

嵌入式操作系统

3 基于arm的Linux启动代码分析

陈香兰 (xlanchen@ustc.edu.cn)

计算机应用教研室@计算机学院
嵌入式系统实验室@苏州研究院
中国科学技术大学
Fall 2014

November 27, 2014

Outline

- 1 Linux的源代码组织
- 2 ARMlinux的编译过程和产生的映像
 - 阅读Makefile
 - 源代码根目录下的vmlinux的生成
 - zImage的生成
 - bootpImage的生成
- 3 ARMlinux的启动分析
- 4 小结和作业

基于arm的Linux启动代码分析

- 源码：来自Linux-2.6.26
- 了解linux的源码组织
 - ▶ 看目录结构
- 了解linux的内核代码结构
 - ▶ 看Makefile
- 了解基于arm的linux的boot image的结构
- 掌握arm的启动流程
 - ▶ 阅读启动源码文件

Outline

- 1 Linux的源代码组织
- 2 ARMlinux的编译过程和产生的映像
- 3 ARMlinux的启动分析
- 4 小结和作业

考虑Arch为arm，了解源码组织

- 观察Linux源码的根目录
- 观察arch目录
- 观察arch下的arm目录
- 观察include目录
- 观察init目录

Outline

- 1 Linux的源代码组织
- 2 ARMlinux的编译过程和产生的映像
 - 阅读Makefile
 - 源代码根目录下的vmlinux的生成
 - zImage的生成
 - bootpImage的生成
- 3 ARMlinux的启动分析
- 4 小结和作业

查看make帮助信息

- 查看make的帮助信息，了解armlinux相关的产物：

```
make ARCH=arm help
```

其中

Architecture specific targets (arm):

- * zImage - Compressed kernel image (arch/arm/boot/zImage)
- Image - Uncompressed kernel image (arch/arm/boot/Image)
- * xipImage - XIP kernel image, if configured (arch/arm/boot/xipImage)
- uImage - U-Boot wrapped zImage
- bootImage - Combined zImage and initial RAM disk
(supply initrd image via make variable INITRD=<path>)
- install - Install uncompressed kernel
- zinstall - Install compressed kernel
Install using (your) ~/bin/installkernel or
(distribution) /sbin/installkernel or
install to \$(INSTALL_PATH) and run lilo

Outline

- 1 Linux的源代码组织
- 2 ARMlinux的编译过程和产生的映像
 - 阅读Makefile
 - 源代码根目录下的vmlinux的生成
 - zImage的生成
 - bootpImage的生成
- 3 ARMlinux的启动分析
- 4 小结和作业

阅读Makefile

- 阅读下列相关的Makefile和链接描述文件，了解上述产物是如何生成的：

- ① Linux源代码根目录下的Makefile
- ② arch/arm/Makefile
- ③ arch/arm/boot/Makefile
- ④ arch/arm/boot/compressed/Makefile
- ⑤ arch/arm/boot/bootp/Makefile
- ⑥ arch/arm/kernel/vmlinux.lds
- ⑦ ...

1、阅读Linux源代码根目录下的Makefile

● 找到缺省目标all

```
# The all: target is the default when no target is given on the
# command line.
# This allow a user to issue only ' make' to build a kernel including modules
# Defaults vmlinux but it is usually overridden in the arch makefile
all: vmlinux
```

● 找到vmlinux目标，并阅读

```
# vmlinux image - including updated kernel symbols
vmlinux: $(vmlinux-lds) $(vmlinux-init) $(vmlinux-main) vmlinux.o $(kallsyms.o) FORCE
ifndef CONFIG_HEADERS_CHECK
    $(Q)$(MAKE) -f $(srctree)/Makefile headers_check
endif
ifndef CONFIG_SAMPLES
    $(Q)$(MAKE) $(build)=samples
endif
    $(call vmlinux-modpost)
    $(call if_changed_rule,vmlinux__)
    $(Q)rm -f .old_version
```

1、阅读Linux源代码根目录下的Makefile

- 解释：关于

`$(call if_changed_rule,vmlinux__)`



`rule_vmlinux__`

```
# Link of vmlinux
# If CONFIG_KALLSYMS is set .version is already updated
# Generate System.map and verify that the content is consistent
# Use + in front of the vmlinux_version rule to silent warning with make -j2
# First command is ' :' to allow us to use + in front of the rule
define rule_vmlinux__
:
$(if $(CONFIG_KALLSYMS),, +$(call cmd,vmlinux_version))

$(call cmd,vmlinux__)
$(Q)echo ' cmd_$@ := $(cmd_vmlinux__)' > $(@D)/.$(@F).cmd

$(Q)$(if $($($(quiet)cmd_sysmap), \
    echo ' $($($(quiet)cmd_sysmap) System.map' &&) \
$(cmd_sysmap) $@ System.map; \
if [ $$? -ne 0 ]; then \
    rm -f $@; \
    /bin/false; \
fi;
$(verify_kallsyms)
endef
```

