임베디드 시스템 엔지니어를 위한 리눅스 커널 분석

남상규
http://ruby.medison.co.kr/~halite
선임 연구원
(주**)**메디슨
초음파 연구소

halite (at) medison.com

임베디드 시스템 엔지니어를 위한 리눅스 커널 분석

지은이 남상규

펴냄 $Date: 2002/05/21 01:10:08 $

이 문서는 리눅스 커널을 임베디드 시스템에 포팅하려는 엔지니어들을 위한 기본 지식 습득을 위해 만들어졌다. 리눅스 커널 자체의 원론적인 것 보다는 임베디드 시스템에 리눅스 커널을 포팅할 때 엔지니어가 리눅스 커널에 쉽게 접근하기 위한 정보나 혹은 방법을 제공하는 것이 목적이다. 그러므로 OS에 대한 이론 보다는 OS가 만들어지는 방법이나 부팅되는 순서 메모리에 적재되고 실행되는 순서 등에 대해 기술하고 더불어 커널을 만들기 위해 필요한 도구들의 사용법에 대해 알아본다.

원문은 *http://ruby.medison.co.kr/~halite* (http://ruby.medison.co.kr/~halite) 에있고 가장 먼저 업데이트될 것이다.

틀린 내용이 있을 수도 있다. 이런 것에 대해선 `<halite (at) medison.com>` 으로 연락 바란다.

차례

1. 리눅스 커널 컴파일하기 6
1.1. 리눅스 일반 6
1.2. LXR 11
1.3. 소스 코드 얻기 11
1.4. 소스 코드 풀기 11
1.5. 컴파일 준비 12
1.6. 커널 설정 13
1.7. 커널 컴파일 14
1.8. 커널 테스트 및 설치 14
2. Makefile 분석 17
2.1. 부팅 과정의 이해[1] 17
2.2. 커널 이미지 파일의 구조 18
2.3. bzimage가 만들어지는 과정 추적-Makefile 분석 20
2.4. bzImage가 만들어지는 과정 추적-Log 분석 49
2.5. 단계별 자세한 분석 56
3. 크로스 컴파일러 만들기 65
3.1. 크로스? 65
3.2. 툴체인 65
4. ARM 리눅스 72
4.1. ARM 프로세서 MMU(Memory Management Unit) 72
4.2. Assabet 보드용 커널 컴파일 80
4.3. ARM 리눅스 Makefile 분석 82
4.4. 소스 분석 101
5. 리눅스 커널 부팅 126
5.1. 커널 시작 126
5.2. lock_kernel() 129
5.3. setup_arch() 132
5.4. trap_init() 133
5.5. init_IRQ() 134
5.6. sched_init() 134
5.7. init() 135
5.8. dmesg 정리 137
6. 디바이스 드라이버 144
6.1. 디바이스 번호 144
6.2. 샘플 디바이스 드라이버 146
6.3. 모듈 동작의 이해 149
6.4. 알아야할 것 들 153
A. SEGA DreamCast Linux 158
A.1. LinuxSH 158
A.2. 드림캐스트에서 리눅스 실행해 보기 158

B. 리눅스에 시스템 콜 만들어 넣기 160
B.1. 시스템 콜의 흐름 160
B.2. IDT(Interrupt Descriptor Table) 160
B.3. 시스템 콜 테이블 160
B.4. 시스템 콜 추가 161
C. Inline Assembly 164
C.1. 인라인 어셈블리 기초 164
C.2. 사례 분석 171

표 목록

1-1. v2.4.16 커널 컴파일을 위한 최소 요구 사항 13
4-1. 섹션 디스크립터 필드 75
4-2. 페이지 디스크립터 필드 76
4-3. 2레벨 디스크립터 포맷 77
4-4. 2레벨 디스크립터 접근 권한 77
4-5. 메모리 시스템 속성 78
4-6. 캐시, 쓰기 버퍼 비트의 의미 79
4-7. 접근 권한 80
5-1. 예상 결과 130
5-2. 가능한 결과 130

그림 목록

2-1. mbr의 구조 17
2-2. bzImage의 구조 19
4-1. 변환 테이블 1레벨 디스크립터 접근 74
4-2. 1레벨 디스크립터 포맷 74
4-3. 섹션 변환 75
4-4. 2레벨 디스크립터 접근 76
4-5. 2레벨 디스크립터 포맷 77
4-6. 큰 페이지 변환 78
4-7. 작은 페이지 변환 78
4-9. ARM 리눅스 커널 이미지 메모리 맵 102
B-1. IDT 구조 160
B-2. fork()가 실행될 때의 흐름 161

1장. 리눅스 커널 컴파일하기

리눅스 시스템에 있어서 가장 중요한 커널을 컴파일해 돌려 보는 방법을 주변에서 가장 찾기 쉬운 시스템인 PC에서 경험해 보고 기본적인 방법 차례를 습득한다.

1.1. 리눅스 일반

리눅스의 배경이나 리눅스가 왜 좋은지 그리고 리눅스 소스 디렉토리의 구조는 어떻게 생겼는지 등을 기술한다.

1.1.1. 왜 리눅스인가?

리눅스가 왜 많이 사용되고 각광 받는가? 이에 대한 대답은 수 많이 존재할 것이지만 조금 간추려보면 대충 아래와 같은 몇몇 답으로 축약할 수 있을 것이다.

공짜

아마 이말 이상의 좋은 매력은 없을 것이다. 리눅스가 갖는 가장 좋은 매력 중 하나가 바로 공짜가 아닐까 생각해본다. 만약 리눅스가 유료였다면 현재 만큼 성장할 수 있었을까? 절대로 그렇지 않다고 생각한다. 게다가 리눅스는 GNU의 정신을 따르기 때문에 Copyleft란 말을 쓴다(copyright의 반대). 누구든 어떤 식으로든 사용 가능한 셈이다.

공짜로 성공하거나 유명해 진 것은 많지만 컴퓨터 분야에서 가장 유명한 것이 바로 리눅스 일 것이다. 이런 리눅의 배경은 다음과 같은 몇줄로 압축할 수 있다.

- 리누스 토발즈가 개발, Copyleft
- 1991년 0.01이 첫 발표됨
- 레드햇, 데비안, 슬랙웨어 등의 배포본이 있음.
- 많은 회사 들에 의해 지원됨

특징

- 멀티태스킹
- 멀티유저
- 멀티프로세서
- 많은 아키텍쳐 지원
- 페이징

- 하드 디스크 용 다이나믹 캐시
- 공유 라이브러리
- POSIX 1003.1 지원
- 여러 형태의 실행 파일 지원
- 진짜 386 프로텍티드 모드
- 수치프로세서 에뮬레이션
- 여러 나라 별 키보드, 언어 지원
- 여러 파일 시스템 지원
- TCP/IP, SLIP, PPP
- BSD 소켓
- System V IPC
- 버추얼 콘솔

단점

- 모놀리딕 커널-마이크로 커널에 비해 많은 부분이 커널레벨에서 구현된다.
- 초보에겐 어렵다-시스템 프로그래머를 위한 것이다
- 잘 짜여진 구조가 아니다(성능에 초점이 맞춰져 있다)

매력

- 많은 시스템에 의해 성능이 검증되어 있다
- 우리 스스로가 커널을 조정하거나 수정할 수 있다

1.1.2. 리눅스 소스 디렉토리 구조

리눅스 커널 소스는 압축된 것이 약 20MB 이상이다. 그러므로 수 많은 디렉토리와 수 많은 파일을 포함하고 있다. 필자가 완료한 몇몇 아주 큰 프로젝트에서도 전체 소스 코드를 압축해 봤자 10MB를 넘기기는 힘들었다. 그러나 양으로도 리눅스 커널은 이미 함부로 대하기 힘든 상대임을 나타내는데, 여기서는 리눅스 커널 소스의 압축을 풀면 생기는 디렉토리에 대해 알아보자.

한 가지 주의 해야할 점은 리눅스 커널이 지금 이 순간에도 계속 변하고 있으므로 아래의 내용이 언제 바뀌어 틀리게될지 모른다. 그러므로 항상 최신 버전을 소스 코드를 참조하기 바란다. 아래의 내용은 2.4.16 ~ 2.4.18의 것을 참조해 설명한 것이다.

모든 코드는 /usr/src/linux에서 시작한다고 가정한다. 이 글에선 이 위치를 $(TOPDIR)이라 표현한다.

Documents

커널에 관계된 문서들이 들어 있다. 커널을 분석하거나 할 때 필요한 정보는 여기를 먼저 보고나서 다른 곳을 찾는 것이 빠른 정보를 얻는 길이다.

kernel

커널의 핵심 코드,스스템콜,스케쥴러,시그널 핸들링

ipc

전통적 프로세스간의 통신,세마포어,공유메모리,메세지큐

lib

커널 라이브러리 함수(printk와 같은 것 들)

mm

버추얼 메모리 관리,페이징, 커널 메모리 관리

scripts

코드 사이의 의존성을 만들어주는 등의 스크립트나 실행 파일이 모여있다.

arch

아키텍쳐에 관계된 코드가 들어있다.

- alpha
- arm
- m68k
- mips
- ppc
- sparc
- i386
    - boot

      부트스프랩코드,메모리/디바이스 설정

    - kernel

      커널 시작점,컨텍스트 스위칭

    - lib

    - math-emu
    - mm

      각 아키텍쳐에 관계된 메모리 코드

- ...

fs

버추얼 파일 시스템 인터페이스,여러 파일시스템 지원

- coda
- ext2
- hpfs
- msdos
- nfs
- ntfs
- ufs
- ...

init

커널이 실행되기 위한 모든 코드,프로세스 0,프로세스 생성...

include

커널에 관계된 헤더 파일,asm-*은 아키텍쳐 관련,linux는 리눅스 커널 관련

- asm-alpha
- ...
- asm-i386
- linux
- net
- scsi
- video
- ...

net

많은 종류의 네트웍 프로토콜 지원,소켓 지원

- 802
- appletalk
- decnet
- ethernet
- ipv6
- unix
- sunrpc
- x25
- ...

driver

하드웨어에 대한 드라이버

- block

  하드디스크 같은 블럭 디바이스

- cdrom
- char

  시리얼 포트,모뎀,tty 같은 문자 디바이스

- net

  네트웍 카드

- pci

  PCI 버스 컨드롤

- pnp
- sbus
- scsi

  SCSI 카드

- sound

  사운드 카드

- viceo
- ...

1.2. LXR

리눅스 소스코드는 압축된 양만 해도 약 20MB 정도되는 방대한 양이다. 게다가 많은 양의 소스 코드에 산재해 있는 많은 함수나 정의를 찾기란 여간 힘든 것이 아니니다.

예를 들어 root 파일 시스템을 마운트하는 부분에 대해 분석하고 싶다면 과연 어디서 부터 시작해야할 것인가? 시작하는 곳을 찾는 것 까진 했다고 해도 그럼 역으로 어디서 마운트를 실행하는가?

소스 코드 분석에 가장 필요한 것이 내가 알고 싶은 함수/변수/정의가 어디에 있는가와 어디에서 불려 지는가 일것이다.

보통은 grep을 사용해 찾아보고 하나씩 열어 보는 방법을 동원하는데 이게 너무 힘들다. 그래서 리눅스 커널 관련 프로젝트 중에 하나인 lxr 프로젝트를 소개 한다.

lxr은 linux cross reference 정도의 의미로 이해하면 되겠고 리눅스 소스 코드의 모든 함수/변수/정의 등에 대한 크로스 레퍼런스를 온라인으로 제공한다. URL은 *http://lxr.linux.no* (http://lxr.linux.no)다. 여기에서 원하는 버전을 사용해 찾아가면 쉽게 접근할 수 있을 것이다.

