Thread Local в C++

Как он есть

Евгений Ерохин

DINS

Кто этот маЛчик, и что он нам тут затирает ;-)

Старший разработчик в DINS

- Разработка сервиса передачи видео в реальном времени
- Семейство протоколов RTP (Real-time Transport Protocol)
- Реализация протоколов по RFC

Кто этот маЛчик, и что он нам тут затирает ;-)

Опыт за плечами, Paragon Software 10+ лет ведущий разработчик направления, тимлид.

- Драйвера файловых систем под macOS
- Имплементация файловых систем
- Система снепшотинга блочных (дисковых) устройств
- Бут загрузчик
- Реверс инжиниринг
- И еще оч много всего системного =)

Thread Local

??? !!!

Что возьмем за основу?

Linux

- На сегодняшний день имеет наиболее обширную поддержку thread_local.
- Наибольшее разнообразие видов имплементаций

Немного об эволюции

Становление **thread_local**

- **POSIX** pthread_getspecific / pthread_setspecific
- **__thread** не стандартное расширение **GCC**
- **thread_local** стандартизовано с **C++11**

Thread Local

Так кто же ты такой?

Snippet 1 https://godbolt.org/z/hm82DC

Как все начиналось

Поддержка в **POSIX**

- Создать ключ **pthread_key_create**
- Сохранить ключ в глобальную переменную
- Задаем значение **pthread_setspecific**
- Получаем значение **pthread_getspecific**
- Не забыть удалить ключ **pthread_key_delete**
- И да! Значение может быть только **void***

Как избавиться от этих сложностей?

Нужна поддержка со стороны компилятора

- Нестандартное расширение **__thread**
- Стандартизация в C++11 **thread_local**

Где можно использовать новые Thread Local

Global scope

```
thread_local unsigned t_somevar;
```

Structure / class

```
struct A {
    static thread_local unsigned t_somevar;
};
```

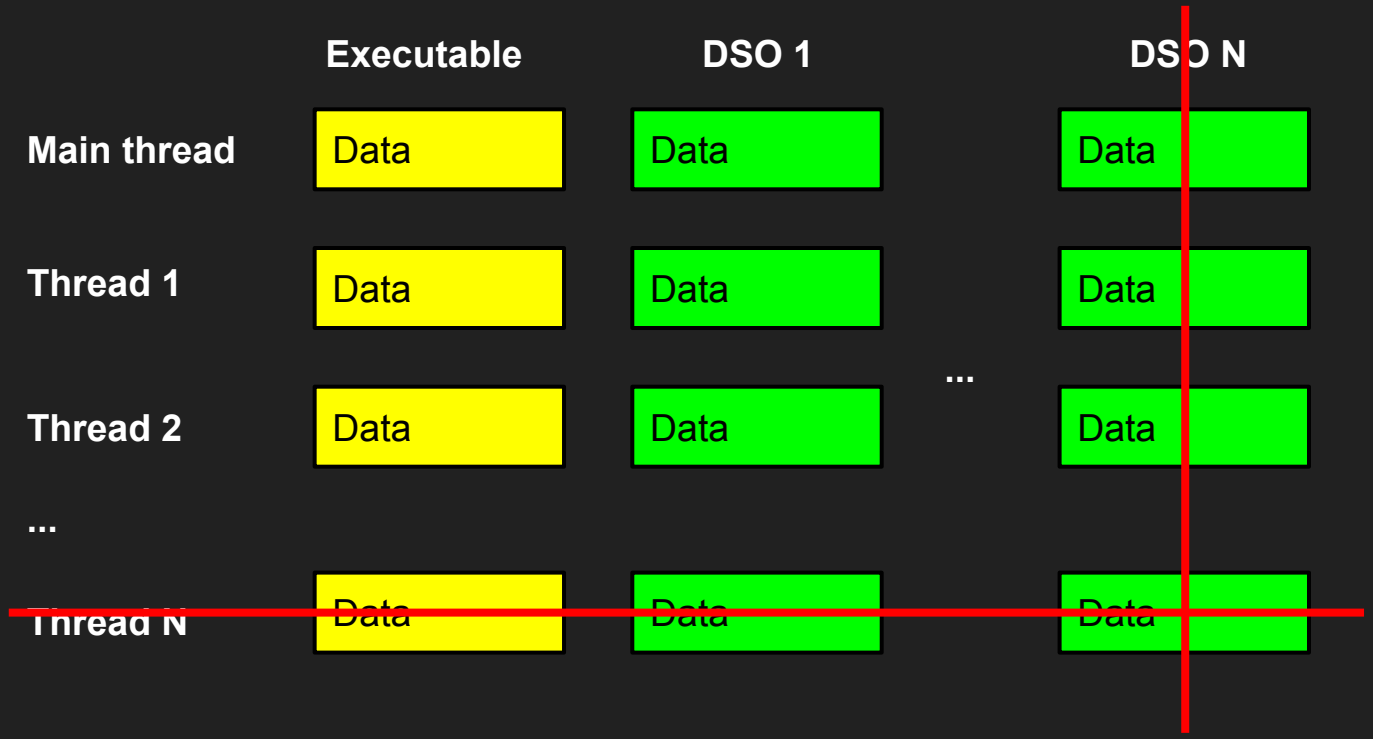
Переменная в функции

```
unsigned fn() {
    thread_local unsigned t_somevar = 0;
    t_somevar++;
    return t_somevar;
}
```

