Kernel configuration and building in Linux 2.5

Kai Germaschewski
University of Iowa
`kai@germaschewski.name`

Sam Ravnborg
Ericsson Diax A/S
`sam@ravnborg.org`

Abstract

The development phase of Linux 2.5 brought substantial changes to the kernel configuration process, the actual kernel build and, in particular, implementation and building of loadable modules.

The first part of this paper will give an overview of the user-visible changes which occured in Linux 2.5, on the one hand for users which build kernels themselves, on the other hand for developers which maintain drivers or other parts of the kernel, in order to help porting to Linux 2.5/2.6.

The second part of the paper deals with the actual design and implementation of the current kbuild, showing how *GNU make* is actually flexible enough to allow for nice condensed Makefile fragments which per subdirectory describe which objects to build into the kernel or as loadable modules. The paper ends with an outlook showing possible approaches for implementing additional features.

The paper also explains the improvements in handling loadable kernel modules, including symbol versioning, and the necessary build system changes.

1 Introduction and history

Why is a kernel build system necessary at all, and why does the Linux kernel use its own special solution?

As the Linux kernel evolved from a student's terminal emulation program towards a full-featured UNIX-like kernel, changes to the way it was built became necessary and were integrated, so the kernel build system basically followed the evolutionary development of the kernel itself.

In the science world, in particular people running numerical simulations, many people consider a build system completely unnecessary, they just run

```
f77 code.f
./a.out
```

However, this approach obviously doesn't scale to large projects. To keep projects maintainable, some kind of modularization occurs the code is divided into a number of source files and as the project is growing further, a directory hierarchy is introduced which helps organizing the code even further.

During development, normally only one or a few files are edited and then the developer wants to rebuild the program, in this case the kernel *vmlinux*, be it just for compile-time checks or testing.

First of all, one does not want to enter all the commands manually for each build, so some type of script is necessary to record those commands. Next, it is actually a waste to recompile every file if only few have changed. Smart programmers recognized this a long time ago and invented a tool called *make* which is still the most popular build tool used today. We assume in this paper that the audience is familiar with the basics of *make*.

So even Linux 0.01 came already with a Makefile which took care of building the kernel.

As time passed and Linux matured, new features were incorporated into the build system, such as

- **Automatic generation of dependency information.** *make* only handles simple dependencies like the dependency of an object file on the corresponding source automatically, other prerequisites as for example included header files need to be added to the Makefile explicitly, a task which can be (and was) automated.
- **Configurability.** As the code base for the Linux kernel expanded, a need for a user selectable configuration became apparent and was introduced before release of Linux 1.0. This system allows the user to answer questions with respect to which components are desired, and then only builds those components into the kernel.
- **Different architectures and cross–compilation.** Linux introduced support for different architectures, which means the kernel is build from a large arch-independent code base as well as some machine-specific low-level code. It is also often necessary to cross–compile the kernel, i.e. do the compilation on a different platform than it is actually run on.
- **Loadable modules.** Within Linux 1.3, support for loadable kernel modules was introduced, which again needed special support in configuration and building of those objects.

In particular the high configurability and support for loadable modules distinguish the Linux kernel from most other projects, and it thus comes as no surprise that its build system also evolved away from a standard Makefile. However, *make* is still the underlying tool used for building the kernel. In fact, the extensibility of the *GNU* version of *make* [1] in conjunction with some support scripts / C code renders it possible to meet the goals listed above.

2 A dummy's guide to kbuild

This section is addressed to users and will explain how to use the kernel build system in Linux-2.5/2.6. "Users" (as opposed to "developers") here mean people who download the linux kernel tree source, possibly apply patches and then build and install their own kernels. Of course, since kernel developers need to build and run kernels, too, this section is of relevance for them as well.

The build system is based on *GNU make*, i.e. all commands are given to *make* by invoking it as

```
make <target>
```

Contrary to many userspace packages which are using autoconf/automake, there is no preceding `./configure` necessary, the necessary configuration process is embedded into the build process.

The actual targets are in part platform-specific, for example on i386 one typical wants to build the boot image `bzImage` and modules. A list of supported targets for the platform can be obtained from `make help`.

Arch maintainers should setup their arch-specific Makefile in a way that invoking *make* without parameters will build the commonly used boot target for the architecture, for example on i386 just typing

```
  make
```

will build bzImage and modules (the latter only when `CONFIG_MODULES` is selected, of course) which is what is typically needed.

If one just runs *make* after unpacking the kernel source tarball, *make* will actually just error out, asking you to configure your kernel first by running `make *config`. (In Linux-2.4 and before, it would invoke `make config` for you, but this is the wrong choice in 99% of the cases, since nobody likes answering a straight sequence of a couple of hundred questions…)

To generate a new kernel configuration, it is recommended to use `make menuconfig`, `make xconfig` (which uses Qt now) or `make gconfig` (uses gtk).

