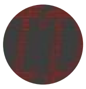

ILDER

// @Linux中国 :领走翻译去了——LCTT

@网路冷眼
【linux-insides: 深入理解Linux内核Makefile】 http://t.cn/RLoS1Um 本文不是告诉你如何构建和安装Linux内核的，而是详细解释Linux内核的构建过程的，由此掌握Linux内核Makefile的工作机制。 @Linux中国 @伯乐头条

Process of the Linux kernel building

Introduction

I will not tell you how to build and install custom Linux kernel on your machine, you can find many many resources that will help you to do it. Instead, we will know what does occur when you are typed `make` in the directory with Linux kernel source code in this part. When I just started to learn source code of the Linux kernel, the Makefile file was a first file that I've opened. And it was scary :) This makefile contains `1591` lines of code at the time when I wrote this part and it was third release candidate.

This makefile is the the top makefile in the Linux kernel source code and kernel build starts here. Yes, it is big, but moreover, if you've read the source code of the Linux kernel you can noted that all directories with a source code has an own makefile. Of course it is not real to describe how each source files compiled and linked. So, we will see compilation only for the standard case. You will not find here building of the kernel's documentation, cleaning of the kernel source code, tags generation, cross-compilation related stuff and etc. We will start from the `make` execution with the standard kernel configuration file and will finish with the building of the bzImage.

It would be good if you're already familiar with the make util, but I will anyway try to describe all code that will be in this part.

So let's start.

Preparation before the kernel compilation

There are many things to preparate before the kernel compilation will be started. The main point here is to find and configure the type of compilation, to parse command line arguments that are passed to the `make` util and etc. So let's dive into the top `Makefile` of the Linux kernel.

The Linux kernel top `Makefile` is responsible for building two major products: vmlinux (the resident kernel image) and the modules (any module files). The Makefile of the Linux kernel starts from the definition of the following variables:

```
VERSION = 4
PATCHLEVEL = 2
SUBLEVEL = 0
EXTRAVERSION = -rc3
NAME = Hurr durr I'ma sheep
```

These variables determine the current version of the Linux kernel and are used in the different places, for example in the forming of the `KERNELVERSION` variable:

```
KERNELVERSION = $(VERSION)$(if $(PATCHLEVEL),.$(PATCHLEVEL)$(if $(SUBLEVEL),.$(SUBLEVEL)))$(EXTRAVERSION)
```

After this we can see a couple of the `ifeq` condition that check some of the parameters passed to `make`. The Linux kernel `makefiles` provides a special `make help` target that prints all available targets and some of the command line arguments that can be passed to `make`. For example: `make V=1` - provides verbose builds. The first `ifeq` condition checks if the `V=` option is passed to make:

checks if the `V=n` option is passed to make:

```
ifeq ("$(origin V)", "command line")
  KBUILD_VERBOSE = $(V)
endif
ifndef KBUILD_VERBOSE
  KBUILD_VERBOSE = 0
endif

ifeq ($(KBUILD_VERBOSE),1)
  quiet =
  Q =
else
  quiet=quiet_
  Q = @
endif

export quiet Q KBUILD_VERBOSE
```

If this option is passed to `make` we set the `KBUILD_VERBOSE` variable to the value of the `V` option. Otherwise we set the `KBUILD_VERBOSE` variable to zero. After this we check value of the `KBUILD_VERBOSE` variable and set values of the `quiet` and `Q` variables depends on the `KBUILD_VERBOSE` value. The `@` symbols suppress the output of the command and if it will be set before a command we will see something like this: `CC scripts/mod/empty.o` instead of the `Compiling .... scripts/mod/empty.o`. In the end we just export all of these variables. The next `ifeq` statement checks that `O=/dir` option was passed to the `make`. This option allows to locate all output files in the given `dir`:

```
ifeq ($(KBUILD_SRC),)

ifeq ("$(origin O)", "command line")
  KBUILD_OUTPUT := $(O)
endif

ifneq ($(KBUILD_OUTPUT),)
saved-output := $(KBUILD_OUTPUT)
KBUILD_OUTPUT := $(shell mkdir -p $(KBUILD_OUTPUT) && cd $(KBUILD_OUTPUT) \
                                && /bin/pwd)
$(if $(KBUILD_OUTPUT),, \
     $(error failed to create output directory "$(saved-output)"))

sub-make: FORCE
    $(Q)$(MAKE) -C $(KBUILD_OUTPUT) KBUILD_SRC=$(CURDIR) \
    -f $(CURDIR)/Makefile $(filter-out _all sub-make,$(MAKECMDGOALS))

skip-makefile := 1
endif # ifneq ($(KBUILD_OUTPUT),)
endif # ifeq ($(KBUILD_SRC),)
```