А еще :-D

- Начиная с C++17 делать inline
- И в GNU совместимых компиляторах делать с атрибутом weak

Что мы должны учесть



Начало времени жизни TL переменной

Началом времени жизни TL переменной является одно из следующих условий (без учета оптимизаций)

- Создание нового треда
- Загрузка нового DSO содержащего TL

Конец времени жизни TL переменной

Концом времени жизни TL переменной являются

- Завершение треда
- Выгрузка DSO содержащего TL

Поддержка в ELF

Новые секции

- **.tdata**
- **.tbss**

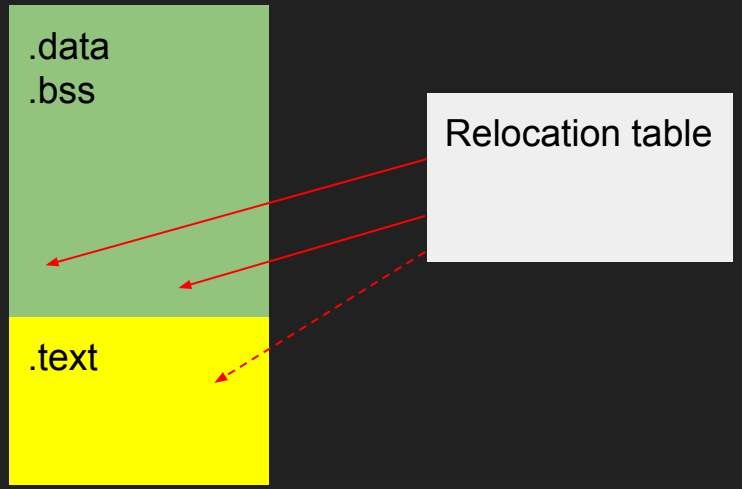
Поддержка в ELF

Новые релокации

- **R_X86_64_DTPMOD64**
- **R_X86_64_DTPOFF64**

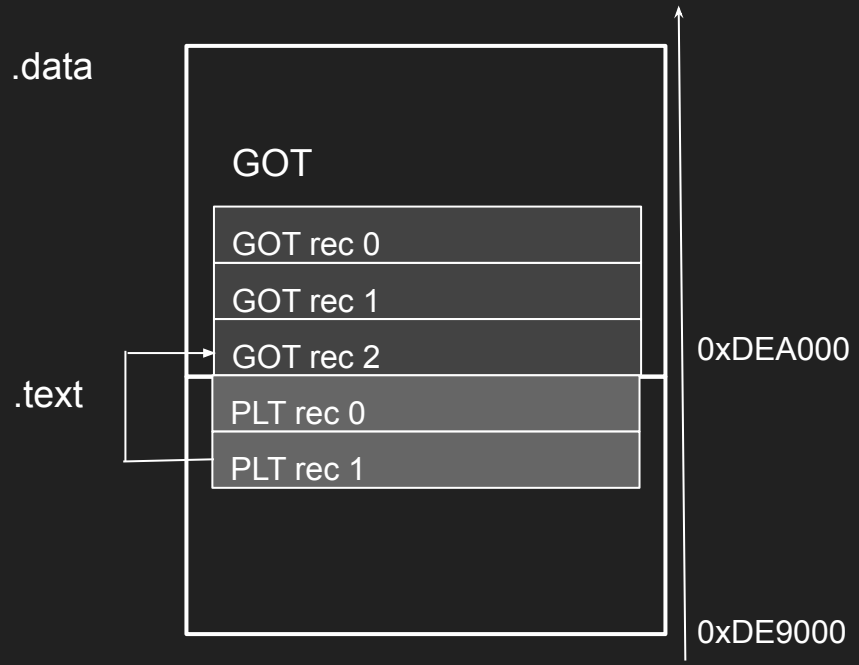
Или объединенные в одну запись

- **R_X86_64_TLSLD**



Global Offset Table (GOT)

Организация **Position Independent Code (PIC)**



Глоссарий

PIC - Position Independent Code
GOT - Global Offset Table
PLT - Procedure Linkage Table

Thread Local Storage (TLS)

Блоки в памяти, в которых хранятся Тред-специфические данные.

- **Static** - создается при старте программы и тредов
- **Dynamic** - создается при загрузке DSO при помощи **dlopen**

Алгоритм формирования Static TLS

Статический TLS формируется динамическим линковщиком на запуске программы

1. Добавляем TLS исполняемого файла (секции .tbss .tdata)
2. Загружаем прилинкованные DSO
3. Добавляем TLS от загруженных DSO (секции .tbss .tdata)
4. Выделяем один блок на все участвующие модули
5. Некоторые! Рантаймы могут добавлять запасной кусок (surplus)

