Инструменты для разработки на C++

Отладка, профилирование, утечки памяти, статический анализ

Мотивирующая цитата

As soon as we started programming, we found to our surprise that it wasn’t as easy to get programs right as we had thought. We had to discover debugging. I can remember the exact instant when I realized that a large part of my life from then on was going to be spent in finding mistakes in my own programs.

— Maurice Wilkes, 1949

Код для семинара

https://drive.google.com/folderview?id=0B-KXDlig5kwVeDVpZE9vV1N5bnM&usp=sharing

Жизненный цикл разработчика и бага

1. Кем-то замечена неправильная работа программы
2. Разработчик собирает информацию о баге (**как?**)
3. Разработчик делает предположение о причине возникновения бага
4. Разработчик проверяет предположение (**как?**)
5. Если проверка успешна, то можно править баг, если нет то сделать выводы перейти к (2)
6. Пишется код для исправления бага
7. ...

Основные способы сбора информации о программе

- Исполнение ручных/авто тестов
- Логирование
- Счетчики
- Трассировка (DTrace и аналоги)
- Проверки времени исполнения
- Core dumps
- Отчеты об ошибках программы
- Профилирование
- Изучение программы в отладчике
- Статический анализ кода
- Чтение кода

Исполнение ручных/авто тестов

- Тесты позволяют выбрать подсистемы/модули/классы для исследования причины возникновения бага (локализация)
- В идеале сломанный тест позволяет точно восстановить алгоритм работы программы при выполнении теста и найти неверный code path
- Тест, дающий стабильное воспроизведение бага - ключ к его исправлению, т.к. можно подвергать программу самым разным проверкам во время прогона теста

Логирование

- Логи работы приложения позволяют проследить работу программы во времени без изменения времени исполнения отдельных ее участков
- Пример хорошей записи в логе:
  - Дата,время,id/имя процесса,id потока,подсистема,функция,строка: информационное сообщение, достаточно полно описывающее логируемое событие
- Каждая запись в лог обычно сопровождается тегом, обозначающим уровень ее важности. Пример: DEBUG,INFO,WARNING,ERROR,ALERT
- При чтении лога можно фильтровать записи в зависмости от тега, даты, процесса, потока и т.д.

Счетчики

- На определенных code paths программа инкрементирует глобальный счетчик, отвечающий за одну из метрик программы
- Примеры:
  - Суммарное время исполнения HTTP запросов каждым тредом
  - Суммарное время ожидания ответа от базы данных каждым тредом
  - Кол-во исполнений code path X
  - Кол-во исполнений slow paths в code path X
  - Кол-во рестартов бэкенда
  - Счетчики времени обработки запросов (для гистограммы)
- Счетчики используются при изучении поведения программы, оптимизации производительности, изучении зависаний, паттернов использования продукта

Трассировка

Похожа на логирование. Сложно сформулировать отличия.

- **strace** - лог системных вызовов, выполняющихся процессом
- **ltrace** - лог вызовов в динамические библиотеки
- **DTrace** - подключение обработчиков к точкам трассировки и выполнение произвольных операций с данными, поставляемыми этими точками
- tcpdump - лог определенных типов пакетов из сетевого стека
- **BPF** - userspace фреймворк для чтения и модификации сетевых пакетов в ядре Linux. Обработчики пакетов JIT-компилируются в ядре.

Много других инструментов

Проверки времени исполнения

- Проверки, выполняющиеся всегда (самодиагностика, sanity checks)
- Проверки, выполняющиеся только в отладочной сборке программы (используемой для разработки, тестирования)
- При срабатывании проверки можно падать - можно не падать. В production обычно стараются не падать. Ругаются в лог и возвращают ошибку
- Проверки времени исполнения для отладочной сборки в gcc STL https://gcc.gnu.org/onlinedocs/libstdc++/manual/debug_mode_using.html#debug_mode.using.mode

Core dumps

Это файл или несколько файлов, представляющих снимок состояния программы:

- Содержимое памяти программы
- Потоки, их регистры, стеки
- Информация об исполняемом файле и используемых библиотеках для открытия дампа с помощью отладчика и source level отладки
- Системные данные, на хранящиеся в памяти программы, например маска сигналов, маска активных сигналов

Не кросс-платформенны. Но есть Google breakpad.