We check the `KBUILD_SRC` that represent top directory of the source code of the linux kernel and if it is empty (it is empty every time while makefile executes first time) and the set the `KBUILD_OUTPUT` variable to the value that passed with the `O` option (if this option was passed). In the next step we check this `KBUILD_OUTPUT` variable and if we set it, we do following things:

- Store value of the `KBUILD_OUTPUT` in the temp `saved-output` variable;
- Try to create given output directory;
- Check that directory created, in other way print error;
- If custom output directory created sucessfully, execute `make` again with the new directory (see `-C` option).

The next `ifeq` statements checks that `C` or `M` options was passed to the make:

```
ifeq ("$(origin C)", "command line")
  KBUILD_CHECKSRC = $(C)
endif
ifndef KBUILD_CHECKSRC
  KBUILD_CHECKSRC = 0
endif

ifeq ("$(origin M)", "command line")
```

```
  KBUILD_EXTMOD := $(M)
endif
```

The first `C` option tells to the `makefile` that need to check all `c` source code with a tool provided by the `$CHECK` environment variable, by default it is sparse. The second `M` option provides build for the external modules (will not see this case in this part). As we set this variables we make a check of the `KBUILD_SRC` variable and if it is not set we set `srctree` variable to `.`:

```
ifeq ($(KBUILD_SRC),)
        srctree := .
endif

objtree := .
src     := $(srctree)
obj     := $(objtree)

export srctree objtree VPATH
```

That tells to `Makefile` that source tree of the Linux kernel will be in the current directory where `make` command was executed. After this we set `objtree` and other variables to this directory and export these variables. The next step is the getting value for the `SUBARCH` variable that will represent tewhat the underlying archicecture is:

```
SUBARCH := $(shell uname -m | sed -e s/i.86/x86/ -e s/x86_64/x86/ \
                  -e s/sun4u/sparc64/ \
                  -e s/arm.*/arm/ -e s/sa110/arm/ \
                  -e s/s390x/s390/ -e s/parisc64/parisc/ \
                  -e s/ppc.*/powerpc/ -e s/mips.*/mips/ \
                  -e s/sh[234].*/sh/ -e s/aarch64.*/arm64/ )
```

As you can see it executes uname utils that prints information about machine, operating system and architecture. As it will get output of the `uname` util, it will parse it and assign to the `SUBARCH` variable. As we got `SUBARCH`, we set the `SRCARCH` variable that provides directory of the certain architecture and `hfr-arch` that provides directory for the header files:

```
ifeq ($(ARCH),i386)
        SRCARCH := x86
endif
ifeq ($(ARCH),x86_64)
        SRCARCH := x86
endif

hdr-arch  := $(SRCARCH)
```

Note that `ARCH` is the alias for the `SUBARCH`. In the next step we set the `KCONFIG_CONFIG` variable that represents path to the kernel configuration file and if it was not set before, it will be `.config` by default:

```
KCONFIG_CONFIG  ?= .config
export KCONFIG_CONFIG
```

and the shell that will be used during kernel compilation:

```
CONFIG_SHELL := $(shell if [ -x "$$BASH" ]; then echo $$BASH; \
      else if [ -x /bin/bash ]; then echo /bin/bash; \
      else echo sh; fi ; fi)
```

The next set of variables related to the compiler that will be used during Linux kernel compilation. We set the host compilers for the `c` and `c++` and flags for it:

```
HOSTCC       = gcc
HOSTCXX      = g++
HOSTCFLAGS   = -Wall -Wmissing-prototypes -Wstrict-prototypes -O2 -fomit-frame-pointer -std=gnu89
HOSTCXXFLAGS = -O2
```