TCB & TLS

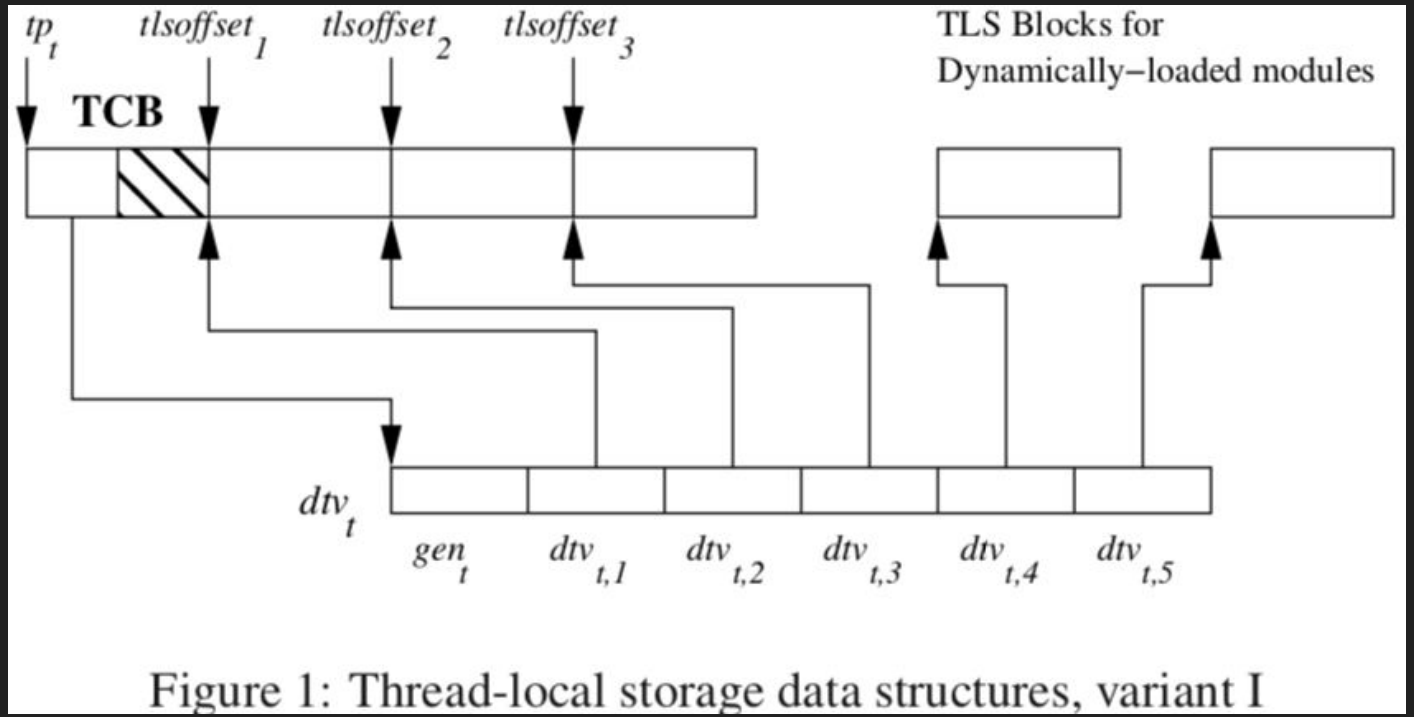


Figure 1: Thread-local storage data structures, variant I

Ulrich Drepper: https://www.uclibc.org/docs/tls.pdf

Два “Традиционных” варианта TLS

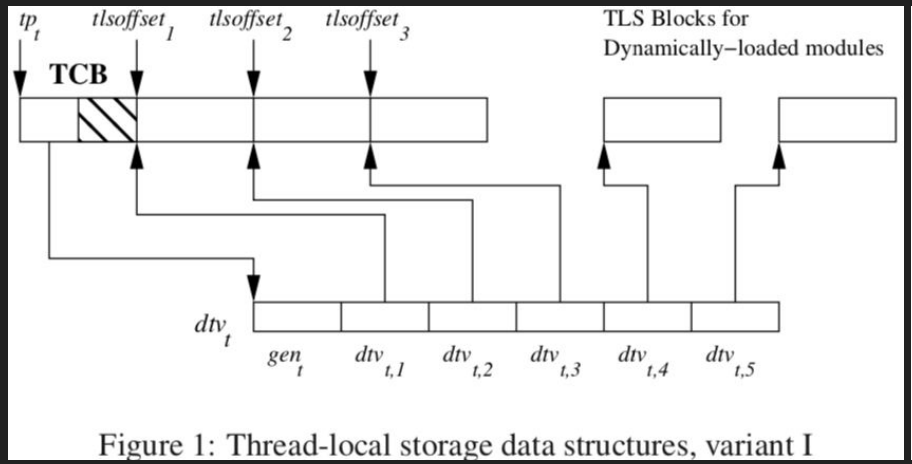