예를 들어 커널의 시작인 start_kernel()을 분석하는 중에 처음 나오는 함수인 lock_kernel()이 어디에 있는지 알고 싶다면 lock_kernel()을 클릭해본다. 그럼 lock_kernel이 함수로 정의된 소스 코드, 매크로로 정의된 소스 코드 그리고 lock_kernel이 불린 위치가 차례로 열거 된다.

정의를 알고 싶으면 정의에 해당하는 것을 클릭하면 정의되어 있는 소스 코드로 가게되고 불린 위치를 원하면 불린 위치가 적힌 곳을 클리하면 해당 소스 코드로 이동하게 된다.

1.3. 소스 코드 얻기

리눅스 커널 소스 코드는 *http://www.kernel.org* 에서 구할 수 있다. 미러 사이트가 있으니 가장 빠른 곳에서 받으면 된다.

커널 소스는 각 버전 별로 구분된 디렉토리 내에 있고 패치와 각 버전의 완전한 소스코드가 같이 있다. 이 중에 원하는 버전의 완전한 코드를 받는다. 보통 gz과 bz2의 압축으로 되어 있는데 bz2가 약간 더 작기 때문에 필자는 이 타입을 선호한다.

1.4. 소스 코드 풀기

먼저 커널을 컴파일하기 위해 root의 권한을 가져야한다. 시스템에 root로 로그인하거나 su를 사용해 root가 된후 다음 절차를 시작한다.

보통의 커널은 /usr/src 밑에 위치하게 된다. 필자는 여기에 각 버전 번호를 가지고 디렉토리를 만들고 각각에 압축을 풀어 놓고 사용 중이다. 현재 사용되고 있는 커널 버전을 linux란 이름으로 링크시켜 놓고 사용 중이다. 즉 /usr/src/linux는 현재 사용되고 있는 커널을 가리키게 된다.

```
[root@localhost src]# ls -l
```

```
lrwxrwxrwx    1 root     root           12 12월 24 15:23 linux -> linux-
2.4.16
lrwxrwxrwx    1 root     root           14 12월 24 15:22 linux-2.4 -
> linux-2.4.7-10
drwxr-xr-x   14 root     root         8192  2월  5 11:00 linux-2.4.16
drwxr-xr-x   16 root     root         4096 12월 24 15:22 linux-2.4.7-10
drwxr-xr-x    7 root     root           40 12월 24 15:21 redhat
```

이렇게한 이유는 /usr/include에 linux와 asm 이란 리눅스 커널 소스코드 내의 디렉토리를 링크하게 되어 있는데 사용하는 커널 버전이 자주 바뀌거나 여러 커널을 같이 사용 중이라면 변하지 않는 패스를 사용하지 않는 한엔 linux와 asm의 링크가 자꾸 변하게되 불편을 감수해야하기 때문에 이 두 링크는 언제나 /usr/src/linux/include/linux와 /usr/src/linux/include/asm을 가리키도록 해놓고 /usr/src/linux만을 변경해 주도록 했다.

```
[root@localhost include]# ls -l linux asm
lrwxrwxrwx    1 root     root           26  1월 17 10:46 asm -
> /usr/src/linux/include/asm
lrwxrwxrwx    1 root     root           28  1월 17 10:45 linux -
> /usr/src/linux/include/linux
```

압축을 풀기 전에 주의해야할 것은 압축된 커널의 소스코드는 모두 linux란 이름의 디렉토리로 시작하므로 만약 /usr/src에 linux란 링크나 디렉토리가 있는 상태에서 여기에 압축을 풀어버리면 다른 버전의 코드를 엎어쓸 가능성이 있으므로 조심해야한다.

우선 /usr/src/linux의 링크를 없애고 tar를 사용해 압축을 푼 후 /usr/src/linux를 해당 버전으로 바꿔준다. 절차는 다음과 같다.

- cd /usr/src
- rm -f linux
- tar xvjf somewhere/linux-2.4.16.tar.bz2
- mv linux linux-2.4.16
- ln -s linux-2.4.16 linux

1.5. 컴파일 준비

소스 코드의 압축을 푼 후 /usr/src/linux로 이동해 혹시 있을지 모르는 것들을 지우고 컴파일이 제대로 되게 하기 위해 make mrproper 라고 명령을 실행한다. mrproper는 소스 코드를 처음 깔았을 때와 같은 상태로 돌려준다고 생각하면 된다. 만약 커널 설정 등을 해놓은 상태에서 이 명령을 실행하면 설정이 사라지기 때문에 필요한 때만 주의를 기울여 실행해야한다.

cd /usr/src/linux/Documentation 하고 vi Changes 를 한다. Changes엔 현재 커널을 컴파일하고 사용하기 위해 필요한 툴들의 버전 정보가 있다. Current Minimal Requirements 항목을 보고 자신의 시스템을 한번쯤 체크해 보기 바란다. 대부분의 경우는 만족할 것이다. 2.4.16 버전은 다음과 같은 사항을 만족해야한다.

표 **1-1. v2.4.16** 커널 컴파일을 위한 최소 요구 사항

툴	버전	확인 방법
Gnu C	2.95.3	gcc --version
Gnu make	3.77	make --version
binutils	2.9.1.0.25	ld -v
util-linux	2.10o	fdformat --version
modutils	2.4.2	insmod -V
e2fsprogs	1.19	tune2fs
reiserfsprogs	3.x.0j	reiserfsck 2>&1|grep reiserfsprogs
pcmcia-cs	3.1.21	cardmgr -V
PPP	2.4.0	pppd --version
isdn4k-utils	3.1pre1	isdnctrl 2>&1|grep version

1.6. 커널 설정

커널 설정은 몇 가지 방법이 있다. 고전적인 방법, 텍스트 기반의 메뉴를 이용하는 방법, X-Window 상에서 GUI를 이용하는 방법이다. 원하는 방법 중 하나를 택해 사용하면 된다. 필자는 손에 익은 대로 menuconfig를 주로 사용한다.

각각은 다음과 같이 실행된다.

- make config
- make menuconfig
- make xconfig

커널 설정에 관한 자세한 내용은 여기서 다루지 않는다. 커널 설정에 관한 자세한 것은 *http://www.kldp.org* 를 참조하기 바란다.

커널의 설정이 끝나면 /usr/src/linux/.config가 만들어진다. 이 파일의 내용을 보면 다음과 같다.

```
CONFIG_X86=y
...
CONFIG_MK7=y
...
CONFIG_MODULES=y
...
CONFIG_NET=y
...
# CONFIG_ACPI_DEBUG is not set
```

```
...
CONFIG_PARPORT=m
...
```

모두 CONFIG_로 시작하고 뒤에 각 항목의 이름이 붙는다. 예를 들어 위에서 CONFIG_MK7은 AMD의 Athlon CPU를 의미한다. 그리고 y 혹은 m 아니면 #으로 막혀져 있는 것이 있는데 y는 커널에 직접 포함 되도록 설정한 항목을 의미하고 m은 module로 설정한 것, #으로 막힌 것은 사용되지 않는 것을 의미한다.

1.7. 커널 컴파일

커널을 컴파일 해보자. 컴파일 순서는 다음과 같다.

- make dep
- make modules
- make bzImage
- make modules_install

make dep로 소스 파일과 헤더와의 의존성을 검사해 /usr/src/linux/.depend를 만든다.

make modules는 설정에서 module로 선택한 것들을 *.o의 형태로 만들어 준다.

make bzImage는 커널 자체를 만들어 준다. make zImage를 하는 경우 커널의 크기가 너무 커서 에러가 날 수도 있다. 이 경우엔 더 많은 부분을 모듈로 만들거나 bzImage를 사용해야한다.

make modules_install은 만들어진 module을 /lib/modules/2.4.16에 설치해 준다. 설치와 함께 depmod를 실행해 module 간의 의존성도 만들준다.

1.8. 커널 테스트 및 설치

커널 컴파일이 끝난 후 만들어진 커널이 제대로 동작하는지 확인해본 후 기본 커널로 설치해 사용해야한다. 그렇지 않고 무턱대고 원래의 잘돌아가던 커널을 대체해버리면 혹시라도 에러있는 커널인 경우엔 부팅이 안되는 위급한 사태가 발생하게 된다. 그러므로 만들어진 커널을 먼저 테스트 후 사용하기 바란다.

테스트 방법엔 2가지 정도를 추천한다. 하나는 플로피를 사용하는 것이고 나머지 하나는 LILO를 사용하는 것이다.

플로피를 사용하는 방법

> 플로피를 사용하기 위해 make zdisk를 사용해 만들어진 커널을 플로피에 담는다. 복사가 끝나면 플로피를 사용해 부팅해 정상적으로 동작하는지 확인한다. 가장 안전한 방법 중

하나로 생각되고 문제가 있으면 플로피를 제거하고 다시 부팅해 커널 설정 등을 다시 하고 테스트를 하면 된다.

LILO를 사용하는 방법

LILO를 사용해 하는 방법은 아래와 같이 /etc/lilo.conf를 우선 수정한다. 먼저 염두에 둬야할 것은 만들어진 커널 이미지를 어디에 무슨 이름으로 복사할 것인지 결정해 두고 일을 진행해야한다는 것이다. 필자의 경우 보통은 $(TOPDIR)/arch/i386/boot/bzImage[1]를 /boot/test_img로 복사하고 테스트 한다.

아래는 필자의 lilo.conf다. 기본 부팅용 항목을 그대로 복사하고 image와 label 만을 바꿔 하나더 등록해 부팅한다. 부팅 후엔 LILO에 test라 입력해 새로 만든 테스트 커널을 실험해 보면 된다.

```
prompt
timeout=50
default=linux
boot=/dev/hda
map=/boot/map
install=/boot/boot.b
#message=/boot/message
lba32
vga=0x030a

image=/boot/bzImage-2.4.16
label=linux
initrd=/boot/initrd-2.4.16.img
read-only
root=/dev/hda1
append="mem=nopentium hdd=ide-scsi"

image=/boot/test_img
label=test
initrd=/boot/initrd-2.4.16.img
read-only
root=/dev/hda1
append="mem=nopentium hdd=ide-scsi"
```

커널의 설치는 간단하다. 커널 이미지와 맵 파일을 복사하고 lilo.conf를 수정해주면 된다. 복사할 파일은 $(TOPDIR)/arch/i386/boot/bzImage와 $(TOPDIR)/System.map 두 개다.

필자는 bzImage를 /boot/bzImage-2.4.16과 같이 버전 이름을 사용해 복사하고 System.map을 /boot/Sysstem.map-2.4.16과 같이 복사한다. 그리고 /boot/System.map-2.4.16을 /boot/System.map 으로 심볼릭 링크를 만들어 준다. 커널은 /boot/System.map을 찾아 사용하기 때문이다.

주석

1. $(TOPDIR)은 커널 소스 디렉토리를 가리킨다. 즉 /usr/src/linux.

2장. Makefile 분석

필자는 리눅스를 공부하면서 처음엔 소스 코드를 찾아 스케쥴링부터 보고 다음엔 시스템 콜을 보고 하는 식으로 접근했다. 그러나 실제 임베디드 시스템에 적용하는데는 그런 단계가 먼저 필요한 것이 아니라 커널 이미지가 어떻게 구성됐으며 부팅 과정에서 어떤 식으로 실행되고 어떤 절차를 거쳐 리눅스란 os를 구성하는가가 먼저 필요하단 것을 느꼈다.

이런 생각은 자연스레 커널을 만들 때 어떤 식으로 커널이 만들어지는지를 알고 있어야 다른 임베디드 시스템에 커널을 만들어 넣을 때도 쉽게 접근할 수 있다는 생각을 갖게 만들었다.

이 장에선 커널 이미지 파일의 구조와 Makefile을 통해 커널이 만들어지는 과정을 추적해 본다.