Next we will meet the `CC` variable that represent compiler too, so why do we need in the `HOST*` variables? The `CC` is the target compiler that will be used during kernel compilation, but `HOSTCC` will be used during compilation of the set of the `host` programs (we will see it soon). After this we can see definition of the `KBUILD_MODULES` and `KBUILD_BUILTIN` variables that are used for the determination of the what to compile (kernel, modules or both):

```
KBUILD_MODULES :=
KBUILD_BUILTIN := 1

ifeq ($(MAKECMDGOALS),modules)
  KBUILD_BUILTIN := $(if $(CONFIG_MODVERSIONS),1)
endif
```

Here we can see definition of these variables and the value of the `KBUILD_BUILTIN` will depens on the `CONFIG_MODVERSIONS` kernel configuration parameter if we pass only `modules` to the `make`. The next step is including of the:

```
include scripts/Kbuild.include
```

`kbuild` file. The Kbuild or `Kernel Build System` is the special infrastructure to manage building of the kernel and its modules. The `kbuild` files has the same syntax that makefiles. The scripts/Kbuild.include file provides some generic definitions for the `kbuild` system. As we included this `kbuild` files we can see definition of the variables that are related to the different tools that will be used during kernel and modules compilation (like linker, compilers, utils from the binutils and etc...):

```
AS      = $(CROSS_COMPILE)as
LD      = $(CROSS_COMPILE)ld
CC      = $(CROSS_COMPILE)gcc
CPP     = $(CC) -E
AR      = $(CROSS_COMPILE)ar
NM      = $(CROSS_COMPILE)nm
STRIP       = $(CROSS_COMPILE)strip
OBJCOPY     = $(CROSS_COMPILE)objcopy
OBJDUMP     = $(CROSS_COMPILE)objdump
AWK     = awk
...
...
...
```

After definition of these variables we define two variables: `USERINCLUDE` and `LINUXINCLUDE`. They will contain paths of the directories with headers (public for users in the first case and for kernel in the second case):

```
USERINCLUDE    := \
        -I$(srctree)/arch/$(hdr-arch)/include/uapi \
        -Iarch/$(hdr-arch)/include/generated/uapi \
        -I$(srctree)/include/uapi \
        -Iinclude/generated/uapi \
        -include $(srctree)/include/linux/kconfig.h

LINUXINCLUDE    := \
        -I$(srctree)/arch/$(hdr-arch)/include \
        ...
```

And the standard flags for the C compiler:

```
KBUILD_CFLAGS   := -Wall -Wundef -Wstrict-prototypes -Wno-trigraphs \
           -fno-strict-aliasing -fno-common \
           -Werror-implicit-function-declaration \
           -Wno-format-security \
           -std=gnu89
```

It is the not last compiler flags, they can be updated by the other makefiles (for example kbuilds from `arch/`). After all of these, all variables will be exported to be available in the other makefiles. The following two the `RCS_FIND_IGNORE` and the

`RCS_TAR_IGNORE` variables will contain files that will be ignored in the version control system:

```
export RCS_FIND_IGNORE := \( -name SCCS -o -name BitKeeper -o -name .svn -o    \
               -name CVS -o -name .pc -o -name .hg -o -name .git \) \
               -prune -o
export RCS_TAR_IGNORE := --exclude SCCS --exclude BitKeeper --exclude .svn \
              --exclude CVS --exclude .pc --exclude .hg --exclude .git
```

That's all. We have finished with the all preparations, next point is the building of `vmlinux`.

Directly to the kernel build

As we have finished all preparations, next step in the root makefile is related to the kernel build. Before this moment we will not see in the our terminal after the execution of the `make` command. But now first steps of the compilation are started. In this moment we need to go on the 598 line of the Linux kernel top makefile and we will see `vmlinux` target there:

```
all: vmlinux
    include arch/$(SRCARCH)/Makefile
```

Don't worry that we have missed many lines in Makefile that are placed after `export RCS_FIND_IGNORE.....` and before `all: vmlinux.....`. This part of the makefile is responsible for the `make *.config` targets and as I wrote in the beginning of this part we will see only building of the kernel in a general way.