Figure 1: Thread-local storage data structures, variant I

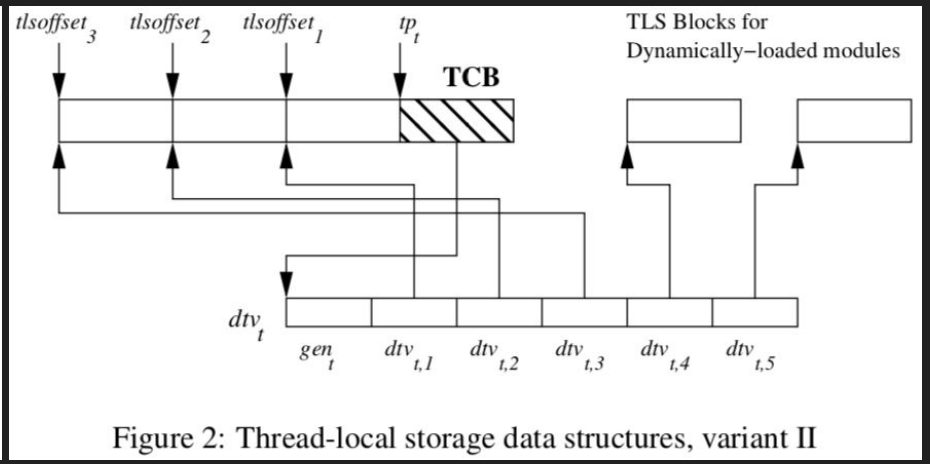


Figure 2: Thread-local storage data structures, variant II

Ulrich Drepper: https://www.uclibc.org/docs/tls.pdf

TLS static & dynamic вариант 2

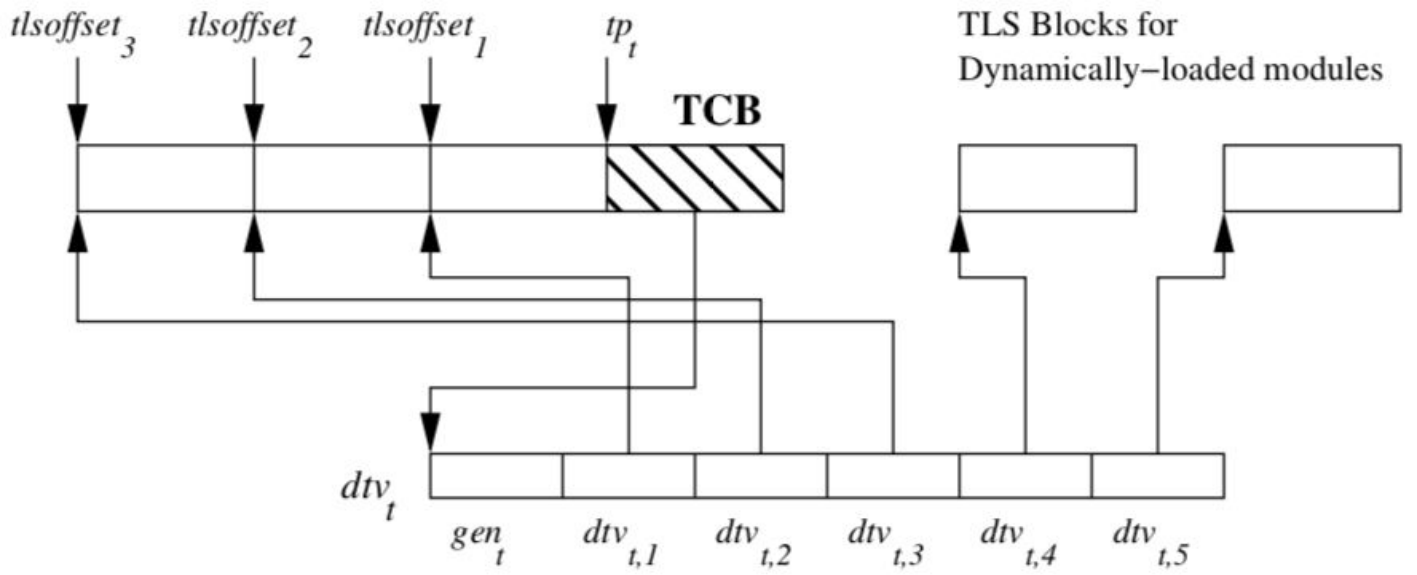


Figure 2: Thread-local storage data structures, variant II

TLS

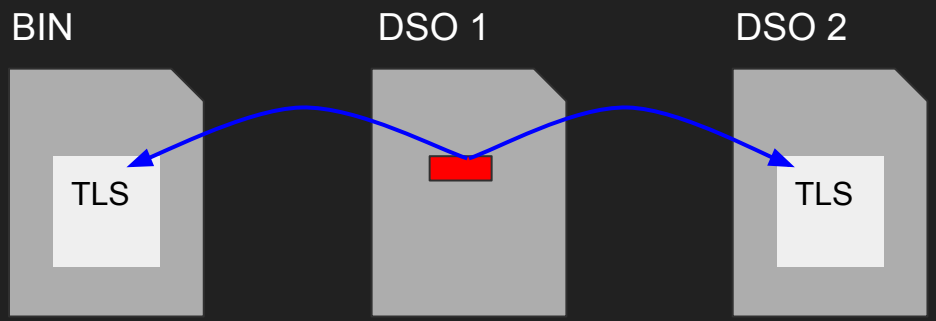
Модели доступа

- **General Dynamic**
- **Local Dynamic**
- **Initial Exec**
- **Local Exec**

General Dynamic

Наиболее общая модель и самая тяжеловесная.

