Return of the Segment: Thread Local Storage

John Baldwin

BSDCan

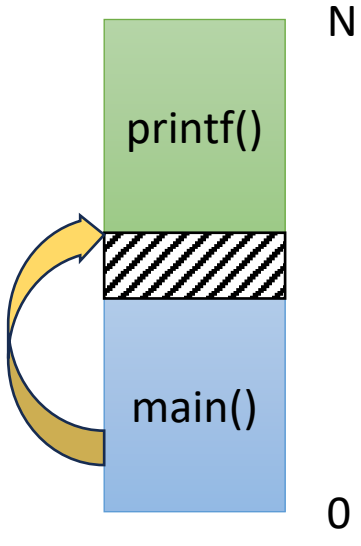
20 June 2026

The Problem: Mapping Symbols to Addresses

- Source code refers to functions and data objects by symbol names
- Machine code refers to functions and data by addresses

```
#include <stdio.h>

int main(void)
{
    printf("hello world\n");
    return (0);
}
```



Thread Local Storage (TLS)

- Previous talks have focused on symbols that (mostly) resolve to a single address such as global variables
- The introduction of threads to UNIX required a new type of symbol with per-thread storage
  - `errno` must be stored per-thread to permit concurrent system calls within a single process
- Threaded programs may also require per-thread storage for other variables
  - The storage is allocated when a thread is created and released after the thread terminates

But Let's Talk About `errno`

- Before the introduction of threads, `errno` was just a simple global variable in <errno.h>

```
extern int errno;
```

But Let's Talk About `errno`

- Before the introduction of threads, `errno` was just a simple global variable in <errno.h>
- In commit f70177e76e60 in January of 1996 a bespoke per-thread variant was added

```
#ifdef      _THREAD_SAFE
extern      int * __error();
#define     errno (* __error())
#else
extern int errno;
#endif
```

But Let's Talk About `errno`

- Before the introduction of threads, `errno` was just a simple global variable in <errno.h>
- In commit f70177e76e60 in January of 1996 a bespoke per-thread variant was added
- Two years later, commit 1b46cb523df3 switched to the bespoke variant always

```
__BEGIN_DECLS
int * __error();
__END_DECLS
#define    errno (* __error())
```

How does `errno` work now?

- Each reference to `errno` calls the `__error()` function and dereferences the pointer it returns

```
__BEGIN_DECLS
int * __error();
__END_DECLS
#define    errno (* __error())
```

How does `errno` work now?

- Each reference to `errno` calls the `__error()` function and dereferences the pointer it returns
- `__error()` returns a pointer to per-thread storage

```
__BEGIN_DECLS
int * __error();
__END_DECLS
#define    errno (* __error())
```

How does `__error()` work?

```
int __libsys_errno;

static int *
__error_unthreaded(void)
{
        return (&__libsys_errno);
}

static int *(*__error_selector)(void) = __error_unthreaded;

int *
__error(void)
{
        return (__error_selector());
}
```

This is basically the old errno variable

__error_unthreaded returns a pointer to a global variable

Function pointer initially points to __error_unthreaded

_error invokes the __error_selector function pointer

What about threads?

```
extern int __libsys_errno;

__weak_reference(__error_threaded, __erro
int *
__error_threaded(void)
{
        struct pthread *curthread;

        if (_thr_initial != NULL) {
                curthread = _get_curthread();
                if (curthread != NULL && curthread != _thr_initial)
                        return (&curthread->error);
        }
        return (&__libsys_errno);
}
```