2.1. 부팅 과정의 이해[1]

i386 계열의 pc를 중심으로 부팅 과정을 알아본다. 임베디드 시스템에선 PC와는 다른 부팅 과정이 필요할 것이지만 pc에서의 부팅 과정을 이해한 후엔 훨씬 접근이 쉬울 것이다. 또 대체적인 큰 항목들은 pc든 임베디드 시스템이든 같기 때문에 좋은 예가 될 것이다.

pc의 전원을 처음 넣으면 CPU는 ROM에서 BIOS 코드를 읽어 실행하기 시작한다. 처음 코드는 PC의 기본 적인 초기화를 할 것이고 이어 각종 HW의 초기화를 실행할 것이다.

기본 초기화가 끝나면 VGA를 통해 화면이 보이기 시작할 것이고 RAM 체크, HDD 인식, PnP 세팅 등을 실행한다. 여기 까지가 넓은 의미에서의 HW 초기화라고 봐야할 것이다.

모든 초기화와 세팅이 끝난 후는 기본적인 HW인 플로피나 하드디스크가 사용 가능한 상태가 된다. 이제 bios는 부팅 가능한 순서가 지정된 디바이스를 찾아 부팅을 시도한다. 부팅 가능하면그 디바이스의 첫 512 bytes를 읽어 실행한다.

하드 디스크의 첫 섹터를 MBR[2]이라 부르고 이 섹터만이 부팅에 사용된다. 하드 디스크가 여러 개 있더라도 정해진 첫 드라이브의 MBR만이 사용된다. 이 섹터는 로더 프로그램과 파티션 테이블 정보를 담고 있다. 로더는 일반적으로 부트 섹터를 읽어 부트한다. 실제로 MBR과 부트 섹터는 MBR이 파티션 정보를 담고 있다는 것을 제외하고는 기능적인 차이가 없다.

mbr의 첫 446 바이트(0x1BE)는 로더 프로그램이고 그 뒤 64 바이트는 파티션 테일블 정보를 담고 있다. 마지만 두 바이트는 매직 넘버를 갖고 있고 이 숫자는 이 섹터가 진짜 부트 섹터인지 판별할 때 사용된다.

그림 **2-1. mbr**의 구조

위 그림에서 보듯이 처음 읽혀지는 코드의 크기는 512 bytes 이므로 충분하다고는 볼수 없을 것이다. 그러므로 이 작은 용량엔 실제 코드를 읽어 실행하도록 하는 기능만을 넣는 것이 보통이다. 나머지는 해당 OS 파티션의 부트 섹터에 기록된다.

여기 까지가 일반적인 PC의 부팅과정이다. 이제 부터는 어떤 OS가 깔렸는지에 따라 부팅이 달라지게 된다. lilo와 같은 부트 로더가 실행되 원하는 OS를 부트한다.

이 책에서는 리눅스에 대해 다루므로 BIOS에 의해 처음 읽혀 실행되는 것이 LILO(혹은 GRUB)가 될 것이다. lilo를 설치할 때 LILO는 MBR에 자신을 위한 로더를 기록해 부팅할 때 LILO가 실행 되도록 할 것이다. lILO의 첫 스테이지 부트 섹터는 여러 다른 곳에서 LILO의 나머지 부분을 읽어 들인다.

다음은 필자의 lilo.conf로 lilo를 실행하면 install에 지정된 /boot/boot.b가 MBR에 써진다. boot.b는 lilo의 첫 스테이지와 두번째 스테이지가 묶여진 file로 첫 스테이지는 부트 섹터에 기록 되도록 512 바이트이고 두번째 스테이지는 그 나머지 부분이다. 실제로 필자의 컴퓨터 boot.b는 4566 바이트로 앞 512 바이트가 첫번째 스테이지, 나머지 부분인 4054 바이트가 두번째 스테이지가된다.

```
prompt
timeout=50
default=linux
boot=/dev/hda
map=/boot/map
install=/boot/boot.b
lba32
vga=0x030a

image=/boot/bzImage-2.4.16
label=linux
initrd=/boot/initrd-2.4.16.img
read-only
root=/dev/hda1
append="mem=nopentium hdd=ide-scsi"
```

그러나 /boot/boot.b를 hexedit와 같은 것으로 보면 앞부분에 있어야할 중요한 정보들이 비어 있는 것을 알 수 있다. 이 것은 lilo를 실행해 MBR에 boot.b를 등록할 때 정보가 채워지게 된다.

일단 bios가 mbr을 읽어 LILO를 기동하면 LILO는 lilo.conf를 사용해 등록된 메뉴를 갖고 사용자의 입력에 따라 해당 os를 시작하게 된다. 필자의 lilo.conf에 의하면 선택 가능한 OS는 "linux"로 커널 이미지는 /boot/bzImage-2.4.16임을 알수 있다. 즉 LILO는 사용자가 "linux"를 선택했을 때 /boot/bzImage-2.4.16를 읽어들이고 실행해 주는 역할을 한다.

2.2. 커널 이미지 파일의 구조

이미 커널을 컴파일 해본 사람은 최종 커널 이미지 파일이 압축되어 있단 것을 알 수있을 것이다. [3]이제 LILO가 메모리에 커널을 읽어 올리고 실행해 주기 까지의 과정을 살펴볼 것이다.

그에 앞서 커널 이미지 파일의 구조에 대해 먼저 알아본다. 구조를 알아야 LILO가 커널을 어떤 식으로 부팅하게 해주는지 이해가 빠를 것이다.

zImage나 bzImage나 구조는 같다. 단지 메모리에 올려지는 위치나 동작 방식이 약간 차이가 있기 때문에 이것이 감안된 각각에 맞는 코드가 사용될 뿐이다.

그림 **2-2. bzImage**의 구조

그림 2-2에서 회색 부분이 진짜 리눅스 커널이고 압축되어 있는 상태다. 여기에 압축을 풀어주기 위한 'head + misc'가 앞부분에 붙어 있고 다시 이 한 덩어리의 이미지에 메모리에 올려진 커널의 압축이 제대로 풀리도록 미리 준비하는 "setup"과 부팅할 때 사용되는 "bootsect"가 붙어 최종 커널 이미지 파일을 만든다.

bbootsect나 bsetup의 앞에 붙은 b는 bzImage의 앞에 붙은 b와 같은 의미로 "big kernel"을 의미한다. bbootsect는 플로피로 부팅될 때 즉 부트로더가 없이 커널이 직접 읽혀져 부팅될 때 필요한 부트 섹터다. lilo에 의해 부팅되는 경우는 필요 없는 부분이다.

2.2.1. 커널의 부팅

lilo에 의해 부팅이 시작되면 LILO는 bzImage를 하드 디스크에서 읽어 메모리에 올려 놓고 LILO에 의한 부팅일 경우 필요 없는 bbootsect를 건너뛴 bsetup에서부터 실행되도록 해준다. LILO의 역할은 bsetup에 실행 권을 넘겨주는데 까지다.

실행된 bsetup은 메모리 세팅을 마치고 압축된 커널 이미지의 압축을 풀기 위한 코드로 실행을 옮긴다. "head + misc"로 표시된 부분이고 이곳이 실행되면 piggy.o는 압축이 풀려 실행 가능한 리눅스 커널이 메모리에 존재하게 된다. 압축이 풀릴 때 화면에 "Uncompressing Linux... "란 메시지가 출력된다.

그러나 압축이 풀렸다고 해서 바로 커널을 실행하는 것은 아니고 메모리 낭비를 막기 위한 정리를 한번 다시 하고 나서 커널이 실행된다. 압축이 풀리고나면 처음 LILO에 의해 읽혀진 커널 이미지는 필요 없게 된다. 그러므로 이 부분을 내버려두면 그만큼 메모리 낭비이므로 압축 풀린 커널 이미지를 옮겨와 되도록 많은 메모리를 사용하도록 한 후 커널을 실행해 준다. 커널이 실행되기 시작하면 "ok, booting the kernel."이 출려되고 커널에 의한 출력이 화면에 나타나게 된다.

2.2.2. zimage와 bzImage의 차이

zimage와 bzImage는 무슨 차이가 있는걸까? 필자는 처음에 z와 b의 의미 때문에 gzip으로 압축하거나 아니면 bzip2로 압축한 것의 차이인줄 알았지만 압축은 gzip으로 같고 단지 z는 압축했단 의미고 b는 'big kernel'이란 뜻인걸 알았다. 왜 이렇게 나눠졌는가?

이미 1.7절에서 언급했던 것 처럼 커널의 크기가 너무 커서 압축 후에도 일정 크기를 넘어가면 zImage 대신 bzImage를 사용해야한다고 했는데 이유는 다음과 같다.

pc가 처음 만들어질 땐 OS로 도스가 사용됐고 이 때 M$의 유명한 분이 640KB면 충분하다고 했단 소릴 들은적이 있을 것이다. 처음 PC가 만들어질 때의 CPU는 8086으로 16bit CPU 였다. 이 프로세서가 지원하는 최대의 메모리는 1MB였기 때문에 모든 어드레스 스페이스가 1MB 내로 제한됐다. 그러므로 램을 640kb 사용하고 나머지 영역엔 MGA, VGA와 같은 다른 디바이스를 할당해 줬다.

문제는 여기서 시작되는데 AT시절의 PC 기본 구조는 현재까지도 계속 유지되고 있기 때문에 PC가 처음 부팅되면 하위 1MB 만을 사용한다고 생각하면 된다. 보호모드라고 알고 있는 386 이상의 cpu가 가진 기능을 사용하지 않고 리얼모드란 8086 호환 모드를 사용하기 때문인데 이는 OS가 보호모드를 사용할 상태를 만들고 전환하기 전까지는 계속 리얼모드로 남아있기 때문이다.

리눅스 커널의 크기가 커서 커널을 읽어들이는 프로그램 크기나 시스템에서 사용되는 약간의 메모리를 제외한 나머지 램의 빈공간에 읽어 들이지 못하면 하위 1MB가 아니라 그 이상의 연속된 메모리에 커널을 읽어 들이고 압축을 푸는 등의 일을 해야할 것이다. 반대로 남은 용량에 커널이 들어갈 수 있다면 당연히 읽어 들이고 압축을 풀면 끝날 것이고...

이렇게 메모리에 처음 적재되고 압축 풀리고 하는 절차와 위치가 다르기 때문에 zImage와 bzImage오 나뉜 것이고 커널 이미지 파일의 앞부분 bootsect와 setup이 각각에 따라 맞는 것으로 합쳐지게된다. 그리고 bzimage의 경우 하위 1M는 사용하지 못하는데 리눅스에선 그렇다!

컴파일 단계에서 make zImage 했을 경우 System is too big. Try using bzImage or modules. 라고 에러가 난다면 더 많은 부분을 module로 만들거나 bzImage를 사용해야한다.

2.3. bzimage가 만들어지는 과정 추적-Makefile 분석

bzimage가 만들어지는 과정을 살펴 보고 이를 따라가면서 Makefile의 자세한 내용을 알아본다. 정확한 것은 2.4.2절을 참조하기 바란다.

시작은 물론 $(topDIR)/Makefile로 부터 시작한다.

커널 makefile은 몇 부분으로 나눌 수 있다.

1. 기본 정보 정의
2. 커널 설정
3. 커널 소스 의존성 만들기
4. 모듈 만들기

5. 커널 실행 파일 만들기
6. 모듈 설치하기

각 부분이 명확하게 구분되는 것은 아니지만 make의 동작을 이해하는 사람이라면 대충 구분을 지어 이해할 수 있을 것이다. 구분은 커널을 컴파일 하는 절차에 따라 나눈 것으로 이해 하면 쉬울 것이다. 일반적으로 많이 쓰이지 않는 부분은 넘어가고 중요한 부분만을 자세히 이해하자.

시작에 앞서 사용되는 Makefile들을 설명해 놓는다. 이것 들을 참조로 추적해나가므로 시작하기에 앞서 한번 쯤 훑어 보는 것도 좋을 것이다.

