

LLVM [MemProf] и методы профилирования памяти

Алексей Веселовский

Профилирование памяти

Профилирование **использования** памяти

Профилирование **использования** памяти

Самый простой вопрос:

Профилирование **использования** памяти

Самый простой вопрос: сколько памяти использует?

Профилирование **использования** памяти

Самый простой вопрос: сколько памяти использует?

Например очень важно если памяти ограничено, и вы за неё платите.

Профилирование **использования** памяти

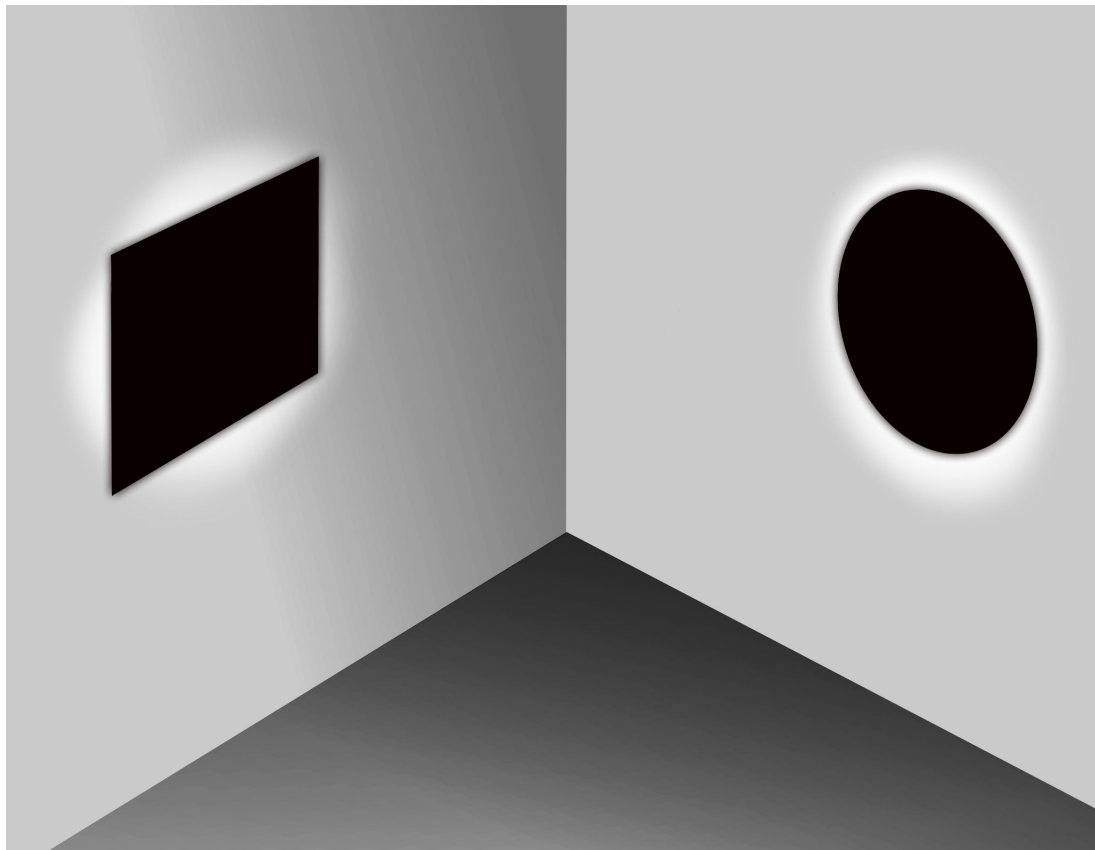
Самый простой вопрос: сколько памяти использует?

Например очень важно если памяти ограничено, и вы за неё платите.

AWS Lambda

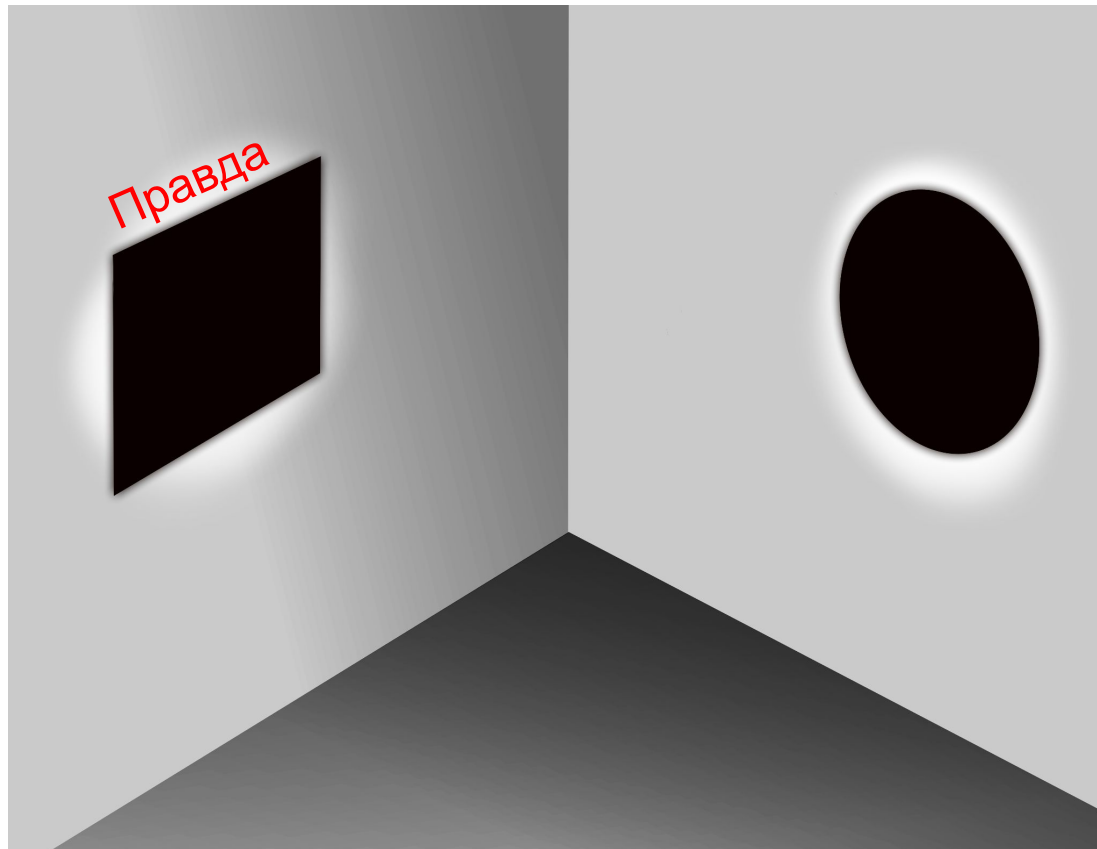
Профилирование **использования** памяти

Самый простой вопрос: сколько памяти использует?



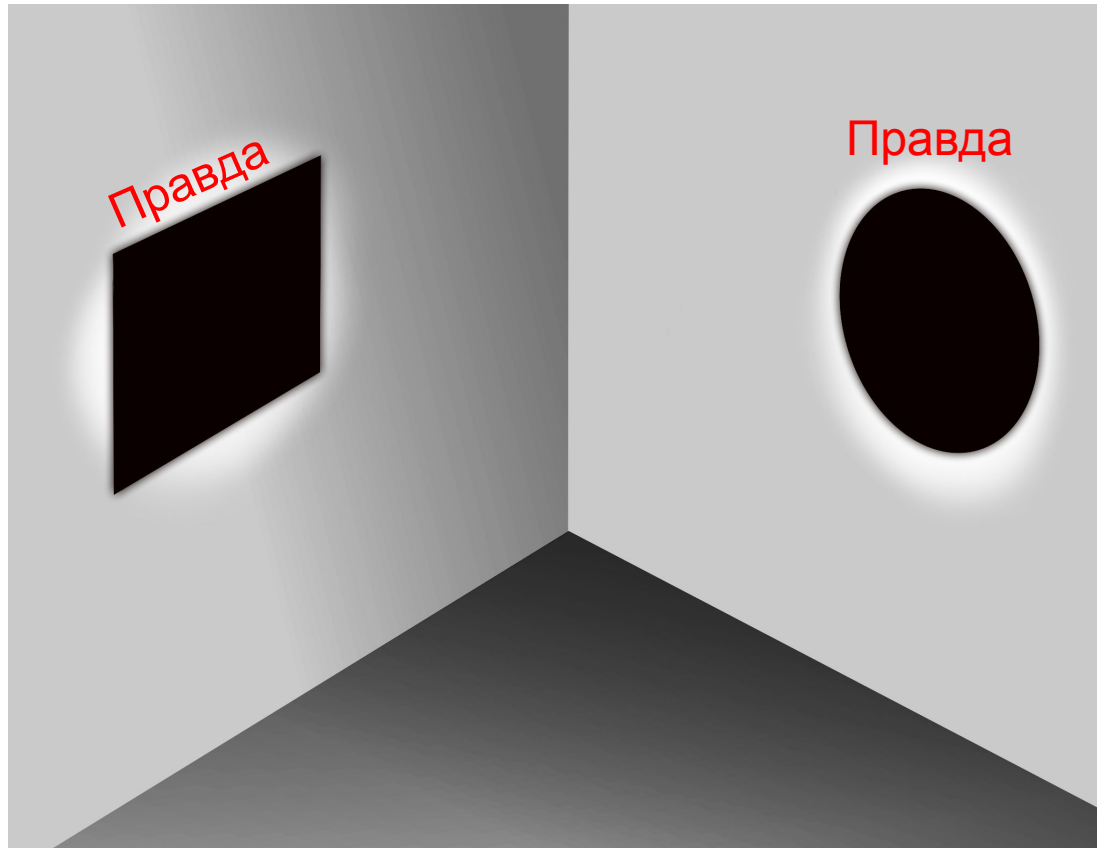
Профилирование **использования** памяти

Самый простой вопрос: сколько памяти использует?



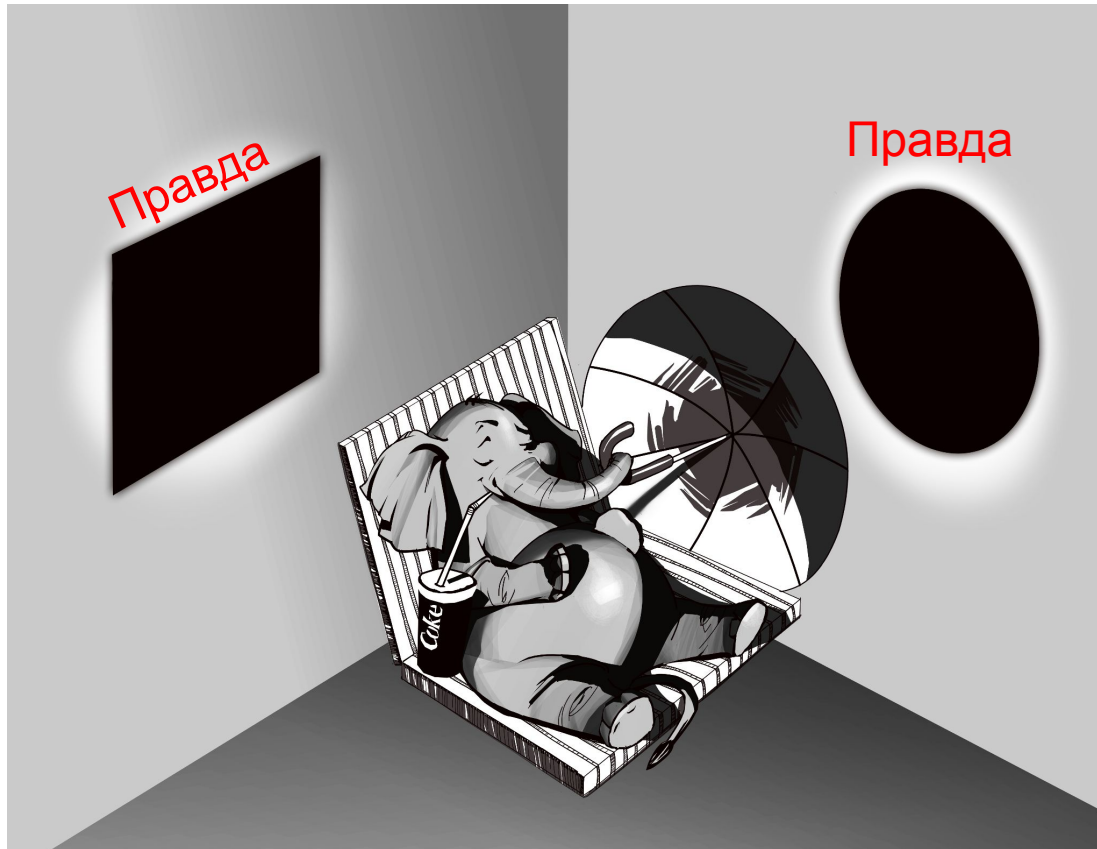
Профилирование **использования** памяти

Самый простой вопрос: сколько памяти использует?



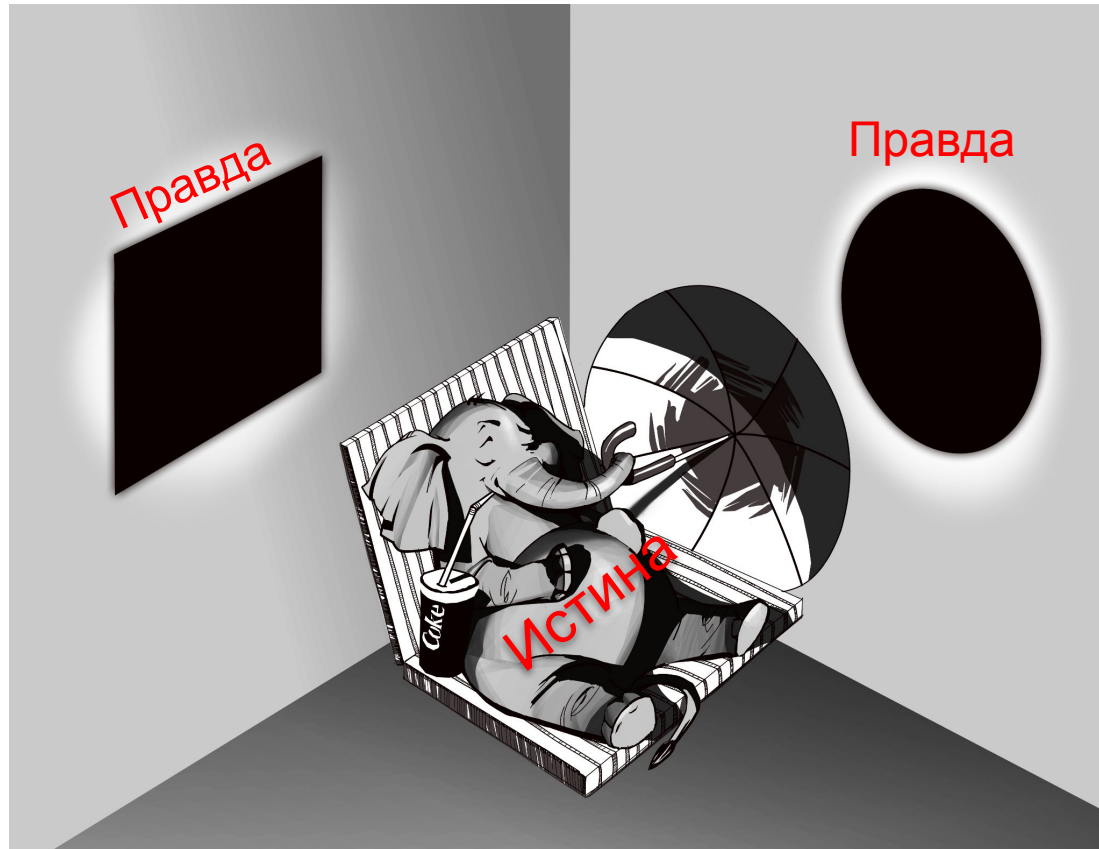
Профилирование **использования** памяти

Самый простой вопрос: сколько памяти использует?



