Московский авиационный институт
(национальный исследовательский университет)

Институт № 8 «Информационные технологии и прикладная математика»

Кафедра вычислительной математики и программирования

Лабораторная работа №3 по курсу «Дискретный анализ»

Студент: К.С. Саженов
Преподаватель: Н. С. Капралов
Группа: М8О-208Б-19
Дата:
Оценка:
Подпись:

Москва 2020

Лабораторная работа №3

Задача: Для реализации словаря из предыдущей лабораторной работы, необходимо провести исследование скорости выполнения и потребления оперативной памяти. В случае выявления ошибок или явных недочётов, требуется их исправить.

1. Введение

Разработка современного ПО включает в себя большое количество различных процессов. Один из таких процессов, который помогает понять, какое конкретное место является, т. н., bottleneck(англ. горлышко бутылки)— узким местом программы, является анализ производительности программ, а именно profiling (англ. Профилировние) — позволяет узнать, какие участки кода занимают больше всего процессорного/пользовательского времени, coverage (англ. покрытие) — позволяет узнать, какие части кода сколько раз выполняются, а также memcheck(англ. Проверка памяти) — позволяет определить утечки памяти. На примере программы из предыдущей лабораторной работы — реализация словаря с использованием PATRICIA Trie — я проведу детальный анализ производительности данной программы.

2. Анализ времени исполнения

Valgrind(callgrind) & kcachegrind

При выборе утилиты я хотел найти наиболее многофункциональную и удобную для пользования новичкам программу. Эти два критерия совместили в себе две программы соответственно — valgrind и *kcachegrind*(содержится в среде рабочего стола KDE). Valgrind включает в себя несколько утилит(некоторые из них будут рассмотрены далее), одной из которых является *callgrind*.

Используя утилиту *kcachegrind*, данный симбиоз программ позволяет крайне детально оценивать скорость выполнения тех или иных мест в программе. Одними из основных полезных функций для визуализации вызовов в данной графической утилиты является построение графов вызовов и карты вызовов.

Первая позволяет проследить вызовы, а вторая оценить время выполнения каждого вызова.

Однако стоит понимать, что указанную программу следует компилировать с флагом компиляции(-g)

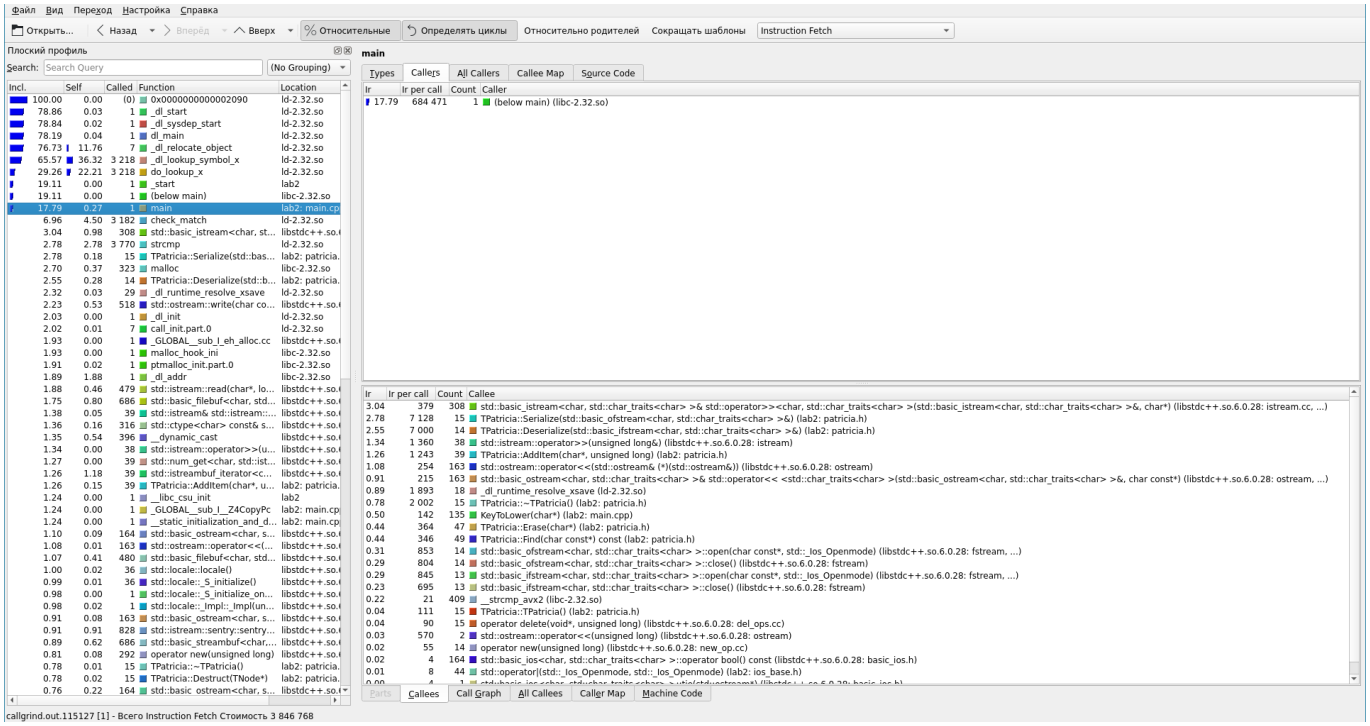
Запустим профайлер следующим образом:

```
╭─sakost@sakost-pc ~/university/2 course/diskran/lab2
╰─$ valgrind --tool=callgrind --dump-instr=yes --simulate-cache=yes --collect-jumps=yes cmake-build-debug/lab2 < data.txt
```