However in most cases, it is easier to adapt an existing kernel configuration to the current kernel than to create a new one from scratch. This is done by copying the *.config* file into the top-level directory of the source tree. kbuild will recognize that the .config file may need adaption for the current kernel source and automatically run `make oldconfig` for you, which makes sure that *.config* is consistent with the current rules and asks the user about the value of previously not existing options.

So the normal sequence for building a kernel is just

```
  cp /my/old/.config .config
  make
```

where one could insert a `make *config` between those two steps if a change of configuration options is desired.

The last remaining step is the installation of the newly built kernel. The procedure to install the boot image depends of course on the bootloader used.

For *lilo*, the kernel boot image *bzImage* should be copied to a certain location (typically */boot*), then */etc/lilo.conf* may need an appropriate entry and finally */sbin/lilo* must be run.

For *grub*, copying the kernel image to */boot* and possibly editing */etc/grub.conf* should suffice.

An important change is that on i386 *bzImage*/*zImage* can not be directly booted from a floppy disk anymore. Instead the targets `zdisk` and `fdimage` create a boot floppy disk and a boot disk image, respectively. Those targets now require mtools and syslinux to be installed.

Since the actual installation of the boot image varies as described above, one can give the `install` target to `make`, which will invoke a user– or distro–provided script, `~/bin/installkernel` or `/sbin/installkernel` which can be customized for the local setup.

Installing modules is simpler, just invoking `make modules_install` will do the necessary work. By default this will install into `/lib/modules/‘uname -r‘/`, though this can be customized by setting `INSTALL_MOD_PATH`, e.g. if one wants to collect the modules for transfer onto a different machine.

This is basically all knowledge which is needed to build a Linux kernel—everything else is handled automatically by the build system. Applying patches, editing files, changing configuration options or adding compiler flags—

the build system will notice the change and rebuild whatever is needed. The one exception to this rule is changing the architecture (by setting the `ARCH` variable), which needs an explicit `make distclean` to work correctly.

3 kbuild for kernel developers

3.1 kbuild in the daily work

Since developers tend to build kernels and modules a lot, the previous section of course also applies to them, in particular the fact that just running `make` will recognize all changes and rebuild whatever is necessary to generate a consistent *vmlinux* and modules.

Some additional features exist to support the development / debugging process:

- `make some/path/file.o` will rebuild the single file given, using compiler flags (e.g. `-DMODULE`) according to the current *.config*.
- `make some/path/file.i` will generate a preprocessed version of `/some/path/file.c`, again using compiler flags for the current configuration.
- `make some/path/file.s` will generate a file containing the raw assembler code for `some/path/file.[cS]`.
- `make some/path/file.lst` (little known but very useful) gives interspersed assembler code with the C source, relocated to the correct virtual address when a current *System.map* exists.

Another useful feature for the daily work, which has existed for a long time, is the ability to override the `SUBDIRS` variable on the command line, which will force *make* to only descend into the given subtree. This can be very useful for faster build times, but it bypasses some dependencies and thus does not guarantee to result in a consistent state.

So while e.g. working on the *hisax* ISDN driver, it's useful to call *make* as

```
make SUBDIRS=drivers/isdn/hisax \
     modules
```

for compile checks etc. However, before installing a new kernel and modules, the authors advise to always run a full `make bzImage/vmlinux/modules` (or otherwise, do not complain ;).

3.2 Integrating a driver

Basically each subdirectory in the Linux kernel tree contains a file called *Makefile*, which is included by *make* during the kernel build process. However, these Makefiles are different from regular Makefiles in that they normally don't have any targets or rules, but only set variables which tell the build process what should be built and the latter takes control of the actual compiling and linking.

In conjunction with the Makefile there normally exists a *Kconfig* file, these files were introduced with the configurator rewrite by Roman Zippel and replace the old *Config.in/Config.help* files used during the configuration phase of the kernel build.

This paper does not intend to elaborate on the new kernel configuration system, however the following examples will provide some basic usage guidance.

The most common case is adding a new driver which is built from a single source file.

```
config TIGON3
    tristate "Broadcom Tigon3 support"
    depends on PCI
    help
      This driver supports Broadcom Tigon3 based gigabit Ethernet cards.

      If you want to compile this driver as a module ( = code which can be
      inserted in and removed from the running kernel whenever you want),
      say M here and read <file:Documentation/modules.txt>.  This is
      recommended.  The module will be called tg3.
```

Figure 1: *Kconfig* fragment for the Tigon3 driver.

Figure 1 shows the *Kconfig* fragment for the Tigon3 driver, which defines a config option `TIGON3` (the corresponding variable will be given the name `CONFIG_TIGON3`), which is a tristate, i.e. can have the values `y`, `m`, or `n` with the obvious meanings (a config option which has been turned off, actually has the value `""`, to be correct here). The fragment `depends on PCI` states that this option is only selectable when the option `PCI` is also set (that is, if the kernel supports the PCI bus).