1、阅读Linux源代码根目录下的Makefile

```
# Rule to link vmlinux - also used during CONFIG_KALLSYMS
# May be overridden by arch/${ARCH}/Makefile
quiet_cmd_vmlinux__  ?= LD $@
    cmd_vmlinux__  ?= $(LD) $(LDFLAGS) $(LDFLAGS_vmlinux) -o $@ \
        -T $(vmlinux-lds) $(vmlinux-init) \
        --start-group $(vmlinux-main) --end-group \
        $(filter-out $(vmlinux-lds) $(vmlinux-init) $(vmlinux-main) vmlinux.o FORCE
, $^)
```

- 链接描述文件？
- 链接顺序：
 - ▶ vmlinux-init
 - ▶ vmlinux-main

1、阅读Linux源代码根目录下的Makefile

```
# Build vmlinux
# -----
# vmlinux is built from the objects selected by $(vmlinux-init) and
# $(vmlinux-main). Most are built-in.o files from top-level directories
# in the kernel tree, others are specified in arch/$(ARCH)/Makefile.
# Ordering when linking is important, and $(vmlinux-init) must be first.
#
# vmlinux
# ^
# |
# +--< $(vmlinux-init)
# | +--< init/version.o + more
# |
# +--< $(vmlinux-main)
# | +--< driver/built-in.o mm/built-in.o + more
# |
# +--< kallsyms.o (see description in CONFIG_KALLSYMS section)
```

● 参见 “Documentation/kbuild/makefiles.txt”

— 6.7 Custom kbuild commands

When kbuild is executing with `KBUILD_VERBOSE=0`, then only a shorthand of a command is normally displayed.

To enable this behaviour for custom commands kbuild requires two variables to be set:

`quiet_cmd_<command>` - what shall be echoed
`cmd_<command>` - the command to execute

Example:

```
#
quiet_cmd_image = BUILD $$
cmd_image = $(obj)/tools/build $(BUILDFLAGS) \
            $(obj)/vmlinux.bin > $$

targets += bzImage
$(obj)/bzImage: $(obj)/vmlinux.bin $(obj)/tools/build FORCE
    $(call if_changed,image)
    @echo ' Kernel: $$ is ready'
```

When updating the `$(obj)/bzImage` target, the line

```
BUILD arch/i386/boot/bzImage
```

will be displayed with `" make KBUILD_VERBOSE=0"` .

● 注意：

- ▶ vmlinux-init
- ▶ vmlinux-main

```
vmlinux-init := $(head-y) $(init-y)
vmlinux-main := $(core-y) $(libs-y) $(drivers-y) $(net-y)
vmlinux-all := $(vmlinux-init) $(vmlinux-main)
vmlinux-lds := arch/$(SRCARCH)/kernel/vmlinux.lds
```

- ▶ vmlinux-dirs

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

Outline

- 1 Linux的源代码组织
- 2 ARMlinux的编译过程和产生的映像
 - 阅读Makefile
 - 源代码根目录下的vmlinux的生成
 - zImage的生成
 - bootpImage的生成
- 3 ARMlinux的启动分析
- 4 小结和作业

主要目标文件vmlinux的编译

- vmlinux



```
# The actual objects are generated when descending,  
# make sure no implicit rule kicks in  
$(sort $(vmlinux-init) $(vmlinux-main)) $(vmlinux-lds): $(vmlinux-dirs) ;
```



```
# Handle descending into subdirectories listed in $(vmlinux-dirs)  
# Preset locale variables to speed up the build process. Limit locale  
# tweaks to this spot to avoid wrong language settings when running  
# make menuconfig etc.  
# Error messages still appears in the original language
```

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



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

- 不妨以core-y为例，

观察体系相关和体系无关部分的代码是如何被包含进来的

ARMLinux的启动和初始化代码：vmlinux-init

```
vmlinux-init := (head-y)(init-y)
```

① head-y = ?