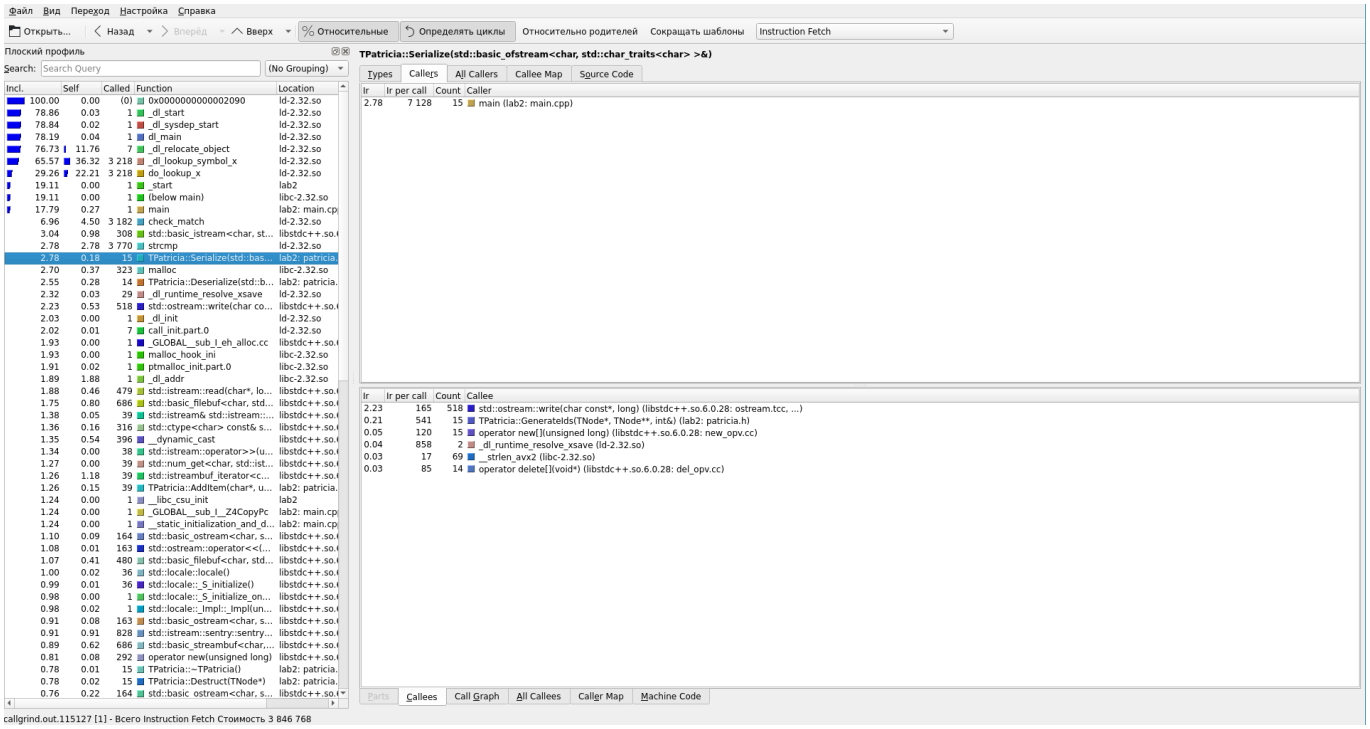
Сгенерировался файл callgrind.out.XXX – откроем его:

```
╭─sakost@sakost-pc ~/university/2 course/diskran/lab2
╰─$ kcachegrind callgrind.out.115127
```

