DRuntime and You

David Nadlinger (@klickverbot)
ETH Zurich
Agenda

- **Warmup**: TypeInfo and ModuleInfo
- Exception handling
- Garbage collection
- Thread-local storage
- Fibers
- **Interlude**: C program startup

- Shared libraries
- Linker-level dead code elimination (--gc-sections)
Packages

- **object**: Top-level module, imported automatically
- **core.***: User interface, C standard library/operating system bindings
- **etc.***: Also user-facing, currently just etc.linux.memoryerror
- **gc.***: Garbage collector implementation
- **rt.***: Compiler support code, runtime initialization

- **gcc.*
- **ldc.*
class TypeInfo {
    string toString();
    size_t toHash();
    int opCmp(Object o);
    bool opEquals(Object o);
    size_t getHash(in void* p);

    TypeInfo next();
    uint flags();

    void[] init();
    size_t tsize();
    size_t talign();

    bool equals(in void* p1, in void* p2);
    int compare(in void* p1, in void* p2);
    void swap(void* p1, void* p2);
    void destroy(void* p);
    void postblit(void* p);

    OffsetTypeInfo[] offTi();

    void* rtInfo();
}
```c
struct ModuleInfo {
    string name();
    uint flags();
    void function() tlsctor();
    void function() tlsdtor();
    void function() ctor();
    void function() dtor();
    void function() unitTest();
    ModuleInfo*[] importedModules();
    TypeInfo_Class[] localClasses();
    void function() ictor();
    void* xgetMembers();
    uint index();
    static int opApply(scope int delegate(ModuleInfo* ) dg);
};
```

module foo.bar;

class C {
    this() { x = 10; }
    int x;
}

void main() {
    auto c = cast(C)Object.factory("foo.bar.C");
    assert(c != null && c.x == 10);
}
Exception Handling

• Two main tasks: Stack unwinding, finding landing pads (catch/finally/scopes with destructors)
• Compiler- and platform-specific
• DMD/Win32: Structured Exception Handling (SEH), rt.deh_win32
• DMD/Win64 and Posix: Custom implementation, rt.deh_win64_posix
• GDC and LDC (except Win64): libunwind does heavy lifting, we provide *personality function*, see gcc.deh and ldc.eh
• LDC/Win64: SEH, ldc.eh2
• Backtrace generation
Garbage collection

• Mark-and-Sweep collector:
  • Mark phase: Transitively mark all reachable objects as live
  • Sweep phase: Free those allocations that have not been marked
    (potentially also reclaim entire page, etc.)

• Potential GC roots:
  • Stack
  • (Shared) globals
  • TLS globals
  • (Explicitly added roots/ranges using core.memory.GC interface)
Thread-Local Storage (TLS)

module test;
int myGlobal;
int foo() {
    return myGlobal;
}

• Linux x86_64 static TLS model:

_D4test3fooFZi:
    mov eax, dword ptr fs:[_D4test8myGlobali@TPOFF]
    ret
Thread-Local Storage (TLS)

module test;
int myGlobal;
int foo() {
    return myGlobal;
}

• Linux x86_64 global dynamic TLS model:

_D4test3fooFZi:
    lea rdi, qword ptr [rip + _D4test8myGlobali@TLSGD]
    call   __tls_get_addr@PLT
    mov    eax, dword ptr [rax]
    pop    rdx
ret
TLS on OS X

OS X had native TLS only since 10.7 (which LDC requires), DMD has a custom implementation:

- Emit TLS variables to named __tls_data and __tlscoa1 nt sections
- rt.sections_osx:
  - getTLSBlock(): Lazily create pthreads TLS variable
  - getTLSBlockAlloc(): Read that variable, if TLS not yet initialized for this thread copy initializers from above sections
  - __tls_get_addr(): Takes an address in either of the two sections, translates it to thread-local copy. Calls emitted by compiler.
TLS on OS X

LDC on OS X:
• Use default LLVM implementation
• Need to use functions from dyld_priv.h to get GC ranges
  • Might be a problem for App Store deployment?
  • API uses Apple-specific Blocks extension
import core.thread : Fiber;
void f() {
    writeln("In f(), yielding execution");
    Fiber.yield();
    writeln("Back in f() again");
}

auto fiber = new Fiber(&f);
fiber.call();
writeln("In caller");
writeln("In caller");
fiber.call();
Fibers

• Cooperative, user-space multitasking
• Just save the registers to the stack, switch out stack and instruction pointers, load registers from new stack
• Need to keep TLS and EH intact (easy in theory; in practice however…)

```c
int tlsGlobal = 42;
void bar() {
  writeln(tlsGlobal);
  Fiber.yield();
  writeln(tlsGlobal);
}
```
Pop quiz: You are writing a C program on GNU/Linux using GCC. What's the name of the first function that is executed when your program starts?
Pop quiz: You are writing a C program on GNU/Linux using GCC. What's the name of the first function that is executed when your program starts?

