Link time dead code and data elimination using GNU toolchain

Denys Vlasenko

It makes sense to run the same software on embedded devices
as we run on desktops (for example, Linux kernel),
in order to leverage an enormous investment of talent, time and
in some cases, money, which went into creating and improving it.

However, embedded devices are generally more resource-constrained
than desktops or servers. To make the software more suited
for embedded use, it needs to be put on a diet.

This talk will show you how to eliminate one of the typical
sources of code bloat - public functions which are not reachable
through other modules of the program. On typical programs, this
technique eliminates up to 10% of code/data.

A detour: an admin plays with old, unused server...

- It couldn't boot Linux - problem recognizing a SCSI disk controller.
  Added the PCI ID for the controller to aic7xxx driver.
  Bingo! It works... but why this driver is ~300 kb of code?
- What good geek should do?
  - Investigate it!
  - The driver has a lot of debugging code controlled by a CONFIG option. The code is always compiled, it is just never called if option is off...
- What good geek should do?
  - Fix it!
  - It took five days to refactor the code and stop it from being linked in if option is off. 100K of code is shaved off.
- What good geek should do?
  - Submit it!
  - It took three YEARS to push the fix to mainline kernel.
    (admittedly, the admin was not focusing on the effort, but still...)

This does not scale!

Humans should not do optimizations by hand if machine can do it!

- Why was the code not dropped by compiler/linker?
  - ELF object module (.o file) consists of blocks called sections.
  - Compiler puts all executable code produced by compiling C code into section called .text, r/o data into section called .rodata, r/w data into .data, and uninitialized data into .bss. Example:

    module1.c
    ```
    int f(void) { return 0; }
    void unused_func(void) { a_lot_of_code; }
    ```
    main.c
    ```
    int main(int argc, char **argv) { return f(); }
    ```
  - module1.o: section .text contains code of both f() and unused_func()
  - main.o: section .text contains code of main()
  - module1.o (and a lot of other .o files) is put into a lib.a
  - we are creating executable: `gcc -o executable_name main.o lib.a`