Наблюдаем интерфейс программы:



Слева отображается таблица функций, отранжированных по умолчанию по времени выполнения по убыванию. Нас интересуют функции, написанные нами — возможно, анализ данных функций, позволит нам понять, каким образом мы можем оптимизировать нашу программу. Выбрав одну из функций, например TPatricia::Serialize, справа можно наблюдать целый набор инструментов, предоставляющий подробную информацию о ней:

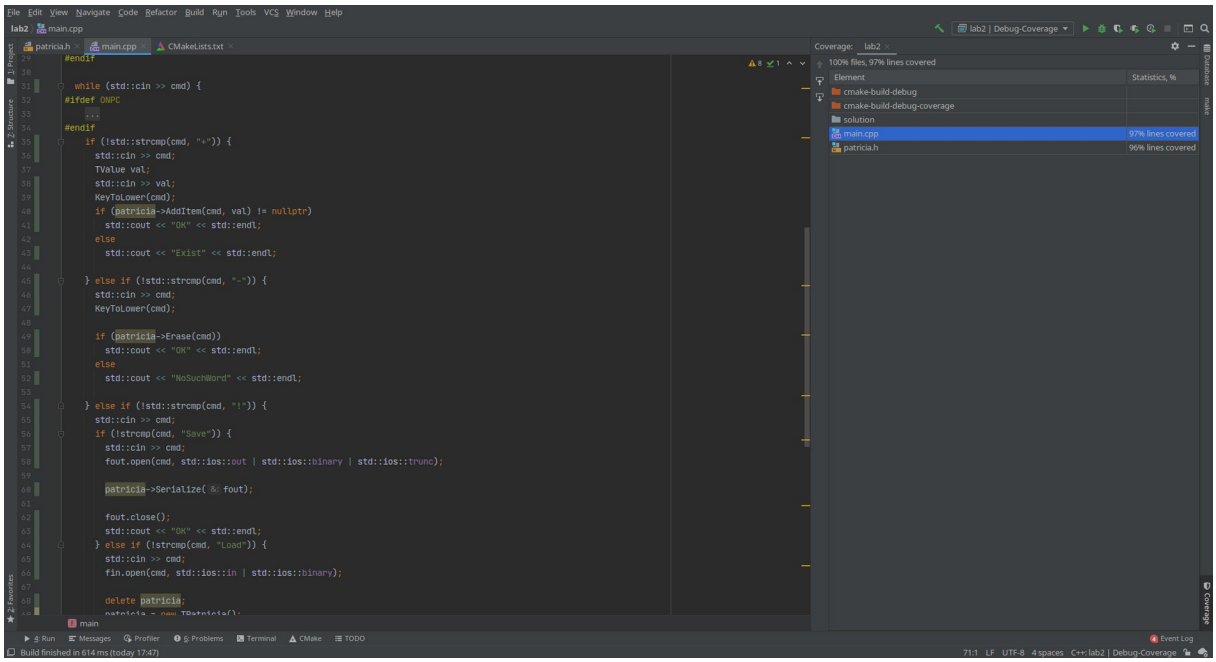


Одни из самых хорошо интерпретируемых полей здесь, как и упоминалось выше, callee map и call graph. Судя по данным графикам, наиболее «долгая» функция — встроенная функция записи в файл. Её оптимизировать, к сожалению, на данном этапе моего обучения не предоставляется возможным.