Профилирование **использования** памяти

Самый простой вопрос: сколько памяти использует?



Профилирование **использования** памяти

Самый простой вопрос: сколько памяти использует?

Профилирование **использования** памяти

Самый простой вопрос: сколько памяти использует?

Приложение => process

Профилирование **использования** памяти

Самый простой вопрос: сколько памяти использует?

Приложение => process

process => virtual address space + что-то ещё (timers, files, sockets...)

Профилирование **использования** памяти

Самый простой вопрос: сколько памяти использует?

```
$ man top
```

Профилирование **использования** памяти

Самый простой вопрос: сколько памяти использует?

```
$ man top
```

| | Private | Shared |
|-------------|--|--------------------------------------|
| | 1 | 2 |
| Anonymous | . stack . malloc() . brk()/sbrk() . mmap(PRIVATE, ANON) | . POSIX shm* . mmap(SHARED, ANON) |
| -----+----- | | |
| File-backed | . mmap(PRIVATE, fd) . pgms/shared libs | . mmap(SHARED, fd) |
| | 3 | 4 |

Профилирование **использования** памяти

Самый простой вопрос: сколько памяти использует?

```
$ man top
```

```
VIRT -- Virtual Memory Size (KiB)
```

```
The total amount of virtual memory used by the task. It includes all code, data and shared libraries plus pages that have been swapped out and pages that have been mapped but not used.
```

```
See `OVERVIEW, Linux Memory Types' for additional details.
```

| | Private | Shared |
|-----------------------|---------|----------------------|
| Anonymous | 1 | 2 |
| . stack | | |
| . malloc() | | |
| . brk()/sbrk() | | . POSIX shm* |
| . mmap(PRIVATE, ANON) | | . mmap(SHARED, ANON) |
| -----+----- | | |
| File-backed | | |
| . mmap(PRIVATE, fd) | | . mmap(SHARED, fd) |
| . pgms/shared libs | | |
| | 3 | 4 |

Профилирование **использования** памяти

Самый простой вопрос: сколько памяти использует?

```
$ man top
```

```
CODE -- Code Size (KiB)
```

```
The amount of physical memory currently devoted to executable code, also known as the Text Resident Set size or TRS.
```

```
See `OVERVIEW, Linux Memory Types' for additional details.
```

| | Private | Shared |
|-------------|-----------------------|----------------------|
| | 1 | 2 |
| Anonymous | . stack | |
| | . malloc() | |
| | . brk()/sbrk() | . POSIX shm* |
| | . mmap(PRIVATE, ANON) | . mmap(SHARED, ANON) |
| | -----+----- | |
| File-backed | . mmap(PRIVATE, fd) | . mmap(SHARED, fd) |
| | . pgms/shared libs | |
| | 3 | 4 |

Профилирование **использования** памяти

Самый простой вопрос: сколько памяти использует?

```
$ man top
```

```
DATA -- Data + Stack Size (KiB)
```

```
The amount of private memory reserved by a process. It is also known as the Data Resident Set or DRS. Such memory may not yet be mapped to physical memory (RES) but will always be included in the virtual memory (VIRT) amount.
```

```
See `OVERVIEW, Linux Memory Types' for additional details.
```

| | Private | Shared |
|-------------|--|--------------------------------------|
| | 1 | 2 |
| Anonymous | . stack . malloc() . brk()/sbrk() . mmap(PRIVATE, ANON) | . POSIX shm* . mmap(SHARED, ANON) |
| -----+----- | | |
| File-backed | . mmap(PRIVATE, fd) . pgms/shared libs | . mmap(SHARED, fd) |
| | 3 | 4 |

Профилирование **использования** памяти

Самый простой вопрос: сколько памяти использует?

```
$ man top
```

```
RES -- Resident Memory Size (KiB)
```

```
A subset of the virtual address space (VIRT) representing the non-swapped physical memory a task is currently using. It is also the sum of the `RSan', `RSfd' and `RSsh' fields.
```

```
It can include private anonymous pages, private pages mapped to files (including program images and shared libraries) plus shared anonymous pages. All such memory is backed by the swap file represented separately under SWAP.
```

```
Lastly, this field may also include shared file-backed pages which, when modified, act as a dedicated swap file and thus will never impact SWAP.
```

| | Private | Shared |
|-------------|-----------------------|----------------------|
| | 1 | 2 |
| Anonymous | . stack | |
| | . malloc() | |
| | . brk()/sbrk() | . POSIX shm* |
| | . mmap(PRIVATE, ANON) | . mmap(SHARED, ANON) |
| -----+----- | | |
| | . mmap(PRIVATE, fd) | . mmap(SHARED, fd) |
| File-backed | . pgms/shared libs | |
| | 3 | 4 |

Профилирование **использования** памяти

Самый простой вопрос: сколько памяти использует?

```
$ man top
```

```
SHR -- Shared Memory Size (KiB)
```

```
A subset of resident memory (RES) that may be used by other processes. It will include shared anonymous pages and shared file-backed pages. It also includes private pages mapped to files representing program images and shared libraries.
```

```
See `OVERVIEW, Linux Memory Types' for additional details.
```

| | Private | Shared |
|-----------------------|---------|----------------------|
| Anonymous | 1 | 2 |
| . stack | | |
| . malloc() | | |
| . brk()/sbrk() | | . POSIX shm* |
| . mmap(PRIVATE, ANON) | | . mmap(SHARED, ANON) |
| -----+----- | | |
| File-backed | | |
| . mmap(PRIVATE, fd) | | . mmap(SHARED, fd) |
| . pgms/shared libs | | |
| | 3 | 4 |

Профилирование **использования** памяти

Самый простой вопрос: сколько памяти использует?

```
$ man top
```

```
SWAP -- Swapped Size (KiB)  
The formerly resident portion of a task's address space written to the swap file when physical memory becomes over committed.
```

See `OVERVIEW, Linux Memory Types' for additional details.

| | Private | Shared |
|-------------|--|--------------------------------------|
| | 1 | 2 |
| Anonymous | . stack . malloc() . brk()/sbrk() . mmap(PRIVATE, ANON) | . POSIX shm* . mmap(SHARED, ANON) |
| -----+----- | | |
| File-backed | . mmap(PRIVATE, fd) . pgms/shared libs | . mmap(SHARED, fd) |
| | 3 | 4 |

Профилирование использования памяти

Самый простой вопрос: сколько памяти использует?

```
$ man top
```

```
CODE - the `pgms' portion of quadrant 3
DATA - the entire quadrant 1 portion of VIRT plus all
      explicit mmap file-backed pages of quadrant 3
RES  - anything occupying physical memory which, beginning with
      Linux-4.5, is the sum of the following three fields:
      RSan - quadrant 1 pages, which include any
            former quadrant 3 pages if modified
      RSfd - quadrant 3 and quadrant 4 pages
      RSsh - quadrant 2 pages
RSlk - subset of RES which cannot be swapped out (any quadrant)
SHR  - subset of RES (excludes 1, includes all 2 & 4, some 3)
SWAP - potentially any quadrant except 4
USED - simply the sum of RES and SWAP
VIRT - everything in-use and/or reserved (all quadrants)
```

| | Private | Shared |
|-------------|--|--------------------------------------|
| | 1 | 2 |
| Anonymous | . stack . malloc() . brk()/sbrk() . mmap(PRIVATE, ANON) | . POSIX shm* . mmap(SHARED, ANON) |
| -----+----- | | |
| File-backed | . mmap(PRIVATE, fd) . pgms/shared libs | . mmap(SHARED, fd) |
| | 3 | 4 |

Профилирование **использования** памяти

Самый простой вопрос: сколько памяти использует?

```
$ man top
```

```
Note: Even though program images and shared libraries are considered private to a process, they will be accounted for as shared (SHR) by the kernel.
```

```
CODE - the `pgms' portion of quadrant 3
DATA - the entire quadrant 1 portion of VIRT plus all
      explicit mmap file-backed pages of quadrant 3
RES   - anything occupying physical memory which, beginning with
      Linux-4.5, is the sum of the following three fields:
      RSan - quadrant 1 pages, which include any
            former quadrant 3 pages if modified
      RSfd - quadrant 3 and quadrant 4 pages
      RSsh - quadrant 2 pages
RSlk - subset of RES which cannot be swapped out (any quadra
SHR - subset of RES (excludes 1, includes all 2 & 4, some 3
SWAP - potentially any quadrant except 4
USED - simply the sum of RES and SWAP
VIRT - everything in-use and/or reserved (all quadrants)
```

| | Private | Shared |
|-------------|--|--------------------------------------|
| | 1 | 2 |
| Anonymous | . stack . malloc() . brk()/sbrk() . mmap(PRIVATE, ANON) | . POSIX shm* . mmap(SHARED, ANON) |
| | -----+ | -----+ |
| File-backed | . mmap(PRIVATE, fd) . pgms/shared libs | . mmap(SHARED, fd) |
| | 3 | 4 |

Профилирование **использования** памяти

Самый простой вопрос: сколько памяти использует?

```
$ man top
```

```
VIRT -- Virtual Memory Size (KiB)
```

```
The total amount of virtual memory used by the task. It includes all code, data and shared libraries plus pages that have been swapped out and pages that have been mapped but not used.
```

```
See `OVERVIEW, Linux Memory Types' for additional details.
```

Профилирование **использования** памяти

Самый простой вопрос: сколько памяти использует?

```
$ man top
```

```
VIRT -- Virtual Memory Size (KiB)
```

```
The total amount of virtual memory used by the task. It includes all code, data and shared libraries plus pages that have been swapped out and pages that have been mapped but not used.
```

```
See `OVERVIEW, Linux Memory Types' for additional details.
```

???!??!

Профилирование **использования** памяти

Самый простой вопрос: сколько памяти использует?

Linux memory overcommit

Профилирование **использования** памяти

Самый простой вопрос: сколько памяти использует?

Linux memory overcommit == airline ticket overbooking

Профилирование **использования** памяти

Самый простой вопрос: сколько памяти использует?

Linux memory overcommit == airline ticket overbooking

```
uint8_t* arr = new uint8_t[TiB];
```

Профилирование **использования** памяти

Самый простой вопрос: сколько памяти использует?

Linux memory overcommit == airline ticket overbooking

```
uint8_t* arr = new uint8_t[TiB]; // OK
```

Профилирование **использования** памяти

Самый простой вопрос: сколько памяти использует?

Linux memory overcommit == airline ticket overbooking

```
uint8_t* arr = new uint8_t[TiB]; // OK

std::random_device rd;
std::mt19937 gen(rd());
std::uniform_int_distribution<> distrib(0, TiB-1);
```

Профилирование **использования** памяти

Самый простой вопрос: сколько памяти использует?

Linux memory overcommit == airline ticket overbooking

```
uint8_t* arr = new uint8_t[TiB]; // OK

std::random_device rd;
std::mt19937 gen(rd());
std::uniform_int_distribution<> distrib(0, TiB-1);

while (!stop) arr[distrib(gen)] = 1;
```

Профилирование **использования** памяти

Самый простой вопрос: сколько памяти использует?

Linux memory overcommit == airline ticket overbooking

```
uint8_t* arr = new uint8_t[TiB]; // OK

std::random_device rd;
std::mt19937 gen(rd());
std::uniform_int_distribution<> distrib(0, TiB-1);

while (!stop) arr[distrib(gen)] = 1; // I'm Feeling Lucky
```

Профилирование **использования** памяти

Самый простой вопрос: сколько памяти использует?

Linux memory overcommit == airline ticket overbooking

```
uint8_t* arr = new uint8_t[TiB]; // OK

std::random_device rd;
std::mt19937 gen(rd());
std::uniform_int_distribution<> distrib(0, TiB-1);

while (!stop) arr[distrib(gen)] = 1; // I'm Feeling Lucky
```

Профилирование **использования** памяти

Самый простой вопрос: сколько памяти использует?

Вообще говоря, это настройка ядра:

Профилирование **использования** памяти

Самый простой вопрос: сколько памяти использует?

Вообще говоря, это настройка ядра: `sysctl vm.overcommit_memory`

Профилирование **использования** памяти

Самый простой вопрос: сколько памяти использует?

Вообще говоря, это настройка ядра: `sysctl vm.overcommit_memory`

- 0 - **Heuristic overcommit handling.** Obvious overcommits of address space are refused. Used for a typical system. It ensures a seriously wild allocation fails while allowing overcommit to reduce swap usage. root is allowed to allocate slightly more memory in this mode. This is the default.

Профилирование **использования** памяти

Самый простой вопрос: сколько памяти использует?

Вообще говоря, это настройка ядра: `sysctl vm.overcommit_memory`

- 1 - **Always overcommit.** Appropriate for some scientific applications. Classic example is code using **sparse arrays** and just relying on the virtual memory consisting almost entirely of zero pages.

Профилирование **ИСПОЛЬЗОВАНИЯ** памяти

Самый простой вопрос: сколько памяти использует?

Вообще говоря, это настройка ядра: `sysctl vm.overcommit_memory`

- 2 - **Don't overcommit.** The total address space commit for the system is not permitted to exceed swap + a configurable amount (default is 50%) of physical RAM. Depending on the amount you use, in most situations this means a process will not be killed while accessing pages but will receive errors on memory allocation as appropriate.

Useful for applications that want to guarantee their memory allocations will be available in the future without having to initialize every page.

Профилирование **использования** памяти

Правильным ответом на вопрос “сколько приложение использует памяти?” будет

Профилирование **использования** памяти

Правильным ответом на вопрос “сколько приложение использует памяти?” будет

“А вы с какой целью интересуетесь?”

Профилирование **использования** памяти

Правильным ответом на вопрос “сколько приложение использует памяти?” будет

“А вы с какой целью интересуетесь?”

“А вам эта информация, чтобы что?”

Профилирование использования памяти

Правильным ответом на вопрос “сколько приложение использует памяти?” будет

“А вы с какой целью интересуетесь?”

“А вам эта информация, чтобы что?”



Профилирование использования памяти

Правильным ответом на вопрос “сколько приложение использует памяти?” будет

“А вы с какой целью интересуетесь?”

“А вам эта информация, чтобы что?”

Наприме на самом деле, возможно хотят ответить на вопрос “а хватит ли RAM” для запуска на вот этой машине?



Профилирование использования памяти

Правильным ответом на вопрос “сколько приложение использует памяти?” будет

“А вы с какой целью интересуетесь?”

“А вам эта информация, чтобы что?”

Наприме на самом деле, возможно хотят ответить на вопрос “а хватит ли RAM” для запуска на вот этой машине?

Тогда не важно сколько приложение ест, важно сколько будет ВСЕГО потрачено памяти



Профилирование использования памяти

Правильным ответом на вопрос “сколько приложение использует памяти?” будет

“А вы с какой целью интересуетесь?”

“А вам эта информация, чтобы что?”

Наприме на самом деле, возможно хотят ответить на вопрос “а хватит ли RAM” для запуска на вот этой машине?

Тогда не важно сколько приложение ест, важно сколько будет ВСЕГО потрачено памяти

X11 Client + X11 Server – два разных приложения



Профилирование **использования** памяти

Правильным ответом на вопрос “сколько приложение использует памяти?” будет матрица вроде:

Профилирование **использования** памяти

Правильным ответом на вопрос “сколько приложение использует памяти?” будет

матрица вроде:

| | MIN | MEAN | MEDIAN | SD | MAX |
|------|-----|------|--------|----|-----|
| VIRT | | | | | |
| CODE | | | | | |
| SHR | | | | | |
| DATA | | | | | |
| RES | | | | | |

Профилирование **использования** памяти

Самый простой вопрос: сколько памяти использует?