根Makefile中以include的方式包含了arm体系结构相关部分的Makefile

```
# Read arch specific Makefile to set KBUILD_DEFCONFIG as needed.  
# KBUILD_DEFCONFIG may point out an alternative default configuration  
# used for 'make defconfig'  
include $(srctree)/arch/$(SRCARCH)/Makefile  
export KBUILD_DEFCONFIG
```

在这个Makefile中

```
#Default value  
head-y      := arch/arm/kernel/head$(MMUEXT).o arch/arm/kernel/init_task.o  
textofs-y   := 0x00008000
```

其中，变量MMUEXT的定义如下

```
# defines filename extension depending memory manement type.  
ifeq ($(CONFIG_MMU),)  
MMUEXT      := -nommu  
endif
```

head-nommu还是head

ARMLinux的启动和初始化代码：vmlinux-init

```
vmlinux-init := (head - y)(init-y)
```

- ② `init-y = ?`
在根Makefile中

```
# Objects we will link into vmlinux / subdirs we need to visit
init-y          := init/
drivers-y       := drivers/ sound/
net-y          := net/
libs-y         := lib/
core-y         := usr/
```

为便于阅读，了解关于命令输出的相关内容

```
# Beautify output
# -----
#
# Normally, we echo the whole command before executing it. By making
# that echo $(quiet$(cmd)), we now have the possibility to set
# $(quiet) to choose other forms of output instead, e.g.
#
# quiet_cmd_cc_o_c = Compiling $(RELDIR)/$@
# cmd_cc_o_c = $(CC) $(c_flags) -c -o $@ $<
#
# If $(quiet) is empty, the whole command will be printed.
# If it is set to "quiet_", only the short version will be printed.
# If it is set to "silent_", nothing will be printed at all, since
# the variable $(silent_cmd_cc_o_c) doesn't exist.
#
# A simple variant is to prefix commands with $(Q) - that's useful
# for commands that shall be hidden in non-verbose mode.
#
# $(Q)ln $@ :<
#
# If KBUILD_VERBOSE equals 0 then the above command will be hidden.
# If KBUILD_VERBOSE equals 1 then the above command is displayed.

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

考虑体系结构相关的产物 I

- 我们已经知道vmlinux是如何生成的，下面看看体系结构相关的产物，例如zImage是如何生成的？
- `make <default>`, OR
`make zImage`, OR
...
- 在arch/arm/Makefile中

考虑体系结构相关的产物 II

```
...  
# Default target when executing plain make  
ifeq ($(CONFIG_XIP_KERNEL),y)  
KBUILD_IMAGE := xipImage  
else  
KBUILD_IMAGE := zImage  
endif  
  
all: $(KBUILD_IMAGE)  
  
...  
# Convert bzImage to zImage  
bzImage: zImage  
  
zImage Image xipImage bootpImage uImage: vmlinux  
$(Q)$(MAKE) $(build)=$(boot) MACHINE=$(MACHINE) $(boot)/$@  
  
zinstall install: vmlinux  
$(Q)$(MAKE) $(build)=$(boot) MACHINE=$(MACHINE) $@
```

- ▶ **z**代表压缩；**b**代表大内核
- ▶ 对于arm，没有大小内核之分

考虑体系结构相关的产物 III

- 各种image都需要到boot目录下生成
 - ▶ 压缩zImage or 非压缩Image or bootp or ...

观察boot目录

```
tree arch/arm/boot/
```

```
arch/arm/boot/
```

```
├── bootp
│   ├── bootp.lds
│   ├── initrd.S
│   ├── init.S
│   ├── kernel.S
│   └── Makefile
├── compressed
│   ├── big-endian.S
│   ├── head-clps7500.S
│   ├── head-17200.S
│   ├── head.S
│   ├── head-sa1100.S
│   ├── head-shark.S
│   ├── head-sharps1.S
│   ├── head-xscale.S
│   ├── ll_char_wr.S
│   ├── Makefile
│   ├── Makefile.debug
│   ├── misc.c
│   ├── ofw-shark.c
│   ├── piggy.S
│   └── vmlinux.lds.in
├── install.sh
└── Makefile
```


Outline

- 1 Linux的源代码组织
- 2 ARMlinux的编译过程和产生的映像
 - 阅读Makefile
 - 源代码根目录下的vmlinux的生成
 - **zImage的生成**
 - bootpImage的生成
- 3 ARMlinux的启动分析
- 4 小结和作业

arch/arm/boot/zImage的生成

❶ 在arch/arm/Makefile中：

观察zImage

```
...  
# Default target when executing plain make  
ifeq ($(CONFIG_XIP_KERNEL),y)  
KBUILD_IMAGE := xipImage  
else  
KBUILD_IMAGE := zImage  
endif  
  
all: $(KBUILD_IMAGE)  
  