3. Анализ покрытия кода

1. gcov

Одной из основной программой для проверки кода на покрытие в IDE CLion является утилита gcov. При помощи среды разработки можно в режиме написания кода просмотреть, какие строки сколько раз исполнялись(красные — 0, желтые — частично вызывается, зеленые — исполняется постоянно):



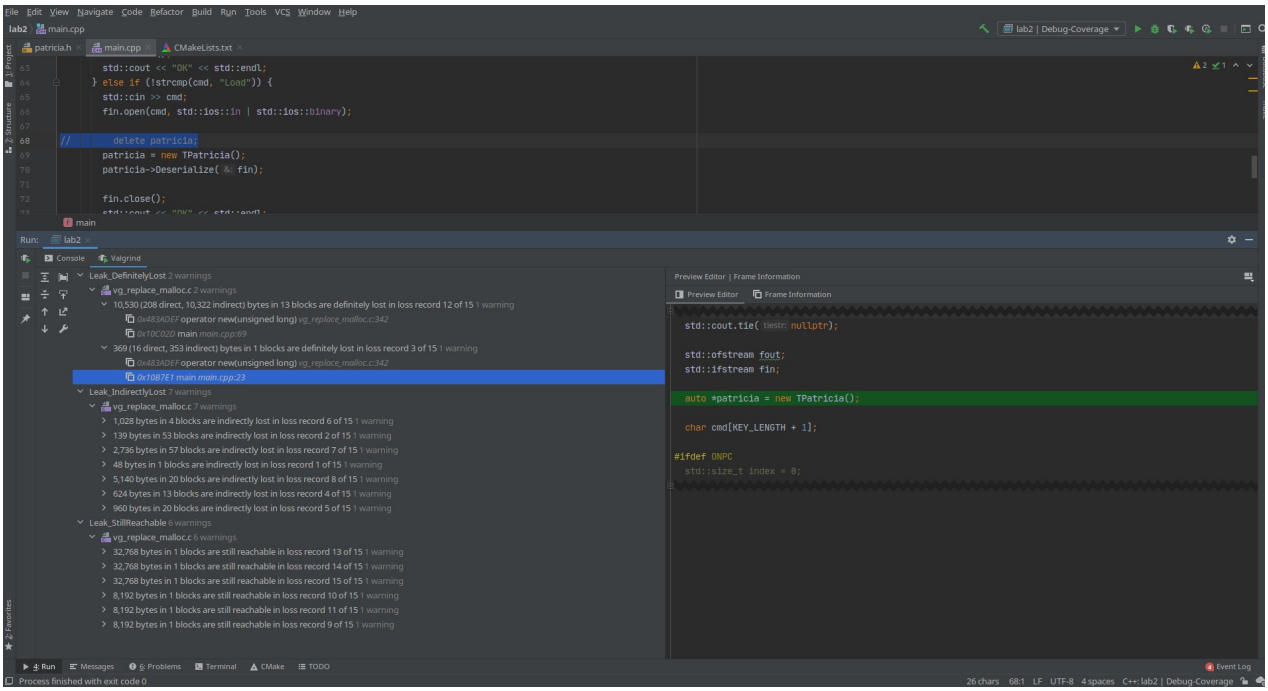
Делаем вывод, что в нашем тесте обрабатываются все возможные случаи, которые возможны на чекере. Следовательно, если ошибок нет, значит наша программа работает корректно для всех случаев входных данных.

