Linux kernel exploration: the secrets of Kconfig/Kbuild

Linux kernel config/build system, as known as kconfig/kbuild, have existed for quite a long time. As config/build infrastructure of linux kernel, it exists since linux kernel code is migrated to git; from searchable history of internet, they have been there since 2001 at least. But, as a supporting infrastructure, it is seldom under spotlight, people tend to focus on the cutting-edge techniques.

Kernel developers use these two utilities in their daily work without any perception, probably they seldom get interested in it. Driven by the cause that I want to know exactly how a PC-based linux OS is booted from power on, I think it is necessary to understand how linux kernel is compiled.

This article will focus on the internal process of kconfig/kbuild, show you: how .config file is produced, and how vmlinux/bzImage file is produced; also introduce a smart trick used for dependency tracking.

Kconfig

The first step of building a kernel always is configuration. With the help of kconfig, linux kernel is highly modular and customizable. Kconfig provide many config targets for user, shown as following:

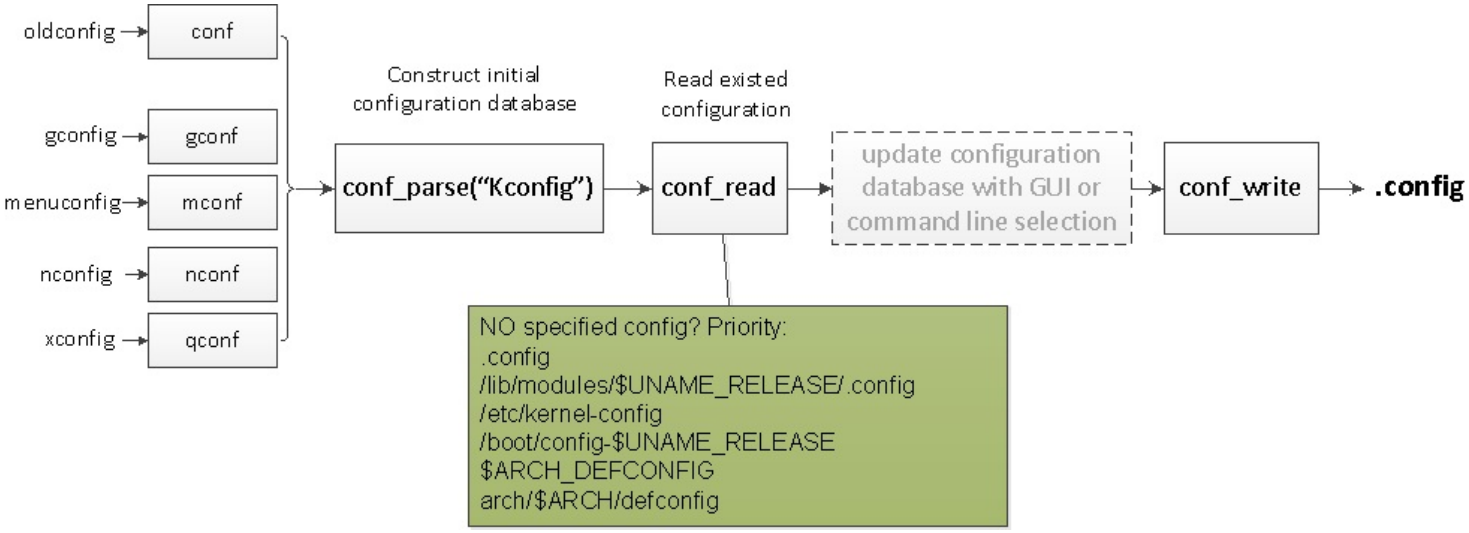
```
config          - Update current config utilising a line-oriented program
nconfig         - Update current config utilising a ncurses menu based program
menuconfig      - Update current config utilising a menu based program
xconfig         - Update current config utilising a Qt based front-end
gconfig         - Update current config utilising a GTK+ based front-end
oldconfig       - Update current config utilising a provided .config as base
localmodconfig  - Update current config disabling modules not loaded
localyesconfig  - Update current config converting local mods to core
defconfig       - New config with default from ARCH supplied defconfig
savedefconfig   - Save current config as ./defconfig (minimal config)
allnoconfig     - New config where all options are answered with no
allyesconfig    - New config where all options are accepted with yes
allmodconfig    - New config selecting modules when possible
alldefconfig    - New config with all symbols set to default
randconfig      - New config with random answer to all options
listnewconfig   - List new options
olddefconfig    - Same as oldconfig but sets new symbols to their default value without prompting
kvmconfig       - Enable additional options for kvm guest kernel support
xenconfig       - Enable additional options for xen dom0 and guest kernel support
tinyconfig      - Configure the tiniest possible kernel
```