Мы рассмотрели только то, что видно на уровне OS

Профилирование **использования** памяти

Самый простой вопрос: сколько памяти использует?

Мы рассмотрели только то, что видно на уровне OS

Но внутри каждого приложения свой персональный **особенный** ад.

Профилирование **использования** памяти

Самый простой вопрос: сколько памяти использует?

Мы рассмотрели только то, что видно на уровне OS

Но внутри каждого приложения свой персональный **особенный** ад.

```
uint8_t* p = new uint8_t[GiB];  
delete[] p;
```

Профилирование **использования** памяти

Самый простой вопрос: сколько памяти использует?

Мы рассмотрели только то, что видно на уровне OS

Но внутри каждого приложения свой персональный **особенный** ад.

```
uint8_t* p = new uint8_t[GiB];  
delete[] p;  
// сколько теперь памяти занято?
```

Профилирование **использования** памяти

Самый простой вопрос: сколько памяти использует?

Но внутри каждого приложения свой персональный **особенный** ад.

```
uint8_t* p = new uint8_t[GiB];  
delete[] p;  
// сколько теперь памяти занято?
```

Профилирование **использования** памяти

Самый простой вопрос: сколько памяти использует?

Но внутри каждого приложения свой персональный **особенный** ад.

Менеджер памяти может и не отдать в систему память.

```
uint8_t* p = new uint8_t[GiB];  
delete[] p;  
// сколько теперь памяти занято?
```

Профилирование **использования** памяти

Самый простой вопрос: сколько памяти использует?

Но внутри каждого приложения свой персональный **особенный** ад.

Также мы не знаем трогал он память, или нет.

```
uint8_t* p = new uint8_t[GiB];  
delete[] p;  
// сколько теперь памяти занято?
```

Профилирование **использования** памяти

Самый простой вопрос: сколько памяти использует?

Но внутри каждого приложения свой персональный **особенный** ад.

Профилирование **использования** памяти

Самый простой вопрос: сколько памяти использует?

Внутри процесса нет “барьеров” наподобие тех, что есть в OS (выделенные виртуальные адресные пространства).

Профилирование **использования** памяти

Самый простой вопрос: сколько памяти использует?

Внутри процесса нет “барьеров” наподобие тех, что есть в OS (выделенные виртуальные адресные пространства). В erlang’е есть, но это другая история...

Профилирование **использования** памяти

Самый простой вопрос: сколько памяти использует?

Внутри процесса нет “барьеров” наподобие тех, что есть в OS (выделенные виртуальные адресные пространства). В erlang’е есть, но это другая история...

Поэтому понять кто же “сожрал” всю память уже внутри приложения – существенно сложнее.

Профилирование **использования** памяти

Самый простой вопрос: сколько памяти использует?

Внутри процесса нет “барьеров” наподобие тех, что есть в OS (выделенные виртуальные адресные пространства). В erlang’е есть, но это другая история...

Поэтому понять кто же “сожрал” всю память уже внутри приложения – существенно сложнее.

Причем “сожрать” можно двояко – кто-то много выделил, а кто-то держит (умными указателями)

Профилирование **использования** памяти

Самый простой вопрос: сколько памяти использует?

Внутри процесса нет “барьеров” наподобие тех, что есть в OS (выделенные виртуальные адресные пространства). В erlang’е есть, но это другая история...

Поэтому понять кто же “сожрал” всю память уже внутри приложения – существенно сложнее.

Причем “сожрать” можно двояко – кто-то много выделил, а кто-то держит (умными указателями)

А часть памяти может просто утечь.

Профилирование **использования** памяти

Самый простой вопрос: сколько памяти использует?

Внутри процесса нет “барьеров” наподобие тех, что есть в OS (выделенные виртуальные адресные пространства). В erlang’е есть, но это другая история...

Поэтому понять кто же “сожрал” всю память уже внутри приложения – существенно сложнее.

Причем “сожрать” можно двояко – кто-то много выделил, а кто-то держит (умными указателями)

А часть памяти может просто утечь.

А ещё интересно кто как и как часто тербит менеджер памяти.

Профилірованне **іспользавання** памяці

Профиліровшчыкі спешат на дапамогу!

Профилирование **использования** памяти

Профилеровщики спешат на помощь!

Позволяют заглянуть внутрь приложения и попытаться что-то понять.

Valgrind massif

Valgrind massif

```
$ cat wc.cpp
```

Valgrind massif

```
$ cat wc.cpp
#include <iostream>
#include <map>
#include <string>

int main() {
    std::string in;
    std::map<std::string, int> dict;
    while (std::cin >> in)
        dict[in]++;
    for (auto [k, v] : dict)
        std::cout << k << ": " << v << std::endl;
    return 0;
}
```

Valgrind massif

```
$ cat wc.cpp
#include <iostream>
#include <map>
#include <string>

int main() {
    std::string in;
    std::map<std::string, int> dict;
    while (std::cin >> in)
        dict[in]++;
    for (auto [k, v] : dict)
        std::cout << k << ": " << v << std::endl;
    return 0;
}

$ g++ -g wc.cpp
```

Valgrind massif

```
$ cat wc.cpp
#include <iostream>
#include <map>
#include <string>

int main() {
    std::string in;
    std::map<std::string, int> dict;
    while (std::cin >> in)
        dict[in]++;
    for (auto [k, v] : dict)
        std::cout << k << ": " << v << std::endl;
    return 0;
}

$ g++ -g wc.cpp
$ valgrind --tool=massif ./a.out < test_file.txt
```

Valgrind massif

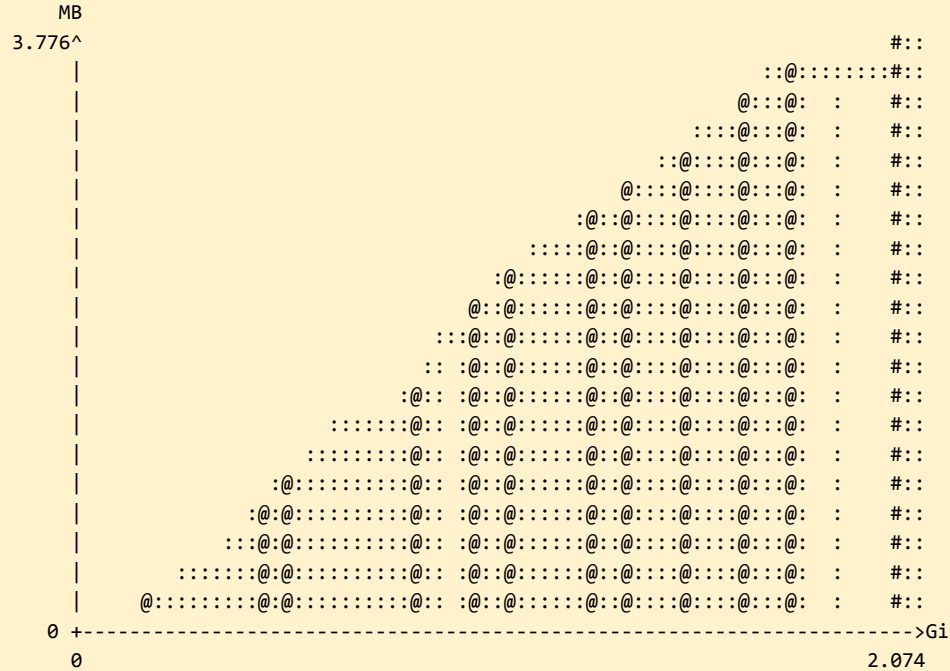
```
$ cat wc.cpp
#include <iostream>
#include <map>
#include <string>

int main() {
    std::string in;
    std::map<std::string, int> dict;
    while (std::cin >> in)
        dict[in]++;
    for (auto [k, v] : dict)
        std::cout << k << ": " << v << std::endl;
    return 0;
}

$ g++ -g wc.cpp
$ valgrind --tool=massif ./a.out < test_file.txt
$ ms_print massif.out.265087 | less
```

Valgrind massif

```
$ ms_print massif.out.265087 | less
```



Number of snapshots: 89

Detailed snapshots: [5, 6, 18, 20, 31, 35, 39, 46, 49, 59, 69, 79, 84 (peak)]

Valgrind massif

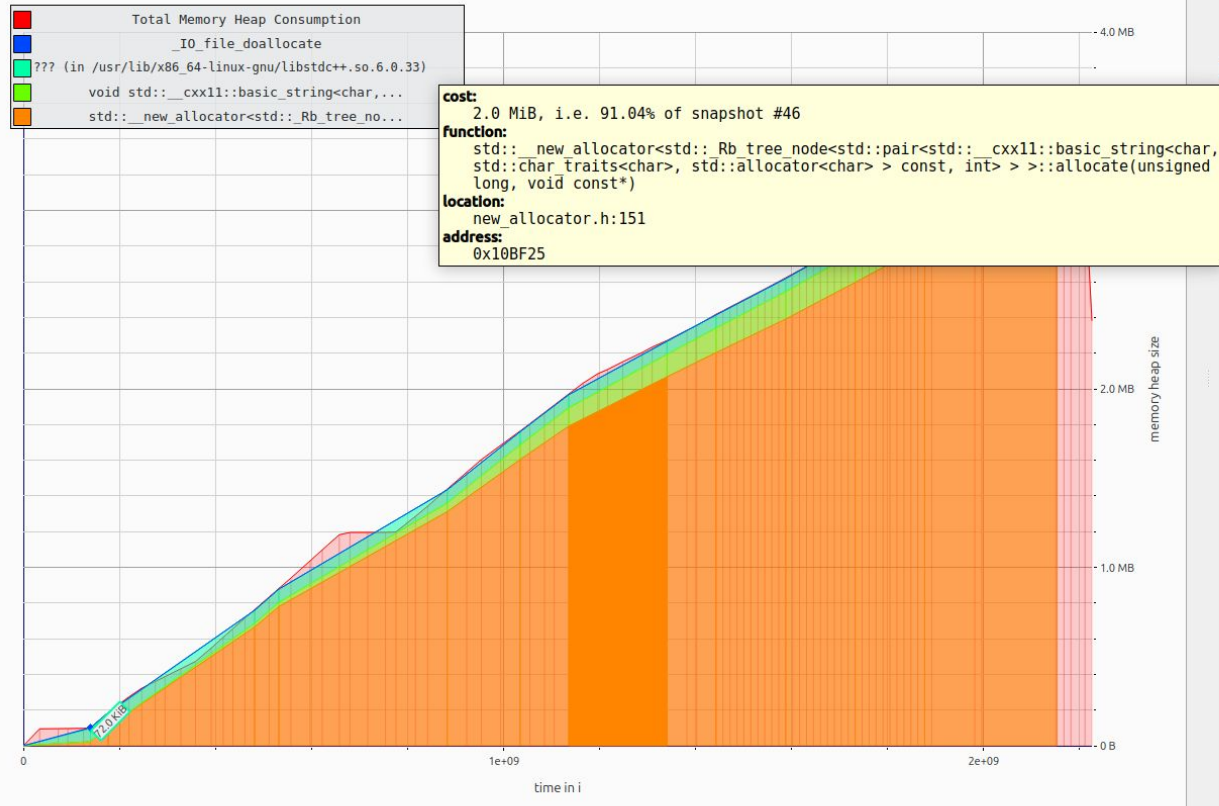
```
$ massif-visualizer massif.out.265087
```

Valgrind massif

```
$ massif-visualizer massif.out.265087
```

Memory consumption of ./a.out

Peak of 3.0 MiB at snapshot #84



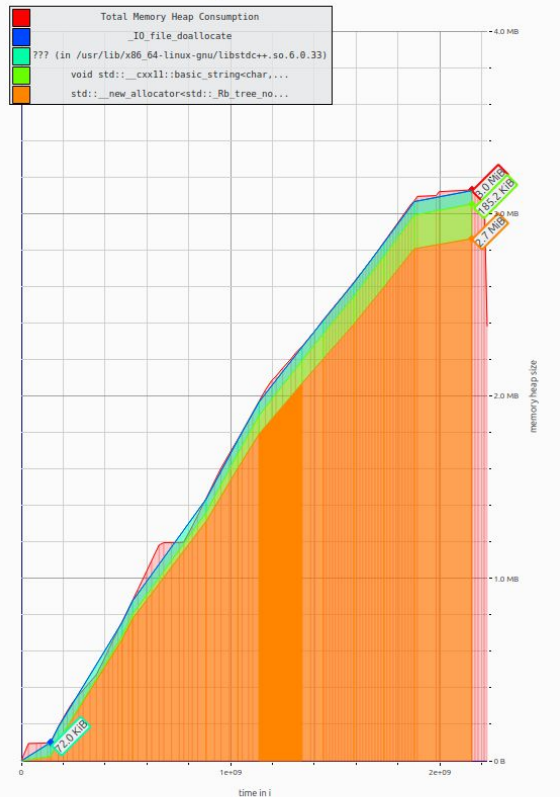
- 2.0 MiB: Snapshot #41
- 2.0 MiB: Snapshot #42
- 2.1 MiB: Snapshot #43
- 2.1 MiB: Snapshot #44
- 2.1 MiB: Snapshot #45
- 2.2 MiB: Snapshot #46
 - 2.0 MiB: std::_new_allocator<std::Rb_tree_node<std::pair<std::cxx11::basic_string<ch...
 - 122.6 KiB: void std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char>...
 - 72.0 KiB: ??? (in /usr/lib/x86_64-linux-gnu/libstdc++.so.6.0.33)
 - 4.0 KiB: in 2 places, all below massif's threshold (1.00%)
- 2.2 MiB: Snapshot #47
- 2.2 MiB: Snapshot #48
- 2.3 MiB: Snapshot #49
 - 2.1 MiB: std::_new_allocator<std::Rb_tree_node<std::pair<std::cxx11::basic_string<ch...
 - 134.4 KiB: void std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char>...
 - 72.0 KiB: ??? (in /usr/lib/x86_64-linux-gnu/libstdc++.so.6.0.33)
 - 4.0 KiB: in 2 places, all below massif's threshold (1.00%)
- 2.3 MiB: Snapshot #50
- 2.3 MiB: Snapshot #51
- 2.4 MiB: Snapshot #52
- 2.4 MiB: Snapshot #53
- 2.4 MiB: Snapshot #54
- 2.4 MiB: Snapshot #55
- 2.4 MiB: Snapshot #56
- 2.5 MiB: Snapshot #57
- 2.5 MiB: Snapshot #58
- 2.5 MiB: Snapshot #59
- 2.5 MiB: Snapshot #60
- 2.5 MiB: Snapshot #61
- 2.6 MiB: Snapshot #62
- 2.6 MiB: Snapshot #63
- 2.6 MiB: Snapshot #64
- 2.6 MiB: Snapshot #65
- 2.6 MiB: Snapshot #66
- 2.7 MiB: Snapshot #67
- 2.7 MiB: Snapshot #68
- 2.7 MiB: Snapshot #69
 - 2.5 MiB: std::_new_allocator<std::Rb_tree_node<std::pair<std::cxx11::basic_string<ch...
 - 165.6 KiB: void std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char>...
 - 72.0 KiB: ??? (in /usr/lib/x86_64-linux-gnu/libstdc++.so.6.0.33)
 - 4.0 KiB: in 2 places, all below massif's threshold (1.00%)
- 2.7 MiB: Snapshot #70
- 2.8 MiB: Snapshot #71
- 2.8 MiB: Snapshot #72
- 2.8 MiB: Snapshot #73
- 2.8 MiB: Snapshot #74
- 2.8 MiB: Snapshot #75
- 2.9 MiB: Snapshot #76
- 2.9 MiB: Snapshot #77
- 2.9 MiB: Snapshot #78