...  
zImage Image xipImage bootpImage uImage: vmlinux  
    $(Q)$(MAKE) $(build)=$(boot) MACHINE=$(MACHINE) $(boot)/$@  
  
...
```

zImage是缺省目标，而且是一个phony target。
(其他image也是phony target)

arch/arm/boot/zImage的生成

② 在arch/arm/boot/Makefile中：

▶ 3: zImage

```
$(obj)/zImage: $(obj)/compressed/vmlinux FORCE
$(call if_changed,objcopy)
@echo '   Kernel: $@ is ready'
```

▶ 2: compressed/vmlinux

```
$(obj)/compressed/vmlinux: $(obj)/Image FORCE
$(Q)$(MAKE) $(build)=$(obj)/compressed $@
```

▶ 1: Image

```
$(obj)/Image: vmlinux FORCE
$(call if_changed,objcopy)
@echo '   Kernel: $@ is ready'
```

Image所依赖的vmlinux即源代码根目录下生成的vmlinux

arch/arm/boot/zImage的生成

③ 在arch/arm/boot/compressed/Makefile中：

▶ compressed/vmlinux

```
$(obj)/vmlinux: $(obj)/vmlinux.lds $(obj)/$(HEAD) $(obj)/piggy.o \  
    $(addprefix $(obj)/, $(OBJS)) FORCE  
$(call if_changed,ld)  
@:
```

▶ 其中，变量HEAD和OBJS？

```
HEAD = head.o
```

```
OBJS = misc.o
```

此外，根据配置情况，OBJS还可能包含其他目标文件

▶ 关于piggy.o

```
$(obj)/piggy.gz: $(obj)/../Image FORCE  
$(call if_changed,gzip)
```

```
$(obj)/piggy.o: $(obj)/piggy.gz FORCE
```

arch/arm/boot/zImage的生成

❶ 了解compressed/vmlinux的链接情况

- ▶ 观察文件：

`arch/arm/boot/compressed/vmlinux.lds.in`

- ▶ 若已经缺省编译过，则可以看看文件：

`arch/arm/boot/compressed/vmlinux.lds`

Outline

- 1 Linux的源代码组织
- 2 ARMlinux的编译过程和产生的映像
 - 阅读Makefile
 - 源代码根目录下的vmlinux的生成
 - zImage的生成
 - bootpImage的生成
- 3 ARMlinux的启动分析
- 4 小结和作业

arch/arm/boot/bootImage的生成

- 在arch/arm/boot/Makefile中：

- ▶ bootpImage

```
$(obj)/bootp/bootp: $(obj)/zImage initrd FORCE
    $(Q)$ (MAKE) $(build)=$(obj)/bootp $@
@:
```

```
$(obj)/bootpImage: $(obj)/bootp/bootp FORCE
    $(call if_changed,objcopy)
    @echo '  Kernel: $@ is ready'
```

bootImage是zImage和initrd的合成体。
具体由arch/arm/boot/bootp目录负责生成。

arch/arm/boot/bootp 目录

- 目录下的内容

arch/arm/boot/bootp/

- bootp.lds
- initrd.S
- init.S
- kernel.S
- Makefile

1、arch/arm/boot/bootp/Makefile

```
#
# linux/arch/arm/boot/bootp/Makefile
# ...

LDFLAGS_bootp := -p --no-undefined -X \
                 --defsym initrd_phys=$(INITRD_PHYS) \
                 --defsym params_phys=$(PARAMS_PHYS) -T
AFLAGS_initrd.o := -DINITRD=\\" $(INITRD)\\

targets := bootp init.o kernel.o initrd.o

# Note that bootp.lds picks up kernel.o and initrd.o
$(obj)/bootp: $(src)/bootp.lds $(addprefix $(obj)/,init.o kernel.o initrd.o) FORCE
    $(call if_changed,ld)
    @:

# kernel.o and initrd.o includes a binary image using
# .incbin, a dependency which is not tracked automatically

$(obj)/kernel.o: arch/arm/boot/zImage FORCE

$(obj)/initrd.o: $(INITRD) FORCE

PHONY += $(INITRD) FORCE
```

2、关于INITRD

```
define archhelp
echo ' * zImage      - Compressed kernel image (arch/${ARCH}/boot/zImage)'
echo '      Image      - Uncompressed kernel image (arch/${ARCH}/boot/Image)'
echo ' * xipImage      - XIP kernel image, if configured (arch/${ARCH}/boot/xipImage)'
echo '      uImage       - U-Boot wrapped zImage'
echo '      bootImage     - Combined zImage and initial RAM disk'
echo '                      (supply initrd image via make variable INITRD=<path>)'
echo '      install       - Install uncompressed kernel'
echo '      zinstall      - Install compressed kernel'
echo '                      Install using (your) ~/bin/installkernel or'
echo '                      (distribution) /sbin/installkernel or'
echo '                      install to ${INSTALL_PATH} and run lilo'
endif
```

3、arch/arm/boot/bootp下的kernel.S和initrd.S

文件：kernel.S

