Hell Freezes Over

Freezing Limbo modules to reduce Inferno's memory footprint

David Boddie <david@boddie.org.uk>

https://www.boddie.org.uk/david

9th International Workshop on Plan 9

About me

Background:

- Physics/Astronomy/Mathematics not Computer Science/Engineering
- Technical Writer/Developer

Interest in Inferno:

- Interested from a computing history perspective
- Then wanted to try using something other than Linux
- Now mostly trying to port Inferno to different devices

Context

Small devices:

- Some flash memory, not much RAM
- Examples:
  - STM32F405 – Cortex-M4, 192 KB RAM, 1 MB flash
  - RP2040 – Cortex-M0+, 264 KB RAM
- Both use the Thumb-2 instruction set

I'm using the STM32 MCU.

See caerwynj/inferno-os for Inferno on RP2040.

Building Inferno on STM32

Toolchain for Thumb instructions:

- Compiler: `tc`
- Assembler: `5a`
- Linker: `tl`

References:

- Notes on the Inferno Thumb Compiler
- More Thumb Compiler Notes

Problems

Two issues:

1. A lot of data is copied into RAM
2. Loading a Limbo module has an overhead

Thoughts:

- Not much we can do about issue 1
  - Modify the compiler and linker
  - Use a custom allocator?
- Issue 2 might be worth exploring

Limbo modules

Dis files contain features like these:

- Magic number
- Code
- Types
- Data
- Links
- Imports
- Handlers

See the Dis object file man page.

Loading Limbo modules

- `readmod()` in `libinterp/readmod.c`:
  - Allocates RAM for the contents of a Dis file
  - `parsemod()` in `libinterp/load.c`:
    - Decodes each section of the file
    - This involves allocating RAM for data structures
- Frees the RAM containing the file data

Result is a `Module` struct with pointers to allocated memory.

Code section

Just instructions, non-self-modifying:

```
; disdump dis/pause.dis
newcw       , $0, 44(fp)
recv        44(fp), 40(fp)
movp        0(mp), 44(fp)
ret
```

Why not move them into flash memory?

Why not move them into flash memory?

Decoding the Dis file expands the instructions.

- So need to expand them at kernel compile-time
- Compile them into the kernel as read-only data (“*freeze*” them)
- Then just point the interpreter at them

The interpreter normally patches instructions at run-time.

- It creates `Inst` structures and patches branches
- The compiler can't know where each `Inst` will be at run-time
- But we can decide where in flash memory to put them
- So we can patch the instructions ourselves at kernel compile-time

Practical problems

- How to freeze Dis files?
- How does the interpreter know which files are frozen?
- How does the interpreter find frozen modules?
- How do we automate this?

How to freeze a Dis file?

- There is already code to read Dis files in `libinterp/load.c`
  - Create a `freeze` tool and borrow existing code
  - Expand Dis code and compile it into the kernel
- What do we put in the root file system instead of the original file?
  - Could put only the code section in the kernel
  - Easier to put the whole file in the kernel
  - Leave a placeholder in the root file system

```
; xd dis/pause.fdis
0000000 c000fd15 2f646973 2f706175 73652e64
0000010 69730000
0000013
; cat dis/pause.fdis
��/dis/pause.dis;
```

Which files are frozen and which are normal?

- Use a different magic number and a token to identify the frozen module
- Change the loader to do something different in `parsemod()`:
  - Normal magic number?
    - Execute the function as normal
  - New magic number?
    - Execute `loadfrozen()` instead
    - A new function in `libinterp/loadfrozen.c`

Finding frozen modules

We have:

- Converted Dis files in the kernel
- Placeholder files in the root file system

Interpreter loads a placeholder file:

- Checks for the new magic number and reads the token
- Searches a linked list of structs to find the matching token and data
- Calls the `_loadfrozen()` function
  - Does what `parsemod()` normally does
  - Except that the code is already unpacked and ready to use

Missing pieces

We need to initialize the linked list of frozen modules:

- Implement `addfrozenmod()` to register frozen modules
- A new function in `libinterp/loadfrozen.c`

Still need to:

- Describe all the frozen modules
- Register them when the kernel starts

Let's do both of these together.

Registering frozen modules

The configuration file (`stm32f405.conf`) contains a `root` section with entries like this:

```
    /dis/pause.dis  /dis/pause.fdis
```

In the `mkfile`:

- `mkfrozen` finds `.fdis` entries in the configuration file
  - Runs `freeze` on the corresponding `.dis` files
  - Creates a `frozen.h` file with calls to register modules
  - This is included by `frozen.c` in `initfrozen()`
- `listfrozen` creates a build rule for the frozen modules

Registering frozen modules

At run-time:

- `main()` calls `initfrozen()`
- Calls `addfrozenmod()` for each frozen module
  - Thanks to the macro in the generated `frozen.h` file

The interpreter has a linked list of frozen modules, so:

- It can load a placeholder file
- Check the magic number
- Look up the token to find frozen module data

Summary

- A frozen module is like a normal Dis file
  - but with different magic number
  - and expanded code section
- A placeholder file refers to a frozen module
  - using a token (basically just the original path)
- The interpreter
  - checks for the new magic number
  - uses the token to look up the frozen module data
  - loads it differently (no need to unpack code from flash)
- The build system
  - finds frozen module declarations in the configuration file
  - creates frozen module data
  - generates code to register frozen modules at run-time

Caveats

Code itself read-only, but…

- Sometimes assumed to be in allocated memory
  - `freemod()` function in `libinterp/load.c`
  - `devprog` file system – `ps` breaks…
- Introduce a `frozen` flag in the `Module` struct

Frozen modules not completely frozen.

- All the other sections still need RAM
- Takes time to track down uses of these sections

Benefits

Reduced allocation overhead:

- No need to allocate RAM to load a Dis file
- Less RAM used for instructions, especially for big modules

Indirect benefits:

- Less RAM used for root file system
- A lot of that could/should be in flash anyway
- Compiler changes?

What's next?

- Make the linker's `-t` option work
  - Puts some strings in the text section of the kernel
  - Already done, saves a few kilobytes
- Move more constant data into the kernel's text section
  - Early days, slow progress
- Use a different toolchain to build kernels?
  - Like Inferno on RP2040
- Compile to native code instead of just expanding Dis code
  - Could be expensive in terms of flash memory
  - Not looked into feasibility
  - Will no doubt require attention to allocation issues

Thanks

- Charles Forsyth for compiler hints and tips
- Michael Engel for discussions about microcontrollers
- inferno-os and 9fans participants

Resources:

- https://github.com/dboddie/inferno-os/tree/stm32f405
- https://dboddie.gitlab.io/inferno-diary