Among these targets, I bet "menuconfig" is the most popular one.

These targets are processed by different host programs which are provided by kernel itself, and built during kernel building. Some of targets have GUI for user's convenience, while most don't. The kconfig related tools & source code mostly reside under scripts/kconfig/ of kernel source, from scripts/kconfig/Makefile, we can see there are several host programs, like conf, mconf, nconf, etc. Except `conf`, each of them is responsible for one of the GUI-based config targets, so, `conf` actually deals with most of them.

Logically, kconfig infrastructure has two parts, first one invents a new language used to defines the configuration items(See the `Kconfig` files under kernel source); second one is the tool used to parse the kconfig language, deal with all the configuration actions.

For most of the config targets, they have roughly the same internal process, here is the illustration:



A tip first: all configuration items have default value.

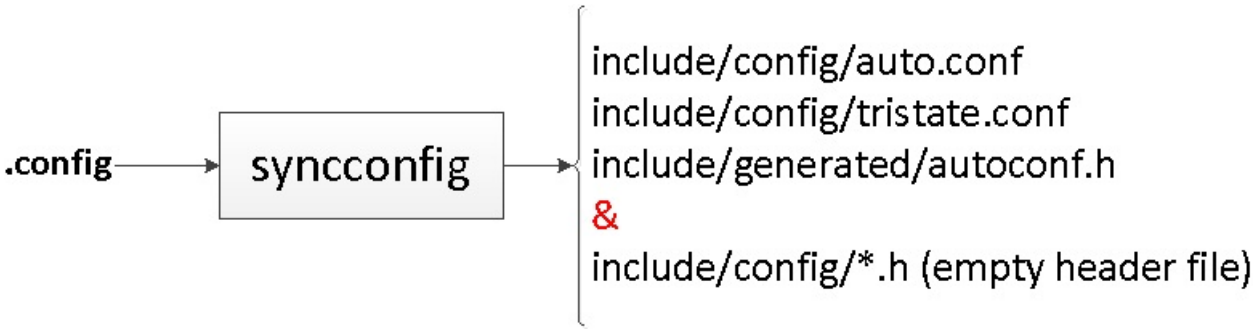
The first step is reading the `Kconfig` file under source root, to construct an initial configuration database; then it will read an existed configuration file according to the priority:

> .config
> /lib/modules/$(shell,uname -r)/.config
> /etc/kernel-config
> /boot/config-$(shell,uname -r)
> ARCH_DEFCONFIG
> arch/$(ARCH)/defconfig

to update the initial database; if you are doing GUI-based configuration via menuconfig, or command line based configuration via oldconfig, the database will be updated again according to your customization; finally, dump the configuration database into the `.config` file.

But, `.config` file is not the final fodder for kernel building, that is why `syncconfig` target exists. `syncconfig` was named `silentoldconfig`, and listed in the config targets. It is for internal use, not users, and it doesn't do what the old name tells, so it is renamed and dropped from the list.