The same storage from libsys

Installed as the _error_selector function pointer

If no other threads yet, fall back to the global

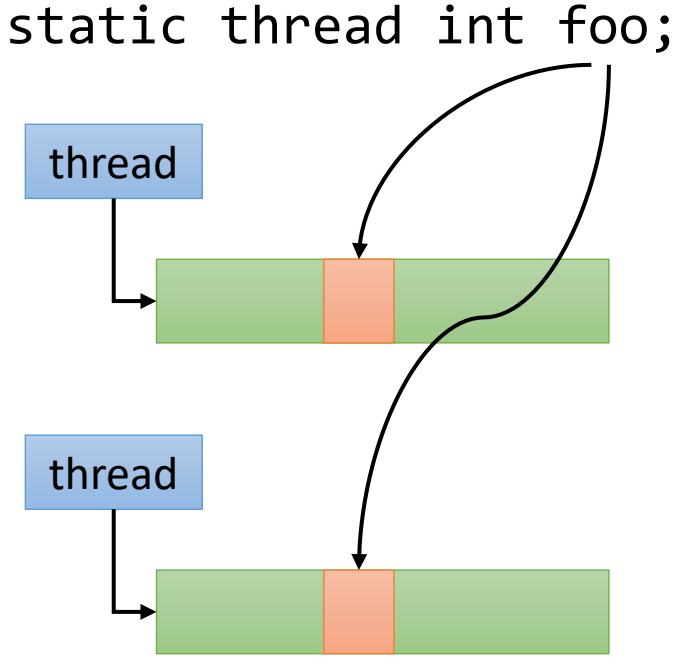
Main thread still uses the global

errno

- So every access to errno calls a function which returns a pointer that then has to be dereferenced to access the symbol
- … And the compiler has to repeat this every time you use errno
- Writing this much code each time you need a new per-thread variable does not scale
  - And where do you store it, the thread structure is private to the implementation
- POSIX threads includes an API (pthread_key) for per-thread storage, but it is a bit clumsy to use

Ideal Model

- A nicer solution than the `errno` hack or POSIX thread keys would be native thread-local symbols
- Each thread would be associated with a block of storage
- Thread-local symbols would be mapped to a relative offset inside of the block for the currently executing thread

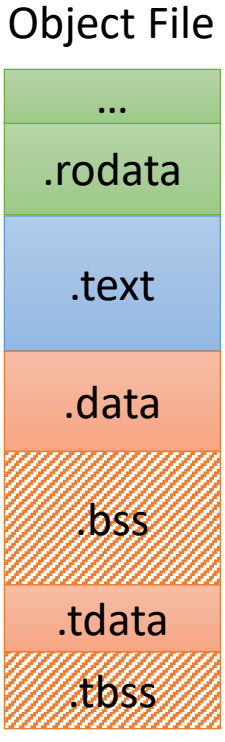


Thread Local Storage in Modern C

- C11 and C++11 permit using the `thread_local` storage class to declare variables with per-thread storage
  - GNU C supported this via `__thread` in older language standards
- The toolchain and runtime must cooperate to implement this
- ELF grew various extensions to describe TLS symbols including new relocation types, sections, and segments
- Runtime loader has to allocate per-thread storage for TLS symbols and map TLS symbol accesses to per-thread addresses
  - Requires explicit cooperation with libc and the threading library (e.g. libpthread)

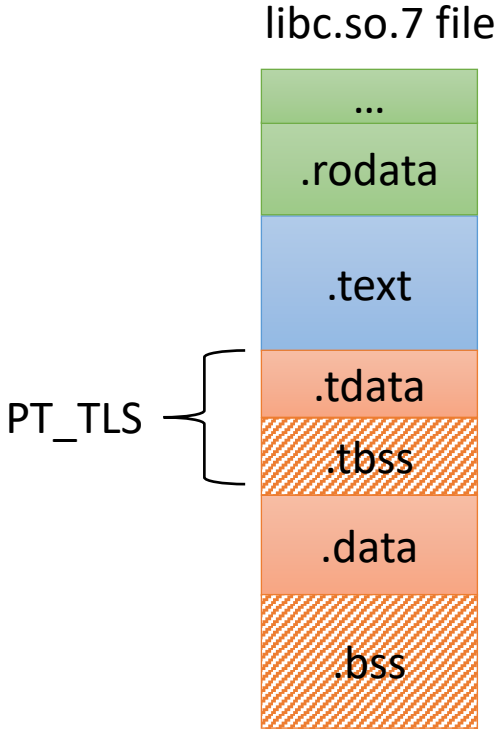
ELF Sections

- New ELF sections describe TLS variables in object files and linked output files
- .tdata contains the initial values of TLS variables with non-zero values (similar to .data)
- .tbss contains the size of storage needed for zero-initialized TLS variables (similar to .bss)
- No equivalents for .text or .rodata



