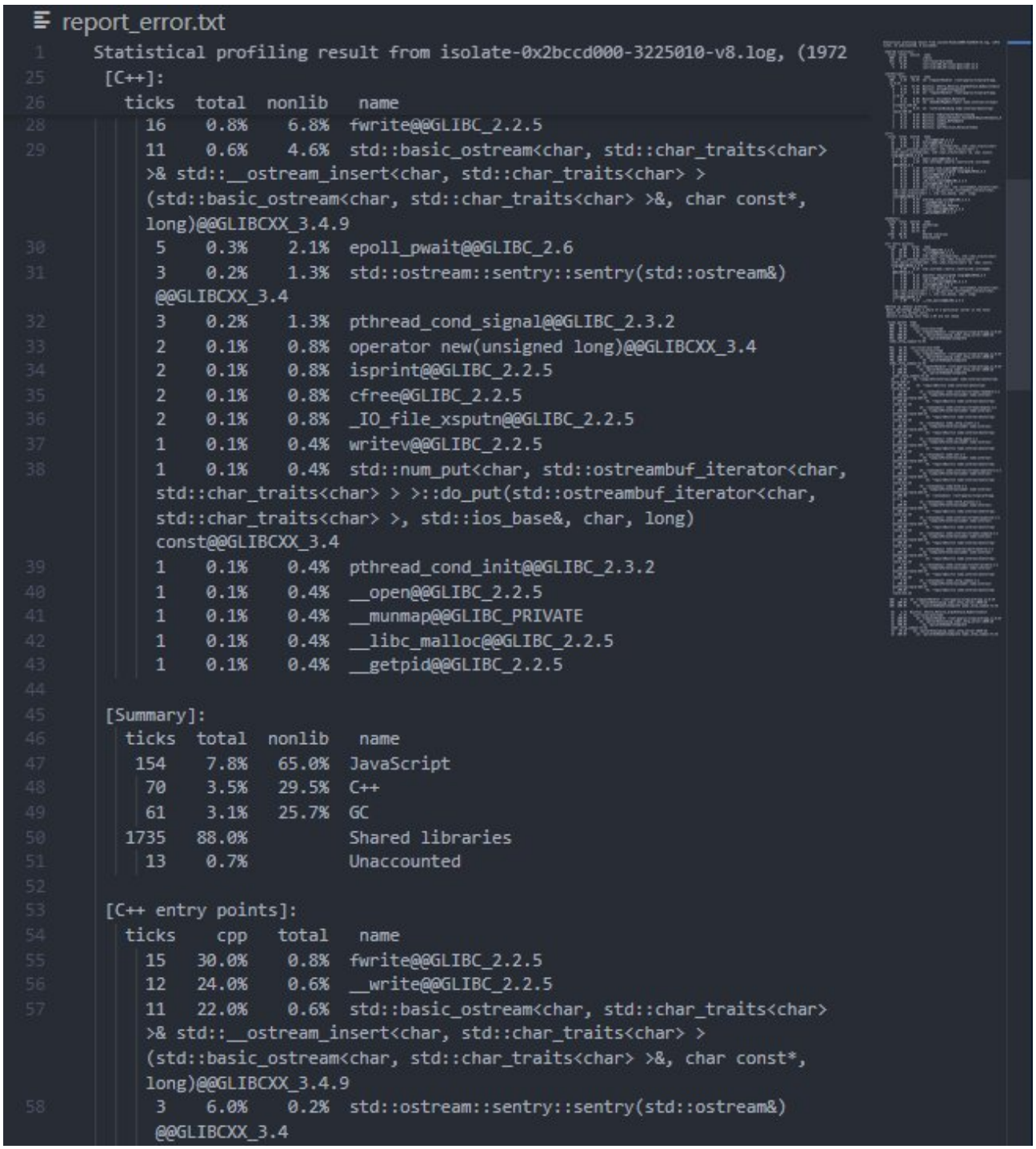
Результаты динамического анализа:
Clinic.js Doctor – Инструмент зафиксировал критическую деградацию всех показателей системы (Рис. 9.1).