The `all:` target is the default when no target is given on the command line. You can see here that we include architecture specific makefile there (in our case it will be arch/x86/Makefile). From this moment we will continue from this makefile. As we can see `all` target depends on the `vmlinux` target that defined a little lower in the top makefile:

```
vmlinux: scripts/link-vmlinux.sh $(vmlinux-deps) FORCE
```

The `vmlinux` is is the Linux kernel in an statically linked executable file format. The scripts/link-vmlinux.sh script links combines different compiled subsystems into vmlinux. The second target is the `vmlinux-deps` that defined as:

```
vmlinux-deps := $(KBUILD_LDS) $(KBUILD_VMLINUX_INIT) $(KBUILD_VMLINUX_MAIN)
```

and consists from the set of the `built-in.o` from the each top directory of the Linux kernel. Later, when we will go through all directories in the Linux kernel, the `Kbuild` will compile all the `$(obj-y)` files. It then calls `$(LD) -r` to merge these files into one `built-in.o` file. For this moment we have no `vmlinux-deps`, so the `vmlinux` target will not be executed now. For me `vmlinux-deps` contains following files:

```
arch/x86/kernel/vmlinux.lds arch/x86/kernel/head_64.o
arch/x86/kernel/head64.o    arch/x86/kernel/head.o
init/built-in.o             usr/built-in.o
arch/x86/built-in.o         kernel/built-in.o
mm/built-in.o               fs/built-in.o
ipc/built-in.o              security/built-in.o
crypto/built-in.o           block/built-in.o
lib/lib.a                   arch/x86/lib/lib.a
lib/built-in.o              arch/x86/lib/built-in.o
drivers/built-in.o          sound/built-in.o
firmware/built-in.o         arch/x86/pci/built-in.o
arch/x86/power/built-in.o   arch/x86/video/built-in.o
net/built-in.o
```

The next target that can be executed is following:

```
$(sort $(vmlinux-deps)): $(vmlinux-dirs) ;
$(vmlinux-dirs): prepare scripts
    $(Q)$(MAKE) $(build)=$@
```

As we can see the `vmlinux-dirs` depends on the two targets: `prepare` and `scripts`. The first `prepare` defined in the top `Makefile` of the Linux kernel and executes three stages of preparations:

```
prepare: prepare0
prepare0: archprepare FORCE
    $(Q)$(MAKE) $(build)=.
archprepare: archheaders archscripts prepare1 scripts_basic

prepare1: prepare2 $(version_h) include/generated/utsrelease.h \
                   include/config/auto.conf
    $(cmd_crmodverdir)
prepare2: prepare3 outputmakefile asm-generic
```

The first `prepare0` expands to the `archprepare` that exapnds to the `archheaders` and `archscripts` that defined in the `x86_64` specific Makefile. Let's look on it. The `x86_64` specific makefile starts from the definition of the variables that are related to the archicteture-specific configs (defconfig and etc.). After this it defines flags for the compiling of the 16-bit code, calculating of the `BITS` variable that can be `32` for `i386` or `64` for the `x86_64` flags for the assembly source code, flags for the linker and many many more (all definitions you can find in the arch/x86/Makefile). The first target is `archheaders` in the makefile generates syscall table:

```
archheaders:
    $(Q)$(MAKE) $(build)=arch/x86/entry/syscalls all
```

And the second target is `archscripts` in this makefile is:

```
archscripts: scripts_basic
    $(Q)$(MAKE) $(build)=arch/x86/tools relocs
```

We can see that it depends on the `scripts_basic` target from the top Makefile. At the first we can see the `scripts_basic` target that executes make for the scripts/basic makefile:

```
scripts_basic:
    $(Q)$(MAKE) $(build)=scripts/basic
```

The `scripts/basic/Makefile` contains targets for compilation of the two host programs: `fixdep` and `bin2`:

```
hostprogs-y := fixdep
hostprogs-$(CONFIG_BUILD_BIN2C)     += bin2c
always      := $(hostprogs-y)

$(addprefix $(obj)/,$(filter-out fixdep,$(always))): $(obj)/fixdep
```

First program is `fixdep` - optimizes list of dependencies generated by the gcc that tells make when to remake a source code file. The second program is `bin2c` depends on the value of the `CONFIG_BUILD_BIN2C` kernel configuration option and very little C program that allows to convert a binary on stdin to a C include on stdout. You can note here strange notation: `hostprogs-y` and etc. This notation is used in the all `kbuild` files and more about it you can read in the documentation. In our case the `hostprogs-y` tells to the `kbuild` that there is one host program named `fixdep` that will be built from the will be built from `fixdep.c` that located in the same directory that `Makefile`. The first output after we will execute `make` command in our terminal will be result of this `kbuild` file:

```
$ make
  HOSTCC  scripts/basic/fixdep
```