Отчет об ошибке

Как правило, является архивом, состоящим из:

- Логов приложения
- Счетчиков приложения
- Core minidump приложения
- Важная информаци о приложении - версия сборки, установленные плагины, сериализованное состояние модулей, подсистем, ...
- Важная информаци о системе - ОС, железо, запущенные процессы, вывод top, содержимое реестра, ...

Профилировщики памяти

- В языках с прямым доступом к памяти и без сборки мусора отслеживают:
  - использование непроаллоцированной памяти
  - использование чужой памяти
  - использование непроинициализированной память
  - утечки памяти
  - создаваемые объекты (типы, места создания, кол-во, размер,...)
- В языках без прямого доступа к памяти и со сборкой мусора отслеживают:
  - создаваемые объекты (типы, места создания, кол-во, размер,...)
  - мусор (+наблюдение за поведением сборщика мусора)
  - ссылки между объектами

Valgrind

- Виртуальная машина для исполнения нативных Linux программ
- Имеет возможность выполнять любые преобразования с исходной программой перед ее исполнением
- Не требует вмешательства компилятора для поддержки профилирования (но удобно иметь отладочную информацию -g)
- Производительность программы падает в 20-30 раз, возрастает потребление памяти
- Позволяет подключать плагины (tools), анализирующие исполняющуюся программу

Что отслеживает Valgrind memcheck tool

- Использование непроинициализированной памяти
- Использование динамической памяти после освобождения
- Использование динамической памяти до ее начала/после конца
- Утечки динамической памяти
- Передача непроинициализированной/не выделенной памяти в системный вызов
- Неверные освобождения динамической памяти

Что не отслеживает Valgrind memcheck tool

- Т.к. стеки и область статических данных непрерывны, то проверка границ в них не работает, например, не отслеживаются:
  - Ошибки переполнения буфера
  - Доступ за пределами массива или переменной на стеке (или если статические)
- Ошибки, связанные с прямым доступом к памяти, например:
  - Модификация данных на стеке другого треда
  - Неверная интерпретация участка памяти (reinterpret_cast<T*>(K*))
- То есть Valgrind работает на уровне непрерывных буферов в памяти программы, к которым разрешен прямой доступ

Практика

Воспроизведем все основные проблемы в работе с памятью в Valgrind

valgrind --leak-check=full --track-origins=yes ./bin/mem

valgrind --leak-check=full --track-origins=yes ./bin/memtricky

Как Valgrind ищет memory leak’и

- При завершении программы Valgrind знает обо всех аллокациях и освобождениях памяти
- Для поиска не утекших указателей Valgrind эвристически формирует root set указателей из значений:
  - Регистров общего назначения всех тредов
  - Проинициализированных, выравненных данных размером с машинный указатель в валидной памяти программы (включая стеки, проаллоцированные участки кучи)
- Valgrind находит все проаллоцированные указатели, к которым нет доступа из root set и считает их утекшими

Valgrind LEAK SUMMARY

- Definitely lost: в программе нет ни одного указателя на блок данных
- Indirectly lost: в программе есть указатель на блок данных из definitely lost блока
- Possibly lost: найдена цепочка указателей, ведущая к блоку с данными, но как минимум один из указателей в цепочке указывает в середину следующего блока в цепочке
- Still reachable: указатель на начало блока доступен из root set