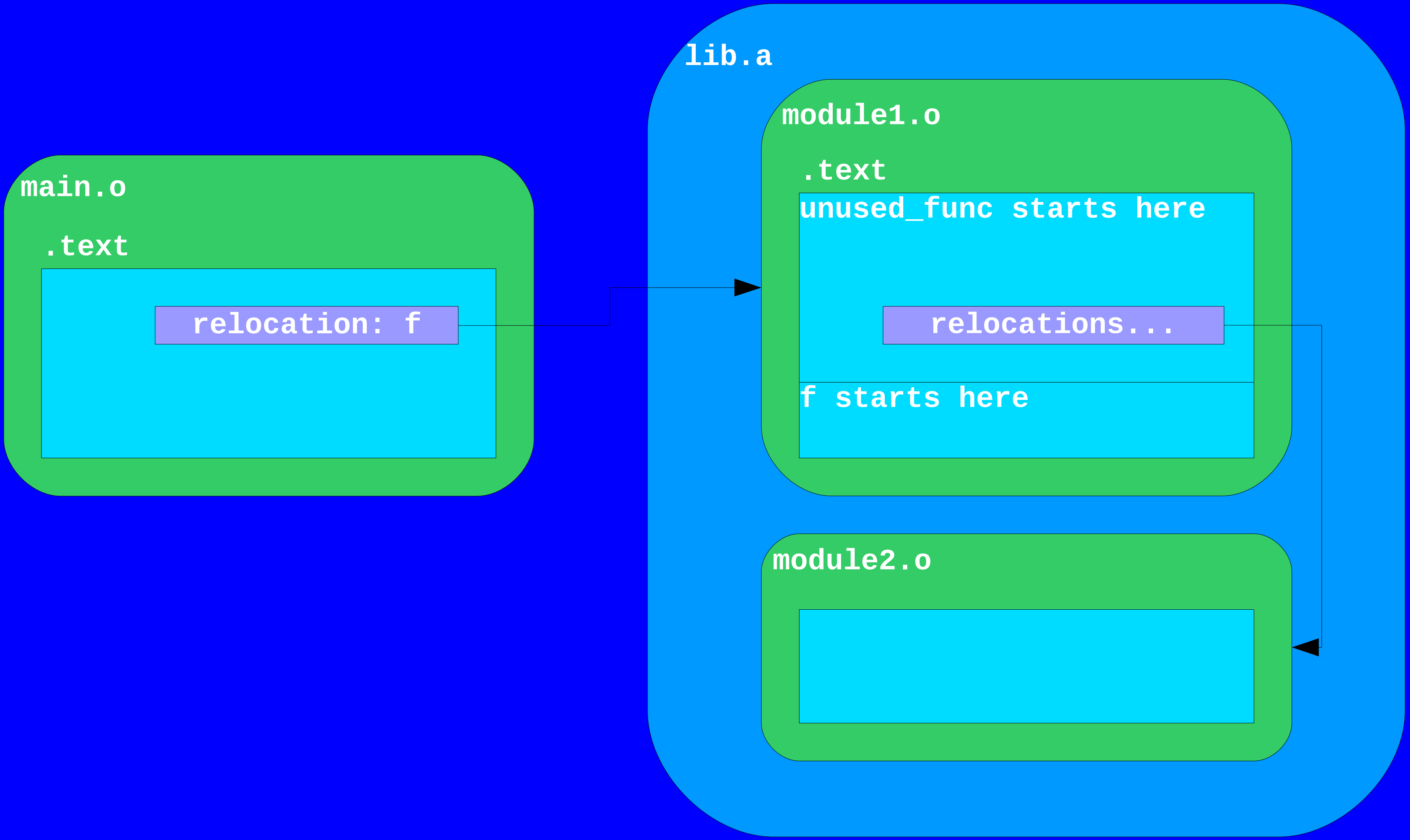
Welcome to the 1970s

- How exactly executable is created?
  - Linker sees that main.o has an undefined reference to label f.
  - Linker scans lib.a and finds that module1.o has this label defined.
  - Linker adds all sections from it, including entire .text section from module1.o, to the executable.
  - Therefore unused_func() body is included too.

- **WHY???**
  - Linker has no way of eliminating unused_func() code because it is no longer known which part of .text section in module1.o corresponds to which function. It is only known where f() function starts inside it, but as far as linker is concerned, f() may jump around entire .text section. Linker has to include entire .text section, or risk losing parts of f() body!
  - This is how linking works in Unix since the beginning.

Come on, Turbo Pascal twenty years ago did better than this!