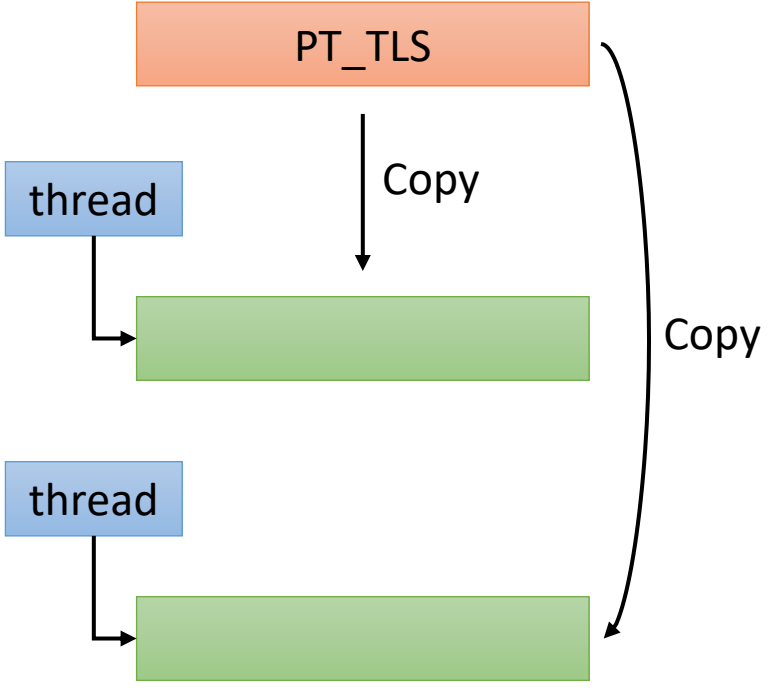
ELF Segment

- Content of .tdata section mapped into memory by PT_LOAD similar to .data
- New ELF segment PT_TLS describes a region of address space similar to PT_GNU_RELRO
- The memory range covers the initial TLS block (a template), including both .tdata and .tbss
- Each binary or shared library can have its own TLS block



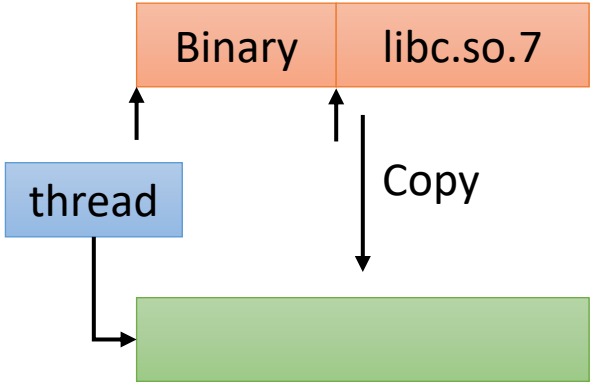
Mapping TLS Blocks to Per-Thread Storage

- For a static binary, we can allocate a single block of storage for each thread sized by the PT_TLS of the binary
- Per-thread storage initialized by cloning PT_TLS-mapped block
- The C startup code in libc allocates this storage for the initial thread prior to main()
- Thread creation allocates new storage for each thread



Mapping TLS Blocks to Per-Thread Storage

- For dynamic binaries, there can be multiple PT_TLS blocks, so the per-thread storage has to be sized to the sum
- The runtime loader has to manage the layout of the storage by assigning starting offsets to each PT_TLS block
- Thread creation is the same, just copying from a larger template