The Makefile fragment for the Tigon3 driver

```
obj-$(CONFIG_TIGON3) += tg3.o
```

is very short and though a little awkward at first, a very elegant way to quickly express what files are supposed to be built. The idea dates back to Micheal Elizabeth Castain's *dancing Makefiles* [2] and was globally introduced into the kernel by Linus shortly before the release of kernel 2.4.

What happens is that depending on the config option `CONFIG_TIGON3`, the value `tg3.o` is appended to either of the variables `obj-y`, `obj-m` or `obj-`.

The meaning of those special variables is as follows:

- `obj-y`. All objects listen in `obj-y` will be compiled as built-in objects, and will finally be linked into *vmlinux*.
- `obj-m`. All objects listed in `obj-m` and not not listed in `obj-y` will be compiled as modules (so they actually end up being called e.g. `tg3.ko` in 2.5/2.6).
- `obj-`. All objects listed in `obj-` and not in `obj-y` or `obj-m` will be ignored by kbuild.

Since the build system does not have any further information on `tg3.o`, it will try to build it from a source file called `tg3.c` (or an assembler source `tg3.S`, which only happens in the architecture dependent part of the kernel, though).

This is all what is needed to integrate a simple driver into the kernel build, other than of course writing the driver (`tg3.c`) itself.

It is also possible to list more than one object to be built in the Makefile statement. The Makefile line dealing with the *eepro100* driver looks like the following:

```
obj-$(CONFIG_EEPRO100) += \
        eepro100.o mii.o
```

If this driver is selected, the *mii.o* support module also needs to be compiled, which is achieved by simply appending it to the statement.

Other network drivers will, if selected, also add *mii.o* to the list of objects to be built—this is fine, the build system handles this case. It is even possible that a support module like *mii.o* got added to the list of built-in objects `obj-y` and `obj-m`—again, the build system recognizes this fact and just compiles the built-in version, which will also be usable for the drivers compiled modular.

The new *e100* driver examplifies two more features. *drivers/net/Makefile* only contains the line

```
obj-$(CONFIG_E100) += e100/
```

which tells kbuild that it should descend into the *e100/* subdirectory if the option `CONFIG_E100` is set. What to do there will then be determined by *drivers/net/e100/Makefile* (Figure 2):

The first line after the comment looks familiar, it advises the build system to build *e100.o* built-in/modular depending on the value of `CONFIG_E100`. When `CONFIG_E100` equals “m” the e100 driver is built as a module and will be named em e100.ko.

The next line then states that *e100.o* is a composite object which should be linked from the listed individual object files—these object files will automatically compiled with the appropriate flags.

As a last point, instead of using the variable `<modname>-objs` to declare the components of the module *<modname>.o*, the variant `<modname>-y` can be used, which allows for easy definition of optional parts to a composite modules, as seen in the example in Figure 3.

4 What is new in Linux-2.5/2.6’s kbuild?

In this section, we describe some of the steps in the evolution of the kernel build system during the development phase of Linux 2.5. One purpose is to show how this evolution could actually be divided into small, Linus-compatible “piece-meal” patches without the famous “flag-day” patches and with only little breakage along the way.

We will also show how using the extensions provided by *GNU make* were actually exploited to provide a better build system while still using a standard tool instead of creating a specialized build solution for the kernel from scratch.

We start by comparing *drivers/isdn/Makefile* in 2.4 and 2.5 (Figure 4), where many of the improvements are easily seen. (a) shows the *Makefile* as it is present in Linux 2.4.20, and (b) shows the simpler variant present in 2.5. kbuild has been adapted incrementally to allow the more concise syntax. The following sections will describe the internals that eventually allowed for the layout seen in (b).

4.1 `O_TARGET` / linking objects in subdirectories

First of all, we start with a short description of what the kbuild interal implementation, which is hidden in the top-level *Makefile* and *scripts/Makefile.** typically does in a subdirectory: From the kbuild *Makefile* located in the subdirectory we obtain a list of what to build from the variables `obj-y` (built-in) and `obj-m` (modular) as explained in the previous section. The default target in *scripts/Makefile.build* is `__build` and the corresponding rule, shown in Figure 5, defines what work needs to be done. Important here is that we build `O_TARGET` or `L_TARGET`, respectively, when building *vmlinux* and `obj-m` when compiling modules. As opposed to 2.4, in 2.5 `O_TARGET` is a kbuild internal variable

```
#
# Makefile for the Intels E100 ethernet driver

obj-$(CONFIG_E100) += e100.o

e100-objs := e100_main.o e100_config.o e100_phy.o \
             e100_eeprom.o e100_test.o
```

Figure 2: `/drivers/net/e100/Makefile`

```
#
# Makefile for the Linux X.25 Packet layer.
#

obj-$(CONFIG_X25) += x25.o

x25-y          := af_x25.o x25_dev.o x25_facilities.o x25_in.o \
               x25_link.o x25_out.o x25_route.o x25_subr.o \
               x25_timer.o x25_proc.o
x25-$(CONFIG_SYSCTL)    += sysctl_net_x25.o
```