Here is the illustration of what syncconfig does:



syncconfig takes .config as input, outputs many other files which can be put into 3 categories:

i. auto.conf & tristate.conf

   Used for Makefile text processing. For example, you may see many statements like:

   ```
   obj-$(CONFIG_GENERIC_CALIBRATE_DELAY) += calibrate.o
   ```

   in component's makefile.

ii. autoconf.h

   Apparently it is used in C language source file.

iii. empty header files under include/config/

   used for configuration dependency tracking during kbuild, which is another topic will be explained later.

After configuration, we will know which file and code piece are not compiled.

kbuild

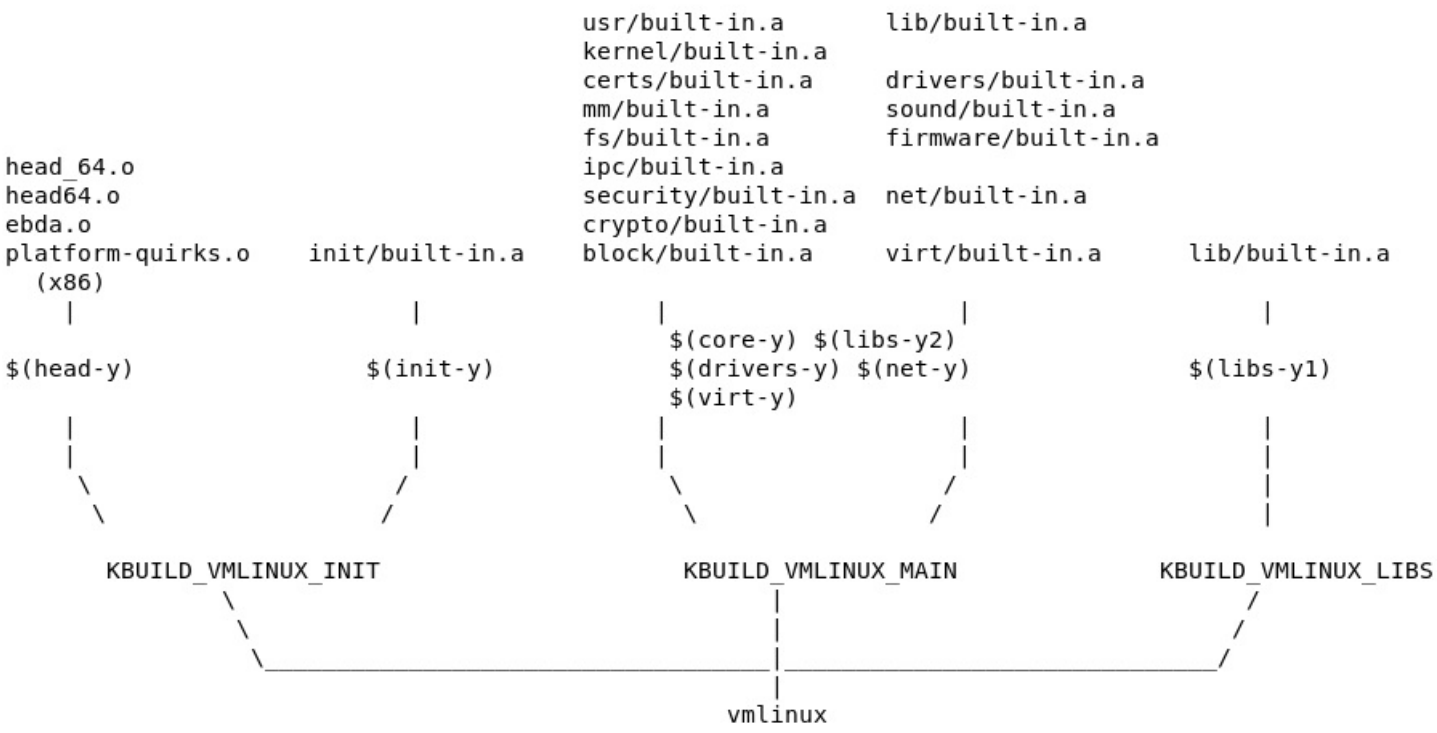
Large software projects management are usually simplified by dividing them into major components, these components often dwell in their own directories and managed by their own makefiles. One way to build an entire system of components employs a top-level makefile that invokes the makefile for each component in the proper order. This approach is called recursive make, because the top-level makefile invokes make recursively on each component's makefile. Recursive make is a common technique for handling component-wise builds. Kbuild is an excellent practice of recursive-make.