```
.globl kernel_start
kernel_start:
    .incbin " arch/arm/boot/zImage"
    .globl kernel_end
kernel_end:
    .align 2
```

文件：initrd.S

```
.type initrd_start,#object
.globl initrd_start
initrd_start:
    .incbin INITRD
    .globl initrd_end
initrd_end:
```

4、arch/arm/boot/bootp/init.S

文件：init.S

```
.section .start,#alloc,#execinstr
.type _start, #function
.globl _start
```

_start:

```
...
mov r5, #4      @ Size of initrd tag (4 words)
stmia r9, {r5, r6, r7, r8, r10}
b kernel_start @ call kernel
...
```

where?

5、arch/arm/boot/bootp/bootp.lds

链接脚本：

```
...
OUTPUT_ARCH(arm)
ENTRY(_start)
SECTIONS
{
    . = 0;
    .text : {
        _stext = .;
        *(.start)
        *(.text)
        initrd_size = initrd_end - initrd_start;
        _etext = .;
    }

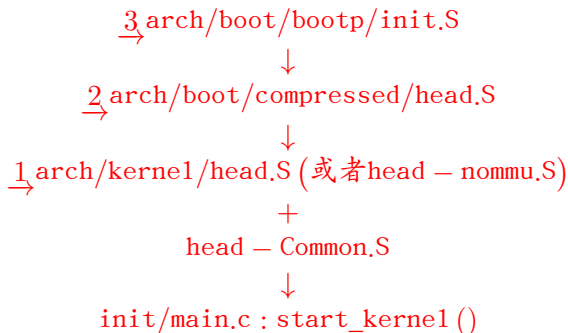
    .stab 0 : { *(.stab) }
    .stabstr 0 : { *(.stabstr) }
    .stab.excl 0 : { *(.stab.excl) }
    .stab.exclstr 0 : { *(.stab.exclstr) }
    .stab.index 0 : { *(.stab.index) }
    .stab.indexstr 0 : { *(.stab.indexstr) }
    .comment 0 : { *(.comment) }
}
```

Outline

- 1 Linux的源代码组织
- 2 ARMlinux的编译过程和产生的映像
- 3 ARMlinux的启动分析**
- 4 小结和作业

armlinux启动的推断

- 以zImage和initrd的合成体bootImage为例：



- zImage ? Image ?
- 阅读documentation/arm/booting
- 阅读documentation/arm/IXP4xx

ARMlinux的启动

- Bootloader：参考文件Documentation/arm/Booting
- In order to boot ARM Linux, a bootloader is required.
 - ▶ Bootloader is a small program that runs before the main kernel.
 - ▶ The bootloader is expected to
 - ★ initialize various devices, and eventually
 - ★ call the Linux kernel, passing information to the kernel.
- 以zImage为例，进入zImage执行的方法：
 - ① 直接从flash中的zImage启动
 - ② 把zImage取到RAM中再启动。





















ARMlinux的启动

进入Linux中关于arm的启动代码前的约定：

In either case, the following conditions must be met:

- ▶ Quiesce all DMA capable devices so that memory does not get corrupted by bogus network packets or disk data. This will save you many hours of debug.
- ▶ CPU register settings
 - `r0 = 0,`
 - `r1 = machine type number discovered in (3) above.`
 - `r2 = physical address of tagged list in system RAM.`
- ▶ CPU mode
 - All forms of interrupts must be disabled (IRQs and FIQs)
 - The CPU must be in SVC mode. (A special exception exists for Angel)
- ▶ Caches, MMUs
 - The MMU must be off.
 - Instruction cache may be on or off.
 - Data cache must be off.
- ▶ The boot loader is expected to call the kernel image by jumping directly to the first instruction of the kernel image.

Arm寄存器集

Modes						
Privileged modes						
Exception modes						
User	System	Supervisor	Abort	Undefined	Interrupt	Fast interrupt
R0	R0	R0	R0	R0	R0	R0
R1	R1	R1	R1	R1	R1	R1
R2	R2	R2	R2	R2	R2	R2
R3	R3	R3	R3	R3	R3	R3
R4	R4	R4	R4	R4	R4	R4
R5	R5	R5	R5	R5	R5	R5
R6	R6	R6	R6	R6	R6	R6
R7	R7	R7	R7	R7	R7	R7
R8	R8	R8	R8	R8	R8	 R8_fiq
R9	R9	R9	R9	R9	R9	 R9_fiq
R10	R10	R10	R10	R10	R10	 R10_fiq
R11	R11	R11	R11	R11	R11	 R11_fiq
R12	R12	R12	R12	R12	R12	 R12_fiq
R13	R13	 R13_svc	 R13_abt	 R13_und	 R13_irq	 R13_fiq
R14	R14	 R14_svc	 R14_abt	 R14_und	 R14_irq	 R14_fiq
PC	PC	PC	PC	PC	PC	PC
CPSR	CPSR	CPSR	CPSR	CPSR	CPSR	CPSR
		 SPSR_svc	 SPSR_abt	 SPSR_und	 SPSR_irq	 SPSR_fiq

 indicates that the normal register used by User or System mode has been replaced by an alternative register specific to the exception mode

ARMLinux的启动入口点

- 根据所使用的image的不同，入口可能是
 - ① bootpImage : Bootp的Init.S
 - ② zImage : Compressed的head.S
 - ③ Image : Kernel的head.S

确定bootpImage的入口

- bootpImage在什么时候、如何加载到目标机器上？
- 确定bootpImage的入口
 - ▶ bootp/Makeifile
 - ▶ bootp/bootp.lds
 - ▶ bootp/init.S
- init.S的主要流程：
 - ① Move initrd to its right place.
 - ② Setup the initrd parameters to pass to the kernel.
 - ③ Call the kernel

结合compressed/head.S确定zImage的入口

- 提示：从bootpImage是如何开始执行zImage的？
- 确定zImage的入口
 - ▶ compressed/Makefile
 - ▶ compressed/vmlinux.lds.in
 - ▶ compressed/head.S

❶ 入口 start