As `script_basic` target was executed, the `archscripts` target will execute `make` for the arch/x86/tools makefile with the `relocs` target:

```
$(Q)$(MAKE) $(build)=arch/x86/tools relocs
```

The `relocs_32.c` and the `relocs_64.c` will be compiled that will contain relocation information and we will see it in the `make` output:

```
HOSTCC  arch/x86/tools/relocs_32.o
HOSTCC  arch/x86/tools/relocs_64.o
HOSTCC  arch/x86/tools/relocs_common.o
HOSTLD  arch/x86/tools/relocs
```

There is checking of the `version.h` after compiling of the `relocs.c`:

```
$(version_h): $(srctree)/Makefile FORCE
    $(call filechk,version.h)
    $(Q)rm -f $(old_version_h)
```

We can see it in the output:

```
CHK     include/config/kernel.release
```

and the building of the `generic` assembly headers with the `asm-generic` target from the `arch/x86/include/generated/asm` that generated in the top Makefile of the Linux kernel. After the `asm-generic` target the `archprepare` will be done, so the `prepare0` target will be executed. As I wrote above:

```
prepare0: archprepare FORCE
    $(Q)$(MAKE) $(build)=.
```

Note on the `build`. It defined in the scripts/Kbuild.include and looks like this:

```
build := -f $(srctree)/scripts/Makefile.build obj
```

or in our case it is current source directory - `.`:

```
$(Q)$(MAKE) -f $(srctree)/scripts/Makefile.build obj=.
```

The scripts/Makefile.build tries to find the `Kbuild` file by the given directory via the `obj` parameter, include this `Kbuild` files:

```
include $(kbuild-file)
```

and build targets from it. In our case `.` contains the Kbuild file that generates the `kernel/bounds.s` and the `arch/x86/kernel/asm-offsets.s`. After this the `prepare` target finished to work. The `vmlinux-dirs` also depends on the second target - `scripts` that compiles following programs: `file2alias`, `mk_elfconfig`, `modpost` and etc... After scripts/host-programs compilation our `vmlinux-dirs` target can be executed. First of all let's try to understand what does `vmlinux-dirs` contain. For my case it contains paths of the following kernel directories:

```
init usr arch/x86 kernel mm fs ipc security crypto block
drivers sound firmware arch/x86/pci arch/x86/power
arch/x86/video net lib arch/x86/lib
```

We can find definition of the `vmlinux-dirs` in the top Makefile of the Linux kernel:

```
vmlinux-dirs    := $(patsubst %/,%,$(filter %/, $(init-y) $(init-m) \
             $(core-y) $(core-m) $(drivers-y) $(drivers-m) \
             $(net-y) $(net-m) $(libs-y) $(libs-m)))

init-y      := init/
drivers-y   := drivers/ sound/ firmware/
net-y       := net/
libs-y      := lib/
...
```

```
...
...
```

Here we remove the `/` symbol from the each directory with the help of the `patsubst` and `filter` functions and put it to the `vmlinux-dirs`. So we have list of directories in the `vmlinux-dirs` and the following code:

```
$(vmlinux-dirs): prepare scripts
    $(Q)$(MAKE) $(build)=$@
```

The `$@` represents `vmlinux-dirs` here that means that it will go recursively over all directories from the `vmlinux-dirs` and its internal directories (depens on configuration) and will execute `make` in there. We can see it in the output:

```
  CC      init/main.o
  CHK     include/generated/compile.h
  CC      init/version.o
  CC      init/do_mounts.o
  ...
  CC      arch/x86/crypto/glue_helper.o
  AS      arch/x86/crypto/aes-x86_64-asm_64.o
  CC      arch/x86/crypto/aes_glue.o
  ...
  AS      arch/x86/entry/entry_64.o
  AS      arch/x86/entry/thunk_64.o
  CC      arch/x86/entry/syscall_64.o
```