When talking about kbuild, we actually refer to different kinds of makefiles:

- Makefile: the top Makefile locates in source root.
- .config: the kernel configuration file.
- arch/$(ARCH)/Makefile: the arch Makefile, supplement to top makefile.
- scripts/Makefile.*: common rules etc. for all kbuild Makefiles.
- kbuild Makefiles: there are about 500 of these.

Top makefile includes arch makefile, reads the .config file, then descends into sub-directories, invokes `make` on each component's makefile, with the help of routines defined in scripts/Makefile.*, builds up each intermediate object, links all the intermediate objects into vmlinux. Kernel document Documentation/kbuild/makefiles.txt gives a comprehensive description of all aspects of these makefiles.

Take x86-64 for example, let's see how vmlinux is produced:



(The illustration is an update base on a blog, thanks to the author for letting me use it freely)

All the .o files that finally go into `vmlinux` first go into its own built-in.a, which is indicated via variable `KBUILD_VMLINUX_INIT`, `KBUILD_VMLINUX_MAIN`, `KBUILD_VMLINUX_LIBS`, then they are collected into the `vmlinux` file.

Now take a look at how recursive-make is implemented in linux kernel, with help of simplified makefile code:

```
# In top Makefile
vmlinux: scripts/link-vmlinux.sh $(vmlinux-deps)
    +$(call if_changed,link-vmlinux)

# Variable assignments
vmlinux-deps := $(KBUILD_LDS) $(KBUILD_VMLINUX_INIT) $(KBUILD_VMLINUX_MAIN) $(KBUILD_VMLINUX_LIBS)

export KBUILD_VMLINUX_INIT := $(head-y) $(init-y)
export KBUILD_VMLINUX_MAIN := $(core-y) $(libs-y2) $(drivers-y) $(net-y) $(virt-y)
export KBUILD_VMLINUX_LIBS := $(libs-y1)
```

```
export KBUILD_LDS          := arch/$(SRCARCH)/kernel/vmlinux.lds

init-y          := init/
drivers-y       := drivers/ sound/ firmware/
net-y           := net/
libs-y          := lib/
core-y          := usr/
virt-y          := virt/

# Transform to corresponding built-in.a
init-y          := $(patsubst %/, %/built-in.a, $(init-y))
core-y          := $(patsubst %/, %/built-in.a, $(core-y))
drivers-y       := $(patsubst %/, %/built-in.a, $(drivers-y))
net-y           := $(patsubst %/, %/built-in.a, $(net-y))
libs-y1         := $(patsubst %/, %/lib.a, $(libs-y))
libs-y2         := $(patsubst %/, %/built-in.a, $(filter-out %.a, $(libs-y)))
virt-y          := $(patsubst %/, %/built-in.a, $(virt-y))

# Setup the dependency. vmlinux-deps are all intermediate objects, vmlinux-dirs
# are phony targets, so every time comes to this rule, the recipe of vmlinux-dirs
# will be executed. Refer "4.6 Phony Targets" of `info make`
$(sort $(vmlinux-deps)): $(vmlinux-dirs) ;

# Variable vmlinux-dirs is the directory part of each built-in.a
vmlinux-dirs    := $(patsubst %/,%,$(filter %/, $(init-y) $(init-m) \
                     $(core-y) $(core-m) $(drivers-y) $(drivers-m) \
                     $(net-y) $(net-m) $(libs-y) $(libs-m) $(virt-y)))

# The entry of recursive make
$(vmlinux-dirs):
    $(Q)$(MAKE) $(build)=$@ need-builtin=1
```