Статический анализ, cppcheck

- Работает на уровне исходного кода
- Как правило, делает то, что не может сделать динамический анализ
- Проверяет все code paths
- Может понимать где прямой доступ в проаллоцированную память валиден, а где нет - нужен для поиска проблем, которые не находит Valgrind
- Декларирует, что проверяет правильность использования STL и другое https://sourceforge.net/p/cppcheck/wiki/ListOfChecks/
- Но его легко обмануть, усложнив код
- Компромисс между true positives и false positives

Пробуем cppcheck

sudo apt-get install cppcheck

cppcheck --enable=all ./src/mem.cpp

cppcheck --enable=all ./src/memtricky.cpp

GDB

- Подключается к другим процессам для наблюдения и модификации их состояния (даже к процессам на другом компьютере с другой архитектурой)
- Предназначен для отладки нативных процессов linux. Для языков, исполняющихся в языковых виртуальных машинах используются свои отладчики
- Без отладочной информации позволяет отлаживаться только на уровне ассемблерных инструкций
- Крайне сложно отлаживать оптимизированные программы (-O2)
- Позволяет смотреть содержимое Linux core dump’ов
- DDD - графический фронтенд к GDB

GDB старт отладки

Запускаем процесс сразу под отладчиком

$ gdb ./bin/mem

(gdb) run

Присоединяемся с запущенному процессу

$./bin/profsys &
$ gdb -p $!

GDB: вход, выход

Пауза программы и выход в консоль отладки

ctrl+c

(gdb)

Продолжение исполнения программы

(gdb) continue | c

Выход из отладчика

(gdb) quit | q

Полезные команды

- info threads | info thr
- thread THREAD_IX
- backtrace | bt
- frame STACK_FRAME_IX
- up
- down
- list | l
- info locals
- print CPP_EXPRESSION | p CPP_EXPRESSION

Полезные команды

- b FUNCTION_NAME | b SOURCE_NAME:LINE | b Foo<int>::bar(int)
- i break
- d BREAKPOINT_IX
- d breakpoints