Valgrind massif

```
$ massif-visualizer massif.out.265087
```

Memory consumption of ./a.out

Peak of 3.0 MiB at snapshot #84



```
2.0 MiB: Snapshot #41
2.0 MiB: Snapshot #42
2.1 MiB: Snapshot #43
2.1 MiB: Snapshot #44
2.1 MiB: Snapshot #45
2.2 MiB: Snapshot #46
  2.0 MiB: std:: new_allocator<std::Rb_tree_node<std::pair<std::cxx11::basic_string<char, std::char_traits<char>, std::allocator<char>> const, int>>>>::allocate(unsigned long, void ...
    2.0 MiB: allocate (alloc_traits.h:482)
      2.0 MiB: std::Rb_tree<std::cxx11::basic_string<char, std::char_traits<char>, std::allocator<char>>, std::pair<std::cxx11::basic_string<char, std::char_traits<char>, std::allocat...
        2.0 MiB: std::Rb_tree_node<std::pair<std::cxx11::basic_string<char, std::char_traits<char>, std::allocator<char>> const, int>>>*> std::Rb_tree<std::cxx11::basic_string<...
          2.0 MiB: std::Rb_tree<std::cxx11::basic_string<char, std::char_traits<char>, std::allocator<char>>, std::pair<std::cxx11::basic_string<char, std::char_traits<char>, std::...
            2.0 MiB: std::Rb_tree_iterator<std::pair<std::cxx11::basic_string<char, std::char_traits<char>, std::allocator<char>> const, int>> std::Rb_tree<std::cxx11::basic_s...
              2.0 MiB: std::map<std::cxx11::basic_string<char, std::char_traits<char>, std::allocator<char>>, int, std::less<std::cxx11::basic_string<char, std::char_traits<char>, ...
                2.0 MiB: main (wc.cpp:10)
  122.6 KiB: void std::cxx11::basic_string<char, std::char_traits<char>, std::allocator<char>>::_M_construct<char*>(char*, char*, std::forward_iterator_tag) (in /usr/lib/x86_64-linux-gn...
  72.0 KiB: ??? (in /usr/lib/x86_64-linux-gnu/libstdc++.so.6.0.33)
  4.0 KiB: in 2 places, all below massif's threshold (1.00%)
2.2 MiB: Snapshot #47
2.2 MiB: Snapshot #48
2.3 MiB: Snapshot #49
  2.1 MiB: std:: new_allocator<std::Rb_tree_node<std::pair<std::cxx11::basic_string<char, std::char_traits<char>, std::allocator<char>> const, int>>>>::allocate(unsigned long, void ...
  134.4 KiB: void std::cxx11::basic_string<char, std::char_traits<char>, std::allocator<char>>::_M_construct<char*>(char*, char*, std::forward_iterator_tag) (in /usr/lib/x86_64-linux-gn...
  72.0 KiB: ??? (in /usr/lib/x86_64-linux-gnu/libstdc++.so.6.0.33)
  4.0 KiB: in 2 places, all below massif's threshold (1.00%)
2.3 MiB: Snapshot #50
2.3 MiB: Snapshot #51
2.4 MiB: Snapshot #52
2.4 MiB: Snapshot #53
2.4 MiB: Snapshot #54
2.4 MiB: Snapshot #55
2.4 MiB: Snapshot #56
2.5 MiB: Snapshot #57
2.5 MiB: Snapshot #58
2.5 MiB: Snapshot #59
2.5 MiB: Snapshot #60
2.5 MiB: Snapshot #61
2.6 MiB: Snapshot #62
2.6 MiB: Snapshot #63
2.6 MiB: Snapshot #64
2.6 MiB: Snapshot #65
2.6 MiB: Snapshot #66
2.7 MiB: Snapshot #67
2.7 MiB: Snapshot #68
2.7 MiB: Snapshot #69
  2.5 MiB: std:: new_allocator<std::Rb_tree_node<std::pair<std::cxx11::basic_string<char, std::char_traits<char>, std::allocator<char>> const, int>>>>::allocate(unsigned long, void ...
  165.6 KiB: void std::cxx11::basic_string<char, std::char_traits<char>, std::allocator<char>>::_M_construct<char*>(char*, char*, std::forward_iterator_tag) (in /usr/lib/x86_64-linux-gn...
  72.0 KiB: ??? (in /usr/lib/x86_64-linux-gnu/libstdc++.so.6.0.33)
  4.0 KiB: in 2 places, all below massif's threshold (1.00%)
2.7 MiB: Snapshot #70
2.8 MiB: Snapshot #71
```

Valgrind massif

Точный (не сэмплирующий по таймеру) профилировщик

Valgrind massif

Точный (не сэмплирующий по таймеру) профилировщик

Датасет прореживается чтобы одновременно он был и компактным и полным интересных событий

Valgrind massif

Точный (не сэмплирующий по таймеру) профилировщик

Датасет прореживается чтобы одновременно он был и компактным и полным интересных событий

Внешний относительно работы приложения

Valgrind massif

Точный (не сэмплирующий по таймеру) профилировщик

Датасет прореживается чтобы одновременно он был и компактным и полным интересных событий

Внешний относительно работы приложения

ОЧЕНЬ МЕДЛЕННЫЙ!

TCMalloc Heap Profiler

TCMalloc Heap Profiler

“Внутренний” – менеджера памяти кусок

TSMalloc Heap Profiler

“Внутренний” – менеджера памяти кусок

Не каждый TSMalloc одинаково полезен

TSMalloc Heap Profiler

“Внутренний” – менеджера памяти кусок

Не каждый TSMalloc одинаково полезен

нам нужен тот, что является частью gperftools

TCMalloc Heap Profiler

“Внутренний” – менеджера памяти кусок

Не каждый TCMalloc одинаково полезен

нам нужен тот, что является частью gperftools

Можно использовать просто через LD_PRELOAD

TCMalloc Heap Profiler

“Внутренний” – менеджера памяти кусок

Не каждый TCMalloc одинаково полезен

нам нужен тот, что является частью gperftools

Можно использовать просто через LD_PRELOAD

И да, он тоже точный

TCMalloc Heap Profiler

```
$ cat wc.cpp
#include <iostream>
#include <map>
#include <string>

int main() {
    std::string in;
    std::map<std::string, int> dict;
    while (std::cin >> in)
        dict[in]++;
    for (auto [k, v] : dict)
        std::cout << k << ": " << v << std::endl;
    return 0;
}
$ g++ -g wc.cpp
```

TCMalloc Heap Profiler

```
$ cat wc.cpp
#include <iostream>
#include <map>
#include <string>

int main() {
    std::string in;
    std::map<std::string, int> dict;
    while (std::cin >> in)
        dict[in]++;
    for (auto [k, v] : dict)
        std::cout << k << ": " << v << std::endl;
    return 0;
}
$ g++ -g wc.cpp
$ export HEAPPROFILE=/path/to/report/my_report
```

TCMalloc Heap Profiler

```
$ cat wc.cpp
#include <iostream>
#include <map>
#include <string>

int main() {
    std::string in;
    std::map<std::string, int> dict;
    while (std::cin >> in)
        dict[in]++;
    for (auto [k, v] : dict)
        std::cout << k << ": " << v << std::endl;
    return 0;
}
$ g++ -g wc.cpp
$ export HEAPPFILE=/path/to/report/my_report
$ LD_PRELOAD=/usr/lib/x86_64-linux-gnu/libtcmalloc.so.4.5.16 ./a.out < test_file.txt
```

TCMalloc Heap Profiler

```
$ cat wc.cpp
#include <iostream>
#include <map>
#include <string>

int main() {
    std::string in;
    std::map<std::string, int> dict;
    while (std::cin >> in)
        dict[in]++;
    for (auto [k, v] : dict)
        std::cout << k << ": " << v << std::endl;
    return 0;
}
$ g++ -g wc.cpp
$ export HEAPPROFILE=/path/to/report/my_report
$ LD_PRELOAD=/usr/lib/x86_64-linux-gnu/libtcmalloc.so.4.5.16 ./a.out < test_file.txt
$ pprof --text --alloc_space ./a.out my_report.0001.heap
```

TCMalloc Heap Profiler

```
$ pprof --text --alloc_space ./a.out my_report.0001.heap
```

TCMalloc Heap Profiler

```
$ pprof --text --alloc_space ./a.out my_report.0001.heap
```

```
File: a.out
```

```
Type: alloc_space
```

```
Showing nodes accounting for 3165.38kB, 99.84% of 3170.41kB total
```

```
Dropped 16 nodes (cum <= 15.85kB)
```

| flat | flat% | sum% | cum | cum% | |
|-----------|--------|--------|-----------|--------|---|
| 2795.06kB | 88.16% | 88.16% | 2795.06kB | 88.16% | std::__new_allocator::allocate |
| 370.31kB | 11.68% | 99.84% | 370.31kB | 11.68% | std::__cxx11::basic_string::_M_construct |
| 0 | 0% | 99.84% | 3170.41kB | 100% | _libc_start_call_main |
| 0 | 0% | 99.84% | 3170.41kB | 100% | _libc_start_main_impl |
| 0 | 0% | 99.84% | 3170.41kB | 100% | main |
| 0 | 0% | 99.84% | 3170.41kB | 100% | start |
| 0 | 0% | 99.84% | 2980.22kB | 94.00% | std::_Rb_tree::_Auto_node::_Auto_node |
| 0 | 0% | 99.84% | 185.16kB | 5.84% | std::_Rb_tree::_M_construct_node |
| 0 | 0% | 99.84% | 2980.22kB | 94.00% | std::_Rb_tree::_M_create_node |
| 0 | 0% | 99.84% | 2980.22kB | 94.00% | std::_Rb_tree::_M_emplace_hint_unique |
| 0 | 0% | 99.84% | 2795.06kB | 88.16% | std::_Rb_tree::_M_get_node |
| 0 | 0% | 99.84% | 185.16kB | 5.84% | std::__new_allocator::construct (inline) |
| 0 | 0% | 99.84% | 2795.06kB | 88.16% | std::allocator_traits::allocate (inline) |
| 0 | 0% | 99.84% | 185.16kB | 5.84% | std::allocator_traits::construct (inline) |
| 0 | 0% | 99.84% | 2980.22kB | 94.00% | std::map::operator[] |
| 0 | 0% | 99.84% | 370.31kB | 11.68% | std::pair::pair |

TCMalloc Heap Profiler

```
$ pprof --gv --alloc_space ./a.out my_report.0001.heap
```

TCMalloc Heap Profiler

\$ pprof



ASAN/LSAN

Тоже кое-что могут.

ASAN/LSAN

Тоже кое-что могут.

Они заменяют системный менеджер памяти своим (в этом плане они как TCMalloc)

ASAN/LSAN

Тоже кое-что могут.

Они заменяют системный менеджер памяти своим (в этом плане они как TCMalloc)

Так что знают и сколько памяти занято, и хранят `stacktrace` для каждой аллокации

ASAN/LSAN

Тоже кое-что могут.

Они заменяют системный менеджер памяти своим (в этом плане они как TCMalloc)

Так что знают и сколько памяти занято, и хранят `stacktrace` для каждой аллокации

```
// Prints stack traces for all live heap allocations ordered by total
// allocation size until top_percent of total live heap is shown. Top_percent
// should be between 1 and 100. At most max_number_of_contexts contexts
// (stack traces) are printed.
// Experimental feature currently available only with ASan on Linux/x86_64.
void SANITIZER_CDECL __sanitizer_print_memory_profile(size_t top_percent, size_t max_number_of_contexts);
```

ASAN/LSAN

```
$ cat wc.cpp
#include <iostream>
#include <map>
#include <string>
#include <sanitizer/asan_interface.h>

int main() {
    std::string in;
    std::map<std::string, int> dict;
    while (std::cin >> in) {
        dict[in]++;
    }
    int i=0;
    for (auto [k, v] : dict) {
        std::cout << k << ": " << v << std::endl;
    }
    __sanitizer_print_memory_profile(100, 1000);
    return 0;
}
```

ASAN/LSAN

```
$ cat wc.cpp
#include <iostream>
#include <map>
#include <string>
#include <sanitizer/asan_interface.h>

int main() {
    std::string in;
    std::map<std::string, int> dict;
    while (std::cin >> in) {
        dict[in]++;
    }
    int i=0;
    for (auto [k, v] : dict) {
        std::cout << k << ": " << v << std::endl;
    }
    __sanitizer_print_memory_profile(100, 1000);
    return 0;
}

$ g++ -fsanitize=address wc.cpp
$ ./a.out < test_file.txt
```

ASAN/LSAN

```
$ cat wc.cpp
```

```
$ ./a.out < test_file.txt
```

```
Live Heap Allocations: 3130715 bytes in 49304 chunks; quarantined: 189600 bytes in 9546 chunks; 2084 other chunks; total chunks: 60934; showing top 100% (at most 1000 unique contexts)
```

2862144 byte(s) (91%) in 39752 allocation(s)

```
#0 0x7f784c8fe548 in operator new(unsigned long)
../../../../src/libsanitizer/asan/asan_new_delete.cpp:95
#1 0x614c930362a3 in std::__new_allocator<std::_Rb_tree_node<std::pair<std::__cxx11::basic_string<char,
std::char_traits<char>, std::allocator<char> > const, int> > >::allocate(unsigned long, void const*)
/usr/include/c++/13/bits/new_allocator.h:151
...
```

189600 byte(s) (6%) in 9546 allocation(s)

```
#0 0x7f784c8fe548 in operator new(unsigned long)
../../../../src/libsanitizer/asan/asan_new_delete.cpp:95
#1 0x7f784c56b85a in void std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char>
>::_M_construct<char*>(char*, char*, std::forward_iterator_tag)
(/lib/x86_64-linux-gnu/libstdc++.so.6+0x16b85a) (BuildId: ca77dae775ec87540acd7218fa990c40d1c94ab1)
...
```

73728 byte(s) (2%) in 1 allocation(s)

```
#0 0x7f784c8fd9c7 in malloc ../../src/libsanitizer/asan/asan_malloc_linux.cpp:69
#1 0x7f784c4b738e (/lib/x86_64-linux-gnu/libstdc++.so.6+0xb738e) (BuildId:
ca77dae775ec87540acd7218fa990c40d1c94ab1)
#2 0x7f784d0cd71e in call_init elf/dl-init.c:74
#3 0x7f784d0cd823 in call_init elf/dl-init.c:120
#4 0x7f784d0cd823 in _dl_init elf/dl-init.c:121
```

ASAN/LSAN

```
// Prints stack traces for all live heap allocations ordered by total
// allocation size until top_percent of total live heap is shown. Top_percent
// should be between 1 and 100. At most max_number_of_contexts contexts
// (stack traces) are printed.
// Experimental feature currently available only with ASan on Linux/x86_64.
void SANITIZER_CDECL __sanitizer_print_memory_profile(size_t top_percent, size_t max_number_of_contexts);
```

ASAN/LSAN

```
// Prints stack traces for all live heap allocations ordered by total
// allocation size until top_percent of total live heap is shown. Top_percent
// should be between 1 and 100. At most max_number_of_contexts contexts
// (stack traces) are printed.
// Experimental feature currently available only with ASan on Linux/x86_64.
void SANITIZER_CDECL __sanitizer_print_memory_profile(size_t top_percent, size_t max_number_of_contexts);

/* Number of bytes, allocated and not yet freed by the application. */
size_t SANITIZER_CDECL __sanitizer_get_current_allocated_bytes(void);
```

ASAN/LSAN

```
// Prints stack traces for all live heap allocations ordered by total
// allocation size until top_percent of total live heap is shown. Top_percent
// should be between 1 and 100. At most max_number_of_contexts contexts
// (stack traces) are printed.
// Experimental feature currently available only with ASan on Linux/x86_64.
void SANITIZER_CDECL __sanitizer_print_memory_profile(size_t top_percent, size_t max_number_of_contexts);

/* Number of bytes, allocated and not yet freed by the application. */
size_t SANITIZER_CDECL __sanitizer_get_current_allocated_bytes(void);

/* Number of bytes, mmaped by the allocator to fulfill allocation requests.
Generally, for request of X bytes, allocator can reserve and add to free
lists a large number of chunks of size X to use them for future requests.
All these chunks count toward the heap size. Currently, allocator never
releases memory to OS (instead, it just puts freed chunks to free
lists). */
size_t SANITIZER_CDECL __sanitizer_get_heap_size(void);
```

ASAN/LSAN

```
// Prints stack traces for all live heap allocations ordered by total
// allocation size until top_percent of total live heap is shown. Top_percent
// should be between 1 and 100. At most max_number_of_contexts contexts
// (stack traces) are printed.
// Experimental feature currently available only with ASan on Linux/x86_64.
void SANITIZER_CDECL __sanitizer_print_memory_profile(size_t top_percent, size_t max_number_of_contexts);

/* Number of bytes, allocated and not yet freed by the application. */
size_t SANITIZER_CDECL __sanitizer_get_current_allocated_bytes(void);

/* Number of bytes, mmaped by the allocator to fulfill allocation requests.
Generally, for request of X bytes, allocator can reserve and add to free
lists a large number of chunks of size X to use them for future requests.
All these chunks count toward the heap size. Currently, allocator never
releases memory to OS (instead, it just puts freed chunks to free
lists). */
size_t SANITIZER_CDECL __sanitizer_get_heap_size(void);

/* Number of bytes, mmaped by the allocator, which can be used to fulfill
allocation requests. When a user program frees memory chunk, it can first
fall into quarantine and will count toward __sanitizer_get_free_bytes()
later. */
size_t SANITIZER_CDECL __sanitizer_get_free_bytes(void);
```

ASAN/LSAN

Можно построить на базе этого сэмплирующий профилировщик.