2.3.1. $(topDIR)/Makefile

아래 makefile에 (1), (2)와 같이 표시된 것은 아래 줄에 대한 설명을 달아 놓은 것으로 Makefile의 끝 부분에 붙어 있는 설명을 참조해가면서 분석하면되겠다.

```
version = 2
patchlevel = 4
sublevel = 16
extraversion =

kernelrelease=$(veRSION).$(PATCHLEVEL).$(SUBLEVEL)$(EXTRAVERSION)

<10122>
arch := $(shell uname -m | sed -e s/i.86/i386/ -e s/sun4u/sparc64/ -
e s/arm.*/arm/ -e s/sa110/arm/)

<10123>
kernelpath=kernel-$(shell echo $(KERNELRELEASE) | sed -e "s/-//")

<10124>
config-shell := $(shell if [ -x "$$BASH" ]; then echo $$BASH; \
  else if [ -x /bin/bash ]; then echo /bin/bash; \
  else echo sh; fi ; fi)

<10125>
topdir := $(shell /bin/pwd)

hpath    = $(topdIR)/include
findhpath = $(hpatH)/asm $(HPATH)/linux $(HPATH)/scsi $(HPATH)/net

<10126>
hostcc   = gcc
hostcflags = -wall -Wstrict-prototypes -O2 -fomit-frame-pointer

<10127>
cross-compile  =
```

```
<10128>
#
# include the make variables (CC, 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
makefiles = $(topdIR)/.config
genksyms = /sbin/genksyms
depmod = /sbin/depmod
modflags = -dmoduLE
cflags-kernel =
perl = perl

<10129>
export version paTCHLEVEL SUBLEVEL EXTRAVERSION KERNELRELEASE ARCH \
config-shell topDIR HPATH HOSTCC HOSTCFLAGS CROSS-COMPILE AS LD CC \
cpp ar nm strip OBJCOPY OBJDUMP MAKE MAKEFILES GENKSYMS MODFLAGS PERL

all: do-it-all

<10130>
#
# make "config" the default target if there is no configuration file or
# "depend" the target if there is no top-level dependency information.
#
ifeq (.config,$(wildcard .config))
include .config
ifeq (.depend,$(wildcard .depend))
include .depend
do-it-all: version vmlinux
else
(10)
configuration = depend
do-it-all: depend
endif
else
(11)
configuration = config
do-it-all: config
endif

(12)
```

```
#
# install-path specifies where to place the updated kernel and system map
# images.  uncomment if you want to place them anywhere other than root.
#
#export install-paTH=/boot
```

(13)

```
#
# install-mod-path specifies a prefix to MODLIB for module directory
# relocations required by build roots.  This is not defined in the
# makefile but the arguement can be passed to make if needed.
#
modlib := $(instaLL-MOD-PATH)/lib/modules/$(KERNELRELEASE)
export modlib

#
# standard cflags
#
cppflags := -d--keRNEL-- -I$(HPATH)

cflags := $(cppflaGS) -Wall -Wstrict-prototypes -Wno-trigraphs -O2 \
  -fomit-frame-pointer -fno-strict-aliasing -fno-common
aflags := -d--asseMBLY-- $(CPPFLAGS)
```

(14)

```
#
# root-dev specifies the default root-device when making the image.
# this can be either FLOPPY, CURRENT, /dev/xxxx or empty, in which case
# the default of fLOPPY is used by 'build'.
# this is i386 specific.
#
export root-dev = CURRENT
```

(15)

```
#
# if you want to preset the SVGA mode, uncomment the next line and
# set svga-mode to whatever number you want.
# set it to -dsvga-MODE=NORMAL-VGA if you just want the EGA/VGA mode.
# the number is the same as you would ordinarily press at bootup.
# this is i386 specific.
#
export svga-mode = -DSVGA-MODE=NORMAL-VGA
```

(16)

```
#
# if you want the RAM disk device, define this to be the size in blocks.
# this is i386 specific.
#
#export ramdisk = -DRAMDISK=512
```

(17)

```
core-files =kernel/kernel.o mm/mm.o fs/fs.o ipc/ipc.o
networks =net/network.o

libs =$(topdir)/lib/lib.a
subdirs =kernel drivers mm fs net ipc lib
```

(18)

```
drivers-n :=
drivers-y :=
drivers-m :=
drivers-  :=

drivers-$(config-aCPI) += drivers/acpi/acpi.o
drivers-$(config-pARPORT) += drivers/parport/driver.o
drivers-y += drivers/char/char.o \
drivers/block/block.o \
drivers/misc/misc.o \
drivers/net/net.o \
drivers/media/media.o
drivers-$(config-aGP) += drivers/char/agp/agp.o
drivers-$(config-dRM) += drivers/char/drm/drm.o
drivers-$(config-nUBUS) += drivers/nubus/nubus.a
drivers-$(config-iSDN) += drivers/isdn/isdn.a
drivers-$(config-nET-FC) += drivers/net/fc/fc.o
drivers-$(config-aPPLETALK) += drivers/net/appletalk/appletalk.o
drivers-$(config-tR) += drivers/net/tokenring/tr.o
drivers-$(config-wAN) += drivers/net/wan/wan.o
drivers-$(config-aRCNET) += drivers/net/arcnet/arcnetdrv.o
drivers-$(config-aTM) += drivers/atm/atm.o
drivers-$(config-iDE) += drivers/ide/idedriver.o
drivers-$(config-fC4) += drivers/fc4/fc4.a
drivers-$(config-sCSI) += drivers/scsi/scsidrv.o
drivers-$(config-fUSION-BOOT) += drivers/message/fusion/fusion.o
drivers-$(config-iEEE1394) += drivers/ieee1394/ieee1394drv.o

ifneq ($(config-cd-NO-IDESCSI)$(CONFIG-BLK-DEV-IDECD)$(CONFIG-BLK-DEV-
SR)$(CONFIG-PARIDE-PCD),)
drivers-y += drivers/cdrom/driver.o
endif

drivers-$(config-sOUND) += drivers/sound/sounddrivers.o
drivers-$(config-pCI) += drivers/pci/driver.o
drivers-$(config-mTD) += drivers/mtd/mtdlink.o
drivers-$(config-pCMCIA) += drivers/pcmcia/pcmcia.o
drivers-$(config-nET-PCMCIA) += drivers/net/pcmcia/pcmcia-net.o
drivers-$(config-nET-WIRELESS) += drivers/net/wireless/wireless-net.o
drivers-$(config-pCMCIA-CHRDEV) += drivers/char/pcmcia/pcmcia-char.o
```

```
drivers-$(config-dIO) += drivers/dio/dio.a
drivers-$(config-sBUS) += drivers/sbus/sbus-all.o
drivers-$(config-zORRO) += drivers/zorro/driver.o
drivers-$(config-fC4) += drivers/fc4/fc4.a
drivers-$(config-aLL-PPC) += drivers/macintosh/macintosh.o
drivers-$(config-mAC) += drivers/macintosh/macintosh.o
drivers-$(config-iSAPNP) += drivers/pnp/pnp.o
drivers-$(config-sGI-IP22) += drivers/sgi/sgi.a
drivers-$(config-vT) += drivers/video/video.o
drivers-$(config-pARIDE) += drivers/block/paride/paride.a
drivers-$(config-hAMRADIO) += drivers/net/hamradio/hamradio.o
drivers-$(config-tC) += drivers/tc/tc.a
drivers-$(config-uSB) += drivers/usb/usbdrv.o
drivers-$(config-iNPUT) += drivers/input/inputdrv.o
drivers-$(config-i2O) += drivers/message/i2o/i2o.o
drivers-$(config-iRDA) += drivers/net/irda/irda.o
drivers-$(config-i2C) += drivers/i2c/i2c.o
drivers-$(config-pHONE) += drivers/telephony/telephony.o
drivers-$(config-mD) += drivers/md/mddev.o
drivers-$(config-bLUEZ) += drivers/bluetooth/bluetooth.o
drivers-$(config-hOTPLUG-PCI) += drivers/hotplug/vmlinux-obj.o
```

(19)

```
drivers := $(driveRS-y)
```

(20)

```
# files removed with ’make clean’
clean-files = \
kernel/ksyms.lst include/linux/compile.h \
vmlinux system.map \
.tmp* \
drivers/char/consolemap-deftbl.c drivers/video/promcon-tbl.c \
drivers/char/conmakehash \
drivers/char/drm/*-mod.c \
drivers/pci/devlist.h drivers/pci/classlist.h drivers/pci/gen-devlist \
drivers/zorro/devlist.h drivers/zorro/gen-devlist \
drivers/sound/bin2hex drivers/sound/hex2hex \
drivers/atm/fore200e-mkfirm drivers/atm/{pca,sba}*{.bin,.bin1,.bin2} \
drivers/scsi/aic7xxx/aicasm/aicasm-gram.c \
drivers/scsi/aic7xxx/aicasm/aicasm-scan.c \
drivers/scsi/aic7xxx/aicasm/y.tab.h \
drivers/scsi/aic7xxx/aicasm/aicasm \
drivers/scsi/53c700-mem.c \
net/khttpd/make-times-h \
net/khttpd/times.h \
submenu*
# directories removed with ’make clean’
clean-dirs = \
modules
```

(21)

```
# files removed with 'make mrproper'
mrproper-files = \
include/linux/autoconf.h include/linux/version.h \
drivers/net/hamradio/soundmodem/sm-tbl-{afsk1200,afsk2666,fsk9600}.h \
drivers/net/hamradio/soundmodem/sm-tbl-{hapn4800,psk4800}.h \
drivers/net/hamradio/soundmodem/sm-tbl-{afsk2400-7,afsk2400-8}.h \
drivers/net/hamradio/soundmodem/gentbl \
drivers/sound/*-boot.h drivers/sound/.*.boot \
drivers/sound/msndinit.c \
drivers/sound/msndperm.c \
drivers/sound/pndsperm.c \
drivers/sound/pndspini.c \
drivers/atm/fore200e-*-fw.c drivers/atm/.fore200e-*.fw \
.version .config* config.in config.old \
scripts/tkparse scripts/kconfig.tk scripts/kconfig.tmp \
scripts/lxdialog/*.o scripts/lxdialog/lxdialog \
.menuconfig.log \
include/asm \
.hdepend scripts/mkdep scripts/split-include scripts/docproc \
$(topdir)/include/linux/modversions.h \
kernel.spec

# directories removed with 'make mrproper'
mrproper-dirs = \
include/config \
$(topdir)/include/linux/modules
```

(22)

```
include arch/$(arcH)/Makefile
```

(23)

```
export cppflags cFLAGS AFLAGS

export networks dRIVERS LIBS HEAD LDFLAGS LINKFLAGS MAKEBOOT ASFLAGS
```

(24)

```
.s.s:
$(cpp) $(aflags) -traditional -o $*.s $<.s.o:
$(cc) $(aflags) -traditional -c -o $*.o $<
version: dummy
@rm -f include/linux/compile.h

boot: vmlinux
@$(make) cflags="$(CFLAGS) $(CFLAGS-KERNEL)" -C arch/$(ARCH)/boot
```

(25)

```
vmlinux: include/linux/version.h $(CONFIGURA-
TION) init/main.o init/version.o linuxsubdirs
$(ld) $(linkflagS) $(HEAD) init/main.o init/version.o \
--start-group \
$(core-files) \
$(drivers) \
$(networks) \
$(libs) \
--end-group \
-o vmlinux
$(nm) vmlinux | grep -v ’\(com-
piled\)\|\(\.o$$\)\|\( [aUw] \)\|\(\.\.ng$$\)\|\(LASH[RL]DI\)’ | sort > System.map
```

(26)

```
symlinks:
rm -f include/asm
( cd include ; ln -sf asm-$(ARCH) asm)
@if [ ! -d include/linux/modules ]; then \
mkdir include/linux/modules; \
fi
```