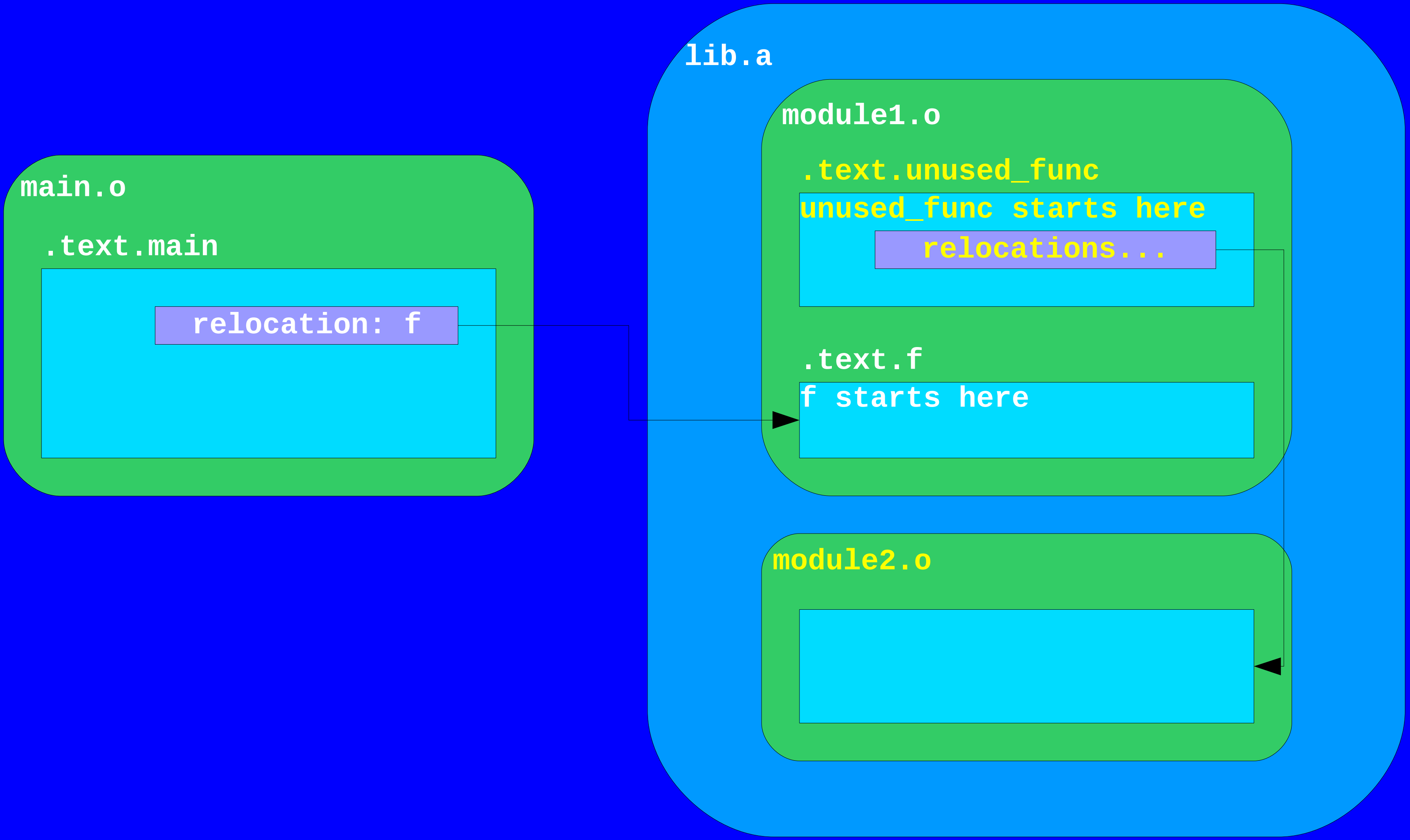
Typical Unix linking



Houston, we have a problem!

- Is there a solution?
  - Yes! Put every function to its own small section!
  - gcc -ffunction-sections -fdata-sections does it.
    f() code goes to .text.f section, unused_func() goes to .text.unused_func section, and so on.
  - Now linker can distinguish between bodies of different functions.
- Are we done?
  - No, linker will still combine all .text.* sections from selected modules into one .text section in the executable - including .text.unused_func.
- How to tell linker to drop unused sections?
  - ld --gc-sections (if you invoke ld from gcc, use gcc -Wl,--gc-sections).
    With this option, ld will not pull in all sections from an .o file, it will pull only those which are needed to resolve a symbol.
    (this is not *exactly* how it works... we will return to this later)

Linking with -ffunction-sections --gc-sections



Pitfalls?

- Things will break if the program uses special sections which are meant to be pulled into the program when other parts of .o files are pulled in, even if these “magic” sections themselves are not referenced.

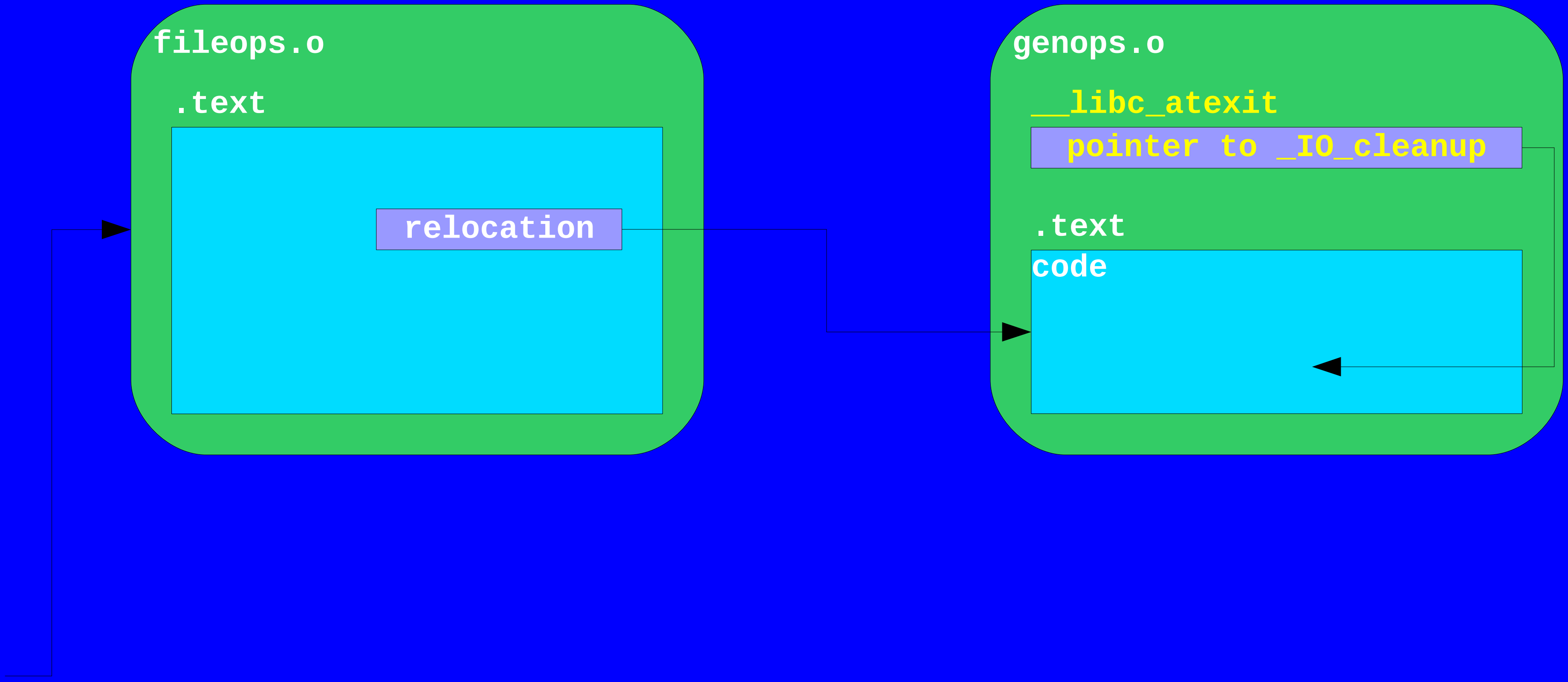
- Example (http://sources.redhat.com/bugzilla/show_bug.cgi?id=3400):

```
#include <stdio.h>
int main() {
   printf("Hello world\n");
}
```