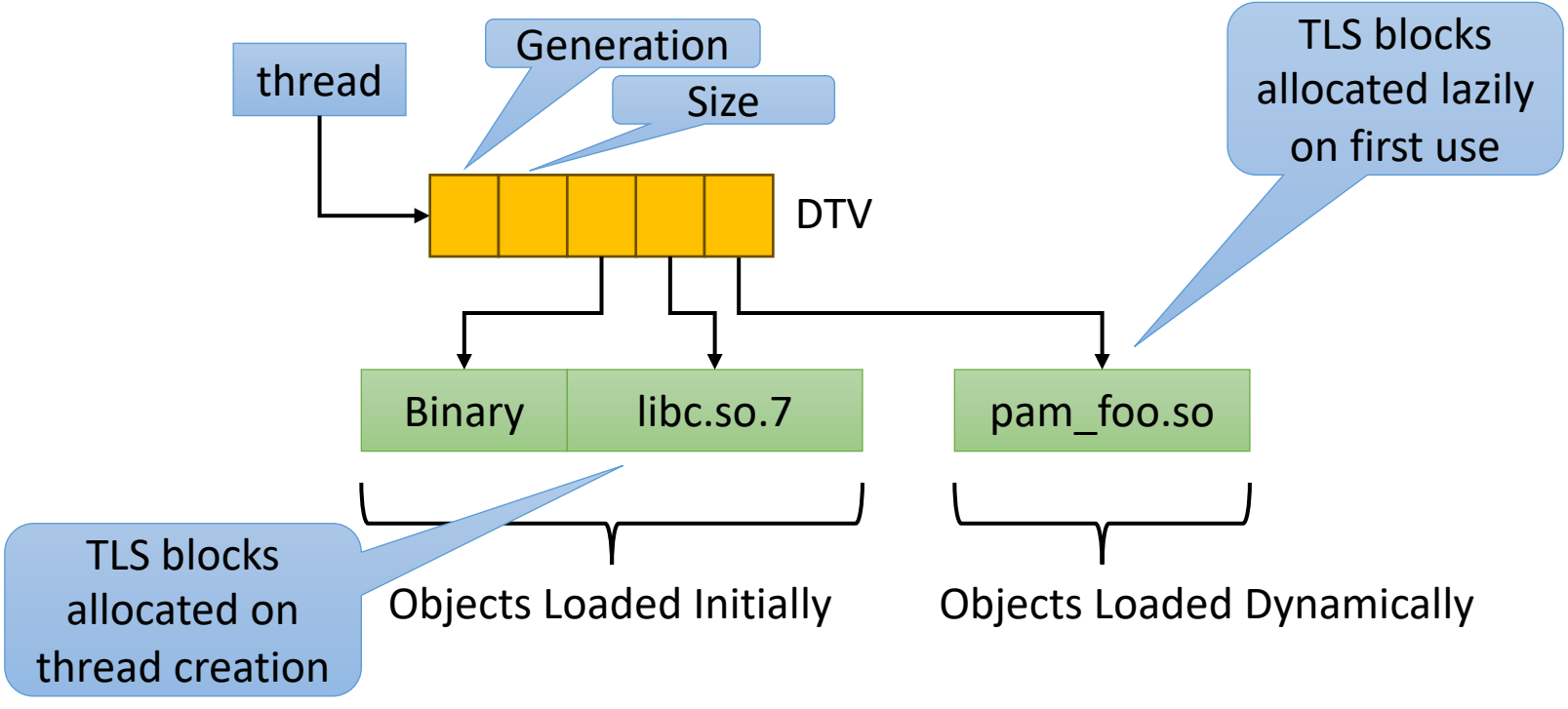
But What About `dlopen()`?

- This is when things get a bit sadder and the complexity goes up
- A DSO opened at runtime via `dlopen()` might contain a PT_TLS segment
- This means that the per-thread storage might need to grow for each thread
  - The runtime loader could pre-allocate some extra space for each thread, but what if you exhaust it?
- Other threads need to still execute concurrently with the `dlopen()`
- Pointers to existing thread local variables need to stay stable
  - For example, a thread might be doing a `memcpy()` to/from a TLS variable concurrent with the `dlopen()`

What's a Little More Indirection Between Friends?

- The solution to the `dlopen()` problem is more indirection
- Each thread allocates an array of pointers to TLS blocks
  - **Dynamic Thread Vector** (DTV)
- Each mapped object (binary, shared library) which defines TLS variables is assigned a unique index in these parallel arrays
- The array entry points to the per-thread TLS block storage for that mapped object
- During `dlopen()`, the runtime loader is able to grow the arrays to append entries for any new TLS blocks

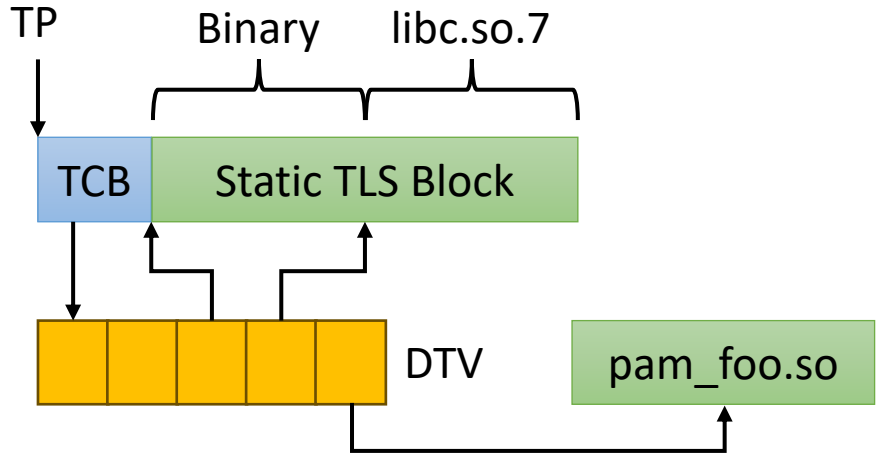
Thread Storage with DTV



How Are These Data Structures Arranged?