ASAN/LSAN

Можно построить на базе этого сэмплирующий профилировщик.

Например в виде подгружаемой через LD_PRELOAD динамической библиотеки

ASAN/LSAN

Можно построить на базе этого сэмплирующий профилировщик.

Например в виде подгружаемой через LD_PRELOAD динамической библиотеки

(на одном из предыдущих докладов я делал это, пишите если хотите повторения на подкасте Pure Virtual Cast :-)

ASAN/LSAN

Можно построить на базе этого сэмплирующий профилировщик.

Например в виде подгружаемой через LD_PRELOAD динамической библиотеки

(на одном из предыдущих докладов я делал это, пишите если хотите повторения на подкасте Pure Virtual Cast :-)

Но хотелось то не сэмплирующего, а точного.

Очевидно у инфраструктуры санитайзеров уже есть всё необходимое для того, чтобы сделать что-то похожее на то, что есть в TCMalloc.

ASAN/LSAN

Можно построить на базе этого сэмплирующий профилировщик.

Например в виде подгружаемой через LD_PRELOAD динамической библиотеки

(на одном из предыдущих докладов я делал это, пишите если хотите повторения на подкасте **Pure Virtual Cast** :-))

Но хотелось то не сэмплирующего, а точного.

Очевидно у инфраструктуры санитайзеров уже есть всё необходимое для того, чтобы сделать что-то похожее на то, что есть в TCMalloc.

Только лучше!

ASAN/LSAN

Можно построить на базе этого сэмплирующий профилировщик.

Например в виде подгружаемой через LD_PRELOAD динамической библиотеки

(на одном из предыдущих докладов я делал это, пишите если хотите повторения на подкасте **Pure Virtual Cast** :-))

Но хотелось то не сэмплирующего, а точного.

Очевидно у инфраструктуры санитайзеров уже есть всё необходимое для того, чтобы сделать что-то похожее на то, что есть в TCMalloc.

Только лучше!

(а самому реализовывать не хочется)

compiler_rt

Мой любимый проект!

[spoiler alert!]

compiler_rt

[spoiler alert!]

| llvm-project / compiler-rt / lib / | |
|------------------------------------|--|
| Name | Last commit message |
| .. | |
| BlocksRuntime | [compiler-rt] Test commit: remove some trailing white spaces. |
| asan | [asan][win] Fix CreateThread leak (#126738) |
| asan_abi | [Sanitizers][ABI] Remove too strong assert in asan_abi_shim (#81696) |
| builtins | [compiler-rt] Add support for big endian for Arm's __negdf2vfp (#127096) |
| cfi | [CFI] Allow LoongArch (#67314) |
| ctx_profile | [ctxprof] Flat profile collection (#130655) |
| dfsan | [sanitizer] VReport BeforeFork/AfterFork (#111900) |
| fuzzer | [libfuzzer] Clarify -max_len behavior on bigger files (#123095) |
| gwp_asan | [gwp_asan] Soft-transition ZXTEST_USE_STREAMABLE_MACROS removal (#121887) |
| hwasan | [asan][hwasan] Link RTUbsan_cxx into shared runtime |
| interception | [win/asan] GetInstructionSize: Support some more 7 or 8 byte instruct... |
| lsan | [compiler-rt] Remove support and workarounds for Android 4 and older (#... |
| memprof | [MemProf] Use correct print_text value (#125793) |
| msan | [Ubsan][Driver] Remove UBSAN C++ runtime from other sanitizers (#121006) |
| nsan | [sanitizer] Replace uptr by usize/SIZE_T in interfaces |
| orc | [ORC-RT] Rename 'orc_rt_*CWrapper*' types and functions to 'orc_rt_*W... |

compiler_rt

| llvm-project / compiler-rt / lib / | |
|------------------------------------|--|
| Name | Last commit message |
| .. | |
| BlocksRuntime | [compiler-rt] Test commit: remove some trailing white spaces. |
| asan | [asan][win] Fix CreateThread leak (#126738) |
| asan_abi | [Sanitizers][ABI] Remove too strong assert in asan_abi_shim (#81696) |
| builtins | [compiler-rt] Add support for big endian for Arm's __negdf2vfp (#127096) |
| cfi | [CFI] Allow LoongArch (#67314) |
| ctx_profile | [ctxprof] Flat profile collection (#130655) |
| dfsan | [sanitizer] VReport BeforeFork/AfterFork (#111900) |
| fuzzer | [libfuzzer] Clarify -max_len behavior on bigger files (#123095) |
| gwp_asan | [gwp_asan] Soft-transition ZXTEST_USE_STREAMABLE_MACROS removal (#121887) |
| hwasan | [asan][hwasan] Link RTUbsan_cxx into shared runtime |
| interception | [win/asan] GetInstructionSize: Support some more 7 or 8 byte instruct... |
| lsan | [compiler-rt] Remove support and workarounds for Android 4 and older (#... |
| memprof | [MemProf] Use correct print_text value (#125793) |
| msan | [Ubsan][Driver] Remove UBSAN C++ runtime from other sanitizers (#121006) |
| nsan | [sanitizer] Replace uptr by usize/SIZE_T in interfaces |
| orc | [ORC-RT] Rename 'orc_rt_*CWrapper*' types and functions to 'orc_rt_*W... |



[MemProf] profiler



teresajohnson committed on Oct 16, 2020

[MemProf] Memory profiling runtime support

See RFC for background:

<http://lists.llvm.org/pipermail/llvm-dev/2020-June/142744.html>

Follow on companion to the clang/llvm instrumentation support in D85948 and committed earlier.

This patch adds the compiler-rt runtime support for the memory profiling.

Note that much of this support was cloned from asan (and then greatly simplified and renamed). For example the interactions with the sanitizer_common allocators, error handling, interception, etc.

[MemProf] profiler



teresajohnson committed on Oct 16, 2020

[MemProf] Memory profiling runtime support

See RFC for background:

<http://lists.llvm.org/pipermail/llvm-dev/2020-June/142744.html>

Follow on companion to the clang/llvm instrumentation support in D85948 and committed earlier.

This patch adds the compiler-rt runtime support for the memory profiling.

ХОРОШО!

Note that much of this support was cloned from asan (and then greatly simplified and renamed). For example the interactions with the sanitizer_common allocators, error handling, interception, etc.

[MemProf] profiler



teresajohnson committed on Oct 16, 2020

[MemProf] Memory profiling runtime support

See RFC for background:

<http://lists.llvm.org/pipermail/llvm-dev/2020-June/142744.html>

Follow on companion to the clang/llvm instrumentation support in D85948 and committed earlier.

Странно...

This patch adds the compiler-rt runtime support for the memory profiling.

ХОРОШО!

Note that much of this support was cloned from asan (and then greatly simplified and renamed). For example the interactions with the sanitizer_common allocators, error handling, interception, etc.

[MemProf] profiler

И чуть дальше:

...

[MemProf] profiler

И чуть дальше:

...

The profile information will be used first for tooling,
and subsequently to guide the compiler optimizer and allocation runtime to
layout heap objects with improved spatial locality.

[MemProf] profiler

И чуть дальше:

...

The profile information will be used first for tooling,
and subsequently to guide the **compiler optimizer** and allocation runtime to
layout heap objects with improved spatial locality.

Профилировка памяти для PGO? То есть для оптимизаций времени компиляции?

[MemProf] profiler

И чуть дальше:

...

The profile information will be used first for tooling, and subsequently to guide the **compiler optimizer** and allocation runtime to layout heap objects with improved spatial locality.

Профилировка памяти для PGO? То есть для оптимизаций времени компиляции?



[MemProf] profiler

Давайте сперва разберемся как MemProf собирает профиль

[MemProf] profiler

Давайте сперва разберемся как MemProf собирает профиль

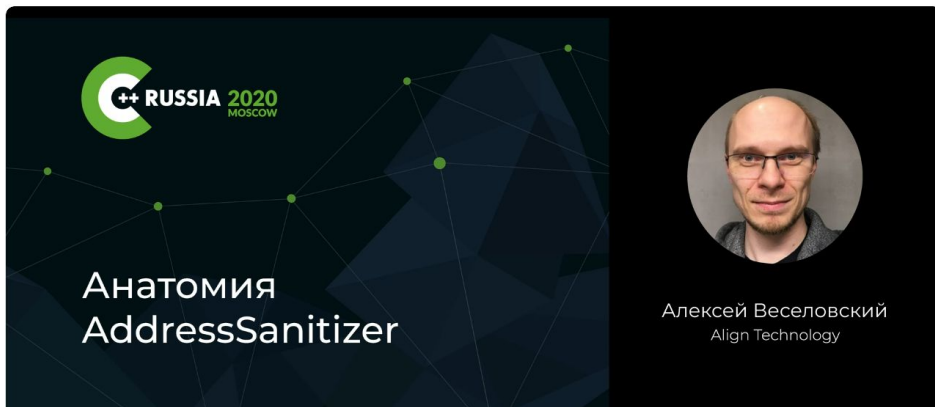
И что за профиль такой оно собирает

[MemProf] profiler

MemProf устройством своим максимально похож на ASAN

[MemProf] profiler

MemProf устройством своим максимально похож на ASAN



[MemProf] profiler

MemProf устройством своим максимально похож на ASAN:

Каждым 64 байтам user memory соответствует 8 байт shadow memory

[MemProf] profiler

MemProf устройством своим максимально похож на ASAN:

Каждым 64 байтам user memory соответствует 8 байт shadow memory:

```
shadow_addr = (address >> 6)*8 + kOffset;
```

[MemProf] profiler

MemProf устройством своим максимально похож на ASAN:

Каждым 64 байтам user memory соответствует 8 байт shadow memory:

```
shadow_addr = (address >> 6)*8 + kOffset;
```

```
address = 16;
```

| | | | | | | | | | | | | | | | | | | | | |
|---|-----|----|----|----|----|-----|----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|--|
| 0 | ... | 16 | 17 | 18 | 19 | ... | 63 | ... | 100 | 101 | 102 | 103 | 104 | 105 | 106 | 107 | 108 | 109 | 110 | |
| | | | | | | | | | | | | | | | | | | | | |

[MemProf] profiler

MemProf устройством своим максимально похож на ASAN:

Каждым 64 байтам user memory соответствует 8 байт shadow memory:

```
shadow_addr = (address >> 6)*8 + kOffset;
```

```
address = 16;
```

| | | | | | | | | | | | | | | | | | | | | |
|---|-----|----|----|----|----|-----|----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|--|
| 0 | ... | 16 | 17 | 18 | 19 | ... | 63 | ... | 100 | 101 | 102 | 103 | 104 | 105 | 106 | 107 | 108 | 109 | 110 | |
| | | | | | | | | | | | | | | | | | | | | |

```
kOffset = 100; // тут начинается shadow memory
```

[MemProf] profiler

MemProf устройством своим максимально похож на ASAN:

Каждым 64 байтам user memory соответствует 8 байт shadow memory:

```
shadow_addr = (address >> 6)*8 + kOffset;
```

```
address = 16;
```

| | | | | | | | | | | | | | | | | | | | | |
|---|-----|----|----|----|----|-----|----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|--|
| 0 | ... | 16 | 17 | 18 | 19 | ... | 63 | ... | 100 | 101 | 102 | 103 | 104 | 105 | 106 | 107 | 108 | 109 | 110 | |
| | | | | | | | | | | | | | | | | | | | | |

```
kOffset = 100; // тут начинается shadow memory
```

```
shadow_addr = (16 >> 6)*8 + 100 =
```

[MemProf] profiler

MemProf устройством своим максимально похож на ASAN:

Каждым 64 байтам user memory соответствует 8 байт shadow memory:

```
shadow_addr = (address >> 6)*8 + kOffset;
```

```
address = 16;
```

| | | | | | | | | | | | | | | | | | | | | |
|---|-----|----|----|----|----|-----|----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|--|
| 0 | ... | 16 | 17 | 18 | 19 | ... | 63 | ... | 100 | 101 | 102 | 103 | 104 | 105 | 106 | 107 | 108 | 109 | 110 | |
| | | | | | | | | | | | | | | | | | | | | |

```
kOffset = 100; // тут начинается shadow memory
```

```
shadow_addr = (16 >> 6)*8 + 100 = (16 / 64)*8 + 100
```

[MemProf] profiler

MemProf устройством своим максимально похож на ASAN:

Каждым 64 байтам user memory соответствует 8 байт shadow memory:

```
shadow_addr = (address >> 6)*8 + kOffset;
```

```
address = 16;
```

| | | | | | | | | | | | | | | | | | | | | |
|---|-----|----|----|----|----|-----|----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|--|
| 0 | ... | 16 | 17 | 18 | 19 | ... | 63 | ... | 100 | 101 | 102 | 103 | 104 | 105 | 106 | 107 | 108 | 109 | 110 | |
| | | | | | | | | | | | | | | | | | | | | |

```
kOffset = 100; // тут начинается shadow memory
```

```
shadow_addr = (16 >> 6)*8 + 100 = (16 / 64)*8 + 100 = 0*8 + 100;
```

[MemProf] profiler

MemProf устройством своим максимально похож на ASAN:

Каждым 64 байтам user memory соответствует 8 байт shadow memory:

```
shadow_addr = (address >> 6)*8 + kOffset;
```

```
address = 16;
```

| | | | | | | | | | | | | | | | | | | | | |
|---|-----|----|----|----|----|-----|----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|--|
| 0 | ... | 16 | 17 | 18 | 19 | ... | 63 | ... | 100 | 101 | 102 | 103 | 104 | 105 | 106 | 107 | 108 | 109 | 110 | |
| | | | | | | | | | | | | | | | | | | | | |

```
kOffset = 100; // тут начинается shadow memory
```

```
shadow_addr = (16 >> 6)*8 + 100 = (16 / 64)*8 + 100 = 0*8 + 100 = 100;
```

[MemProf] profiler

Каждым 64 байтам user memory соответствует 8 байт shadow memory:

```
shadow_addr = (address >> 6)*8 + kOffset;
```

```
address = 16;
```

```
kOffset = 100; // тут начинается shadow memory
```

```
shadow_addr = (16 >> 6)*8 + 100 = (16 / 64)*8 + 100 = 0*8 + 100 = 100;
```

| | | | | | | | | | | | | | | | | | | | | |
|---|-----|----|----|----|----|-----|----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|--|
| 0 | ... | 16 | 17 | 18 | 19 | ... | 63 | ... | 100 | 101 | 102 | 103 | 104 | 105 | 106 | 107 | 108 | 109 | 110 | |
| | | | | | | | | | | | | | | | | | | | | |

Компилятор добавляет доп инструкции к каждому load/store (поэтому нужна инструментация кода!)