- watch CPP_EXPRESSION
- i watch
- d WATCH_IX

Полезные команды

- i registers
- p $REGISTER_NAME
- x /10i $rip
- disassamble

Полезные команды

- next | n
- nexti | ni
- step | s
- stepi | si

Cheat sheets с коммандами GDB

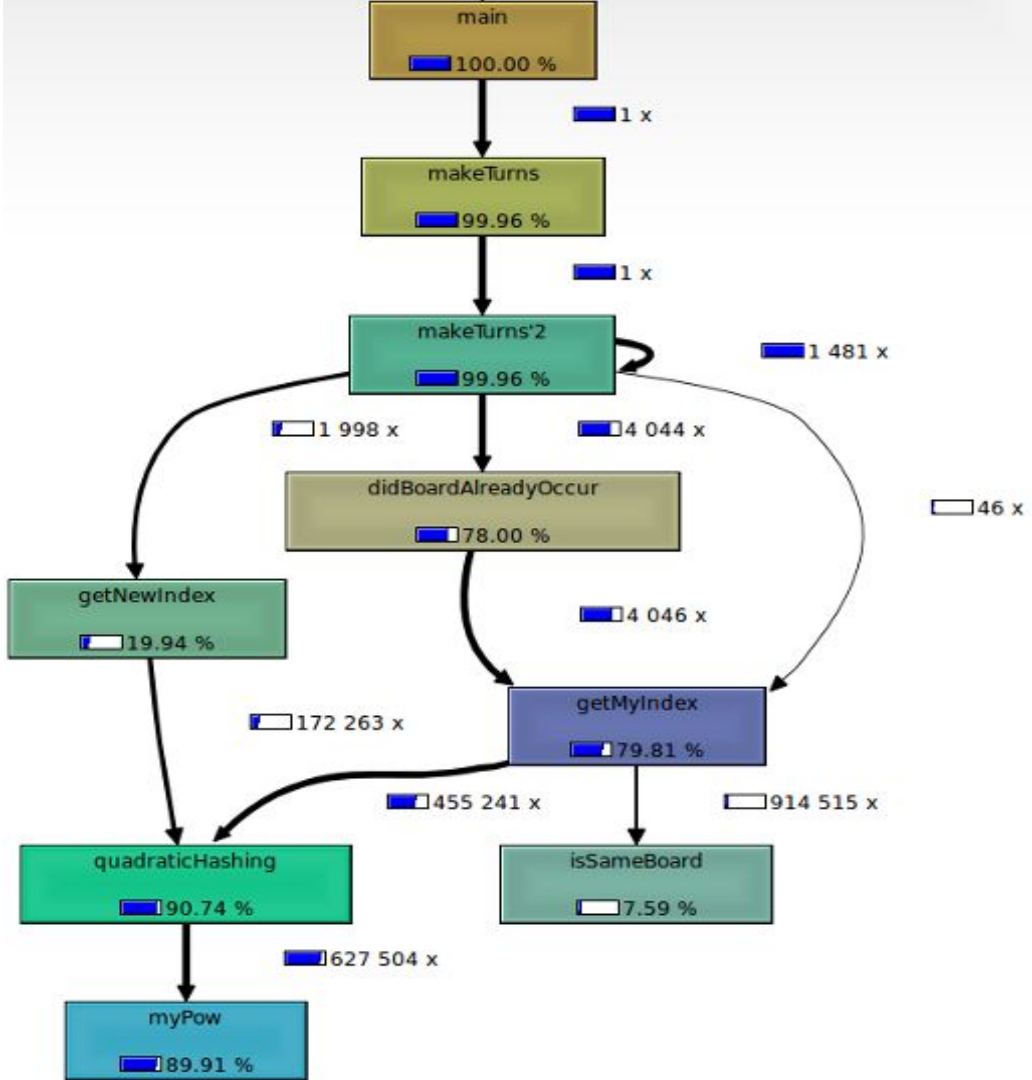
http://darkdust.net/files/GDB%20Cheat%20Sheet.pdf

http://www.cs.berkeley.edu/~mavam/teaching/cs161-sp11/gdb-refcard.pdf

Профилировка CPU

- Valgrind не подходит, т.к. серьезно модифицирует исходную программу и мы уже будем профилировать не ее, а JIT Valgrind’а
- Профилировка как правило, делается так: в программе для каждой функции запоминается длительность ее вызова и собирается backtrace каждого вызова
- Для сбора backtrace и длительностей вызова можно:
  - Модифицировать программу (расширение для компилятора)
  - Периодически прерывать программу и собирать backtrace’ы (не требует расширений)
- Собранные данные можно по-разному визуализировать

Визуализация профиля CPU: call graph



Визуализация профиля CPU: backtrace, text

```
# Overhead  Command      Shared Object                      Symbol
# ........  .......  .................  ..........................
#
    99.83%   profus  profus            [.] fib(unsigned long)
             |
             --- fib(unsigned long)
                |
                |--49.87%-- work3x()
                |           main
                |           __libc_start_main
                |
                |--33.29%-- work2x()
                |           main
                |           __libc_start_main
                |
                 --16.68%-- work1x()
                           main
                           __libc_start_main

     0.15%   profus  [kernel.kallsyms]  [k] 0xffffffff8104f45a

     0.00%   profus  libc-2.19.so       [.] init_cacheinfo

     0.00%   profus  ld-2.19.so         [.] _dl_relocate_object

     0.00%   profus  ld-2.19.so         [.] _dl_lookup_symbol_x

     0.00%   profus  ld-2.19.so         [.] do_lookup_x

     0.00%   profus  libm-2.19.so       [.] 0x000000000001a470

     0.00%   profus  ld-2.19.so         [.] _dl_map_object_from_fd
```

Linux perf

Мощный профилировщик для Linux. Может профилировать всю ОС сразу.

```
$ perf record -g ./bin/profus

$ perf report --stdio -g none -i ./perf.data | c++filt

$ perf report --stdio -g graph -i ./perf.data | c++filt
```