(27)

```
oldconfig: symlinks
$(config-shell) scripts/Configure -d arch/$(ARCH)/config.in

xconfig: symlinks
$(make) -c scripts kconfig.tk
wish -f scripts/kconfig.tk

menuconfig: include/linux/version.h symlinks
$(make) -c scripts/lxdialog all
$(config-shell) scripts/Menuconfig arch/$(ARCH)/config.in

config: symlinks
$(config-shell) scripts/Configure arch/$(ARCH)/config.in

include/config/marKER: scripts/split-include include/linux/autoconf.h
scripts/split-include include/linux/autoconf.h include/config
@ touch include/config/MARKER
```

(28)

```
linuxsubdirs: $(patsubst %, -dir-%, $(SUBDIRS))
```

(29)

```
$(patsubst %, -dir-
%, $(SUBDIRS)) : dummy include/linux/version.h include/config/MARKER
$(make) cflags="$(CFLAGS) $(CFLAGS-KERNEL)" -C $(patsubst -dir-%, %, $@)

$(topdir)/include/linux/version.h: include/linux/version.h
```

```
$(topdir)/include/linux/compile.h: include/linux/compile.h

newversion:
. scripts/mkversion > .tmpversion
@mv -f .tmpversion .version
```

(30)

```
include/linux/compile.h: $(CONFIGURATION) in-
clude/linux/version.h newversion
@echo -n \#define UTS-VERSION \"\#`cat .version` > .ver
@if [ -n "$(confIG-SMP)" ] ; then echo -n " SMP" >> .ver; fi
@if [ -f .name ]; then  echo -n \-`cat .name` >> .ver; fi
@echo ’ ’`date`’"’ >> .ver
@echo \#define lINUX-COMPILE-TIME \"`date +%T`\" >> .ver
@echo \#define lINUX-COMPILE-BY \"`whoami`\" >> .ver
@echo \#define lINUX-COMPILE-HOST \"`hostname`\" >> .ver
@if [ -x /bin/dnsdomainname ]; then \
   echo \#define LINUX-COMPILE-DOMAIN \"`dnsdomainname`\"; \
 elif [ -x /bin/domainname ]; then \
   echo \#define LINUX-COMPILE-DOMAIN \"`domainname`\"; \
 else \
   echo \#define LINUX-COMPILE-DOMAIN ; \
 fi >> .ver
@echo \#define lINUX-COMPILER \"`$(CC) $(CFLAGS) -v 2>&1 | tail -
1`\" >> .ver
@mv -f .ver $@
```

(31)

```
include/linux/version.h: ./Makefile
@echo \#define uTS-RELEASE \"$(KERNELRELEASE)\" > .ver
@echo \#define lINUX-VERSION-CODE ‘expr $(VERSION) \\* 65536 + $(PATCH-
LEVEL) \\* 256 + $(SUBLEVEL)` >> .ver
@echo ’#define kERNEL-
VERSION(a,b,c) (((a) << 16) + ((b) << 8) + (c))’ >> .ver
@mv -f .ver $@

init/version.o: init/version.c in-
clude/linux/compile.h include/config/MARKER
$(cc) $(cflags) $(CFLAGS-KERNEL) -DUTS-MACHINE=’"$(ARCH)"’ -c -
o init/version.o init/version.c
```

(32)

```
init/main.o: init/main.c include/config/MARKER
$(cc) $(cflags) $(CFLAGS-KERNEL) $(PROFILING) -c -o $*.o $<
```

(33)

```
fs lib mm ipc kernel drivers net: dummy
$(make) cflags="$(CFLAGS) $(CFLAGS-KERNEL)" $(subst $@, -dir-$@, $@)
```

(34)

```
# emacs, vi용 tag를 만든다.
tags: dummy
etags ‘find include/asm-$(ARCH) -name ’*.h’‘
find include -type d \( -name "asm-*" -o -name config \) -prune -o -
name ’*.h’ -print | xargs etags -a
find $(subdirs) init -name ’*.[ch]’ | xargs etags -a

# exuberant ctags works better with -I
tags: dummy
ctagsf=‘ctags --version | grep -i exuberant >/dev/null && echo "-I --
initdata,--exitdata,EXPORT-SYMBOL,EXPORT-SYMBOL-NOVERS"‘; \
ctags $$ctagsf ‘find include/asm-$(ARCH) -name ’*.h’‘ && \ find include -
type d \( -name "asm-*" -o -name config \) -prune -o -name ’*.h’ -
print | xargs ctags $$CTAGSF -a && \
find $(subdirs) init -name ’*.[ch]’ | xargs ctags $$CTAGSF -a

ifdef config-modulES
ifdef config-modveRSIONS
modflags += -dmodvERSIONS -include $(HPATH)/linux/modversions.h
endif
```

(35)

```
.phony: modules
modules: $(patsubst %, -mod-%, $(SUBDIRS))

.phony: $(patsubst %, -mod-%, $(SUBDIRS))
$(patsubst %, -mod-
%, $(SUBDIRS)) : include/linux/version.h include/config/MARKER
$(make) -c $(patsubst -mod-
%, %, $@) CFLAGS="$(CFLAGS) $(MODFLAGS)" MAKING-MODULES=1 modules

.phony: modules-install
modules-install: -modinst- $(patsubst %, -modinst-%, $(SUBDIRS)) -modinst-
post

.phony: -modinst-
-modinst-:
@rm -rf $(modlib)/kernel
@rm -f $(modlib)/build
@mkdir -p $(modlIB)/kernel
@ln -s $(topdir) $(MODLIB)/build
```

(36)

```
# if system.map exists, run depmod.  This deliberately does not have a
# dependency on system.map since that would run the dependency tree on
# vmlinux.  this depmod is only for convenience to give the initial
# boot a modules.dep even before / is mounted read-write.  However the
# boot script depmod is the master version.
ifeq "$(strip $(inSTALL-MOD-PATH))" ""
```

```
depmod-opts :=
else
depmod-opts := -b $(INSTALL-MOD-PATH) -r
endif
.phony: -modinst-post
-modinst-post: -modinst-post-pcmcia
if [ -r system.map ]; then $(DEPMOD) -ae -F System.map $(depmod-
opts) $(KERNELRELEASE); fi

# backwards compatibilty symlinks for people still using old versions
# of pcmcia-cs with hard coded pathnames on insmod.  Remove
# -modinst-post-pcmcia for kernel 2.4.1.
.phony: -modinst-post-pcmcia
-modinst-post-pcmcia:
cd $(modlib); \
mkdir -p pcmcia; \
find kernel -path ’*/pcmcia/*’ -name ’*.o’ | xargs -i -r ln -
sf ../{} pcmcia

.phony: $(patsubst %, -modinst-%, $(SUBDIRS))
$(patsubst %, -modinst-%, $(SUBDIRS)) :
$(make) -c $(patsubst -modinst-%, %, $@) modules-install

# modules disabled....

else
modules modules-install: dummy
@echo
@echo "the present kernel configuration has modules disabled."
@echo "type ’make config’ and enable loadable module support."
@echo "then build a kernel with module support enabled."
@echo
@exit 1
endif

clean: archclean
find . \( -name ’*.[oas]’ -o -name core -o -name ’.*.flags’ \) -type f -
print \
| grep -v lxdialog/ | xargs rm -f
rm -f $(clean-fiLES)
rm -rf $(clean-dIRS)
$(make) -c documentation/DocBook clean

mrproper: clean archmrproper
find . \( -size 0 -o -name .depend \) -type f -print | xargs rm -f
rm -f $(mrproper-FILES)
rm -rf $(mrpropeR-DIRS)
$(make) -c documentation/DocBook mrproper
```

(37)

```
distclean: mrproper
rm -f core ‘find . \( -not -type d \) -and \
\( -name ’*.orig’ -o -name ’*.rej’ -o -name ’*~’ \
-o -name ’*.bak’ -o -name ’#*#’ -o -name ’.*.orig’ \
-o -name ’.*.rej’ -o -name ’.SUMS’ -o -size 0 \) -type f -print‘ TAGS tags

backup: mrproper
cd .. && tar cf - linux/ | gzip -9 > backup.gz
sync

sgmldocs:
chmod 755 $(topdIR)/scripts/docgen
chmod 755 $(topdIR)/scripts/gen-all-syms
chmod 755 $(topdIR)/scripts/kernel-doc
$(make) -c $(topDIR)/Documentation/DocBook books

psdocs: sgmldocs
$(make) -c documentation/DocBook ps

pdfdocs: sgmldocs
$(make) -c documentation/DocBook pdf

htmldocs: sgmldocs
$(make) -c documentation/DocBook html

sums:
find . -type f -print | sort | xargs sum > .SUMS

dep-files: scripts/mkdep archdep include/linux/version.h
scripts/mkdep -- init/*.c > .depend
scripts/mkdep -- ‘find $(FINDHPATH) -name SCCS -prune -o -follow -
name \*.h ! -name modversions.h -print‘ > .hdepend
$(make) $(patsubst %,-sfdep-%,$(SUBDIRS)) -FASTDEP-ALL-SUB-
DIRS="$(SUBDIRS)"
ifdef config-modveRSIONS
$(make) update-modverfile
endif

ifdef config-modveRSIONS
modverfile := $(toPDIR)/include/linux/modversions.h
else
modverfile :=
endif
export modverfile

depend dep: dep-files

checkconfig:
```

```
find * -name '*.[hcS]' -type f -print | sort | xargs $(PERL) -
w scripts/checkconfig.pl

checkhelp:
find * -name [cc]onfig.in -print | sort | xargs $(PERL) -
w scripts/checkhelp.pl

checkincludes:
find * -name '*.[hcS]' -type f -print | sort | xargs $(PERL) -
w scripts/checkincludes.pl

ifdef configuratioN
..$(configuration):
@echo
@echo "you have a bad or nonexistent" .$(CONFIGURATION) ": run-
ning 'make" $(CONFIGURATION)"'"
@echo
$(make) $(configURATION)
@echo
@echo "successful. Try re-making (ignore the error that follows)"
@echo
exit 1

#dummy: ..$(configURATION)
dummy:

else

dummy:

endif
```

(38)

```
include rules.make
```

(39)

```
#
# this generates dependencies for the .h files.
#

scripts/mkdep: scripts/mkdep.c
$(hostcc) $(hostCFLAGS) -o scripts/mkdep scripts/mkdep.c

scripts/split-include: scripts/split-include.c
$(hostcc) $(hostCFLAGS) -o scripts/split-include scripts/split-include.c
```

(40)

```
#
# rpm target
```

```
#
# if you do a make spec before packing the tarball you can rpm -ta it
#
spec:
. scripts/mkspec > kernel.spec

#
# build a tar ball, generate an rpm from it and pack the result
# there arw two bits of magic here
# 1) the use of /. to avoid tar packing just the symlink
# 2) removing the .dep files as they have source paths in them that
#    will become invalid
#
rpm: clean spec
find . \( -size 0 -o -name .depend -o -name .hdepend \) -type f -
print | xargs rm -f
set -e; \
cd $(topdir)/.. ; \
ln -sf $(topdir) $(KERNELPATH) ; \
tar -cvz --exclude CVS -f $(KERNELPATH).tar.gz $(KERNELPATH)/. ; \
rm $(kernelpath) ; \
cd $(topdir) ; \
. scripts/mkversion > .version ; \
rpm -ta $(topdir)/../$(KERNELPATH).tar.gz ; \
rm $(topdir)/../$(KERNELPATH).tar.gz
```

<10122> arch는 아래와 같이 uname으로 얻어지는 아키텍쳐를 지칭하는 값을 갖는다. intel 계열에선 i386이 되고 ARM 계열에선 arm이 된다.

<10123> kernelPATH는 kernel-2.4.16이 된다.