Source code in each directory will be compiled and linked to the `built-in.o`:

```
$ find . -name built-in.o
./arch/x86/crypto/built-in.o
./arch/x86/crypto/sha-mb/built-in.o
./arch/x86/net/built-in.o
./init/built-in.o
./usr/built-in.o
...
...
```

Ok, all buint-in.o(s) built, now we can back to the `vmlinux` target. As you remember, the `vmlinux` target is in the top Makefile of the Linux kernel. Before the linking of the `vmlinux` it builds samples, Documentation and etc., but I will not describe it in this part as I wrote in the beginning of this part.

```
vmlinux: scripts/link-vmlinux.sh $(vmlinux-deps) FORCE
    ...
    ...
    +$(call if_changed,link-vmlinux)
```

As you can see main purpose of it is a call of the scripts/link-vmlinux.sh script is linking of the all `built-in.o`(s) to the one statically linked executable and creation of the System.map. In the end we will see following output:

```
  LINK    vmlinux
  LD      vmlinux.o
  MODPOST vmlinux.o
  GEN     .version
  CHK     include/generated/compile.h
  UPD     include/generated/compile.h
  CC      init/version.o
  LD      init/built-in.o
  KSYM    .tmp_kallsyms1.o
  KSYM    .tmp_kallsyms2.o
  LD      vmlinux
  SORTEX  vmlinux
  SYSMAP  System.map
```

and `vmlinux` and `System.map` in the root of the Linux kernel source tree:

```
$ ls vmlinux System.map
System.map  vmlinux
```

That's all, `vmlinux` is ready. The next step is creation of the bzImage.

Building bzImage

The `bzImage` is the compressed Linux kernel image. We can get it with the execution of the `make bzImage` after the `vmlinux` built. In other way we can just execute `make` without arguments and will get `bzImage` anyway because it is default image:

```
all: bzImage
```

in the arch/x86/kernel/Makefile. Let's look on this target, it will help us to understand how this image builds. As I already said the `bzImage` target defined in the arch/x86/kernel/Makefile and looks like this:

```
bzImage: vmlinux
	$(Q)$(MAKE) $(build)=$(boot) $(KBUILD_IMAGE)
	$(Q)mkdir -p $(objtree)/arch/$(UTS_MACHINE)/boot
	$(Q)ln -fsn ../../x86/boot/bzImage $(objtree)/arch/$(UTS_MACHINE)/boot/$@
```

We can see here, that first of all called `make` for the boot directory, in our case it is:

```
boot := arch/x86/boot
```

The main goal now to build source code in the `arch/x86/boot` and `arch/x86/boot/compressed` directories, build `setup.bin` and `vmlinux.bin`, and build the `bzImage` from they in the end. First target in the arch/x86/boot/Makefile is the `$(obj)/setup.elf`:

```
$(obj)/setup.elf: $(src)/setup.ld $(SETUP_OBJS) FORCE
	$(call if_changed,ld)
```

We already have the `setup.ld` linker script in the `arch/x86/boot` directory and the `SETUP_OBJS` expands to the all source files from the `boot` directory. We can see first output:

```
  AS      arch/x86/boot/bioscall.o
  CC      arch/x86/boot/cmdline.o
  AS      arch/x86/boot/copy.o
  HOSTCC  arch/x86/boot/mkcpustr
  CPUSTR  arch/x86/boot/cpustr.h
  CC      arch/x86/boot/cpu.o
  CC      arch/x86/boot/cpuflags.o
  CC      arch/x86/boot/cpucheck.o
  CC      arch/x86/boot/early_serial_console.o
  CC      arch/x86/boot/edd.o
```

The next source code file is the arch/x86/boot/header.S, but we can't build it now because this target depends on the following two header files:

```
$(obj)/header.o: $(obj)/voffset.h $(obj)/zoffset.h
```

The first is `voffset.h` generated by the `sed` script that gets two addresses from the `vmlinux` with the `nm` util:

```
#define VO__end 0xffffffff82ab0000
#define VO__text 0xffffffff81000000
```

They are start and end of the kernel. The second is `zoffset.h` depens on the `vmlinux` target from the arch/x86/boot/compressed/Makefile:

```
$(obj)/zoffset.h: $(obj)/compressed/vmlinux FORCE
    $(call if_changed,zoffset)
```