Figure 3: `net/x25/Makefile`

and needs no longer be defined in the kbuild makefiles. Except for the rare case of building an actual library, `O_TARGET` is used in the built-in case and we find the rule how to make it as

```
$(O_TARGET): $(obj-y) FORCE
     $(call if_changed,link_o_target)
```

So `O_TARGET` is linked from the objects listed in `obj-y`, which contains files locally compiled in the current directory as well as objects which are built in subdirectories by descending. In Figure 6, we see how going from the leaves to the root, the `O_TARGET` in each subdirectory (here always called *built-in.o*) accumulates the objects built below that directory until we finally end up with *vmlinux* at the root of the hierarchy containing all built-in objects generated throughout the tree (this example only shows a small fraction of the objects linked in a normal build).

```
vmlinux
|-- drivers/built-in.o
|   `-- isdn/built-in.o
|       |-- isdn.o
|       |   |-- isdn_common.o
|       |   `-- isdn_net.o
|       |
|       |-- hisax/built-in.o
|       |   |-- hisax.o
|       |   |   |-- config.o
|       |   |   `-- isdnl*.o
|       |   `-- hisax_fcpcipnp.o
|       |
|       `-- icn/built-in.o
|           `-- icn.o
|
`-- fs/built-in.o
```

Figure 6: The hierarchy for linking *vmlinux*

(a)

```
O_TARGET        := vmlinux-obj.o
export-objs     := isdn_common.o

list-multi      := isdn.o
isdn-objs       := isdn_net.o isdn_tty.o isdn_v110.o isdn_common.o
isdn-objs-$(CONFIG_ISDN_PPP)           += isdn_ppp.o
isdn-objs                              += $(isdn-objs-y)

obj-$(CONFIG_ISDN)                     += isdn.o
obj-$(CONFIG_ISDN_PPP_BSDCOMP)         += isdn_bsdcomp.o

mod-subdirs                            := hisax
subdir-$(CONFIG_ISDN_HISAX)            += hisax
subdir-$(CONFIG_ISDN_DRV_ICN)          += icn

obj-y += $(addsuffix /vmlinux-obj.o, $(subdir-y))

include $(TOPDIR)/Rules.make

isdn.o: $(isdn-objs)
        $(LD) -r -o $@ $(isdn-objs)
```

(b)

```
obj-$(CONFIG_ISDN)              += isdn.o
obj-$(CONFIG_ISDN_PPP_BSDCOMP)  += isdn_bsdcomp.o

isdn-y                          := isdn_net_lib.o isdn_fsm.o isdn_tty.o \
                                   isdn_v110.o isdn_common.o
isdn-$(CONFIG_ISDN_PPP)         += isdn_ppp.o

obj-$(CONFIG_ISDN_DRV_HISAX)    += hisax/
obj-$(CONFIG_ISDN_DRV_ICN)      += icn/
```

Figure 4: *drivers/isdn/Makefile* in (a) 2.4.20 and (b) adapted for the build system in 2.5

In Linux 2.4, the name for the object which accumulates all built-in objects in and below the current subdirectory was chosen by setting the variable `O_TARGET` in the local Makefile. In Linux-2.5, it is instead just set to *built-in.o* by the build system. This allows to get rid of the assignment of `O_TARGET` in every subdir Makefile and, more importantly, allows for further clean-up:

In kbuild-2.4, we need to explicitly add the subdirectories to descend into to the variables `subdir-y/m`, and then also add the subdir-generated built-in objects to `obj-y` so that they get linked. This is redundant and error-prone, in 2.5 it is sufficient to just add the objects generated in the subdirectories to the list of objects to be linked, and the build system will deduce from there that it needs to descend into the named subdirectories. To simplify things further, the name of the `O_TARGET` (now always being *built-in.o*) itself is left out and only the trailing slash is kept:

```
__build: $(if $(KBUILD_BUILTIN),$(O_TARGET) $(L_TARGET) $(extra-y)) \
         $(if $(KBUILD_MODULES),$(obj-m)) \
         $(subdir-ym) $(always)
        @:
```

Figure 5: The default `__build` rule from *scripts/Makefile.build*

```
obj-$(CONFIG_...HISAX) += hisax/
obj-$(CONFIG_...ICN)   += icn/
```

4.2 Multi-part modules

As can be seen from Figure 4, a number of statements were necessary for generating a multi-part module, *isdn.o* in that example. First of all, the parts constituting the module need to be declared by assigning them to the variable `isdn-objs`. This step is of course essential and was kept in 2.5.

However, it was also necessary to declare that *isdn.o* is a multi-part module by listing it in the variable `list-multi`. This information is redundant as it can be deduced by checking for the existence of `<module>-objs`, which is now done in 2.5.