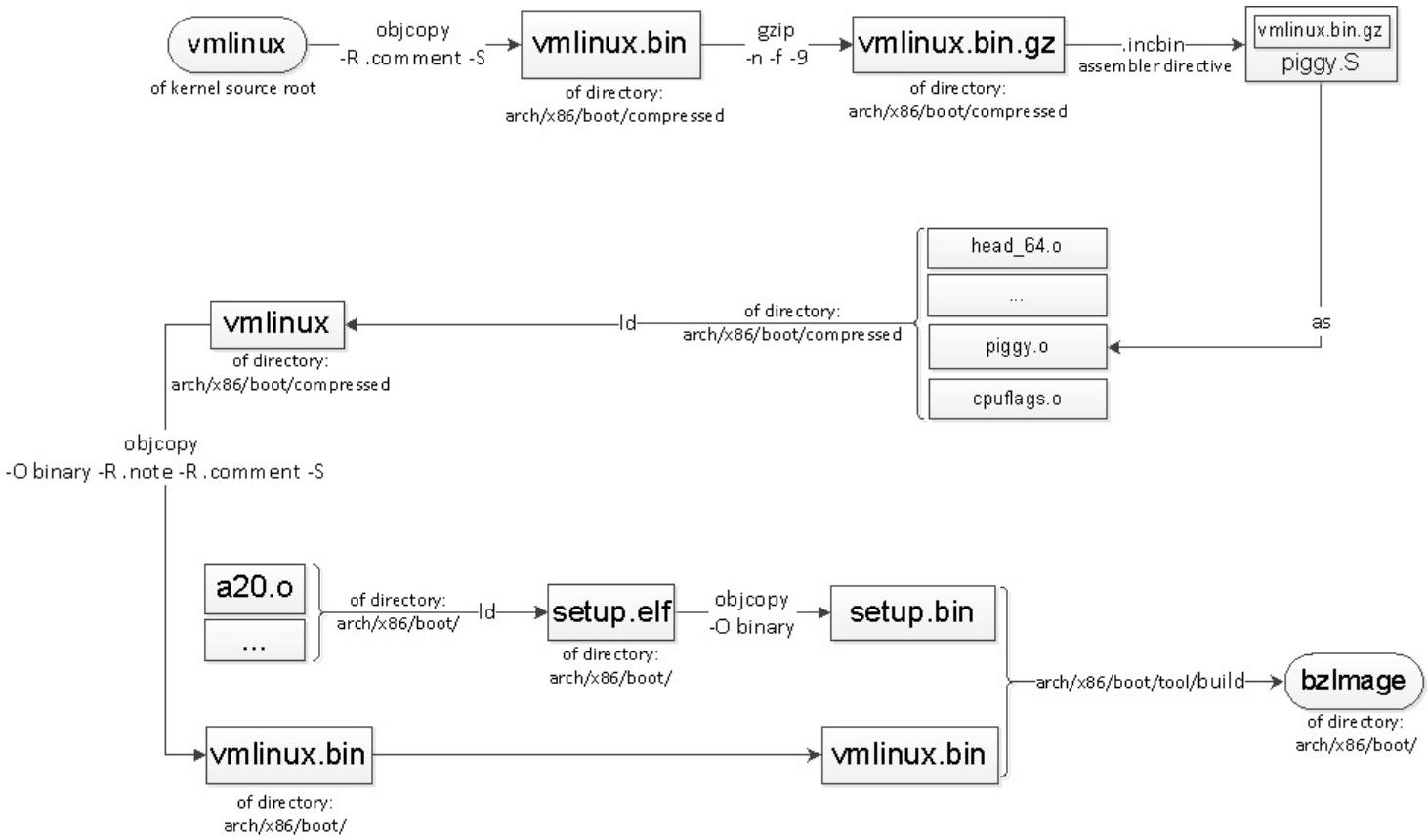
The recipe of recursive-make is expanded as, for example:

```
make -f scripts/Makefile.build obj=init need-builtin=1
```

which means make will go into scripts/Makefile.build to continue the work: build each built-in.a. With the help of scripts/link-vmlinux.sh, finally we got `vmlinux` file under source root.