<10124> 현재 사용 중인 shell을 알아낸다.

<10125> topdir은 커널 소스 코드가 들어있는 최상위 디렉토리

<10126> host가 붙은 것은 커널이 cross compile 되서 다른 아키텍쳐용 바이너리를 만들 수도 있기 때문에 실제 커널을 구성하는 코드 외에 커널을 만들기 위해 필요한 몇 몇 프로그램을 호스트 상에서 돌리기 위한 컴파일러를 지정 하는 것이다.

<10127> 보통의 경우 HOST와 TARGET이 같으면 CROSS-COMPILE에 아무 것도 없으나 target이 다르면 여기에 컴파일러의 prefix를 적어줘야한다. 예를 들어 PDA에 많이 사용되는 arm processor를 TARGET으로한 경우엔 cross-COMPILE = arm-linux- 와 같이 된다.

<10128> 위에서 정의한 CROSS-COMPILE이 컴파일러 등의 prefix로 쓰이는데 arm processor의 경우엔 CC = arm-linux-gcc 와 같이 된다.

<10129> 여기 까지 정의된 변수들은 커널 컴파일 전반에 사용될 것들이므로 아래와 같이 해서 각 디렉토리 등에 들어있는 Makefile로 값을 전달해 준다.

<10130> 리눅스 커널은 컴파일 전에 반드시 설정/의존성설정이 되어있어야 하므로 어느 경우든 두 절차를 검사한다. 먼저 .config가 있다는 것은 커널 설정이 된 상태를 의미하고 .depend는 의존성설정이 끝난 것을 의미한다.

(10) .config는 있지만 .depend는 없는 경우

(11) .config가 없는 경우엔 커널 설정을 먼저하도록 한다.

(12) 커널 컴파일이 끝나고 설치될 디렉토리를 지정한다. 보통은 사용되지 않는다.

(13) module이 설치될 디렉토리를 지정한다. 보통은 /lib/modules/2.4.16 과 같이된다.

(14) i386 아키텍쳐에서 루트 디바이스를 지정한다. currenT는 커널 컴파일 할 당시의 root device를 의미한다.

(15) i386 아키텍쳐에서 초기 부팅시의 화면 모드를 설정한다.

(16) i386에서 램디스크가 필요한 경우 사용한다.

(17) core-fILES는 리눅스 커널을 이루는 근간이 되는 몇 몇 부분을 나타낸다. 리눅스 커널은 아래와 같이 kernel, drivers, mm, fs, ipc, network, lib로 구분된다고 볼수 있다.

(18) 커널 설정할 때 어떤 기능을 yes, no, module로 설정할 수 있는데 yes로하면 driverS-y에, no는 DRIVERS-n에, module은 DRIVERS-m에 모이게 된다.

예를 들어 ACPI 기능을 사용하지 않는다고 했을 경우엔 아래 줄이 driverS- += drivers/acpi/acpi.o가 된다. .config의 내용을 한번 읽어보면 금방 이해될 것이다.

(19) 실제 커널에 포함되는 드라이버는 모두 DRIVERS에 기록된다.

(20) make clean 했을 때 지워지는 file들을 지정한다. clean은 object 등을 지울 뿐 커널 설정 등은 지우지 않는다.

(21) mrproper는 세팅까지도 지워버리고 완전히 초기화 시켜버린다.

(22) 리눅스 커널은 여러 종류의 타켓을 지원하므로 처음 Makefile에서 확인한 아키텍처에 따른 Makefile을 읽어 사용하게 된다. make bzImage 등을 했을 때 사용되는 Makefile은 여기서 include 된다.

(23) 필요한 플래그를 export 해서 하위 디렉토리 등에서 make 할 때도 여기에서 적용된 사항들이 같이 적용될 수 있도록 한다.

(24) 어셈블리 코드 컴파일 방법을 지정

(25) 컴파일된 커널이 일차적으로 하나로 뭉쳐 vmlinux를 만들어낸다. 이 것을 압축하고 부팅에 관계된 코드를 덧붙여주면 커널이 완성된다.

(26) include 디렉토리 내의 심볼릭 링크를 설정한다.

(27) 커널 세팅하는 방법에 따라 설정에 필요한 프로그램을 만들고 설정을 시작한다.

(28) linuxsubdirs는 SUBDIRS에 정의된 것들에서 앞에 -dir-을 붙여 새로운 이름을 하나씩 만들어낸다.

patsubst는 $(patsubst PATTERN, REPLACEMENT, TEXT)의 형식으로 TEXT에서 patterN과 일치하는 부분을 REPLACEMENT로 교체한다.

(29) 각 하위 디렉토리를 make 한다. patsubst에 의해 실제 디렉토리로 이동하게 된다.

(30) 컴파일한 시간, 누가 했는가, gcc 버전 등이 기록된다. 내용은 아래와 같다.

```
#define uts-versioN "#11 2002. 01. 25. (금) 16:35:16 KST"
#define linux-compILE-TIME "16:35:16"
#define linux-compILE-BY "root"
#define linux-compILE-HOST "halite"
#define linux-compILE-DOMAIN ""
#define linux-compILER "gcc version 2.95.3 20010315 (release)"
```

(31) include/linux/version.h는 현재 컴파일 될 리눅스 커널의 버전 정보를 담는 헤더 파일이고 아래 스크립트에 의해 만들어진다. 내용은 아래 3줄과 같다.

```
# #define uts-releASE "2.4.16"
# #define linux-veRSION-CODE 132112
# #define kernel-vERSION(a,b,c) (((a) << 16) + ((b) << 8) + (c))
```

(32) main.c엔 start-kernel()이 들어있고 이 함수는 LILO등에 의해 메모리에 올려진 커널이 불리게되는 시작 위치다.

(33) 하위 디렉토리는 위에 정의된 하위 디렉토리 make 방법에 따라 make 된다. 즉 $(patsubst %, -dir-%, $(SUBDIRS))에 의해 make 된다.

(34) emacs, vi용 tag를 만든다.

(35) module로 지정된 놈들을 다 만들어준다.

.phony를 사용하면 정의된 이름이 file이 아님을 알려주고 퍼포먼스를 올려준다. 자세한 것은 'info make'를 해서 참조 바란다.

(36) 커널을 컴파일할 때 System.map이 만들어지는데 이게 존재하는 경우 module의 dependency를 만들어준다.

(37) distclean은 현재 커널 버전 개발을 끝내고 release하려할 때 실행한다.

(38) 리눅스 커널은 많은 하위 디렉토리가 있고 여기에 각각의 Makefile이 존재하는데 공통으로 사용될 수 있는 것을을 모아 Rules.make로 만들고 이를 사용한다.

(39) 커널 설정을 마친 후 헤더와 소스사이의 의존관계를 만들어주는 실행 파일을 만든다.

(40) rpm 배포용 spec 파일과 rpm 파일을 만든다.

2.3.2. $(topDIR)/arch/i386/Makefile

```
<10122>
#
# i386/makefile
#
# this file is included by the global make-
file so that you can add your own
# architecture-
specific flags and dependencies. Remember to do have actions
```

```
# for "archclean" and "archdep" for cleaning up and making dependen-
cies for
# this architecture
#
# this file is subject to the terms and conditions of the GNU Gen-
eral Public
# license.  see the file "COPYING" in the main directory of this archive
# for more details.
#
# copyright (c) 1994 by Linus Torvalds
#
# 19990713  artur Skawina <skawina@geocities.com>
#           added ’-march’ and ’-mpreferred-stack-boundary’ support
#

<10123>
ld=$(cross-compile)ld -m elf-i386
objcopy=$(cross-coMPILE)objcopy -O binary -R .note -R .comment -S
ldflags=-e stext
linkflags =-t $(toPDIR)/arch/i386/vmlinux.lds $(LDFLAGS)

cflags += -pipe

<10124>
# prevent gcc from keeping the stack 16 byte aligned
cflags += $(shell if $(CC) -mpreferred-stack-boundary=2 -S -o /dev/null -
xc /dev/null >/dev/null 2>&1; then echo "-mpreferred-stack-
boundary=2"; fi)

ifdef config-m386
cflags += -march=i386
endif

ifdef config-m486
cflags += -march=i486
endif

ifdef config-m586
cflags += -march=i586
endif

ifdef config-m586tSC
cflags += -march=i586
endif

ifdef config-m586mMX
cflags += -march=i586
endif
```

```
ifdef config-m686
cflags += -march=i686
endif

ifdef config-mpentIUMIII
cflags += -march=i686
endif

ifdef config-mpentIUM4
cflags += -march=i686
endif

<10125>
ifdef config-mk6
cflags += $(shell if $(CC) -march=k6 -S -o /dev/null -
xc /dev/null >/dev/null 2>&1; then echo "-march=k6"; else echo "-
march=i586"; fi)
endif

<10126>
ifdef config-mk7
cflags += $(shell if $(CC) -march=athlon -S -o /dev/null -
xc /dev/null >/dev/null 2>&1; then echo "-march=athlon"; else echo "-
march=i686 -malign-functions=4"; fi)
endif

ifdef config-mcrusOE
cflags += -march=i686 -malign-functions=0 -malign-jumps=0 -malign-loops=0
endif

ifdef config-mwincHIPC6
cflags += -march=i586
endif

ifdef config-mwincHIP2
cflags += -march=i586
endif

ifdef config-mwincHIP3D
cflags += -march=i586
endif

ifdef config-mcyriXIII
cflags += -march=i586
endif

head := arch/i386/kernel/head.o arch/i386/kernel/init-task.o

subdirs += arch/i386/kernel arch/i386/mm arch/i386/lib
```

```
core-files := arch/i386/kernel/kernel.o arch/i386/mm/mm.o $(CORE-FILES)
libs := $(topdir)/arch/i386/lib/lib.a $(LIBS) $(TOPDIR)/arch/i386/lib/lib.a

ifdef config-math-EMULATION
subdirs += arch/i386/math-emu
drivers += arch/i386/math-emu/math.o
endif

arch/i386/kernel: dummy
$(make) linuxsubdirs SUBDIRS=arch/i386/kernel

arch/i386/mm: dummy
$(make) linuxsubdirs SUBDIRS=arch/i386/mm

makeboot = $(make) -C arch/$(ARCH)/boot

vmlinux: arch/i386/vmlinux.lds

force: ;

.phony: zimage bzimage compressed zlilo bzlilo zdisk bzdisk install \
clean archclean archmrproper archdep

zimage: vmlinux
@$(makeboot) zimage

<10127>
bzimage: vmlinux
@$(makeboot) bzimage

compressed: zimage

zlilo: vmlinux
@$(makeboot) booTIMAGE=zImage zlilo

tmp:
@$(makeboot) booTIMAGE=bzImage zlilo
bzlilo: vmlinux
@$(makeboot) booTIMAGE=bzImage zlilo

zdisk: vmlinux
@$(makeboot) booTIMAGE=zImage zdisk

bzdisk: vmlinux
@$(makeboot) booTIMAGE=bzImage zdisk

install: vmlinux
@$(makeboot) booTIMAGE=bzImage install
```

```
archclean:
@$(makeboot) clean

archmrproper:

archdep:
@$(makeboot) dep
```

<10122> 이 makefile은 $(TOPDIR)/Makefile에 의해 읽여 들여지므로 export된 많은 변수들을 그대로 사용 가능하다.

<10123> ld는 최종 output을 elf-i386의 형태로 만든다.

objcopy는 입력에서 .note, .comment 섹션을 삭제하고 리로케이션 정보와 심볼 정보를 삭제한다. 출력 포맷은 binary. 링크할 땐 $(TOPDIR)/arch/i386/vmlinux.lds란 파일에 기록되어 있는 방법을 따라 링크한다.

<10124> gcc가 스택을 16 byte 단위로 정렬하지 못하도록 한다.

사용되는 옵션은 다음과 같은 의미를 갖는다.