Furthermore, in 2.4 a link rule has to be explicitly given for each multi-part object, which was annoying and error-prone. In the new build system, this link rule is generated by *make*, hitting just about the limits of what *GNU make* is capable of. We use a feature called "static pattern rules," and the code looks like the following:

```
cmd_link_multi-m = $(LD) ... \
  -o $@ $(link_multi_deps)

$(multi-used-m) : \
%.o: $(multi-objs-m) FORCE
  $(call if_changed,link_multi-m)
```

`multi-used-m` contains all multi-part modules we want to be built in the current directory and `multi-objs-m` contains all of the individual objects those are built of. This makes each multi-part module in the directory depend on the set of all components for all multi-part modules in that directory, which is actually too large, as it of course only is dependend its own components; however the latter is not implementable within the restrictions of *GNU make*. When doing the link, the variable `link_multi_deps` recovers the right list of components from the target `$@`, so that linker is invoked correctly.

Another interesting detail is that we here as well as in other places need to uniquify the prerequisites, so that listing a component multiple times doesn't lead to a link error. *GNU make* offers the `sort` function, which throws away duplicates, however it is unfortunately not usable for this purpose since it sorts, i.e. reorders its arguments and thus changes the link/init order. The workaround here is to use the variable `$^` which actually uniquifies the list of prerequisites exactly as needed. Finally, since `$^` lists all prerequisites which as mentioned above exceeds the list of components for the current module, we filter the uniquified list with that list of components to get the information we need.

4.3 Including `Rules.make`

Each subdirectory Makefile in kbuild-2.4 needed to include *$(TOPDIR)/Rules.make* explicitly. In 2.5, when descending into subdirectories, the build system now always calls *make* with the same Makefile, *scripts/Makefile.build*, which then again includes the local subdirectory *Makefile*, so the statement to include *Rules.make* could be dropped.

Furthermore, in 2.5. the build is still organized in a recursive way, i.e. *make* is only invoked to build the objects in the local subdirectory and other instances of *make* are spawned for the underlying directories. However, it does not actually descend into the subdirectories, it always does its work from the top-level directory and prepends the path as necessary. One of the advantages is that the output includes the correct paths, so a compiler warning will not show "*inode.c*: Warning ...", but "*fs/ext2/inode.c*: ...", which makes it easier to recognize where the problem occurs. More importantly, it allows to use relative paths throughout the build, so that paths like "BUG in `/home/kai/src/kernel/v2.5/linux-2.5.isdn/include/linux/fs.h`" are history. Renaming/moving a kernel tree will not cause spurious rebuilds due to changing paths as seen above anymore, and tools like "ccache" can work more effectively.

4.4 Objects exporting symbols

The old module symbol versioning scheme used with Linux 2.4 needed the Makefiles to declare which objects export symbols to modules, which was done by listing them in the variable `export-objs`. In 2.5, module versioning was completely redesigned, removing the need for this explicit declaration. The changes are so complex that they are rewarded their own section in this paper.

Here we conclude the comparison between a 2.4 and 2.5 subdirectory Makefile, where we have shown that all the redundant and deducible information has been removed and the necessary information is revealed to the build system in a very compact form.

Two additional important internal changes, which did not affect the subdirectory Makefile layout will be described in the following:

4.5 Compiling built-in objects and modules in a single pass, recognizing changed command line arguments

The major performance issue for the kernel build are the invocations of *make* (most of the time is of course normally spent compiling / linking, but this cost is independent of the build system used). *make* has to read the local Makefile, the general rules and all the dependencies and figure out the work to be done from there. An obvious way to optimize the performance of the build system is thus to avoid unnecessary invocations. In 2.4, *make* needs to do separate passes for modules and built-in objects and within each directory, it will even call itself again, so an about four-times performance increase is possible by just combining those invocations into a single pass.

The primary reason why kbuild-2.4 needs two passes for built-in and modules lies in its flags handling. This means that it tries to check not only whether prerequisites have changed (e.g. the C source for an object), but also if the compiler flags have changed.

This objective was achieved by generating a *.<target>.flags* file like the following (simplified) for each target built:

```
ifeq (-D__KERNEL__ -DMODULE
            -DEXPORT_SYMTAB,
            $(CFLAGS) -DEXPORT_SYMTAB)))
    FILES_FLAGS_UP_TO_DATE += config.o
endif
```

On a rebuild, the Makefile would read all those *.*.flags* fragments and forces all files which are not listed in `FILES_FLAGS_UP_TO_DATE` to be rebuild.

The flaw of this method is that it cannot handle differing flags for different groups of files, so *make* needs to invoked twice, once for the targets to be built-in with the normal `CFLAGS`, and again for the modular targets with `-DMODULE` added to CFLAGS. In the example above it is also visible that the handling for -DEXPORT_SYMTAB is broken, this method can not detect when a file was added / removed from the list of files exporting symbols, since the -DEXPORT_SYMTAB was hardcoded on both sides of the comparison and thus useless—the only way to fix this within in the old framework would have been to invoke *make* four times, for all combinations of built-in/module and export/no-export.