[MemProf] profiler

Каждым 64 байтам user memory соответствует 8 байт shadow memory:

```
shadow_addr = (address >> 6)*8 + kOffset;
```

```
address = 16;
```

```
kOffset = 100; // тут начинается shadow memory
```

```
shadow_addr = (16 >> 6)*8 + 100 = (16 / 64)*8 + 100 = 0*8 + 100 = 100;
```

| | | | | | | | | | | | | | | | | | | | | |
|---|-----|----|----|----|----|-----|----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|--|
| 0 | ... | 16 | 17 | 18 | 19 | ... | 63 | ... | 100 | 101 | 102 | 103 | 104 | 105 | 106 | 107 | 108 | 109 | 110 | |
| | | | | | | | | | | | | | | | | | | | | |

Компилятор добавляет доп инструкции к каждому load/store (поэтому нужна инструментация кода!)

```
*a = 14;
```

[MemProf] profiler

Каждым 64 байтам user memory соответствует 8 байт shadow memory:

```
shadow_addr = (address >> 6)*8 + kOffset;
```


```
address = 16;
```

```
kOffset = 100; // тут начинается shadow memory
```

```
shadow_addr = (16 >> 6)*8 + 100 = (16 / 64)*8 + 100 = 0*8 + 100 = 100;
```

| | | | | | | | | | | | | | | | | | | | |
|---|-----|----|----|----|----|-----|----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 0 | ... | 16 | 17 | 18 | 19 | ... | 63 | ... | 100 | 101 | 102 | 103 | 104 | 105 | 106 | 107 | 108 | 109 | 110 |
| | | | | | | | | | | | | | | | | | | | |

Компилятор добавляет доп инструкции к каждому load/store (поэтому нужна инструментация кода!)

```
*a = 14;  uint8_t* addr = a;  
uint64_t* shadow_addr = (addr>>6)*8 + kOffset;  
(*shadow_addr)++;  
*a = 14;
```

[MemProf] profiler

Каждым 64 байтам user memory соответствует 8 байт shadow memory:

```
shadow_addr = (address >> 6)*8 + kOffset;
```


```
address = 16;
```

```
kOffset = 100; // тут начинается shadow memory
```

```
shadow_addr = (16 >> 6)*8 + 100 = (16 / 64)*8 + 100 = 0*8 + 100 = 100;
```

| | | | | | | | | | | | | | | | | | | | |
|---|-----|----|----|----|----|-----|----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 0 | ... | 16 | 17 | 18 | 19 | ... | 63 | ... | 100 | 101 | 102 | 103 | 104 | 105 | 106 | 107 | 108 | 109 | 110 |
| | | | | | | | | | | | | | | | | | | | |

Компилятор добавляет доп инструкции к каждому load/store (поэтому нужна инструментация кода!)

```
*a = 14;  uint8_t* addr = a;  
uint64_t* shadow_addr = (addr>>6)*8 + kOffset;  
(*shadow_addr)++;  
*a = 14;
```

Да, shadow memory используется для учета числа обращений к данной ячейке памяти!

[MemProf] profiler

То есть MemProf тоже смотрит использование памяти. Но другое! Как часто её читать-писать.

[MemProf] profiler

То есть MemProf тоже смотрит использование памяти. Но другое! Как часто её читать-писать.

С середины 2024 года есть и другой режим гранулярность: 8 байт memory в 1 байт shadow:

```
shadow_addr = (address >> 3) + kOffset;
```

[MemProf] profiler

То есть MemProf тоже смотрит использование памяти. Но другое! Как часто её читать-писать.

С середины 2024 года есть и другой режим гранулярность: 8 байт memory в 1 байт shadow:

```
shadow_addr = (address >> 3) + kOffset;
```

В этом случае если счетчик достиг максимального значения (255), он дальше не увеличивается.

[MemProf] profiler

То есть MemProf тоже смотрит использование памяти. Но другое! Как часто её читать-писать.

С середины 2024 года есть и другой режим гранулярность: 8 байт memory в 1 байт shadow:

```
shadow_addr = (address >> 3) + kOffset;
```

В этом случае если счетчик достиг максимального значения (255), он дальше не увеличивается.

И тогда это совсем похоже на ASAN.

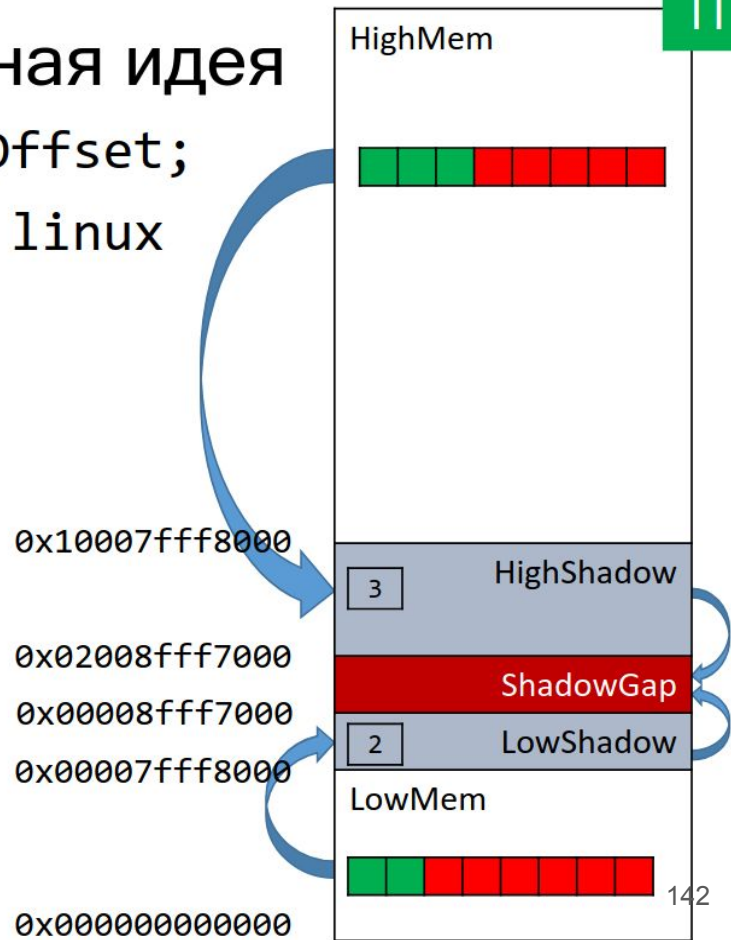
Address Sanitizer основная идея

```
shadow_address = (address >> 3) + kOffset;
```

```
kOffset = 0x7fff8000; // for x86_64 linux
```

| | |
|-------------------------------------|------------|
| [0x10007fff8000, 0x7fffffff8000] | HighMem |
| [0x02008fff7000, 0x10007fff7fff] | HighShadow |
| [0x00008fff7000, 0x02008fff6fff] | ShadowGap |
| [0x00007fff8000, 0x00008fff6fff] | LowShadow |
| [0x000000000000, 0x00007fff7fff] | LowMem |

2020 ГОД



[MemProf] profiler

Давайте попробуем заиспользовать в режиме 8 байт на 1 байт (Histogram mode):

```
$ cat struct.cpp
```

[MemProf] profiler

Давайте попробуем заиспользовать в режиме 8 байт на 1 байт (Histogram mode):

```
$ cat struct.cpp
#include <stdint.h>
struct S {
    uint64_t a;
    char c;
    uint64_t d;
};
int main() {
    S* s = new S{0,0,0};
    for (int i=0; i<10; ++i) s->a += i;
    s->c = 'c';
    s->c = 'p';
    s->c = 'p';
    s->d = s->a + s->c;
    s->d = s->c;
}
```

[MemProf] profiler

Давайте попробуем заиспользовать в режиме 8 байт на 1 байт (Histogram mode):

```
$ cat struct.cpp
#include <stdint.h>
struct S {
    uint64_t a;
    char c;
    uint64_t d;
};
int main() {
    S* s = new S{0,0,0};
    for (int i=0; i<10; ++i) s->a += i;
    s->c = 'c';
    s->c = 'p';
    s->c = 'p';
    s->d = s->a + s->c;
    s->d = s->c;
}
$ clang++ -g -fmemory-profile -mllvm -memprof-histogram struct.cpp
```

[MemProf] profiler

Давайте попробуем заиспользовать в режиме 8 байт на 1 байт (Histogram mode):

```
$ cat struct.cpp
#include <stdint.h>
struct S {
    uint64_t a;
    char c;
    uint64_t d;
};
int main() {
    S* s = new S{0,0,0};
    for (int i=0; i<10; ++i) s->a += i;
    s->c = 'c';
    s->c = 'p';
    s->c = 'p';
    s->d = s->a + s->c;
    s->d = s->c;
}
$ clang++ -g -fmemory-profile -mllvm -memprof-histogram struct.cpp
$ MEMPROF_OPTIONS=print_text=1 ./a.out
```

[MemProf] profiler

Давайте попробуем заиспользовать в режиме 8 байт на 1 байт (Histogram mode):

```
$ cat struct.cpp
#include <stdint.h>
struct S {
    uint64_t a;
    char c;
    uint64_t d;
};
int main() {
    S* s = new S{0,0,0};
    for (int i=0; i<10; ++i) s->a += i;
    s->c = 'c';
    s->c = 'p';
    s->c = 'p';
    s->d = s->a + s->c;
    s->d = s->c;
}
$ clang++ -g -fmemory-profile -mllvm -memprof-histogram struct.cpp
$ MEMPROF_OPTIONS=print_text=1 ./a.out
$ less memprof.profraw.301305
```

[MemProf] profiler

Давайте попробуем заиспользовать в режиме 8 байт на 1 байт (Histogram mode):

```
$ less memprof.profraw.301305
```


[MemProf] profiler

21+5+2 = 28

```
$ less memprof.profraw.301305
```

```
Memory allocation stack id = 1
```

```
  alloc_count 1, size (ave/min/max) 73728.00 / 73728 / 73728
```

```
  access_count (ave/min/max): 0.00 / 0 / 0
```

```
  lifetime (ave/min/max): 267.00 / 267 / 267
```

```
  num migrated: 0, num lifetime overlaps: 0, num same alloc cpu: 0, num same dealloc_cpu: 0
```

```
AccessCountHistogram[9216]: 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

```
Memory allocation stack id = 2
```

```
  alloc_count 1, size (ave/min/max) 24.00 / 24 / 24
```

```
  access_count (ave/min/max): 28.00 / 28 / 28
```

```
  lifetime (ave/min/max): 0.00 / 0 / 0
```

```
  num migrated: 0, num lifetime overlaps: 0, num same alloc cpu: 0, num same dealloc_cpu: 0
```

```
AccessCountHistogram[3]: 21 5 2
```

```
Stack for id 2:
```

```
#0 0x638cf53e359d in operator new(unsigned long)
```

```
/home/valexey/Projects/cpp25/llvm-project/compiler-rt/lib/memprof/memprof_new_delete.cpp:48:
```

```
#1 0x638cf53e49a9 in main /home/valexey/Projects/cpp25/test/struct.cpp:10:9
```

```
#2 0x71273342a1c9 in __libc_start_call_main csu/../sysdeps/nptl/libc_start_call_main.h:58
```

```
#3 0x71273342a28a in __libc_start_main csu/../csu/libc-start.c:360:3
```

```
#4 0x638cf53833b4 in _start (/home/valexey/Projects/cpp25/test/aPB.out+0x283b4)
```

```
Stack for id 1:
```

```
#0 0x638cf53b7e43 in malloc /home/valexey/Projects/cpp25/llvm-project/compiler-rt/lib/mem
```

```
#1 0x7127338b738e (/lib/x86_64-linux-gnu/libstdc++.so.6+0xb738e) (BuildId: ca77dae775ec)
```

```
#2 0x712733ba171e in call_init elf/dl-init.c:74:3
```

```
#3 0x712733ba1823 in call_init elf/dl-init.c:120:14
```

```
#4 0x712733ba1823 in _dl_init elf/dl-init.c:121:5
```

```
#5 0x712733bbb59f (/lib64/ld-linux-x86-64.so.2+0x1f59f) (BuildId: 1c8db5f83bba514f8fd5f)
```

```
struct S {
    uint64_t a; // 21
    char c;     // 5
    uint64_t d; // 2
};

int main() {
    S* s = new S{0,0,0};
    for (int i=0; i<10; ++i) s->a += i;
    s->c = 'c';
    s->c = 'p';
    s->c = 'p';
    s->d = s->a + s->c;
    s->d = s->c;
}
```

[MemProf] profiler

Кажется даже без последующего PGO информация достаточно интересная для ручной оптимизации

```
$ less memprof.profraw.301305
```

```
Memory allocation stack id = 1
```

```
  alloc_count 1, size (ave/min/max) 73728.00 / 73728 / 73728
```

```
  access_count (ave/min/max): 0.00 / 0 / 0
```

```
  lifetime (ave/min/max): 267.00 / 267 / 267
```

```
  num migrated: 0, num lifetime overlaps: 0, num same alloc cpu: 0, num same dealloc_cpu: 0
```

```
AccessCountHistogram[9216]: 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

```
Memory allocation stack id = 2
```

```
  alloc_count 1, size (ave/min/max) 24.00 / 24 / 24
```

```
  access_count (ave/min/max): 28.00 / 28 / 28
```

```
  lifetime (ave/min/max): 0.00 / 0 / 0
```

```
  num migrated: 0, num lifetime overlaps: 0, num same alloc cpu: 0, num same dealloc_cpu: 0
```

```
AccessCountHistogram[3]: 21 5 2
```

```
Stack for id 2:
```

```
#0 0x638cf53e359d in operator new(unsigned long)
```

```
/home/valexey/Projects/cpp25/llvm-project/compiler-rt/lib/memprof/memprof_new_delete.cpp:48:
```

```
#1 0x638cf53e49a9 in main /home/valexey/Projects/cpp25/test/struct.cpp:10:9
```

```
#2 0x71273342a1c9 in __libc_start_call_main csu/./sysdeps/nptl/libc_start_call_main.h:58
```

```
#3 0x71273342a28a in __libc_start_main csu/./csu/libc-start.c:360:3
```

```
#4 0x638cf53833b4 in _start (/home/valexey/Projects/cpp25/test/aPB.out+0x283b4)
```

```
Stack for id 1:
```

```
#0 0x638cf53b7e43 in malloc /home/valexey/Projects/cpp25/llvm-project/compiler-rt/lib/mem
```

```
#1 0x7127338b738e (/lib/x86_64-linux-gnu/libstdc++.so.6+0xb738e) (BuildId: ca77dae775ec)
```

```
#2 0x712733ba171e in call_init elf/dl-init.c:74:3
```

```
#3 0x712733ba1823 in call_init elf/dl-init.c:120:14
```

```
#4 0x712733ba1823 in _dl_init elf/dl-init.c:121:5
```

```
#5 0x712733bbb59f (/lib64/ld-linux-x86-64.so.2+0x1f59f) (BuildId: 1c8db5f83bba514f8fd5f)
```

```
struct S {
    uint64_t a; // 21
    char c;    // 5
    uint64_t d; // 2
};

int main() {
    S* s = new S{0,0,0};
    for (int i=0; i<10; ++i) s->a += i;
    s->c = 'c';
    s->c = 'p';
    s->c = 'p';
    s->d = s->a + s->c;
    s->d = s->c;
}
```

[MemProf] PGO

Но давайте поговорим про PGO

[MemProf] PGO

Но давайте поговорим про PGO

PGO обычно состоит из трех шагов:

1. Сборка для профилирования
2. Сбор профиля
3. Сборка с использованием профиля

[MemProf] PGO

Но давайте поговорим про PGO

PGO обычно состоит из трех шагов:

1. **Сборка для профилирования (done)**
2. **Сбор профиля (done)**
3. Сборка с использованием профиля

[MemProf] PGO

Но давайте поговорим про PGO

PGO обычно состоит из трех шагов:

1. **Сборка для профилирования (done)**
2. **Сбор профиля (done)**
3. Сборка с использованием профиля

Нам осталось собрать с собранным профилем. Но...