- Тред локал переменные линкуются снаружи (extern)
- Мы не знаем находятся ли они в Статическом или Динамическом TLS
- Каждая из тред локал переменных может быть из другого DSO



__tls_get_addr

```
33  void * __tls_get_addr (GET_ADDR_ARGS)
34  {
35    dtv_t *dtv = THREAD_DTV ();
36
37    if (__glibc_unlikely (dtv[0].counter != GL(dl_tls_generation)))
38      return update_get_addr (GET_ADDR_PARAM);
39
40    void *p = dtv[GET_ADDR_MODULE].pointer.val;
41
42    if (__glibc_unlikely (p == TLS_DTV_UNALLOCATED))
43      return tls_get_addr_tail (GET_ADDR_PARAM, dtv, NULL);
44
45    return (char *) p + GET_ADDR_OFFSET;
46  }
```

Релокации

- R_X86_64_DTPMOD64
- R_X86_64_DTPOFF64
- R_X86_64_TLSLD

TLS static & dynamic вариант 2

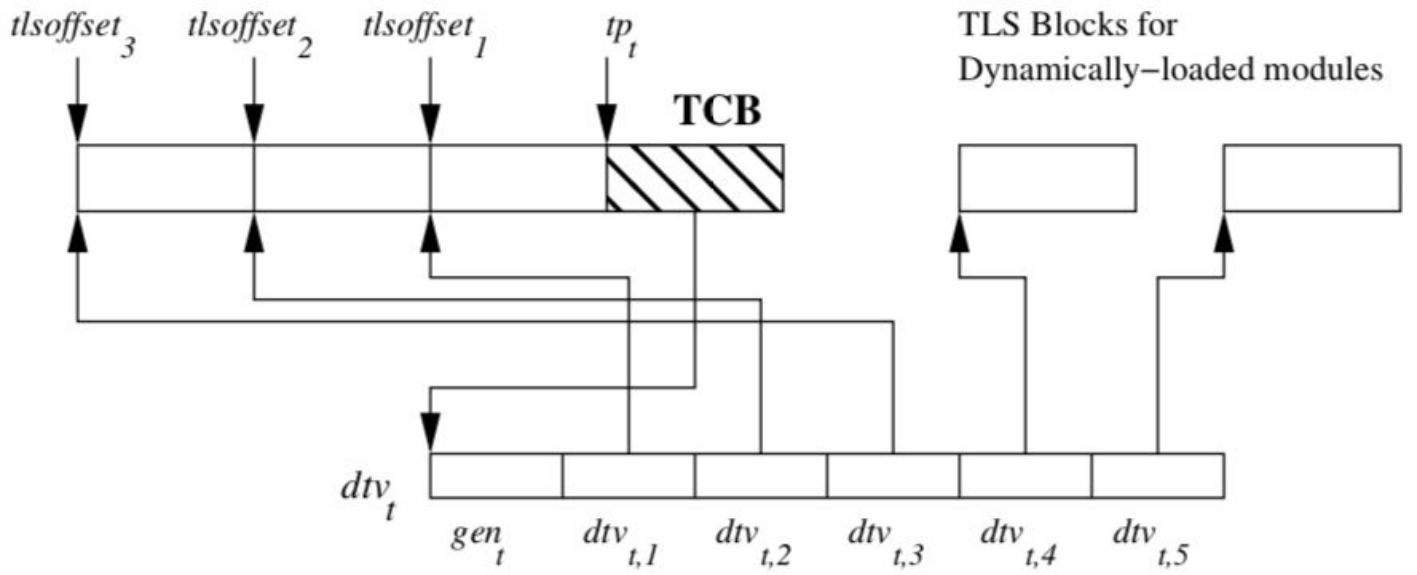


Figure 2: Thread-local storage data structures, variant II

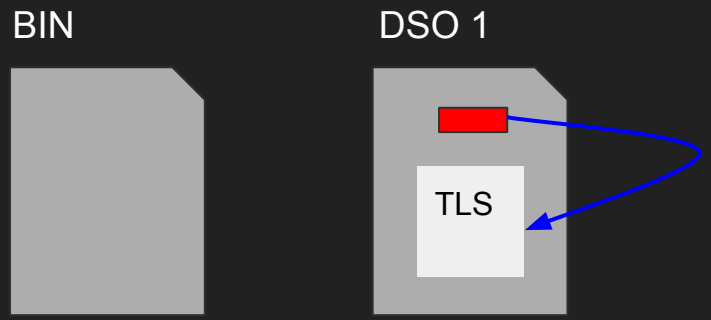
Пример GD

Пример: example_gd
https://godbolt.org/z/QfvmAV

Local Dynamic

Переменные находятся в DSO, мы знаем что они для нас локальны