```
.section ".start", #alloc, #execinstr
/*
 * sort out different calling conventions
 */
.align
start:
.type start,#function
.rept 8
mov r0, r0
.endr

b 1f
.word 0x016f2818 @ Magic numbers to help the loader
.word start @ absolute load/run zImage address
.word _edata @ zImage end address
1:   mov r7, r1 @ save architecture ID
    mov r8, r2 @ save atags pointer
```

❷ 关中断

- ③ 执行具体体系相关代码，并矫正代码的空间感
(不同的体系有不同的代码，通过ld的方式链入)

- ▶ 参见链接描述相关文件vmlinux.lds.in

- ④ 清BSS段
- ⑤ 分配堆栈
- ⑥ 解压缩

```
mov r5, r2 @ decompress after malloc space
mov r0, r5
mov r3, r7
bl decompress_kernel
```

- ⑦ [可选] 复制relocation code，
然后跳转到reloc_start处开始运行

8 call_kernel

```
call_kernel:    bl cache_clean_flush
               bl cache_off
               mov r0, #0 @ must be zero
               mov r1, r7 @ restore architecture number
               mov r2, r8 @ restore atags pointer
               mov pc, r4 @ call kernel
```

kernel是什么？

确定Image的入口

- 确定Image的入口

- ▶ Makefile
- ▶ arch/arm/Makefile
- ▶ arch/arm/kernel/vmlinux.lds.S
若编译过，则代之以arch/arm/kernel/vmlinux.lds

arm/kernel/head.S

❶ 入口 stext

```
/*
 * Kernel startup entry point.
 * -----
 *
 * This is normally called from the decompressor code. The requirements
 * are: MMU = off, D-cache = off, I-cache = dont care, r0 = 0,
 * r1 = machine nr, r2 = atags pointer.
 *
 * This code is mostly position independent, so if you link the kernel at
 * 0xc0008000, you call this at __pa(0xc0008000).
 *
 * See linux/arch/arm/tools/mach-types for the complete list of machine
 * numbers for r1.
 *
 * We're trying to keep crap to a minimum; DO NOT add any machine specific
 * crap here - that's what the boot loader (or in extreme, well justified
 * circumstances, zImage) is for.
 */
.section ".text.head", "ax"
.type stext, %function
ENTRY(stext)
    msr cpsr_c, #PSR_F_BIT | PSR_I_BIT | SVC_MODE @ ensure svc mode
                                                    @ and irqs disabled
    ...
```

arm/kernel/head.S

- lookup processor type
- lookup machine type
- create page tables
- enable mmu
- jump to (`__switch_data`), 即进入head-common.S

在head.S的末尾：