[MemProf] PGO

Но давайте поговорим про PGO

PGO обычно состоит из трех шагов:

1. **Сборка для профилирования (done)**
2. **Сбор профиля (done)**
3. Сборка с использованием профиля

Нам осталось собрать с собранным профилем. Но...

Как **ВООБЩЕ** компилятор сможет нам что-то оптимизировать в плане использования heap?

[MemProf] PGO

```
// Upper bound on lifetime access density (accesses per byte per lifetime sec)
// for marking an allocation cold.
cl::opt<float> MemProfLifetimeAccessDensityColdThreshold(
    "memprof-lifetime-access-density-cold-threshold", cl::init(0.05),
    cl::Hidden,
    cl::desc("The threshold the lifetime access density (accesses per byte per "lifetime sec) must be under to consider an allocation cold"));
```

[MemProf] PGO

```
// Upper bound on lifetime access density (accesses per byte per lifetime sec)
// for marking an allocation cold.
cl::opt<float> MemProfLifetimeAccessDensityColdThreshold(
    "memprof-lifetime-access-density-cold-threshold", cl::init(0.05),
    cl::Hidden,
    cl::desc("The threshold the lifetime access density (accesses per byte per "lifetime sec) must be under to consider an allocation cold"));

// Lower bound on lifetime to mark an allocation cold (in addition to accesses
// per byte per sec above). This is to avoid pessimizing short lived objects.
cl::opt<unsigned> MemProfAveLifetimeColdThreshold(
    "memprof-ave-lifetime-cold-threshold", cl::init(200), cl::Hidden,
    cl::desc("The average lifetime (s) for an allocation to be considered cold"));
```

[MemProf] PGO

```
// Upper bound on lifetime access density (accesses per byte per lifetime sec)
// for marking an allocation cold.
cl::opt<float> MemProfLifetimeAccessDensityColdThreshold(
    "memprof-lifetime-access-density-cold-threshold", cl::init(0.05),
    cl::Hidden,
    cl::desc("The threshold the lifetime access density (accesses per byte per "lifetime sec) must be under to consider an allocation cold"));

// Lower bound on lifetime to mark an allocation cold (in addition to accesses
// per byte per sec above). This is to avoid pessimizing short lived objects.
cl::opt<unsigned> MemProfAveLifetimeColdThreshold(
    "memprof-ave-lifetime-cold-threshold", cl::init(200), cl::Hidden,
    cl::desc("The average lifetime (s) for an allocation to be considered cold"));

// Lower bound on average lifetime accesses density (total life time access
// density / alloc count) for marking an allocation hot.
cl::opt<unsigned> MemProfMinAveLifetimeAccessDensityHotThreshold(
    "memprof-min-ave-lifetime-access-density-hot-threshold", cl::init(1000),
    cl::Hidden,
    cl::desc("The minimum TotalLifetimeAccessDensity / AllocCount for an allocation to be considered hot"));
```

[MemProf] PGO

```
// Upper bound on lifetime access density (accesses per byte per lifetime sec)
// for marking an allocation cold.
cl::opt<float> MemProfLifetimeAccessDensityColdThreshold(
    "memprof-lifetime-access-density-cold-threshold", cl::init(0.05),
    cl::Hidden,
    cl::desc("The threshold the lifetime access density (accesses per byte per "lifetime sec) must be under to consider an allocation cold"));

// Lower bound on lifetime to mark an allocation cold (in addition to accesses
// per byte per sec above). This is to avoid pessimizing short lived objects.
cl::opt<unsigned> MemProfAveLifetimeColdThreshold(
    "memprof-ave-lifetime-cold-threshold", cl::init(200), cl::Hidden,
    cl::desc("The average lifetime (s) for an allocation to be considered cold"));

// Lower bound on average lifetime accesses density (total life time access
// density / alloc count) for marking an allocation hot.
cl::opt<unsigned> MemProfMinAveLifetimeAccessDensityHotThreshold(
    "memprof-min-ave-lifetime-access-density-hot-threshold", cl::init(1000),
    cl::Hidden,
    cl::desc("The minimum TotalLifetimeAccessDensity / AllocCount for an allocation to be considered hot"));
```

Allocation != Call

Тут отличие от обычного PGO – там выделяют горячие ветки исполнения.
Тут – горячие аллоцированные куски памяти

[MemProf] PGO

Кроме всего прочего, нам ещё обещали вот это:

“As a result, **DTLB** and cache utilization will be improved, and program IPC (performance) will be increased due to reduced TLB and cache misses.”

[MemProf] PGO

Кроме всего прочего, нам ещё обещали вот это:

“As a result, **DTLB** and cache utilization will be improved, and program IPC (performance) will be increased due to reduced TLB and cache misses.”

То есть это точно не про то, как компилятор раскладывает код в бинарнике. Это именно что про кучу.

[MemProf] PGO

Кроме всего прочего, нам ещё обещали вот это:

“As a result, **DTLB** and cache utilization will be improved, and program IPC (performance) will be increased due to reduced TLB and cache misses.”

То есть это точно не про то, как компилятор раскладывает код в бинарнике. Это именно что про кучу.

Но кучей заведует менеджер памяти, причем во время исполнения программы. А менеджер памяти не является частью clang/llvm.

[MemProf] PGO

Кроме всего прочего, нам ещё обещали вот это:

“As a result, **DTLB** and cache utilization will be improved, and program IPC (performance) will be increased due to reduced TLB and cache misses.”

То есть это точно не про то, как компилятор раскладывает код в бинарнике. Это именно что про кучу.

Но кучей заведует менеджер памяти, причем во время исполнения программы. А менеджер памяти не является частью clang/llvm.

Последний кусочек пазла:

[MemProf] PGO

Кроме всего прочего, нам ещё обещали вот это:

“As a result, **DTLB** and cache utilization will be improved, and program IPC (performance) will be increased due to reduced TLB and cache misses.”

То есть это точно не про то, как компилятор раскладывает код в бинарнике. Это именно что про кучу.

Но кучей заведует менеджер памяти, причем во время исполнения программы. А менеджер памяти не является частью clang/llvm.

Последний кусочек пазла: TCMalloc (тот что **не** часть gperftools).

[MemProf] PGO

Последний кусочек пазла: TCMalloc (тот что **не** часть gperftools).

[MemProf] PGO

Последний кусочек пазла: TCMalloc (тот что **не** часть gperftools).

В 2021 году в него добавили поддержку хинтинга того, как выделяемый кусок памяти будет использоваться: <https://google.github.io/tcmalloc/temeraire.html>

[MemProf] PGO

Последний кусочек пазла: TCMalloc (тот что **не** часть gperftools).

В 2021 году в него добавили поддержку хинтинга того, как выделяемый кусок памяти будет использоваться: <https://google.github.io/tcmalloc/temeraire.html>

“TCMalloc provides a [hot_cold_variant](#) that applications may use to provide hints as to how frequently an allocation would be accessed. It’s more likely that a larger-sized page may be considered hot by kernel even if some allocations are frequently accessed and placed together with infrequently-accessed data. By specifying hot and cold hints, TCMalloc may separate allocations to improve data locality for *hot* heap, and the effectiveness of [memory tiering](#) by placing *cold* heap on a slower, but cheaper, memory tier. “

[MemProf] PGO

Последний кусочек пазла: TCMalloc (тот что **не** часть gperftools).

В 2021 году в него добавили поддержку хинтинга того, как выделяемый кусок памяти будет использоваться: <https://google.github.io/tcmalloc/temeraire.html>

“TCMalloc provides a [hot_cold_variant](#) that applications may use to provide hints as to how frequently an allocation would be accessed. It’s more likely that a larger-sized page may be considered hot by kernel even if some allocations are frequently accessed and placed together with infrequently-accessed data. By specifying hot and cold hints, TCMalloc may separate allocations to improve data locality for *hot* heap, and the effectiveness of [memory tiering](#) by placing *cold* heap on a slower, but cheaper, memory tier. “

То есть эти хинты позволяют менеджеру памяти держать мухи отдельно, а котлеты отдельно.

[MemProf] PGO

Последний кусочек пазла: TCMalloc (тот что **не** часть gperftools).

В 2021 году в него добавили поддержку хинтинга того, как выделяемый кусок памяти будет использоваться: <https://google.github.io/tcmalloc/temeraire.html>

“TCMalloc provides a [hot_cold variant](#) that applications may use to provide hints as to how frequently an allocation would be accessed. It’s more likely that a larger-sized page may be considered hot by kernel even if some allocations are frequently accessed and placed together with infrequently-accessed data. By specifying hot and cold hints, TCMalloc may separate allocations to improve data locality for *hot* heap, and the effectiveness of [memory tiering](#) by placing *cold* heap on a slower, but cheaper, memory tier. “

То есть эти хинты позволяют менеджеру памяти держать мухи отдельно, а котлеты отдельно. Интерфейс у этого такой:

```
void* operator new(size_t size, tcmalloc::hot_cold_t hot_cold) noexcept(false);  
void* operator new(size_t size, const std::nothrow_t, tcmalloc::hot_cold_t hot_cold) noexcept;
```

hot_cold принимает значения от 0 (очень холодная, редко используемая, в плане доступа) до 255 (очень горячая память).

[MemProf] PGO

```
void* operator new(size_t size, tcmalloc::hot_cold_t hot_cold) noexcept(false);  
void* operator new(size_t size, const std::nothrow_t, tcmalloc::hot_cold_t hot_cold) noexcept;
```

hot_cold принимает значения от 0 (очень холодная, редко используемая, в плане доступа) до 255 (очень горячая память).

[MemProf] PGO

```
void* operator new(size_t size, tcmalloc::hot_cold_t hot_cold) noexcept(false);  
void* operator new(size_t size, const std::nothrow_t, tcmalloc::hot_cold_t hot_cold) noexcept;
```

hot_cold принимает значения от 0 (очень холодная, редко используемая, в плане доступа) до 255 (очень горячая память).

Что делает компилятор:

-

[MemProf] PGO

```
void* operator new(size_t size, tcmalloc::hot_cold_t hot_cold) noexcept(false);  
void* operator new(size_t size, const std::nothrow_t, tcmalloc::hot_cold_t hot_cold) noexcept;
```

hot_cold принимает значения от 0 (очень холодная, редко используемая, в плане доступа) до 255 (очень горячая память).

Что делает компилятор:

- каждая аллокация памяти различается по “каллстеку” (а не просто по строчке кода, где была сделана). Мы это видели примерно во всех профилировщиках памяти

[MemProf] PGO

```
void* operator new(size_t size, tcmalloc::hot_cold_t hot_cold) noexcept(false);  
void* operator new(size_t size, const std::nothrow_t, tcmalloc::hot_cold_t hot_cold) noexcept;
```

hot_cold принимает значения от 0 (очень холодная, редко используемая, в плане доступа) до 255 (очень горячая память).

Что делает компилятор:

- каждая аллокация памяти различается по “каллстеку” (а не просто по строчке кода, где была сделана). Мы это видели примерно во всех профилировщиках памяти
- для каждой аллокации памяти у нас есть профиль со статистикой

[MemProf] PGO

```
void* operator new(size_t size, tcmalloc::hot_cold_t hot_cold) noexcept(false);  
void* operator new(size_t size, const std::nothrow_t, tcmalloc::hot_cold_t hot_cold) noexcept;
```

hot_cold принимает значения от 0 (очень холодная, редко используемая, в плане доступа) до 255 (очень горячая память).

Что делает компилятор:

- каждая аллокация памяти различается по “каллстеку” (а не просто по строчке кода, где была сделана). Мы это видели примерно во всех профилировщиках памяти
- для каждой аллокации памяти у нас есть профиль со статистикой
- исходя из статистики и собственных настроек компилятор решает – горяча или же холодна данная аллокация

[MemProf] PGO

```
void* operator new(size_t size, tcmalloc::hot_cold_t hot_cold) noexcept(false);  
void* operator new(size_t size, const std::nothrow_t, tcmalloc::hot_cold_t hot_cold) noexcept;
```

hot_cold принимает значения от 0 (очень холодная, редко используемая, в плане доступа) до 255 (очень горячая память).

Что делает компилятор:

- каждая аллокация памяти различается по “каллстеку” (а не просто по строчке кода, где была сделана). Мы это видели примерно во всех профилировщиках памяти
- для каждой аллокации памяти у нас есть профиль со статистикой
- исходя из статистики и собственных настроек компилятор решает – горяча или же холодна данная аллокация
- и если она холодна, то заменяет обычный new на new с хинтом, что использоваться эта аллокация будет редко (hot_cold = 1)

[MemProf] PGO

```
void* operator new(size_t size, tcmalloc::hot_cold_t hot_cold) noexcept(false);  
void* operator new(size_t size, const std::nothrow_t, tcmalloc::hot_cold_t hot_cold) noexcept;
```

hot_cold принимает значения от 0 (очень холодная, редко используемая, в плане доступа) до 255 (очень горячая память).

Что делает компилятор:

- каждая аллокация памяти различается по “каллстеку” (а не просто по строчке кода, где была сделана). Мы это видели примерно во всех профилировщиках памяти
- для каждой аллокации памяти у нас есть профиль со статистикой
- исходя из статистики и собственных настроек компилятор решает – горяча или же холодна данная аллокация
- и если она холодна, то заменяет обычный new на new с хинтом, что использоваться эта аллокация будет редко (hot_cold = 1)
- но есть проблема – на каждую “строчку” в коде с new, на самом деле приходится НЕСКОЛЬКО уникальных аллокаций (разный стектрейс). Разной степени горячести.