A more flexible scheme to handle changing command lines within *GNU make* was created:

As an example, we present the rule which is responsible for linking built-in objects into a library in Figure 7. The actual use is pretty simple, instead of writing the command directly into the command part of the rule, it is instead assigned to the variable cmd_link_l_target and the build system takes care of executing the command as necessary, keeping track of changes to the command line itself.

The implementation works as follows: After executing the command, the macro if_changed, records the command line into the file .*<target>.cmd*. As *make* is invoked again during a rebuild, it will include those .*.cmd files. As it tries to decide whether to rebuild L_TARGET, it will find FORCE in the prerequisites, which actually forces it to always rerun the command part of the rule.

However, the command part of the rule now does the actual work: It checks whether any of the prerequisites changed, i.e. $? is non-empty or if the command line changed, which is achieved by the two filter-out statements. Only if either of those two conditions is met, if_changed expands to a command rebuilding the target, otherwise it is empty and the target will not be rebuilt.

The advantage of this method, apart from the easier use in a rule as shown above, is that all the checking is done within the context of the actual rule and not in a unrelated place later in the Makefile. This allows for the use and correct checking of *GNU make*'s per target variables, e.g.

```
modkern_cflags := $(CFLAGS_KERNEL)
$(real-objs-m) : \
  modkern_cflags := $(CFLAGS_MODULE)
```

which sets modkern_cflags to $(CFLAGS_KERNEL) by default, but to $(CFLAGS_MODULE) for objects listed in $(real-objs-m), i.e. for objects compiled as modules. The compilation rule can then just use $(modkern_cflags) to get the right flags for the current object, where the mechanism described above will take care of recognizing changes and acting accordingly.

4.6 Dependencies

Between configuration and building of a kernel, the old kernel build needed the user to run "make dep", which served to purposes: It generated dependency information for the C source files on headers and other included files, and it generated the version checksums for exported symbols.

Both of these task have become unnecessary in 2.5, so the reliance on the user to rerun "make dep" as needed is gone (additionally, the system in 2.4 is broken that in some modversions cases it's not even sufficient to rerun "make dep", the only solution then is to do "make distclean" and start over).

2.4 used a small tool called *mkdep* to generate dependencies for C sources. This tools basically extracted the names of the included files out of the source, but did not actually recursively scan those includes then. So, if *foo.c* includes *foo.h*, which itself includes *bar.h*, *mkdep* would only pick up the dependency of *foo.c* on *foo.h*, but *foo.c* also needs recompiling when

```
cmd_link_l_target = rm -f $@; $(AR) $(EXTRA_ARFLAGS) rcs $@ $(obj-y)

$(L_TARGET): $(obj-y) FORCE
        $(call if_changed,link_l_target)

targets += $(L_TARGET)

[...]

if_changed = $(if $(strip $? \
                          $(filter-out $(cmd_$(1)),$(cmd_$@))\
                          $(filter-out $(cmd_$@),$(cmd_$(1)))),\
        @set -e; \
        $(cmd_$(1)); \
        echo 'cmd_$@ := $(cmd_$(1))' > $(@D)/.$(@F).cmd)
```

Figure 7: Checking for a changed command line

foo.h changes. This problem was solved in 2.4 by assuming that *foo.h* would reside in *include/** (which is mostly, but not always, true). For those files it would generate another set of dependencies, basically:

```
foo.h: bar.h
        @touch $@
```

So as *bar.h* changes, this rule will update the timestamp on *foo.h*, which will then be seen by the rule for *foo.c* and cause *foo.c* to be rebuild.

This method has several disadvantages:

- Changing the timestamp on files which have not actually been modified confuses a number of source management systems.
- It only works for header files in the *include/** subdirectories.
- As *foo.h* is changed to also include *baz.h*, the dependency information does not get updated, so a subsequent change to *baz.h* will erroneously not cause *foo.c* to be recompiled.
- Starting from a clean tree, the user has to wait for the dependency information to be created (for all files, even for entire subsystem which may not be selected in the configuration at all), even though this information is totally useless for a first build—it's only useful for deciding whether a file needs to be rebuilt.

The build system in Linux 2.5 instead uses gcc's -MD flag to generate the dependency information during the build. This flag generates the full list of all files included during the compile, so in the example above it would generate "*foo.o*: *foo.c foo.h bar.h*" (and "*baz.h*" as that gets added). This procedure is much simpler, and it gets around all the disadvantages listed above.

The only quirk which is applied similarly in 2.4 and 2.5 is related to the high configurability of the linux kernel.