  This program uses stdio functions and data (through printf() function). When it is linked via "gcc -Wl,--gc-sections -static", sections containing them are included, but a section named __libc_atexit from genops.o file, which holds a pointer to _IO_cleanup() function, is not referenced by any of them and is not included, even though other parts of genops.o are.

  As a result, atexit function responsible for flushing stdout won't be called on exit, and “hello | cat” command will output nothing.

Magic sections: glibc example



How to fix “magic section” problem?

- In many cases, it can't be fixed. You must not link statically against libraries which use “magic section” technique, and must not use it yourself.
- Bad news: even glibc is using it.
  But linking statically against glibc is not a good idea for a half dozen other reasons anyway...
- Good news: dynamic linking is not affected!
- Maybe we can invent a new section property and mark “magic sections” with it, and teach linker to not drop such sections (IIRC, no such attribute exists as of now. Today it can only be done via *custom linker script*).
- The best, but hardest way, is to get rid of “magic sections”. One method is to add references to them from sections which need their presence.

“Magic sections” in Linux kernel

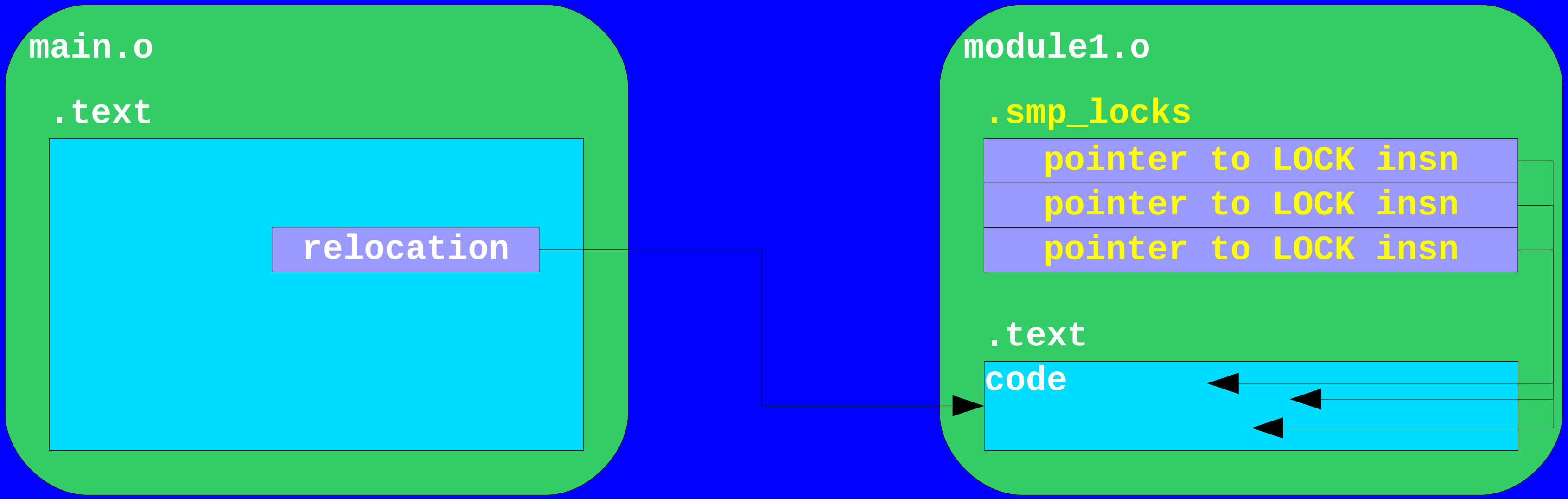
- Linux kernel: we can still use “magic sections”, but kernel linker scripts have to be fixed.