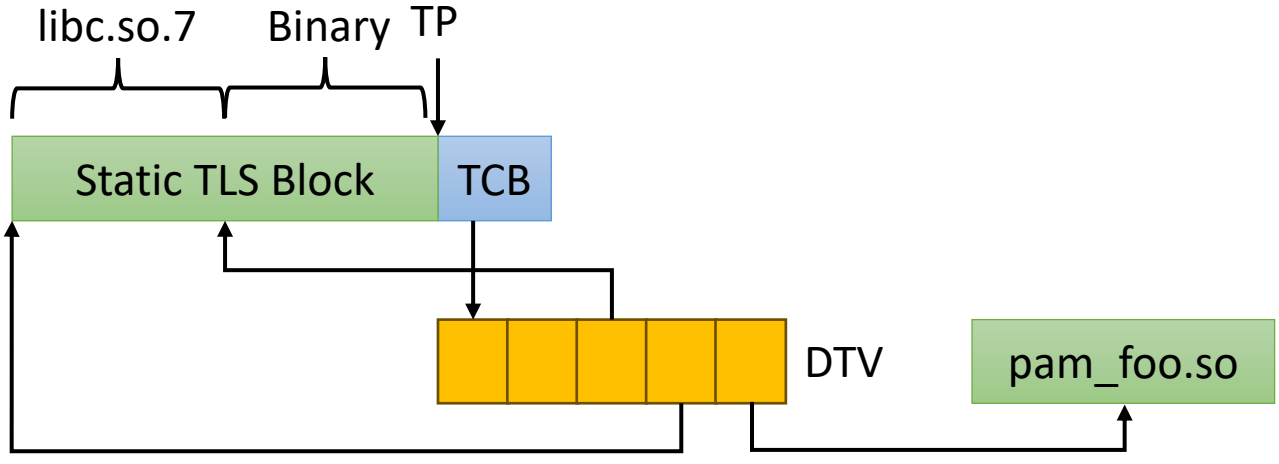
- Each thread has a logical **thread pointer** (TP) register
  - Backed by an architecture-specific register with an optional offset (bias)
- TP points to a **thread control block** (TCB)
  - The contents of the TCB are (mostly) opaque to the toolchain
- A **static TLS block** is allocated adjacent to the TCB for each thread
  - The static TLS block contains storage for TLS symbols defined by the executable and any shared libraries loaded during initialization
- The DTV array is allocated separately and the runtime loader must ensure it can locate the DTV for each thread
  - FreeBSD and Linux store a pointer to the DTV in the TCB
- There are two ways to arrange the allocation containing the TCB and static TLS block

Variant I Layout



- TCB size fixed by the ABI (contents still opaque)
- Static TLS block appended after TCB
- Used by aarch64, armv7, powerpc, powerpc64, riscv64

Variant II Layout



- TCB size is not fixed by the ABI
- Static TLS block prepended before TCB
- Used by amd64 and i386

Accessing TLS Variables

- TLS variables can be accessed in two modes
- Dynamic TLS names a TLS variable by a **module index** and a module-relative **offset**
  - Address of TLS variable = DTV[module index] + offset
  - Address of DTV is not directly accessible in the ABI, so this requires a call into the runtime loader
  - Call also permits lazy DTV growth and lazy allocation of dynamic TLS blocks
- Static TLS names a TLS variable by a single **offset** relative to the static TLS block
  - Variant I: Address = TP + sizeof(TCB) + offset
  - Variant II: Address = TP – offset
- Leads to four different access models

General Dynamic

Similar to _error()

- Access model that works for all use cases including dynamically-loaded modules
- Runtime loader defines a `__tls_get_addr()` function which accepts a module index and offset as arguments and returns a pointer to requested TLS variable
- GOT for module contains two adjacent entries for each TLS variable containing a module index and an offset
  - GOT entries initialized by new dynamic relocations
- Each TLS variable access calls `__tls_get_addr()` to obtain the variable's address
  - Compiler is permitted to cache address of a given TLS variable within a function

General Dynamic: x86-64

```
#include <threads.h>

extern thread_local int x;

int
foo(void)
{
    ret
}
```

x86-64 gcc 16.1 -O2 –fPIC

```
foo():
        subq    $8, %rsp
        data16
        leaq    x@tlsgd(%rip), %rdi
        rex64
        call    __tls_get_addr@PLT
        movl    (%rax), %eax
        addq    $8, %rsp
        ret
```