Using the gcc generated list of dependencies as-is has the drawback that virtually every file in the kernel includes <*linux/config.h*> which then again includes <*linux/autoconf.h*>

If a user reruns `make *config` to change a configuration option, *linux/autoconf.h* will be regenerated. *make* will notice this and rebuild every file which includes autconf.h, i.e. basically all files. This is correct, but extremely annoying if the user just changed some option `CONFIG_THIS_DRIVER` from n to m.

So we use the same trick that "*mkdep*" applied before. We replace the dependency on *linux/autoconf.h* by a dependency on every config option which is mentioned in any of the listed prerexquisites.

The effect is that if a user changes the `CONFIG_THIS_DRIVER` option, only the objects which (themselves, or in any of the included files) reference `CONFIG_THIS_DRIVER` will be rebuilt, which most likely is only this one driver.

5 Modules and the kernel build process

The implementation of loadable kernel modules has been substantially rewritten by Rusty Russell in the development cycle 2.5. These changes are so complex that this paper will not attempt to describe them in detail. Instead, we concentrate on the changes which were done in the build system to accomodate the new concepts.

5.1 Module symbol versions

Loadable modules need to interface with the kernel. They do this by accessing certain data structures and functions which have been marked as exported symbols in the source. That means not all global symbols in the kernel are accessible to modules, but only an explicitly exported API.

These symbols remain unresolved in the loadable module objects at build time and are then resolved at load time, either by an external program, *modutils*, in 2.4, or by an in-kernel loader in 2.5. A common problem is that Linux does not guarantee a stable binary interface to modules, in fact the binary interface often changes between releases in a stable kernel series and even depending on the configuration of the kernel. One simple example is the `struct net_device`, which embeds a `spinlock_t`. If the kernel is configured for uni-processor operation, this lock expands to nothing, so the layout of the `struct net_device changes`. When calling `register_netdev(struct net_device *)` where the in-kernel function `register_netdev()` assumes the SMP layout, though the module set up the argument in the UP layout, we have an obvious mismatch which often leads to hard to explain kernel crashes.

Other operating systems solve this problem by prescribing a stable ABI between kernel and modules, however in Linux it is preferred to not carry around binary compatibility layers and cope with unflexible interfaces, instead since the source is openly accessible, one just needs to recompile the modules so that they match the kernel.

Now, it is easily possible for users to get this wrong and we thus want a way to detect version mismatches and refuse to load the modules or at least warn. This is what "module symbol versioning" accomplishes. The basic idea is to analyze the exported symbols, including the types of the arguments for function calls and generate a checksum for a specific layout. If anything changes in the ABI, the versioning process will generate a different checksum and thus detect the mismatch. The main work in this scheme is done by the program *genksyms*, which is basically a C parser that reads a preprocessed source file and finds the definitions for the exported symbols from there.

This procedure has caused trouble in the build system for a long time. In Linux 2.4, the "`make dep`" stage, apart from building dependency information, preprocesses all source files which export symbols (that is why they need to be specifically declared in the Makefiles) and then generates *include/linux/modversions.h* which mangles the exported symbols with the generated checksum, using the C preprocessor. The kernel will then not export the symbol `register_netdev`, but instead `register_netdev_R43d2381`. A module referencing `register_netdev` will end up with an unresolved symbol `register_netdev_R43d2381`, so loading it into the kernel will work fine. Has the module however built against a different kernel or a different configuration, the checksum has changed and any attempt to load it will result in an error about unresolved symbols.

This implementation was rather fragile, as it relies on the user to rerun "`make dep`" whenever the version information has possibly changed, and even if only one symbol changed, that basically forces a recompilation of every file. In addition, some of the optimizations made in 2.4's build system were actually broken, leading to the well-known fact that it can get into a state where not even running "`make dep`" will recover from generating inconsistent version information, and starting over from "`make mrproper/distclean`" is needed.

Module versioning is still a challenge to the build system in 2.5, the underlying reason for that is that it introduces cross-directory dependencies, which a recursive build system cannot easily handle. For example, the ISDN module *drivers/isdn/hisax/hisax.ko* uses `register_isdn()`, which is exported by *drivers/isdn/isdn_common.o*. So building *hisax.ko* needs knowledge of the checksum generated from *drivers/isdn/isdn_common.o*, but it has no way to make sure that it is up-to-date since it is located in a different subdirectory.

Module versioning is instead implemented as a two stage process, the first stage is the normal build, which also generates all the checksums. After this stage is completed, we can be sure that all checksums are up-to-date now, and then just record this up-to-date information into the modules. This is one of the reasons why modules have been renamed with a ".ko" extension: The first stage just builds the normal ".o" objects, and afterwards a postprocessing step follows, which builds ".ko" modules adding version checksums for unresolved symbols and other information.

In more detail, the following steps are executed:

- **Compiling**

  Knowledge of which source files export symbols is not required up front. As an `EXPORT_SYMBOL(foo)` is encountered, the definition of `EXPORT_SYMBOL` from *include/linux/module.h* will generate special sections with tables containg the name of the symbol, its address and its checksum. Actually, since the checksum is not known at this time, the value of the checksum is set to a symbol called `__crc_foo`. This is a trick which allows to use the linker to record the checksum even after the object file is already compiled.