```
#include " head-common.S"
```

- 关键数据结构：__switch_data
 - ▶ 关键function：__mmap_switched
- 关键流程：
 - ① Clear BSS
 - ② call start_kernel

关于页表的创建

• 几个关键的参数和宏

① PAGE_OFFSET, include/asm-arm/memory.h

```
/*
 * Page offset: 3GB
 */
#ifndef PAGE_OFFSET
#define PAGE_OFFSET UL(0xc0000000)
#endif
```

② PHYS_OFFSET, include/asm-arm/memory.h

```
#ifndef PHYS_OFFSET
#define PHYS_OFFSET (CONFIG_DRAM_BASE)
#endif
```

并且在arch/arm/kernel/head.S中

```
#if (PHYS_OFFSET & 0x001fffff)
#error " PHYS_OFFSET must be at an even 2MiB boundary!"
#endif
```

关于页表的创建

❷ 虚拟地址与物理地址之间的转换，include/asm-arm/memory.h

```
/*  
 * Physical vs virtual RAM address space conversion. These are  
 * private definitions which should NOT be used outside memory.h  
 * files. Use virt_to_phys/phys_to_virt/__pa/__va instead.  
 */  
#ifndef __virt_to_phys  
#define __virt_to_phys(x) ((x) - PAGE_OFFSET + PHYS_OFFSET)  
#define __phys_to_virt(x) ((x) - PHYS_OFFSET + PAGE_OFFSET)  
#endif
```

关于页表的创建

- TEXT_OFFSET：以arm-linux-gcc的-D参数传递并定义
(缺省编译后，grep -r “TEXT_OFFSET” arch/asm可以搜索到)

... -DTEXT_OFFSET=0x00008000 ...

- ▶ arch/arm/Makefile中如下：

```
...
CPPFLAGS_vmlinux.lds = -DTEXT_OFFSET=$(TEXT_OFFSET)
...
textofs-y := 0x00008000
...
ifeq ($(CONFIG_ARCH_SA1100),y)
# SA1111 DMA bug: we don't want the kernel to live in precious DMA-able memory
textofs-$(CONFIG_SA1111) := 0x00208000
endif
...
textofs-$(CONFIG_ARCH_CLPS711X) := 0x00028000
...
# The byte offset of the kernel image in RAM from the start of RAM.
TEXT_OFFSET := $(textofs-y)
...
```

关于页表的创建

- ▶ 在arch/arm/kernel/vmlinux.lds.S中

```
SECTIONS {  
#ifdef CONFIG_XIP_KERNEL  
    . = XIP_VIRT_ADDR(CONFIG_XIP_PHYS_ADDR);  
#else  
    . = PAGE_OFFSET + TEXT_OFFSET;  
#endif  
    .text.head : {  
        _stext = .;  
        _sinittext = .;  
        *(.text.head)  
    }  
    ...
```


关于页表的创建

● __create_page_tables

- ❶ Clear the 16K level 1 swapper page table. (1级大页表： $4\text{GB}=1\text{MB}/\text{项} \times 4\text{K项}$ ， $4\text{K项} \times 4\text{B}/\text{项}=16\text{KB}$)
 - ❷ Create **identity mapping** for first MB of kernel to cater for the MMU enable.
一致映射：LA=PA
这里使用大页表，一项映射1MB
 - ❸ Setup the pagetables for our kernel **direct mapped region**
直接映射：LA-PAGE_OFFSET+PHYS_OFFSET=PA
 - ❹ Map first 1MB of ram in case it contains our boot params.
 - ❺ 返回调用者
- swapper_pg_dir在哪里？

关于页表的创建

● __create_page_tables

```
/*  
 * Setup the initial page tables. We only setup the barest  
 * amount which are required to get the kernel running, which  
 * generally means mapping in the kernel code.  
 *  
 * r8 = machinfo  
 * r9 = cpuid  
 * r10 = procinfo  
 *  
 * Returns:  
 * r0, r3, r6, r7 corrupted  
 * r4 = physical page table address  
 */  
.type __create_page_tables, %function  
__create_page_tables:  
    pgtbl r4 @ page table address
```

关于页表的创建

● __create_page_tables

- ① Clear the 16K level 1 swapper page table. (1级大页表: $4\text{GB} = 1\text{MB}/\text{项} * 4\text{K项}$, $4\text{K项} * 4\text{B}/\text{项} = 16\text{KB}$)