Анализ производительности (Блокировка):
График Event Loop Delay показывает катастрофический скачок задержки до 20,000 мс (20 секунд). Это означает полную блокировку сервера: он не мог обрабатывать никакие входящие запросы.
График CPU Usage показывает загрузку процессора 100%. Это подтверждает наличие синхронного бесконечного цикла, который монополизировал поток выполнения.
Анализ памяти (Утечка):
График Memory Usage демонстрирует рост потребления памяти. Линия идет строго вверх без спадов, что свидетельствует о том, что сборщик мусора не может освободить память из-за постоянного добавления данных в глобальный массив.
Вывод: Комбинация блокировки процессора и утечки памяти привела сервер в неработоспособное состояние за считанные секунды.

Рис. 9.1 – Отчёт Clinic.js

Node.js Profiler – Текстовый отчет подтверждает высокую нагрузку на процесс.

- Зафиксирован резкий рост количества тиков исполнения JavaScript (с 4 до 154), что указывает на выполнение тяжелого кода.
- Появление активности GC (Garbage Collector — 3.1%) подтверждает наличие проблем с памятью: система пытается (безуспешно) очистить кучу от постоянно добавляемых данных.
- Высокая нагрузка на Shared libraries (88%) обусловлена операциями выделения памяти под большие массивы строк внутри цикла утечки.



report_error.txt

```
Statistical profiling result from isolate-0x2bccd000-3225010-v8.log, (1972
 [C++]:
   ticks  total  nonlib   name
     16    0.8%    6.8%  fwrite@@GLIBC_2.2.5
     11    0.6%    4.6%  std::basic_ostream<char, std::char_traits<char>
     >& std::__ostream_insert<char, std::char_traits<char> >
     (std::basic_ostream<char, std::char_traits<char> >&, char const*,
     long)@@GLIBCXX_3.4.9
      5    0.3%    2.1%  epoll_pwait@@GLIBC_2.6
      3    0.2%    1.3%  std::ostream::sentry::sentry(std::ostream&)
      @@GLIBCXX_3.4
      3    0.2%    1.3%  pthread_cond_signal@@GLIBC_2.3.2
      2    0.1%    0.8%  operator new(unsigned long)@@GLIBCXX_3.4
      2    0.1%    0.8%  isprint@@GLIBC_2.2.5
      2    0.1%    0.8%  cfree@GLIBC_2.2.5
      2    0.1%    0.8%  _IO_file_xsputn@@GLIBC_2.2.5
      1    0.1%    0.4%  writev@@GLIBC_2.2.5
      1    0.1%    0.4%  std::num_put<char, std::ostreambuf_iterator<char,
      std::char_traits<char> > >::do_put(std::ostreambuf_iterator<char,
      std::char_traits<char> >, std::ios_base&, char, long)
      const@@GLIBCXX_3.4
      1    0.1%    0.4%  pthread_cond_init@@GLIBC_2.3.2
      1    0.1%    0.4%  __open@@GLIBC_2.2.5
      1    0.1%    0.4%  __munmap@@GLIBC_PRIVATE
      1    0.1%    0.4%  __libc_malloc@@GLIBC_2.2.5
      1    0.1%    0.4%  __getpid@@GLIBC_2.2.5

 [Summary]:
   ticks  total  nonlib   name
    154    7.8%   65.0%  JavaScript
     70    3.5%   29.5%  C++
     61    3.1%   25.7%  GC
   1735   88.0%          Shared libraries
     13    0.7%          Unaccounted

 [C++ entry points]:
   ticks    cpp   total   name
     15   30.0%    0.8%  fwrite@@GLIBC_2.2.5
     12   24.0%    0.6%  __write@@GLIBC_2.2.5
     11   22.0%    0.6%  std::basic_ostream<char, std::char_traits<char>
     >& std::__ostream_insert<char, std::char_traits<char> >
     (std::basic_ostream<char, std::char_traits<char> >&, char const*,
     long)@@GLIBCXX_3.4.9
      3    6.0%    0.2%  std::ostream::sentry::sentry(std::ostream&)
      @@GLIBCXX_3.4
```

Рис. 10.1 – Отчёт Node.js Profiler

Также был проверен маршрут /crash, содержащий код обращения к свойству переменной со значением null.

Был отправлен HTTP-запрос на эндпоинт /crash, после чего сервер не вернул HTTP-ответ (500 Internal Server Error), а аварийно завершил работу (процесс Node.js остановился). В консоли был выведен Stack Trace ошибки TypeError.