vmlinux VS bzImage

Ordinary linux kernel developers may not be clear about the relationship of these two. Take x86-64 for example, here is the relationship:



`vmlinux` of source root is stripped, compressed, put into piggy.S, then linked with other peer objects into arch/x86/boot/compressed/vmlinux. Meanwhile, another file called setup.bin is produced under arch/x86/boot. There may be a optional third file which has relocation info, depends on the configuration `CONFIG_X86_NEED_RELOCS`.

A host program `build`, provided by kernel itself, builds these two(or three) parts into the final bzImage file.

dependency tracking

Kbuild tracks 3 kinds of dependency:

i. All prerequisite files (both *.c and* .h)
ii. CONFIG_ options used in all prerequisite files
iii. Command-line used to compile target

The first one is easy to understand, why the second and third? As a kernel developer, you should often see code piece like this:

```
#ifdef CONFIG_SMP
__boot_cpu_id = cpu;
#endif
```

So, when CONFIG_SMP changed, this piece of code should be recompiled. And the command line for compiling a source file also matters, different command line may result in different object file.

When a .c file use a header file via #include directive, you need write a rule like this:

```
main.o: defs.h
    recipe...
```

But, managing a large project, probably you need write a lot of this kind of rules, which would be tedious & boring. Fortunately, most modern C compilers can write these rules for you, by looking at the "#include" lines in the source file. For GCC, it is just a matter of adding a command line parameter: -MD depfile

```
# In scripts/Makefile.lib
c_flags        = -Wp,-MD,$(depfile) $(NOSTDINC_FLAGS) $(LINUXINCLUDE)     \
                 -include $(srctree)/include/linux/compiler_types.h       \
                 $(__c_flags) $(modkern_cflags)                           \
                 $(basename_flags) $(modname_flags)
```

which would generate a .d file has content like:

```
init_task.o: init/init_task.c include/linux/kconfig.h \
 include/generated/autoconf.h include/linux/init_task.h \
 include/linux/rcupdate.h include/linux/types.h \
 ...
```

Then host program fixdep takes care of the rest two dependency, by taking the `depfile` and command line as input, outputs a `.<target>.cmd` file in makefile syntax, which record the command line and all the prerequisites(include configuration) for a target, looks like this:

```
# The command line used to compile the target
cmd_init/init_task.o := gcc -Wp,-MD,init/.init_task.o.d  -nostdinc ...
...
# The dependency files
deps_init/init_task.o := \
    $(wildcard include/config/posix/timers.h) \
    $(wildcard include/config/arch/task/struct/on/stack.h) \
    $(wildcard include/config/thread/info/in/task.h) \
    ...
  include/uapi/linux/types.h \
  arch/x86/include/uapi/asm/types.h \
  include/uapi/asm-generic/types.h \
  ...
```

During recursive make, `.<target>.cmd` file will be included, providing all the dependency info, helping to decide whether to re-build a target or not.

The secret behind is, `fixdep` will parse the depfile(.d file), then parse all the dependency files inside, text searching all the "CONFIG_" strings, convert them to the corresponding empty header file, and add them to the prerequisites of the target. Everytime configuration changes, the corresponding empty header file will be updated too, so, kbuild can detect that change and re-build the target which depends on it. Because command line is also recorded, it is easy to compare last and current compiling parameters.

Epilogue

Kconfig/Kbuild have been steady for a long time, until the new maintainer Masahiro Yamada joined in at early 2017. He is a nice maintainer and quite productive, so kbuild now is under active development again, and you should not be surprised if you see something different from this article.