- -mpreferred-stack-boundary=2 : 스택을 2^2 byte로 정렬하도록 한다. (=4면 2^4)
- -S : 컴파일 스테이지까지만 하고 어셈블은 하지 않는다.
- -xc : c 언어로 컴파일 한다.

즉 스택 바운더리 정렬이 4 바이트로 가능한지 알아봐서 가능하면 4 바이트 정렬을 사용한다. 만약 4 바이트 정렬을 지원하지 않으면 컴파일 중에 에러가 날 것이다. 이땐 기본 값을 사용한다.

/dev/null이 $(CC)의 입력으로 지정됐으므로 /dev/null을 읽어 컴파일한다. /dev/null을 읽으면 EOF를 돌려주므로 컴파일된 출력은 다음과 같을 것이지만 바로 /dev/null로 출력되어 화면에는 나타나지 않는다.

컴파일되면 다음과 같은 결과가 나온다.

```
.file   "null"
.version    "01.01"
gcc2-compiled.:
.ident  "gcc: (gNU) 2.95.3 20010315 (release)"
```

에러 없이 컴파일이 끝나면 $(CC)의 결과는 true가 될 것이고 에러가 있다면 false가 될 것이다. >/dev/null 2>&1은 출력되는 에러 메시지는 화면에 나오게 하지 않고 결과가 true인지 false인지만을 판별 하기 위해 넣은 것이다.

<10125> amd k6 CPU는 지원하는지 여부에 따라 지원하지 않을 경우엔 i586으로 간주한다.

<10126> athlon을 사용한다고 했지만 지원하는지 판단 후 지원하지 않으면 i686으로 간주하고 정렬을 16 바이트로 한다.

<10127> bzimage를 만들 경우엔 먼저 vmlinux를 만들고 나서 $(TOPDIR)/arch/i386/boot에서 make bzImage를 다시 실행한다.

2.3.3. $(topDIR)/arch/i386/boot/Makefile

```
#
# arch/i386/boot/makefile
#
# this file is subject to the terms and conditions of the GNU Gen-
eral Public
# license.  see the file "COPYING" in the main directory of this archive
# for more details.
#
# copyright (c) 1994 by Linus Torvalds
#

boot-incl = $(topdIR)/include/linux/config.h \
$(topdir)/include/linux/autoconf.h \
$(topdir)/include/asm/boot.h

<10122>
zimage: $(configurE) bootsect setup compressed/vmlinux tools/build
$(objcopy) compressed/vmlinux compressed/vmlinux.out
tools/build bootsect setup compressed/vmlinux.out $(ROOT-DEV) > zImage

<10123>
bzimage: $(configuRE) bbootsect bsetup compressed/bvmlinux tools/build
$(objcopy) compressed/bvmlinux compressed/bvmlinux.out
tools/build -b bbootsect bsetup compressed/bvmlinux.out $(ROOT-
DEV) > bzImage

compressed/vmlinux: $(TOPDIR)/vmlinux
@$(make) -c compressed vmlinux

compressed/bvmlinux: $(TOPDIR)/vmlinux
@$(make) -c compressed bvmlinux

zdisk: $(bootimage)
dd bs=8192 if=$(BOOTIMAGE) of=/dev/fd0

zlilo: $(configure) $(BOOTIMAGE)
if [ -f $(instalL-PATH)/vmlinuz ]; then mv $(INSTALL-
PATH)/vmlinuz $(INSTALL-PATH)/vmlinuz.old; fi
if [ -f $(instalL-PATH)/System.map ]; then mv $(INSTALL-
PATH)/System.map $(INSTALL-PATH)/System.old; fi
cat $(bootimage) > $(INSTALL-PATH)/vmlinuz
cp $(topdir)/system.map $(INSTALL-PATH)/
if [ -x /sbin/lilo ]; then /sbin/lilo; else /etc/lilo/install; fi

install: $(configuRE) $(BOOTIMAGE)
sh -x ./install.sh $(KERNELRELEASE) $(BOOTIM-
AGE) $(TOPDIR)/System.map "$(INSTALL-PATH)"
```

```
<10124>
tools/build: tools/build.c
$(hostcc) $(hostCFLAGS) -o $@ $< -I$(TOPDIR)/include

bootsect: bootsect.o
$(ld) -ttext 0x0 -s --oformat binary -o $@ $<

bootsect.o: bootsect.s
$(as) -o $@ $<

bootsect.s: bootsect.S Makefile $(BOOT-INCL)
$(cpp) $(cppflagS) -traditional $(SVGA-MODE) $(RAMDISK) $< -o $@

<10125>
bbootsect: bbootsect.o
$(ld) -ttext 0x0 -s --oformat binary $< -o $@

bbootsect.o: bbootsect.s
$(as) -o $@ $<

<10126>
bbootsect.s: bootsect.S Makefile $(BOOT-INCL)
$(cpp) $(cppflagS) -D--BIG-KERNEL-- -traditional $(SVGA-
MODE) $(RAMDISK) $< -o $@

setup: setup.o
$(ld) -ttext 0x0 -s --oformat binary -e begtext -o $@ $<

setup.o: setup.s
$(as) -o $@ $<

setup.s: setup.s video.S Makefile $(BOOT-
INCL) $(TOPDIR)/include/linux/version.h $(TOPDIR)/include/linux/compile.h
$(cpp) $(cppflagS) -D--ASSEMBLY-- -traditional $(SVGA-
MODE) $(RAMDISK) $< -o $@

<10127>
bsetup: bsetup.o
$(ld) -ttext 0x0 -s --oformat binary -e begtext -o $@ $<

bsetup.o: bsetup.s
$(as) -o $@ $<

bsetup.s: setup.s video.S Makefile $(BOOT-
INCL) $(TOPDIR)/include/linux/version.h $(TOPDIR)/include/linux/compile.h
$(cpp) $(cppflagS) -D--BIG-KERNEL-- -D--ASSEMBLY-- -traditional $(SVGA-
MODE) $(RAMDISK) $< -o $@
```

```
dep:

clean:
rm -f tools/build
rm -f setup bootsect zImage compressed/vmlinux.out
rm -f bsetup bbootsect bzImage compressed/bvmlinux.out
@$(make) -c compressed clean
```

<10122> 현재 커널은 zImage, bzImage 두 가지가 존재 한다.

zimage는 gzip으로 압축되고 하위 1M 메모리 내에 적재될 수 있는 크기의 커널[4]

bzimage는 gzip으로 압축되고 하위 1M 메모리 내에 적재될 수 없는 크기의 커널

$(objcOPY)는 $(TOPDIR)/arch/i386/Makefile에서 정의된 것을 따른다. 즉 'objcoPY=$(CROSS-COMPILE)objcopy -O binary -R .note -R .comment -S'가 된다.

<10123> bvmlinux를 objcopy를 사용해 심볼 등을 빼고 build란 것을 사용해 최종 커널을 만든다. build는 bbootsect(512 bytes)+bsetup+bvmlinux.out을 합쳐 하나의 bzImage를 만든다 (그림 2-2).

<10124> build 프로그램은 최종 커널을 만드는 유틸리티다. 더 자세한 것은 여기에서 다룬다.

<10125> 어셈블 끝난 bbootsect.o를 링크한다. 사용된 옵션은

-ttext 0x0 : 코드의 시작을 0번지 부터 시작한다고 하고 링크한다. 이렇게 하면 링크된 최종 출력물은 특별한 위치를 가리지 않고 메모리의 아무 위치에나 적재가 가능하고 실행 가능해진다.

-s : 출력물에서 심볼 정보를 모두 없앤다.

--oformat binary : 출력물의 포맷을 바이너리로 한다.

<10126> bootsect.S를 프리컴파일하는데 --BIG-KERNEL--을 정의해 bzImage의 부트 섹터를 만든다. 초기 vga 모드와 램디스크 크기 등을 정보로 전달해 준다.

<10127> bsetup 또한 bbootsect와 같은 방법으로 만들어진다.

2.3.4. $(topDIR)/arch/i386/boot/compressed/Makefile

```
#
# linux/arch/i386/boot/compressed/Makefile
#
# create a compressed vmlinux image from the original vmlinux
#

head = head.o
system = $(topdir)/vmlinux

objects = $(head) misc.o
```

```
zldflags = -e startup-32

<10122>
# zimage-offset is the load offset of the compression loader
# bzimage-offset is the load offset of the high loaded compression loader
#
zimage-offset = 0x1000
bzimage-offset = 0x100000

zlinkflags = -ttext $(ZIMAGE-OFFSET) $(ZLDFLAGS)
bzlinkflags = -ttext $(BZIMAGE-OFFSET) $(ZLDFLAGS)

all: vmlinux

vmlinux: piggy.o $(OBJECTS)
$(ld) $(zlinkflaGS) -o vmlinux $(OBJECTS) piggy.o

<10123>
bvmlinux: piggy.o $(OBJECTS)
$(ld) $(bzlinkflAGS) -o bvmlinux $(OBJECTS) piggy.o

<10124>
head.o: head.s
$(cc) $(aflags) -traditional -c head.S

misc.o: misc.c
$(cc) $(cflags) -c misc.c

<10125>
piggy.o: $(system)
tmppiggy=-tmp-$$$$piggy; \
rm -f $$tmppiggy $$tmppiggy.gz $$tmppiggy.lnk; \
$(objcopy) $(sysTEM) $$tmppiggy; \
gzip -f -9 < $$tmppiggy > $$tmppiggy.gz; \
echo "sections { .data : { input-len = .; LONG(input-data-end - input-
data) input-data = .; *(.data) input-data-
end = .; }}" >gt; $$tmppiggy.lnk; \
$(ld) -r -o piggy.o -b binary $$tmppiggy.gz -b elf32-i386 -
T $$tmppiggy.lnk; \
rm -f $$tmppiggy $$tmppiggy.gz $$tmppiggy.lnk

clean:
rm -f vmlinux bvmlinux -tmp-*
```

<10122> zimage와 bzImage의 메모리 적재 위치가 서로 달라 zImage는 0x1000에서 부터 메모리에 적재되고 bzImage는 0x100000에 적재된다.

<10123> bvmlinux는 vmlinux와 마찬가지로 heas.o, misc.o, piggy.o가 합쳐져 만들어진다. 그러나 링크 플래그가 서로 다르게 설정되어있다.

<10124> 커널의 압축을 풀고 메모리에 적재 하는 등의 일을 하는 부분이다.

<10125> 커널의 핵심 부분이 모두 컴파일되 링크되면 elf type으로 $(TOP-DIR)/vmlinux가 만들어지는데 이것에서 디버깅 정보 등을 없애고 압축해서 만든 것이 piggy.o가 된다. 압축은 gzip으로 한다.

$$$$piggy의 4개의 $는 4자리의 임의의 숫자로 채워진다. 즉 tmppiggy=-tmp-1234piggy와 같이된다. 더불어 $$tmppiggy.gz 은 -tmp-1234piggy.gz, tmppiggy.lnk는 -tmp-1234piggy.lnk와 같이 된다.

piggy.o는 head.o, misc.o와 합쳐져 하나의 다른 file로 만들어져야하므로 다시 링커를 통해 elf-i386 포맷으로 만들어진다.

$(objcOPY)에 사용된 옵션은 다음과 같다.

- -O : output format. 여기선 binary
- -R : 지정된 section 이름을 지운다. .note, .comment는 없앤다.
- -S : input file을 지정한다.

$(ld)에 사용된 옵션은 다음과 같다.

- -m elf-i386 : ld가 elf-i386을 emulation 하도록 지정한다.
- -r : relocatable, 메모리에 적재될 때 재배치 가능하도록 한다.
- -b binary : input file의 format을 말한다. 여기선 $$tmppiggy.gz은 binary
- -T : linker script file을 지정한다.
- -b elf32-i386 : output을 elf-i386 format으로 지정한다.