- Переменные могут находиться как в Статическом так и в Динамическом TLS
- Но мы знаем что они находятся в одном сторадже
- Достаточно получить адрес стораджа для адресации всех



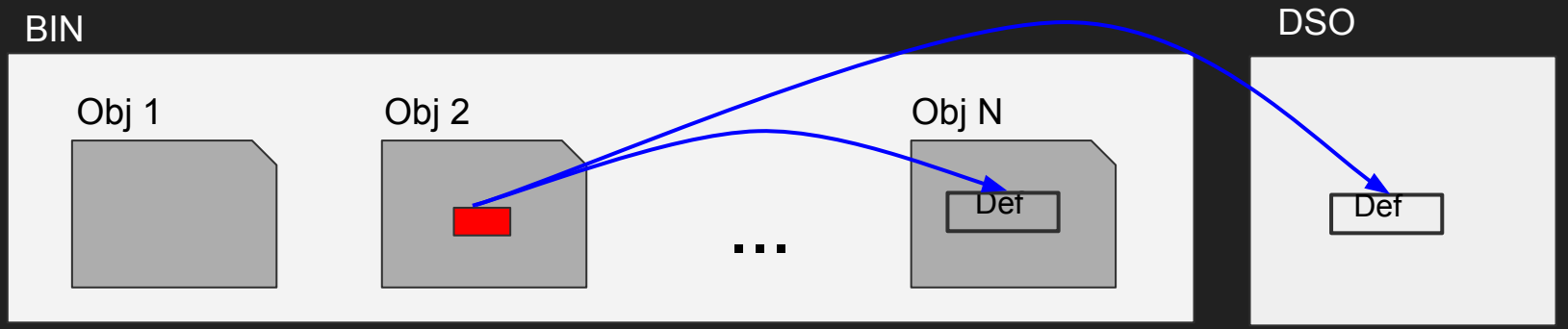
Пример LD

Пример: example_ld
https://godbolt.org/z/g3y8uW

Initial Exec

Переменные находятся в одном из модулей основного исполняемого файла

- Находятся в Статическом TLS (даже если из DSO)
- Получаем адрес из GOT
- Простой индирект доступ



Пример IE

Пример: example_ie
https://godbolt.org/z/BjW68P

Local Exec

Переменные находятся локально в основном исполняемом файле

- Смещение заранее известно относительно регистра **fs**
- Самый дешевый доступ из всех моделей

Пример LE

Пример: example_le
https://godbolt.org/z/YkQUrK

Маленький трюк оптимизации

Если мы уверены что наш DSO не будет загружаться через dlopen.

- На сборке DSO можно задать -ftls-model=initial-exec
- Только в статическом TLS

Инициализация: что нам говорит стандарт!