4. Анализ потребления памяти

1. Valgrind

В основном мире C/C++ для отслеживания утечек памяти используется такая утилита, как Valgrind(memcheck). Она содержит в себе, как и упоминалось выше, несколько инструментов для анализа программ и memcheck — ещё один из них. Valgrind(memcheck) является довольно надежным инструментом, указывающий иногда на, порой, незначительные ошибки, которые даже не приводят к утечкам(но потенциально могли бы)

Интерфейс CLion, который использует valgrind memcheck в качестве инструмента для анализа утечек памяти(я закомментировал выделенный код — будет явная утечка):



Данная утилита верно определила какая часть памяти не освободилась(а за ней ещё несколько, т. к. при удалении данного объекта освобождаются и другие области памяти)

2. **heaptrack**

Имеет схожий с valgrind memcheck функционал и интуитивно понятный интерфейс. Чтобы просмотреть данные по программе, нужно её сначала отдать консольной версии программы:

```
┌─sakost@sakost-pc ~/university/2 course/diskran/lab2
└─$ heaptrack cmake-build-debug/lab2 < data.txt
```

…

/build/heaptrack/src/heaptrack-1.2.0/src/interpret/heaptrack_interpret.cpp:351

ERROR:Failed to create backtrace state for module /home/sakost/university/2: /home/sakost/university/2 / No such file or directory (error code 2)

heaptrack stats:

allocations: 323

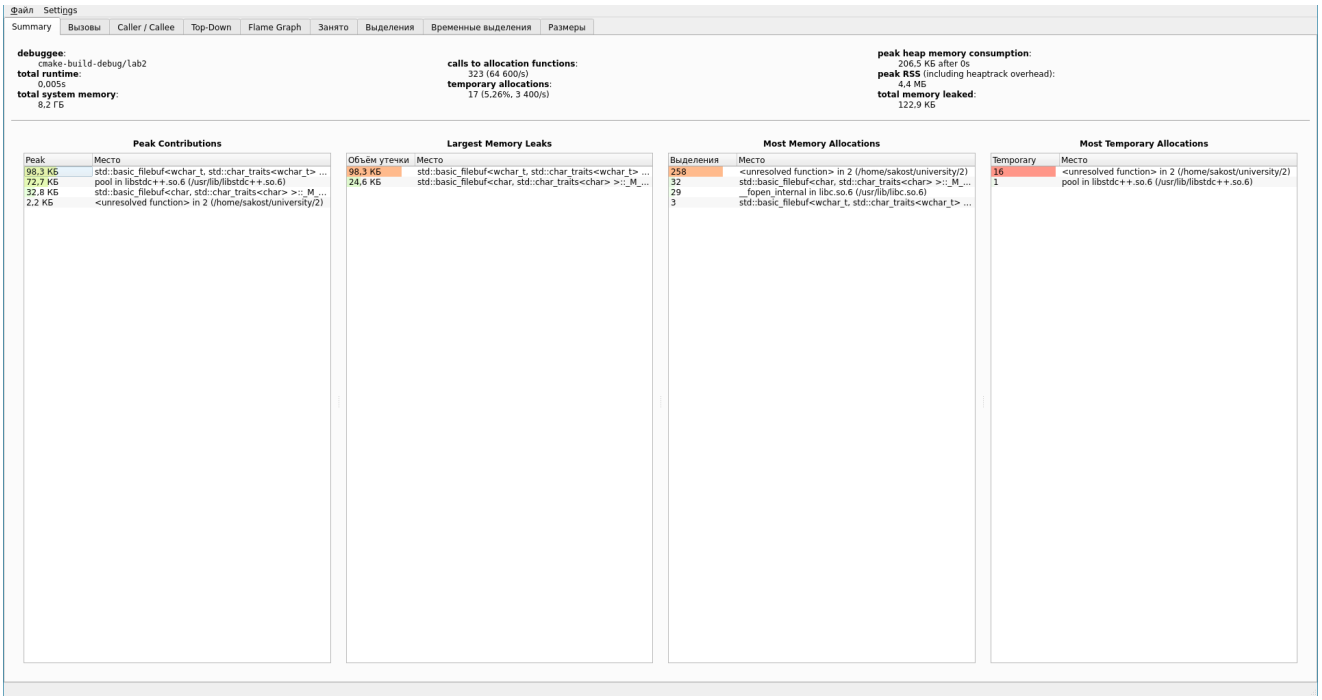
leaked allocations: 6

temporary allocations: 15

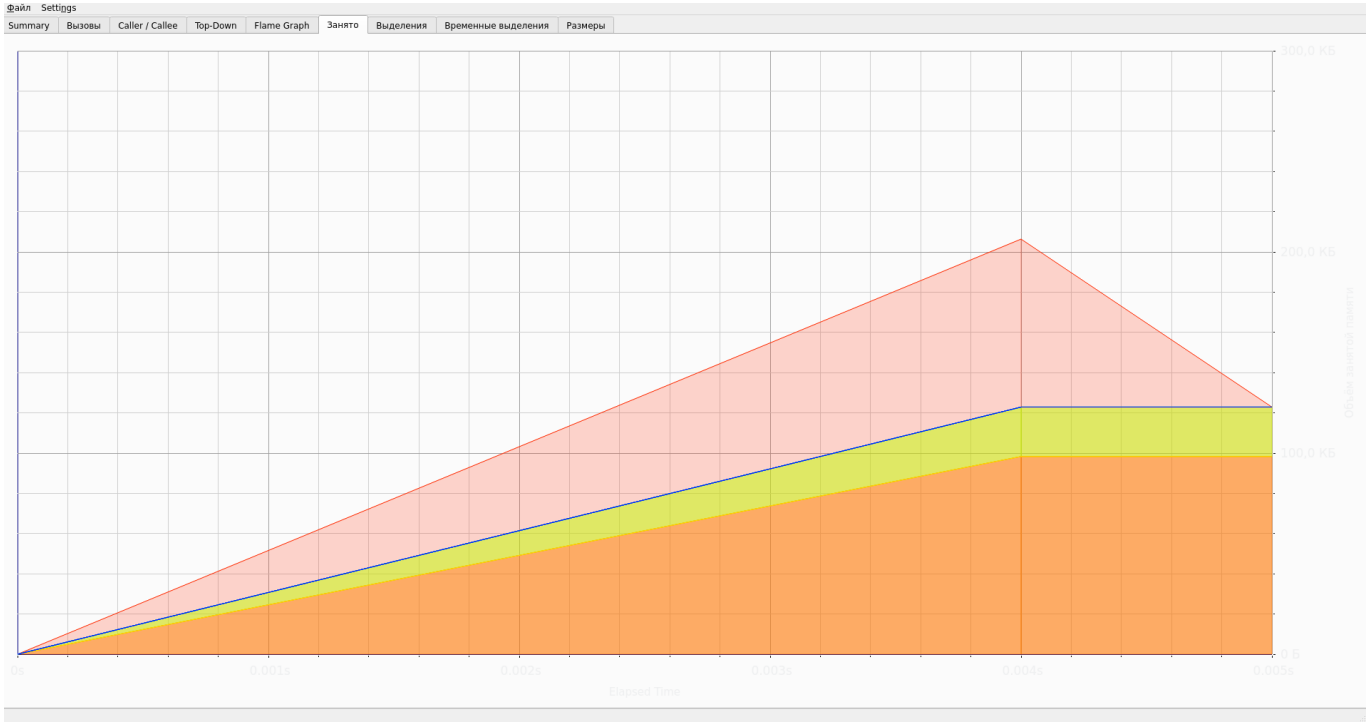
Heaptrack finished! Now run the following to investigate the data:

heaptrack --analyze "/home/sakost/university/2 course/diskran/lab2/heaptrack.lab2.177357.zst"

И запускаем то, что нам выдала консольная утилита. Запустилась графическа версия программы, вот её интерфейс:



Тут также можно посмотреть красивые графики:



5. Вывод

В этой лабораторной работе я проанализировал код программы и саму программу, которую я написал в прошлой лабораторной работе. Я изучил несколько способов анализа поведения программы, несколько утилит, помогающих выловит те или иные ошибки. Как мне показалось, самым функциональным и одним из самых простых в использовании инструментов оказался valgrind. Также, попробовав поизучать heaptrack, я понял, что этот инструмент также заслуживает внимания программистов, хоть он и не является тем инструментом, которым в основном привыкли использовать программисты.

Борьба с ошибками, утечками и багами — это неотъемлемая часть разработки практически любого ПО. Ни одна сложная в написании программа, используемая в повседневной жизни людьми, не была написана сразу без ошибок и для решения проблем использовались, в частности, инструменты, изученные мной. Данные утилиты помогут мне в будущем более оперативно и с меньшими усилиями находить критические части программ.