Why only a single argument to __tls_get_addr?

```
typedef struct {
     unsigned long index;
     unsigned long offset;
} tls_index;

void *__tls_get_addr(tls_index *ti);
```

Local Dynamic

- Optimized version of General Dynamic if TLS variables are known to belong to the current module
  - For example, static variables, or global variables with protected visibility
- Functions fetch the base address of current module's TLS block by passing a pointer to a special pair of GOT entries with an offset of 0 to `__tls_get_addr()`
- Address of individual variables are computed by adding relative offset (constant at static link time) to base address
- A given function reuse the base address to access multiple variables
- Not all that useful in practice

Local Dynamic: x86-64

```
#include <threads.h>

static thread_local int x;
static thread_local int y;

int
foo(void)
{
    ret
}
```

x86-64 gcc 16.1 -O2 –fPIC

```
foo():
        subq    $8, %rsp
        leaq    x@tlsld(%rip), %rdi
        call    __tls_get_addr@PLT
        movl    x@dtpoff(%rax), %edx
        addl    y@dtpoff(%rax), %edx
        addq    $8, %rsp
        movl    %edx, %eax
        ret
```

One call to __tls_get_addr instead of two

Static linker rewrites these static relocations into constants

Initial Exec

- Accesses TLS variables in the static TLS block via offsets relative to TP rather than the DTV
- Only suitable for accesses to TLS variables defined in modules known to be loaded at initialization time
  - Accesses by the executable
  - Accesses by libraries never loaded by dlopen() such as libc
- Single GOT entry contains offset of TLS variable in the static TLS block
- Each TLS variable access fetches the offset of the TLS variable from the GOT and uses this to derive an address from TP

Initial Exec: x86-64

```
#include <threads.h>

extern thread_local int x;

int
foo(void)
{
    return x;
}
```

x86-64 gcc 16.1 -O2 –fPIC -ftls-model=initial-exec

```
foo():
        movq    x@gottpoff(%rip), %rax
        movl    %fs:(%rax), %eax
        ret
```

%fs is TP on x86-64

Local Exec

- Optimized version of Initial Exec that can be used when the executable accesses TLS variables defined in the executable
- The relative offset of these TLS variables is a static constant at link time since the TLS block for the executable is always the TLS block closest to TP
- Each TLS variable access uses constant offset to derive an address from TP
  - No GOT entry, no dynamic relocations
- This is what static binaries end up using

Local Exec: x86-64

```
#include <threads.h>

static thread_local int x;

int
foo(void)
{
    return x;
}
```

x86-64 gcc 16.1 -O2 –fPIE

```
foo():
        movl    %fs:x@tpoff, %eax
        ret
```

Linker Relaxations

- For some relocations, the linker can sometimes replace a slower, more general instruction sequence or memory access with a faster, simpler one
- When this occurs, the linker is effectively relaxing the requirements of the generated code, hence the name **relaxations**
- For TLS there are several possible relaxations when linking executables
  - Relaxing GD -> LE for non-preemptible symbols
  - Relaxing GD -> IE for preemptible symbols
  - Relaxing LD -> LE
  - Relaxing IE -> LE for non-preemptible symbols

More Relocations

- All these new GOT entries need to be populated somehow
- Usually these are initialized by new dynamic relocations processed by the runtime loader
  - For example, module indices are unknown until the runtime loader assigns them so must always be generated by dynamic relocations
- In a few cases the compiler/linker can store constant entries in the GOT
  - Hardcoded offset of 0 used for Local Dynamic
  - If IE can’t be relaxed to LE in an executable, the associated GOT entry can at least be a constant

TLSDESC

- Linker relaxations can only be applied to at static link time
  - Libraries known to never be dlopen()ed can be compiled as IE
- However, most libraries are not dlopen()ed in most processes, but their TLS accesses always use GD which is slow
- TLS descriptors (TLSDESC) is an alternative to GD that permits runtime relaxations instead
- Instead of calling `__tls_get_addr`, the first GOT entry in the pair is replaced with a function pointer, and the second GOT entry is passed as the sole argument to the function pointer
- The runtime loader can use optimized routines for the function pointer in certain cases
  - For example, if the target variable is in the static TLS block, the pointer can avoid the DTV and derive the address from TP directly

Resources

- Ulrich Drepper's Specification for ELF TLS: https://www.akkadia.org/drepper/tls.pdf
- Compiler Explorer: https://godbolt.org
- MaskRay (LLVM developer)'s blog: https://maskray.me/blog/