Dynamic initialization of a block-scope variable with static storage duration or thread storage duration is performed the first time control passes through its declaration; such a variable is considered initialized upon the completion of its initialization. If the initialization exits by throwing an exception, the initialization is not complete, so it will be tried again the next time control enters the declaration. If control enters the declaration concurrently while the variable is being initialized, the concurrent execution shall wait for completion of the initialization.[86] If control re-enters the declaration recursively while the variable is being initialized, the behavior is undefined. [*Example:*

Деструкция: что нам говорит стандарт!

Destructors ([class.dtor]) for initialized objects (that is, objects whose lifetime ([basic.life]) has begun) with static storage duration are called as a result of returning from `main` and as a result of calling `std::exit` ([support.start.term]). Destructors for initialized objects with thread storage duration within a given thread are called as a result of returning from the initial function of that thread and as a result of that thread calling `std::exit`. The completions of the destructors for all initialized objects with thread storage duration within that thread are sequenced before the initiation of the destructors of any object with static storage duration. If the completion of the constructor or dynamic initialization of an object with thread storage duration is sequenced before that of another, the completion of the destructor of the second is sequenced before the initiation of the destructor of the first. If the completion of the constructor or dynamic initialization of an object with static storage duration is sequenced before that of another, the completion of the destructor of the second is sequenced before the initiation of the destructor of the first. [*Note:* This definition permits concurrent destruction. — *end note*] If an object is initialized statically, the object is destroyed in the same

Ну что? Теперь то нам все понятно?

Snippet 1

https://godbolt.org/z/Yh_oZ_

Врапперы

Для переменной **t_var**

_ZTH5t_var - “TLS init function for t_var” **(weak symbol)**

_ZTW5t_var - “TLS wrapper function for t_var“

Пример thread_local объекта

Snippet 2 Class
https://godbolt.org/z/hhaYmq

Суммируем по объектам

- Инициализируется по первому обращению
- Порождает дополнительную неявную Guard переменную **(thread_local)**
- Неявно регистрирует вызова деструктора ”__cxa_thread_atexit”

И еще одно правило

Не смешиваем деструкцию статических объектов с использованием инициализируемых thread_local

А что дальше?

Дескрипторная модель

- В GOT можно хранить указатель на функцию резолвер
- Линкер может выбирать наиболее оптимальный вариант резолвера

http://www.fsfla.org/~lxoliva/writeups/TLS/RFC-TLSDESC-x86.txt
https://android.googlesource.com/platform/bionic/+/HEAD/docs/elf-tls.md

TLS Descriptors (TLSDESC)

```
struct TlsDescriptor { // NB: arm32 reverses these fields
  long (*resolver)(long);
  long arg;
};

char* get_tls_var() {
  // allocated in the .got, uses a dynamic relocation
  static TlsDescriptor desc = R_TLS_DESC(tls_var);
  return (char*)__get_tls() + desc.resolver(desc.arg);
}
```

```
long static_tls_resolver(long arg) {
  return arg;
}
```

Как попробовать

Поддержка в компиляторах через флаг -mtls-dialect=<dialect> (trad-vs-desc on arm64, otherwise gnu-vs-gnu2)

- Поддержка в GCC
- Clang пока не поддерживает (кроме ARM64)

https://godbolt.org/z/-jZR4r

Benchmarks

Производительность!!!???

benchmark

Подведем итоги

Так что у нас в обойме!?

- **POSIX** pthread_* широко поддерживается, но устарел!??
- Новый thread_local эффективный механизм
- НО иногда совсем не бесплатен
- И конечно это еще развивающийся инструмент

А где же место для Thread Local?

Полезные ссылки

- Ulrich Drepper: https://www.uclibc.org/docs/tls.pdf
- Alexandre Oliva http://www.fsfla.org/~lxoliva/writeups/TLS/RFC-TLSDESC-x86.txt
- Android https://android.googlesource.com/platform/bionic/+/HEAD/docs/elf-tls.md
- Еще одно исследование https://chao-tic.github.io/blog/2018/12/25/tls#fnref:main1

Спасибо за внимание!

Евгений Ерохин

mailto: the_gabber@mail.ru

telegram: @the_gabber

GitHub: https://github.com/the-gabber/tls_speech.git