  As the object file has been generated, we check it for the existance of the special section mentioned above. If it exists, the source file did export symbols and *genksyms* is run to obtain the checksums for those symbols. Finally, these checksums are entered into the object using the linker in conjunction with a small linker script.

```
$ nm drivers/isdn/i4l/isdn.ko | grep __crc
86849dd0 A __crc_isdn_ppp_register_compressor
843d2381 A __crc_isdn_ppp_unregister_compressor
66d136e2 A __crc_register_isdn
```

Figure 8: Examining the checksums for exported symbols

The checksums can easily examined at running the command shown in Figure 8.

- **Postprocessing**

  After stage one, we have the checksums for the exported symbols embedded within `vmlinux` and the modules. What is yet to be done is recording the checksums into the consumers, that is adding the checksums for unresolved symbols into the modules.

  This step was initially handled by a small shell script but is now done by a C program for performance reasons, which also handles other postprocessing needs like generating aliases.

  This program basically reads all the exported symbols and their checksums from all modules, and then scans the modules for unresolved symbols. For each unresolved symbol, an entry in a table associating the symbol string with the checksum is made, this table is output as C source *module.mod.c* and compiled and linked into the final `.ko` module object.

  Figure 9 shows an excerpt from *drivers/isdn/hisax/hisax.mod.c* which calls `register_isdn()`. The checksum obviously matches the checksum for the exported symbol in *drivers/isdn/i4l/isdn.ko*, so that the module will load without complaint.

An additional advantage of the new way of handling module version symbols, apart from being cleaner from a build system point of view, is that the actual symbols are not mangled, so it became possible to force a module load even if the checksums do not match—though the kernel will set the taint flag in these case.

The module postprocessing step, introduced mainly for the module symbol versioning, allowed for a number of additional features, i.e. module aliases / device table handling, additional version checks as well as recognition of unresolved symbols during the build stage.

6 Conclusion and Outlook

This paper presented an introduction to using the kernel build system for the Linux kernel 2.5 and 2.6 for users who want to compile their own kernels and developers working on kernel code. We also showed how in the transition from kbuild-2.4 to 2.5, features of *GNU make* could be applied to remove redundant information and allow for simpler Makefile fragments as well as a more consistent and foolproof build system.

Additionally, parts of the internal implementation have been described and an overview over changes related to the new module loader and new module versioning system has been given.

The kernel build system in 2.5 has been improved significantly, but some features remain to be implemented.

```
  static const struct modversion_info ____versions[]
__attribute__((section("__versions"))) = {
        { 0xfa7bbba7, "struct_module" },
        { 0x66d136e2, "register_isdn" },
        { 0x1a1a4f09, "__request_region" },
  [...]
```

Figure 9: Excerpt from *drivers/isdn/hisax/hisax.mod.c*, generated by the postprocessing stage

Separate source and object directories

As opposed to kernel 2.4, source files are not altered or touched during the build in 2.5 anymore, enhancing interoperability with source management systems. The next step is to allow for completely separate source and object directory trees, so that the source can be completely read-only and multiple builds at the same time from the same source are possible. The current code in 2.5 has taken preparatory steps for this feature but work is not completed yet.

Non-recursive build

It is an open question whether it is actually advisable to switch to a non-recursive build system. Obviously, distributing build information with the source files is desirable, this trend is visible in e.g. the split of the global Configure.help file into per-directory fragments which eventually were unified with the new *Kconfig* configuration info. Of course it is essential to keep the build information in the per-subdirectory Makefiles distributed as it is currently, it would be a step back to collapse it into one big file.

However this does not preclude collecting the distributed information when starting a build and generating a global Makefile, which is then used as a main stage. The advantage of this method is that it can handle cross-directory dependencies more easily, whereas the current system has to resort to a two-stage process for module post-processing. On the other hand, a global Makefile which contains also needs to incorporate dependencies for all files will use a significant amount of memory and may turn out to be problematic on low–end systems.

There are two ways to implement a global Makefile: One possibility is using *GNU make* itself, replacing the rules to actually compile / link objects by dummy routines recording the necessary actions into a global Makefile. The second possibility is, as the subdir Makefiles have a very consistent form by now, to write a specialized parser for those files and have that generate a global Makefile.

Whether switching to a non-recursive build system is worth the tradeoffs will be investigated in the Linux 2.7 development cycle.

References

[1] *GNU make* `http://www.gnu.org/software/make/make.html`

[2] Michael Elizabeth Castain: *dancing-makefiles* `http://www.kernel.org/pub/linux/kernel/projects/kbuild/dancing-makefiles-2.4.0-test10.gz`

Proceedings of the Linux Symposium

July 23th–26th, 2003
Ottawa, Ontario
Canada