```
/*
 * Clear the 16K level 1 swapper page table
 */
mov r0, r4
mov r3, #0
add r6, r0, #0x4000
1:  str r3, [r0], #4
    str r3, [r0], #4
    str r3, [r0], #4
    str r3, [r0], #4
    teq r0, r6
    bne 1b

ldr r7, [r10, #PROCINFO_MM_MMUFLAGS] @ mm_mmuflags
```

关于页表的创建

- `__create_page_tables`

- ② Create **identity mapping** for first MB of kernel to cater for the MMU enable.

一致映射：LA=PA

```
/*  
 * Create identity mapping for first MB of kernel to  
 * cater for the MMU enable. This identity mapping  
 * will be removed by paging_init(). We use our current program  
 * counter to determine corresponding section base address.  
 */  
mov r6, pc, lsr #20 @ start of kernel section  
orr r3, r7, r6, lsl #20 @ flags + kernel base  
str r3, [r4, r6, lsl #2] @ identity mapping
```

这里使用大页表，一项映射1MB

关于页表的创建

- `__create_page_tables`

- ⑧ Setup the pagetables for our kernel **direct mapped** region
直接映射： $LA - PAGE_OFFSET + PHYS_OFFSET = PA$

```
/*
 * Now setup the pagetables for our kernel direct
 * mapped region.
 */
add r0, r4, #(KERNEL_START & 0xff000000) >> 18
str r3, [r0, #(KERNEL_START & 0x00f00000) >> 18]!
ldr r6, =(KERNEL_END - 1)
add r0, r0, #4
add r6, r4, r6, lsr #18
l:  cmp r0, r6
    add r3, r3, #1 << 20
    strls r3, [r0], #4
    bls lb
```

关于页表的创建

- `__create_page_tables`

- ④ Map first 1MB of ram in case it contains our boot params.

```
/*  
 * Then map first 1MB of ram in case it contains our boot params.  
 */  
add r0, r4, #PAGE_OFFSET >> 18  
orr r6, r7, #(PHYS_OFFSET & 0xff000000)  
.if (PHYS_OFFSET & 0x00f00000)  
orr r6, r6, #(PHYS_OFFSET & 0x00f00000)  
.endif  
str r6, [r0]
```

关于页表的创建

- `__create_page_tables`
 - ⑤ 返回调用者

```
mov pc, lr
```

关于页表的创建

● __create_page_tables

▶ swapper_pg_dir在哪里？

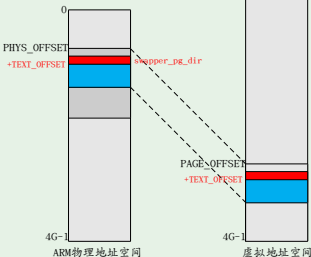
```
...
#define KERNEL_RAM_VADDR (PAGE_OFFSET + TEXT_OFFSET)
#define KERNEL_RAM_PADDR (PHYS_OFFSET + TEXT_OFFSET)
```

```
/*
 * swapper_pg_dir is the virtual address of the initial page table.
 * We place the page tables 16K below KERNEL_RAM_VADDR. Therefore, we must
 * make sure that KERNEL_RAM_VADDR is correctly set. Currently, we expect
 * the least significant 16 bits to be 0x8000, but we could probably
 * relax this restriction to KERNEL_RAM_VADDR >= PAGE_OFFSET + 0x4000.
 */
```

```
#if (KERNEL_RAM_VADDR & 0xffff) != 0x8000
#error KERNEL_RAM_VADDR must start at 0xFFFF8000
#endif
```

```
.globl swapper_pg_dir
.equ swapper_pg_dir, KERNEL_RAM_VADDR - 0x4000
```

```
.macro pgtbl, rd
1dr \rd, =(KERNEL_RAM_PADDR - 0x4000)
.endm
```



Outline

- 1 Linux的源代码组织
- 2 ARMlinux的编译过程和产生的映像
- 3 ARMlinux的启动分析
- 4 小结和作业

小结

- 1 Linux的源代码组织
- 2 ARMlinux的编译过程和产生的映像
 - 阅读Makefile
 - 源代码根目录下的vmlinux的生成
 - zImage的生成
 - bootpImage的生成
- 3 ARMlinux的启动分析
- 4 小结和作业

Project2

- 基于Arm的linux的启动分析

- ▶ 首先进行Makefile的分析，在分析过程中

- ★ 获知不同的启动方案

- ★ 获知典型启动方案的代码结构

- ▶ 选择一种启动方案，进行基于arm的Linux的启动分析，分析直到调用start_kernel为止

难度：bootpImage>zImage>Image

- 提供详细分析报告（要求：比课上的要详细）

Thanks !

The end.