(Hint: It's not "main").
GNU/Linux program startup

- loader calls _start, defined in glibc
- _start calls __libc_start_main (glibc/csu/libc-start.c)
  - Store stack end
  - Set __environ
  - Call global constructors (.ctors, __attribute__((constructor)))
  - main(...)
  - Call global destructors (.dtors, __attribute__((destructor)))
Recap

Need to determine:
• All ModuleInfos
• Stack region
• Global data segments (.data, .bss)
• TLS segments for each thread
• DMD: Exception handling tables
“Old” module registration

- `_Dmodule_ref`: Global linked list of `ModuleInfo` references
- Each object file adds its module using a (C) global constructor
- Simple, portable, does not need any special compiler support
- Still used by LDC on platforms without shared library support, on Solaris/Android/other Posixen by DMD
- For GC ranges, just use `_bss_start`, `_end`, et al.
- DMD: Bracketing symbols for EH tables
- However: Shared libraries
Shared libraries

- Only applies to Posix/ELF for now

- Different use cases:
  - D program linking to D shared libraries
  - D program loading D shared libraries at runtime
  - C program linking to D shared libraries
  - C program loading D shared libraries at runtime

- All require use of shared druntime/Phobos
Module conflict detection

• Want to prohibit defining same D module in two different images, chaos would ensue
• Idea: When loading a shared library:
  • Iterate through all ModuleInfo references
  • For each of them check if the address is in the current image
  • If not, dynamic linker has merged it with same module in other library, fail
• Problem: Copy relocations
Detour: Copy relocations

- What if you have a non-PIC executable (position-dependent code)…
- …that references a data symbol defined in a shared library it links to?

- Fix:
  - Allocate space in the executable's .bss section
  - When loading library, copy symbol from library into that memory
  - Fix up references in library, which is built with PIC

- Breaks our simple module conflict detection!
Module conflict detection, v2

- Want to prohibit defining same D module in two different shared libraries, chaos would ensue
- Idea: When loading a shared library:
  - Iterate through all ModuleInfo references
  - For each of them check if the address is in the current image or in the main executable's BSS section
  - If not, dynamic linker has merged it with same module in other library, fail
- Seems innocent enough, but we'll have fun due to linker bugs
Design constraints

- Want to stay on LLVM IR level for tooling and ease of use
  - Cannot emit arbitrary relocations
- Custom linker scripts are out
- LLVM IR does not support COMDAT symbols in custom sections (arguably a bug, certainly an arbitrary limitation)
_d_dso_registry:
• Checks whether DSO has already been registered
• Uses `dl_iterate_phdr` to locate data/TLS segments
• Checks module collisions
• Registers module with global list, runs constructors, etc.
**--gc-sections**

- Linker-level removal of object file sections that are not referenced by any other code (certain sections are *roots*, see `KEEP` in `ld --verbose`)
- Idea: Put each function/variable into its own section
- Do not want linker to remove ModuleInfo references in `.minfo`, for DMD also custom EH tables
- Custom linker script breaks just using gcc to link, other tooling
- `ld.gold` merges COMDATs before checking their dependencies
--gc-sections

- Unsolved in DMD, WONTFIX for GDC
- Having one .ctor per module (LDC) naturally solves this, pin the ModuleInfo there
- Because of a bug in ld.bfd, cannot use `__bss_start`, `__end` are made local to the main executable; have weak `_d_execBss{Beg, End}Addr`
- LDC binaries (static runtime, release mode) are typically ≈¼ as big as DMD built ones, ≈¼₁₆ the size of default GDC binaries
- Possible alternative: Whole program/link time optimization
Resources

- OS X open source tools:
  - System linker: http://opensource.apple.com/source/ld64
- (Linux) linker internals: http://www.airs.com/blog/archives/38
- Windows TLS: http://www.nynaeve.net/?tag=tls
- Issues with migrating fibers across threads: LDC GitHub #666
- Relevant dlang Bugzilla issues: 879, 11378, 13025
fin