Рис. 11.1 – Клиент не получил ответа от сервера

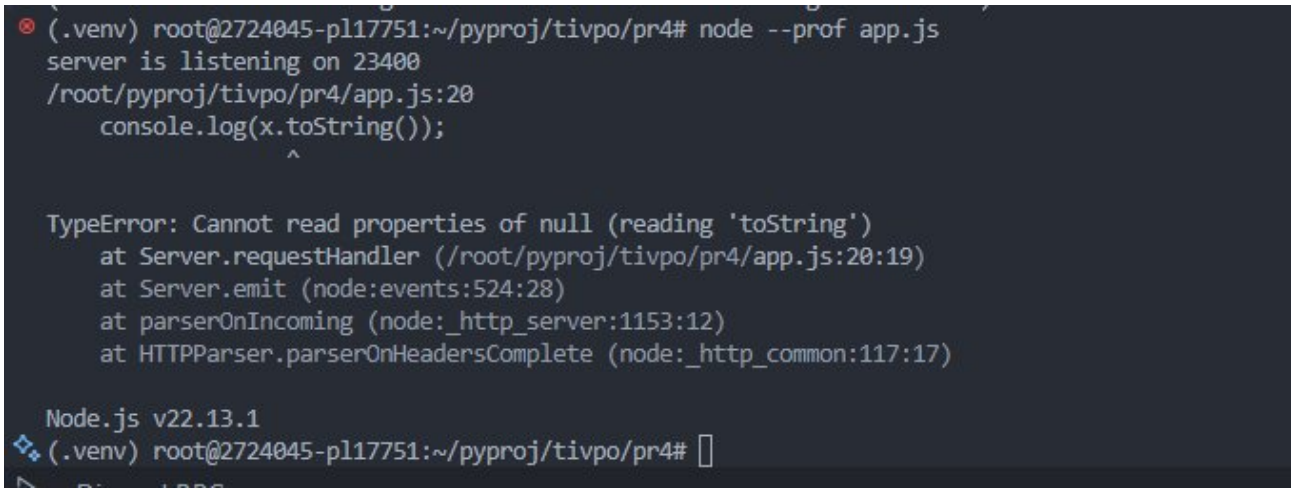


Рис. 11.2 – Полученная ошибка на сервере

Эксперимент показал, что в однопоточной среде Node.js любое необработанное исключение (Uncaught Exception) приводит к падению всего процесса. Это означает полный отказ в обслуживании для всех пользователей до момента ручного перезапуска сервера.

4. Заключение

В ходе выполнения практической работы были изучены и применены на практике методы статического и динамического анализа кода. Эксперимент проводился на проектах Python и JavaScript/Node.js с использованием широкого спектра инструментов (Pylint, ESLint, cProfile, memory_profiler, Clinic.js).

Инструменты *статического* анализа (Pylint, ESLint) являются эффективным средством "первой линии обороны". Они безупречно выявляют нарушения стандартов кодирования (PEP8), опечатки и потенциальные проблемы типизации (Муру).

Однако эксперимент показал критическое ограничение: синтаксическая корректность является строгим требованием. При наличии грубых синтаксических ошибок (например, пропущенное двоеточие) анализаторы аварийно завершают работу, не позволяя проверить остальные аспекты качества кода. Это подтверждает необходимость исправления синтаксиса до запуска глубоких проверок.

Динамический же анализ позволил выявить фатальные проблемы, которые были абсолютно невидимы для *статических* анализаторов, так как код был синтаксически верен:

- Инструмент memory_profiler обнаружил критическую утечку памяти (рост до 5 ГБ), которая привела бы к отказу сервера.
- Инструмент cProfile точно локализовал "узкое место" производительности (искусственную задержку), занимавшую большую часть времени ЦП.
- Только запуск кода позволил выявить отсутствие обработки исключений (ZeroDivisionError), приводящее к падению приложения.

Сравнение инструментов:

Статический анализ не требует запуска и покрывает 100% кодовой базы, но пропускает логические бомбы и утечки ресурсов.

Динамический анализ дает реальную картину потребления ресурсов (CPU/RAM), но требует наличия качественных тестовых сценариев, покрывающих проблемные участки кода.

Итоговые выводы:

Ни один из методов не является исчерпывающим по отдельности. Для обеспечения высокого качества и надежности программного обеспечения необходимо внедрение гибридного подхода в процессы CI/CD:

- Проверка синтаксиса и стиля (Linter).
- Статический анализ типов и безопасности (SAST).
- Запуск профилировщиков и нагрузочных тестов (DAST) для выявления утечек и деградации производительности перед релизом.