The `$(obj)/compressed/vmlinux` target depends on the `vmlinux-objs-y` that compiles source code files from the arch/x86/boot/compressed directory and generates `vmlinux.bin`, `vmlinux.bin.bz2`, and compiles programm - `mkpiggy`. We can see this in the output:

```
  LDS     arch/x86/boot/compressed/vmlinux.lds
  AS      arch/x86/boot/compressed/head_64.o
  CC      arch/x86/boot/compressed/misc.o
  CC      arch/x86/boot/compressed/string.o
  CC      arch/x86/boot/compressed/cmdline.o
  OBJCOPY arch/x86/boot/compressed/vmlinux.bin
  BZIP2   arch/x86/boot/compressed/vmlinux.bin.bz2
  HOSTCC  arch/x86/boot/compressed/mkpiggy
```

Where the `vmlinux.bin` is the `vmlinux` with striped debuging information and comments and the `vmlinux.bin.bz2` compressed `vmlinux.bin.all` + `u32` size of `vmlinux.bin.all`. The `vmlinux.bin.all` is `vmlinux.bin + vmlinux.relocs`, where `vmlinux.relocs` is the `vmlinux` that was handled by the `relocs` program (see above). As we got these files, the `piggy.S` assembly files will be generated with the `mkpiggy` program and compiled:

```
  MKPIGGY arch/x86/boot/compressed/piggy.S
  AS      arch/x86/boot/compressed/piggy.o
```

This assembly files will contain computed offset from a compressed kernel. After this we can see that `zoffset` generated:

```
  ZOFFSET arch/x86/boot/zoffset.h
```

As the `zoffset.h` and the `voffset.h` are generated, compilation of the source code files from the arch/x86/boot can be continued:

```
  AS      arch/x86/boot/header.o
  CC      arch/x86/boot/main.o
  CC      arch/x86/boot/mca.o
  CC      arch/x86/boot/memory.o
  CC      arch/x86/boot/pm.o
  AS      arch/x86/boot/pmjump.o
  CC      arch/x86/boot/printf.o
  CC      arch/x86/boot/regs.o
  CC      arch/x86/boot/string.o
  CC      arch/x86/boot/tty.o
  CC      arch/x86/boot/video.o
  CC      arch/x86/boot/video-mode.o
  CC      arch/x86/boot/video-vga.o
  CC      arch/x86/boot/video-vesa.o
  CC      arch/x86/boot/video-bios.o
```

As all source code files will be compiled, they will be linked to the `setup.elf`:

```
  LD      arch/x86/boot/setup.elf
```

or:

```
ld -m elf_x86_64   -T arch/x86/boot/setup.ld arch/x86/boot/a20.o arch/x86/boot/bioscall.o arch/x86/boot/cmdline.c
```

The last two things is the creation of the `setup.bin` that will contain compiled code from the `arch/x86/boot/*` directory:

```
objcopy  -O binary arch/x86/boot/setup.elf arch/x86/boot/setup.bin
```

and the creation of the `vmlinux.bin` from the `vmlinux`:

```
objcopy  -O binary -R .note -R .comment -S arch/x86/boot/compressed/vmlinux arch/x86/boot/vmlinux.bin
```

In the end we compile host program: arch/x86/boot/tools/build.c that will create our `bzImage` from the `setup.bin` and the `vmlinux.bin`:

```
arch/x86/boot/tools/build arch/x86/boot/setup.bin arch/x86/boot/vmlinux.bin arch/x86/boot/zoffset.h arch/x86/boot
```

Actually the `bzImage` is the concatenated `setup.bin` and the `vmlinux.bin`. In the end we will see the output which familiar to all who once build the Linux kernel from source:

```
Setup is 16268 bytes (padded to 16384 bytes).
System is 4704 kB
CRC 94a88f9a
Kernel: arch/x86/boot/bzImage is ready  (#5)
```

That's all.

Conclusion

It is the end of this part and here we saw all steps from the execution of the `make` command to the generation of the `bzImage`. I know, the Linux kernel makefiles and process of the Linux kernel building may seem confusing at first glance, but it is not so hard. Hope this part will help you to understand process of the Linux kernel building.

Links

- GNU make util
- Linux kernel top Makefile
- cross-compilation
- Ctags
- sparse
- bzImage
- uname
- shell
- Kbuild
- binutils
- gcc
- Documentation
- System.map
- Relocation