- Wait a sec, what is “linker script”? (put your propeller hat on)
  Linker essentially takes input sections, possibly drops some of them, then puts the rest into output file. Linker can merge several input sections into one output section. GNU ld determines which sections to drop, join, or copy without joining, and in what order, by looking in a “linker script”. Usually it uses built-in one, but with option -T you can use your own. Linux kernel does that.

- This linker script fragment says to combine sections named .text[.anything] and .smp_locks into output section .text:
  ```
  .text: { *(.text) *(.text.*) *(.smp_locks) }
  ```
- The fix is to let linker know that some sections are “magic” and must never be dropped using KEEP directive:
  ```
  .text: { *(.text) *(.text.*) KEEP(*(.smp_locks)) }
  ```

Note: **I lied!** ld --gc-sections does not collect only needed sections. It collects all sections from needed .o files, just as it was doing for last 30 years. But before combining them into output file, it drops those sections which are not reachable from “seed” sections.

Magic sections: kernel example



How much do we save?

- Busybox (current git, statically linked against uclibc)

```
   text    data     bss     dec     hex filename
 823890     459    7364  831713   cb0e1 busybox
 816186     443    7256  823885   c924d busybox.gc
 823305     456    7308  831069   cae5d busybox.fs
 813999     440    7208  821647   c898f busybox.fsgc -1.21%
```

- GDB (6.3, dynamically linked against glibc)

```
   text    data     bss     dec     hex filename
1930262   38924  105704 2074890  1fa90a gdb
1930205   38920  105680 2074805  1fa8b5 gdb.gc
1930522   38924  105704 2075150  1faa0e gdb.fs
1872341   38868   88112 1999321  1e81d9 gdb.fsgc -3.64%
```

How much do we save? (cont.)

- Kernel (linux-2.6.33-rc6 monolithic x86-64)

```
   text    data    bss      dec     hex filename
15551696 1900592 896916 18349204 117fc94 vmlinux
15547165 1900592 831380 18279137 116eae1 vmlinux.gc
14408101 1869506 890061 17167668 105f534 vmlinux.fs
14079197 1813040 827812 16720049  ff20b1 vmlinux.fsgc -8.87%
```

- Kernel (linux-2.6.33-rc6 allyesconfig sans debugging, x86-64)

```
    text    data      bss      dec     hex filename
59145131 6103031 19789800 85037962 511938a vmlinux
56899756 6051191 19724264 82675211 4ed860b vmlinux.gc
56299799 6022456 19756228 82078483 4e46b13 vmlinux.fs
52875500 5190455 19701868 77767823 4a2a48f vmlinux.fsgc -8.54%
```

Pushing patches to mainline kernel

- Problem #1: kernel's modpost tool gets confused by 64k+ sections.
  - Easy to fix. The patch is in -mm: http://lkml.org/lkml/2010/1/14/573

- Problem #2: kernel already uses sections named .text.foo for unrelated purposes.
  - Relatively easy to fix by renaming sections. The patch is in kbuild-git already and hopefully will be in mainline: http://lkml.org/lkml/2010/3/8/148

- Problem #3: kernel has magic sections.
  - Easy fix: add KEEP() directives to kernel's linker scripts where they mention these sections.
  - Drawback: this defeats garbage collection logic, dead code which is referenced by magic sections will not be eliminated.
  - More complex, but better performing fix: add dummy references to magic sections from the places they reference.
  - The patch is ready, but want to have #1,#2 accepted into mainline first.

Conclusion

- Compile your modules with -ffunction-sections -fdata-sections – there are no downsides to it!
    - This includes static libraries, not just binaries – make it possible for users of your library to benefit from more efficient dead code removal.
- Link your binaries with --gc-sections, unless you have to link against nasty third-party static library which uses magic sections.

Your questions please.

does it work only on Intel?

- why it does not work so well with kernel modules?

what CONFIG option will control it?

- is this useful for something else than reducing size?

any adverse effects?

- how to add references to “magic sections” so that they work with --gc-sections?