[MemProf] PGO

```
$ cat hotcold.cpp
```

[MemProf] PGO

Одна функция выделяет. Другая использует выделенное двумя разными способами.

```
$ cat hotcold.cpp
int* get_mem(int n) {
    return new int[n];
}

int* use_mem(int flag) {
    int* m;
    if (flag>3) { // hot branch
        m = get_mem(10);
        memset(m, 0, sizeof(int)*10);
        m[1]=flag;
        m[0]=flag-1;
    } else { // cold branch
        m = get_mem(10);
        sleep(10);
        m[0]=flag;
        m[1]=0;
    }
    return m;
}
```

```
int main(int argc, char *argv[]) {
    int* m = use_mem(argc);
    int res = m[0]+m[1];
    delete[] m;

    m = use_mem(argc*4);
    res += m[0]+m[1];
    delete[] m;
    return res;
}
```

[MemProf] PGO

Если дольше 5 секунд, то холодное.

```
$ cat hotcold.cpp
int* get_mem(int n) {
    return new int[n];
}

int* use_mem(int flag) {
    int* m;
    if (flag>3) { // hot branch
        m = get_mem(10);
        memset(m, 0, sizeof(int)*10);
        m[1]=flag;
        m[0]=flag-1;
    } else { // cold branch
        m = get_mem(10);
        sleep(10);
    }
}
```

```
int main(int argc, char *argv[]) {
    int* m = use_mem(argc);
    int res = m[0]+m[1];
    delete[] m;

    m = use_mem(argc*4);
    res += m[0]+m[1];
    delete[] m;
    return res;
}
```

```
$ clang++ -ltcmalloc -g -fmemory-profile-use=t.memprofdata -O1
-Wl,-mllvm,-enable-memprof-context-disambiguation -Wl,-mllvm,-supports-hot-cold-new
-Wl,-mllvm,-optimize-hot-cold-new -mllvm --memprof-ave-lifetime-cold-threshold=5 -fno-inline hotcold.cpp
```

[MemProf] PGO

Посмотрим что получилось (в асме, да)

```
<use_mem(int)>:
  push  r14
  push  rbx
  push  rax
  mov   ebx,edi
  cmp   edi,0x4
  jl    <use_mem(int)+0x2b>
  call  <get_mem(int)>
  xorps xmm0,xmm0
  movups XMMWORD PTR [rax],xmm0
  movups XMMWORD PTR [rax+0x10],xmm0
  mov   QWORD PTR [rax+0x20],0x0
  mov   DWORD PTR [rax+0x4],ebx
  dec   ebx
  mov   DWORD PTR [rax],ebx
  jmp   <use_mem(int)+0x4b>
  call  <get_mem(int) [clone .memprof.1]>
  mov   r14,rax
  mov   edi,0xa
  call  <sleep@plt>
  mov   rax,r14
  mov   DWORD PTR [r14],ebx
  mov   DWORD PTR [r14+0x4],0x0
  add   rsp,0x8
  pop   rbx
  pop   r14
  ret
```

[MemProf] PGO

Склонировал `get_mem` (сделал холодную версию), и заменил её вызов

```
<use_mem(int)>:
    push    r14
    push    rbx
    push    rax
    mov     ebx,edi
    cmp     edi,0x4
    jnl    <use_mem(int)+0x2b>
    call   <get_mem(int)>
    xorps  xmm0,xmm0
    movups XMMWORD PTR [rax],xmm0
    movups XMMWORD PTR [rax+0x10],xmm0
    mov     QWORD PTR [rax+0x20],0x0
    mov     DWORD PTR [rax+0x4],ebx
    dec     ebx
    mov     DWORD PTR [rax],ebx
    jmp    <use_mem(int)+0x4b>
    call   <get_mem(int) [clone .memprof.1]>
    mov     r14,rax
    mov     edi,0xa
    call   <sleep@plt>
    mov     rax,r14
    mov     DWORD PTR [r14],ebx
    mov     DWORD PTR [r14+0x4],0x0
    add     rsp,0x8
    pop     rbx
    pop     r14
    ret
```

```
<get_mem(int) [clone .memprof.1]>:
    mov     edi,0x28
    mov     esi,0x1
    jmp    <operator new[](unsigned long, __hot_cold_t)@plt>

<get_mem(int)>:
    mov     edi,0x28
    jmp    <operator new[](unsigned long)@plt>
```

[MemProf] PGO

Чуть-чуть поменяем пример

```
$ cat hotcold.cpp
int* get_mem(int n) {
    return new int[n];
}

int* use_mem(int flag) {
    int* m = get_mem(10); // <--
    if (flag>3) { // hot branch
        // m = get_mem(10);
        memset(m, 0, sizeof(int)*10);
        m[1]=flag;
        m[0]=flag-1;
    } else { // cold branch
        // m = get_mem(10);
        sleep(10);
        m[0]=flag;
        m[1]=0;
    }
    return m;
}
```

```
int main(int argc, char *argv[]) {
    int* m = use_mem(argc);
    int res = m[0]+m[1];
    delete[] m;

    m = use_mem(argc*4);
    res += m[0]+m[1];
    delete[] m;
    return res;
}
```

[MemProf] PGO

Посмотрим результат - будто бы оптимизация пропала

```
<use_mem(int)>:
  push  r14
  push  rbx
  push  rax
  mov   ebx,edi
  call  <get_mem(int)>
  cmp   ebx,0x4
  jl    <use_mem(int)+0x2b>
  xorps xmm0,xmm0
  movups XMMWORD PTR [rax],xmm0
  movups XMMWORD PTR [rax+0x10],xmm0
  mov   QWORD PTR [rax+0x20],0x0
  mov   DWORD PTR [rax+0x4],ebx
  dec   ebx
  mov   DWORD PTR [rax],ebx
  jmp   <use_mem(int)+0x46>
  mov   edi,0xa
  mov   r14,rax
  call  <sleep@plt>
  mov   rax,r14
  mov   DWORD PTR [r14],ebx
  mov   DWORD PTR [r14+0x4],0x0
  add   rsp,0x8
  pop   rbx
  pop   r14
```

[MemProf] PGO

Посмотрим результат - будто бы оптимизация пропала. Но нет.

```
<use_mem(int)>:
  push  r14
  push  rbx
  push  rax
  mov   ebx,edi
  call  <get_mem(int)>
  cmp   ebx,0x4
  jl    <use_mem(int)+0x2b>
  xorps xmm0,xmm0
  movups XMMWORD PTR [rax],xmm0
  movups XMMWORD PTR [rax+0x10],xmm0
  mov   QWORD PTR [rax+0x20],0x0
  mov   DWORD PTR [rax+0x4],ebx
  dec   ebx
  mov   DWORD PTR [rax],ebx
  jmp   <use_mem(int)+0x46>
  mov   edi,0xa
  mov   r14,rax
  call  <sleep@plt>
  mov   rax,r14
  mov   DWORD PTR [r14],ebx
  mov   DWORD PTR [r14+0x4],0x0
  add   rsp,0x8
  pop   rbx
  pop   r14
```

```
<use_mem(int) [clone .memprof.1]>:
  push  r14
  push  rbx
  push  rax
  mov   ebx,edi
  call  <get_mem(int) [clone .memprof.1]>
  cmp   ebx,0x4
  jl    <use_mem(int) [clone .memprof.1]+0x2b>
  xorps xmm0,xmm0
  movups XMMWORD PTR [rax],xmm0
  movups XMMWORD PTR [rax+0x10],xmm0
  mov   QWORD PTR [rax+0x20],0x0
  mov   DWORD PTR [rax+0x4],ebx
  dec   ebx
  mov   DWORD PTR [rax],ebx
  jmp   <use_mem(int) [clone .memprof.1]+0x46>
  mov   edi,0xa
  mov   r14,rax
  call  <sleep@plt>
  mov   rax,r14
  mov   DWORD PTR [r14],ebx
  mov   DWORD PTR [r14+0x4],0x0
  add   rsp,0x8
  pop   rbx
  pop   r14
  ret
```

[MemProf] PGO

Посмотрим результат - будто бы оптимизация пропала. Но нет. Оно клонировало теперь и use_mem!

```
<use_mem(int)>:
    push    r14
    push    rbx
    push    rax
    mov     ebx,edi
    call    <get_mem(int)>
    cmp     ebx,0x4
    jl     <use_mem(int)+0x2b>
    xorps   xmm0,xmm0
    movups  XMMWORD PTR [rax],xmm0
    movups  XMMWORD PTR [rax+0x10],xmm0
    mov     QWORD PTR [rax+0x20],0x0
    mov     DWORD PTR [rax+0x4],ebx
    dec     ebx
    mov     DWORD PTR [rax],ebx
    jmp     <use_mem(int)+0x46>
    mov     edi,0xa
    mov     r14,rax
    call    <sleep@plt>
    mov     rax,r14
    mov     DWORD PTR [r14],ebx
    mov     DWORD PTR [r14+0x4],0x0
    add     rsp,0x8
    pop     rbx
    pop     r14
```

```
<use_mem(int) [clone .memprof.1]>:
    push    r14
    push    rbx
    push    rax
    mov     ebx,edi
    call    <get_mem(int) [clone .memprof.1]>
    cmp     ebx,0x4
    jl     <use_mem(int) [clone .memprof.1]+0x2b>
    xorps   xmm0,xmm0
    movups  XMMWORD PTR [rax],xmm0
    movups  XMMWORD PTR [rax+0x10],xmm0
    mov     QWORD PTR [rax+0x20],0x0
    mov     DWORD PTR [rax+0x4],ebx
    dec     ebx
    mov     DWORD PTR [rax],ebx
    jmp     <use_mem(int) [clone .memprof.1]+0x46>
    mov     edi,0xa
    mov     r14,rax
    call    <sleep@plt>
    mov     rax,r14
    mov     DWORD PTR [r14],ebx
    mov     DWORD PTR [r14+0x4],0x0
    add     rsp,0x8
    pop     rbx
    pop     r14
    ret
```

[MemProf] PGO

Компилятор не имеет права добавлять свои вызовы. Может только менять существующие (?).

```
<use_mem(int)>:
    push    r14
    push    rbx
    push    rax
    mov     ebx,edi
    call    <get_mem(int)>
    cmp     ebx,0x4
    jl     <use_mem(int)+0x2b>
    xorps   xmm0,xmm0
    movups  XMMWORD PTR [rax],xmm0
    movups  XMMWORD PTR [rax+0x10],xmm0
    mov     QWORD PTR [rax+0x20],0x0
    mov     DWORD PTR [rax+0x4],ebx
    dec     ebx
    mov     DWORD PTR [rax],ebx
    jmp     <use_mem(int)+0x46>
    mov     edi,0xa
    mov     r14,rax
    call    <sleep@plt>
    mov     rax,r14
    mov     DWORD PTR [r14],ebx
    mov     DWORD PTR [r14+0x4],0x0
    add     rsp,0x8
    pop     rbx
    pop     r14
```

```
<use_mem(int) [clone .memprof.1]>:
    push    r14
    push    rbx
    push    rax
    mov     ebx,edi
    call    <get_mem(int) [clone .memprof.1]>
    cmp     ebx,0x4
    jl     <use_mem(int) [clone .memprof.1]+0x2b>
    xorps   xmm0,xmm0
    movups  XMMWORD PTR [rax],xmm0
    movups  XMMWORD PTR [rax+0x10],xmm0
    mov     QWORD PTR [rax+0x20],0x0
    mov     DWORD PTR [rax+0x4],ebx
    dec     ebx
    mov     DWORD PTR [rax],ebx
    jmp     <use_mem(int) [clone .memprof.1]+0x46>
    mov     edi,0xa
    mov     r14,rax
    call    <sleep@plt>
    mov     rax,r14
    mov     DWORD PTR [r14],ebx
    mov     DWORD PTR [r14+0x4],0x0
    add     rsp,0x8
    pop     rbx
    pop     r14
    ret
```

[MemProf] PGO

Компилятор не имеет права добавлять свои вызовы. Может только менять существующие (?).

Итого: для разделения неоднозначности холодная-горячая аллокация компилятор клонирует функции, а иногда и всю цепочку вызовов.

[MemProf] PGO

Компилятор не имеет права добавлять свои вызовы. Может только менять существующие (?).

Итого: для разделения неоднозначности холодная-горячая аллокация компилятор клонирует функции, а иногда и всю цепочку вызовов.

Это работает внутри одного модуля (единицы компиляции)

[MemProf] PGO

Компилятор не имеет права добавлять свои вызовы. Может только менять существующие (?).

Итого: для разделения неоднозначности холодная-горячая аллокация компилятор клонирует функции, а иногда и всю цепочку вызовов.

Это работает внутри одного модуля (единицы компиляции)

Между модулями тоже работает если включено LTO или ThinLTO

[MemProf] PGO

Между модулями тоже работает если включено LTO или ThinLTO.

А если вы ещё не пользуетесь ThinLTO, то посмотрите вот этот доклад от Teresa Johnson (она, кстати, один из авторов [MemProf]):

Teresa Johnson - ThinLTO Whole Program Optimization - Meeting C++ 2020 Center Keynote



The slide, titled "ThinLTO: Incremental Builds", illustrates the workflow of incremental linking. At the top, a row of blue boxes labeled ".o" represents object files, with an arrow pointing to them from the text "Summaries include a hash of each LLVM Module".

Phase 2: Thin Link is shown as a large blue box labeled "Inter-procedural Analysis". Below it, a red box labeled "Analyses Results" is connected to a "Linker" box. This phase is part of a "Distributed build system".

The diagram shows a sequence of operations for each module: `thinlto.bc` (blue box) and `Import list` (green box) are processed. The results of the analysis are then used to generate `Opt` (orange box) and `BE` (red box) files. The `Opt` files are then processed by `BE` blocks, which output `.o` files (blue boxes).

Phase 3: ThinLTO Backends is shown as a large blue box containing the `Opt` and `BE` blocks. The final `.o` files are then processed by a "Traditional Linking" block (orange box) at the bottom.

A video inset on the left shows Teresa Johnson speaking.

[MemProf] PGO

Итого, как пользоваться для PGO:

```
$ clang++ -g -fmemory-profile main.cpp
```

```
$ ./a.out
```

```
<generates memprof.profrw.{PID} file
```

```
$ llvm-profdata merge memprof.profrw.{PID} --profiled-binary a.out -o t.memprofdata
```

```
<generates t.memprofdata>
```

```
$ clang++ -ltcmalloc -g -fmemory-profile-use=t.memprofdata -O1 -Wl,-mllvm,-enable-memprof-context-disambiguation  
-Wl,-mllvm,-supports-hot-cold-new -Wl,-mllvm,-optimize-hot-cold-new -mllvm --memprof-ave-lifetime-cold-threshold=5 -fno-inline main.cpp
```

```
$ ./a.out
```

[MemProf] PGO

Итого, как пользоваться для PGO:

```
$ clang++ -g -fmemory-profile main.cpp
$ ./a.out
<generates memprof.profraw.{PID} file

$ llvm-profdata merge memprof.profraw.{PID} --profiled-binary a.out -o t.memprofdata
<generates t.memprofdata>

$ clang++ -ltcmalloc -g -fmemory-profile-use=t.memprofdata -O1 -Wl,-mllvm,-enable-memprof-context-disambiguation
-Wl,-mllvm,-supports-hot-cold-new -Wl,-mllvm,-optimize-hot-cold-new -mllvm --memprof-ave-lifetime-cold-threshold=5 -fno-inline main.cpp
$ ./a.out
```

Как пользоваться для просмотра глазами:

```
$ clang++ -g -fmemory-profile -mllvm -memprof-histogram struct.cpp

$ MEMPROF_OPTIONS=print_text=1 ./a.out
<generates memprof.profraw.{PID} file in text format>

$ less memprof.profraw.{PID}
```

Спасибо!

Итого, как пользоваться для PGO:

```
$ clang++ -g -fmemory-profile main.cpp
$ ./a.out
<generates memprof.profraw.{PID} file

$ llvm-profdata merge memprof.profraw.{PID} --profiled-binary a.out -o t.memprofdata
<generates t.memprofdata>

$ clang++ -ltcmalloc -g -fmemory-profile-use=t.memprofdata -O1 -Wl,-mllvm,-enable-memprof-context-disambiguation
-Wl,-mllvm,-supports-hot-cold-new -Wl,-mllvm,-optimize-hot-cold-new -mllvm --memprof-ave-lifetime-cold-threshold=5 -fno-inline main.cpp
$ ./a.out
```

Как пользоваться для просмотра глазами:

```
$ clang++ -g -fmemory-profile -mllvm -memprof-histogram struct.cpp

$ MEMPROF_OPTIONS=print_text=1 ./a.out
<generates memprof.profraw.{PID} file in text format>

$ less memprof.profraw.{PID}
```