2.3.5. $(topDIR)/arch/i386/boot/tools/build.c

$(topdir)/arch/i386/boot/tools/build는 커널 이미지 만드는 과정의 최종 단계에서 몇 개의 파일을 합쳐 하나의 커널 이미지를 만들어낸다. 이런 일을 담당하는 프로그램을 분석해야 이미 나온 부팅 과정에서의 동작을 이해할 수 있을 것이다.

최종 만들어지는 이미지는 그림 2-2이 된다. 부팅할 때 bootsect는 build에 의해 기록된 루트 디바이스, setup의 크기, 압축 커널의 크기를 바탕을 부팅 절차를 계속 진행한다.

```
/*
 *  $id: chap2.sgml,v 1.8 2002/02/15 15:59:43 halite Exp $
 *
 *  copyright (c) 1991, 1992  Linus Torvalds
 *  copyright (c) 1997 Martin Mares
 */
```

```
/*
 * this file builds a disk-image from three different files:
 *
 * - bootsect: exactly 512 bytes of 8086 machine code, loads the rest
 * - setup: 8086 machine code, sets up system parm
 * - system: 80386 code for actual system
 *
 * it does some checking that all files are of the correct type, and
 * just writes the result to stdout, removing headers and padding to
 * the right amount. It also writes some system data to stderr.
 */

/*
 * changes by tytso to allow root device specification
 * high loaded stuff by Hans Lermen & Werner Almesberger, Feb. 1996
 * cross compiling fixes by Gertjan van Wingerde, July 1996
 * rewritten by martin Mares, April 1997
 */

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <stdarg.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/sysmacros.h>
#include <unistd.h>
#include <fcntl.h>
#include <asm/boot.h>

typedef unsigned char byte;
typedef unsigned short word;
typedef unsigned long u32;

#define default-maJOR-ROOT 0
#define default-miNOR-ROOT 0

/* minimal number of setup sectors (see also bootsect.S) */
#define setup-sectS 4

byte buf[1024];
int fd;
int is-big-kernel;

void die(const char * str, ...)
{
va-list args;
va-start(args, str);
```

```
vfprintf(stderr, str, args);
fputc('\n', stderr);
exit(1);
}

void file-open(const char *name)
{
if ((fd = open(name, O-RDONLY, 0)) < 0)
die("unable to open `%s': %m", name);
}

void usage(void)
{
die("usage: build [-b] bootsect setup system [rootdev] [> image]");
}

int main(int argc, char ** argv)
{
unsigned int i, c, sz, setup-sectors;
u32 sys-size;
byte major-root, minor-root;
struct stat sb;

<10122>
if (argc > 2 && !strcmp(argv[1], "-b"))
  {
    is-big-kernel = 1;
    argc--, argv++;
  }
if ((argc < 4) || (argc > 5))
usage();
if (argc > 4) {
<10123>
if (!strcmp(argv[4], "CURRENT")) {
if (stat("/", &sb)) {
perror("/");
die("couldn't stat /");
}
major-root = major(sb.st-dev);
minor-root = minor(sb.st-dev);
} else if (strcmp(argv[4], "FLOPPY")) {
if (stat(argv[4], &sb)) {
perror(argv[4]);
die("couldn't stat root device.");
}
major-root = major(sb.st-rdev);
minor-root = minor(sb.st-rdev);
} else {
major-root = 0;
```

```
minor-root = 0;
}
} else {
major-root = dEFAULT-MAJOR-ROOT;
minor-root = dEFAULT-MINOR-ROOT;
}
fprintf(stderr, "Root device is (%d, %d)\n", major-root, minor-root);

<10124>
file-open(argv[1]);
i = read(fd, buf, sizeof(buf));
fprintf(stderr,"Boot sector %d bytes.\n",i);
if (i != 512)
die("boot block must be exactly 512 bytes");
if (buf[510] != 0x55 || buf[511] != 0xaa)
die("boot block hasn't got boot flag (0xAA55)");
buf[508] = minor-root;
buf[509] = major-root;
if (write(1, buf, 512) != 512)
die("write call failed");
close (fd);

<10125>
file-open(argv[2]);     /* Copy the setup code */
for (i=0 ; (c=read(fd, buf, sizeof(buf)))>0 ; i+=c )
if (write(1, buf, c) != c)
die("write call failed");
if (c != 0)
die("read-error on `setup'");
close (fd);

<10126>
setup-sectors = (i + 511) / 512; /* Pad unused space with zeros */
/* for compatibility with ancient versions of LILO. */
if (setup-sectors < SETUP-SECTS)
setup-sectors = SETUP-SECTS;
fprintf(stderr, "Setup is %d bytes.\n", i);
memset(buf, 0, sizeof(buf));
while (i < setup-sectors * 512) {
c = setup-sectors * 512 - i;
if (c > sizeof(buf))
c = sizeof(buf);
if (write(1, buf, c) != c)
die("write call failed");
i += c;
}

<10127>
file-open(argv[3]);
```

```
if (fstat (fd, &sb))
die("unable to stat `%s': %m", argv[3]);
sz = sb.st-size;
fprintf (stderr, "System is %d kB\n", sz/1024);
sys-size = (sz + 15) / 16;
/* 0x28000*16 = 2.5 MB, conservative estimate for the current maximum */
if (sys-size > (is-big-kernel ? 0x28000 : DEF-SYSSIZE))
die("system is too big. Try using %smodules.",
is-big-kernel ? "" : "bzImage or ");
if (sys-size > 0xefff)
fprintf(stderr,"warning: kernel is too big for standalone boot "
    "from floppy\n");
while (sz > 0) {
int l, n;

l = (sz > sizeof(buf)) ? sizeof(buf) : sz;
if ((n=read(fd, buf, l)) != l) {
if (n < 0)
die("error reading %s: %m", argv[3]);
else
die("%s: unexpected EOF", argv[3]);
}
if (write(1, buf, l) != l)
die("write failed");
sz -= l;
}
close(fd);

if (lseek(1, 497, SEEK-SET) != 497)
/* Write sizes to the bootsector */
die("output: seek failed");
buf[0] = setup-sectors;
if (write(1, buf, 1) != 1)
die("write of setup sector count failed");
if (lseek(1, 500, SEEK-SET) != 500)
die("output: seek failed");
buf[0] = (sys-size & 0xff);
buf[1] = ((sys-size >> 8) & 0xff);
if (write(1, buf, 2) != 2)
die("write of image length failed");

return 0;     /* Everything is OK */
}
```

<10122> build의 command line에 -b 옵션을 주면 이는 big kernel 임을 의미하게 된다.

<10123> 루트 디바이스의 major, minor 번호를 알아낸다.

currenT는 /의 major, minor number를 사용한다. 필자의 리눅스 박스는 hda1이 /이므로 major=0x03, minor=0x01이 될것이다.

플로피가 루트 디바이스로 지정됐으면 major=minor=0이 된다.

command line에 아무 것도 지정되지 않으면 기본 값이 사용된다(기본 값은 사실 플로피와 같은 값을 갖는다).

<10124> 부트 섹터 파일을 읽어 512 byte가 아니면 에러를 낸다. 부트 섹터는 정확히 512 byte여야 하기 때문이다. 그리고 MagicNumber를 체크해 정말 부트 섹터인지 확인한다.

또 508(0x1FC), 509(0x1FD) 번째 바이트에 루트 디바이스의 minor, major 번호를 써 넣는다.

수정 후 표준 출력으로 bootsect의 512 byte를 출력한다(원래 512 byte 였으므로 수정 내용을 포함해 그대로 출력될 것이다).

<10125> setup은 크기가 정확히 얼마인지 알수 없으므로 1024 byte 단위로 읽으면서 크기를 변수 i에 기억해 놓는다. 읽은 1KB는 읽는 즉시 표준 출력을 출력된다.

<10126> setup을 512 byte 단위로 끊고 적어도 SETUP-SECTOR(값은 4) 만큼이 되는지 확인해 모자란 부분은 0으로 채워 넣는다. 디스크는 섹터 단위로 입출력한다는 것을 기억하기 바란다.

예를 들어 setup의 크기가 4768 byte라면 4768/512=9.3125 이므로 9 섹터를 차지하고 10번째 섹터는 다 사용하지 않고 조금만 사용하게 된다. 10번째 섹터의 경우 160 byte를 제외한 352 byte 만큼을 0으로 채워 넣는다.

<10127> 압축된 커널의 크기를 계산해 zImage의 경우 0x7F000 보다 큰지 확인하고, bzImage는 0x280000 보다 큰지 확인한다. 만약 지정된 크기보다 크다면 현재로서는 수용할 수 없는 크기의 커널이므로 에러를 낸다. 또 플로피 부팅의 경우 플로피에 들어갈 수 있는 크기인지 확인한다.

1024 byte 단위로 읽어 표준 출력에 출력하고 bootsect의 497(0x1F1)에 setup이 몇 섹터를 차지하는지 기록하고 500(0x1F4)에 압축 커널의 크기를 16 byte 단위로 기록해준다.

2.4. bzImage가 만들어지는 과정 추적-Log 분석

2.4.1. make bzImage 순서 정리

make bzimage를 실행 했을 때 실행되는 순서를 Makefile을 기준으로 나열해 봤다.

1. $(TOPDIR)/Makefile에 포함된 $(TOPDIR)/arch/i386/Makefile에 있는 'bzimage:'로부터 시작
2. 의존 관계에 의해 vmlinux가 먼저 만들어짐

이때 vmlinux가 만들어지면 $(TOPDIR)/arch/i386/boot로 이동해 계속 진행

3. vmlinux의 진행

   의존 관계에 의해 version.h $(CONFIGURATION) init/main.o init/version.o linuxsubdirs가 먼저 만들어짐

4. 의존 관계에 의한 만들기가 끝나면 $(TOPDIR)/vmlinux가 만들어짐
5. 의존 관계에 의해 $(CONFIGURE) bbootsect bsetup compressed/bvmlinux tools/build가 만들어진다
6. compressed/bvmlinux의 진행

   의존 관계에 의해 piggy.o $(OBJECTS)가 먼저 만들어짐

7. piggy.o는 $(TOPDIR)/vmlinux를 압축해 만든다.

순서대로 나열했지만 Makefile의 특성상 하나를 만들기 전에 이미 다른 것이 먼저 만들어져야하는 등의 의존 관계가 있기 때문에 순서가 뒤집힌 것처럼 보일 것이다. 이를 바로 잡아 먼저 만들어지는 순으로 나열해보면 아래와 같다.

1. vmlinux
   a. include/linux/version.h
   b. init/main.o
   c. init/version.o
   d. linuxsubidrs(fs lib mm ipc kernel drivers net)
2. bzimage
   a. bbootsect
   b. bsetup
   c. compressed/bvmlinux
      i. piggy.o
      ii. head.o
      iii. misc.o
   d. tools/build

2.4.2. Log

Makefile을 통해 분석된 것을 이제는 실예를 사용해 분석해보자. 아래 Log는 ’make bzImage 2>&1 | tee log-bzImage.txt’를 사용해 얻은 것이다. 전체는 필요 없는 부분이 너무 많기 때문에 필요 없는 부분은 삭제하거나 축약하고 실었다.

```
<10122>
gcc -D--KERNEL-- -I/usr/src/linux-2.4.16/include -Wall -Wstrict-
prototypes -Wno-trigraphs -O2 -fomit-frame-pointer -fno-strict-aliasing -
fno-common -pipe -mpreferred-stack-boundary=2 -march=i686 -malign-
functions=4    -c -o init/main.o init/main.c
. scripts/mkversion > .tmpversion

<10123>
gcc -D--KERNEL-- -I/usr/src/linux-2.4.16/include -Wall -Wstrict-
prototypes -Wno-trigraphs -O2 -fomit-frame-pointer -fno-strict-aliasing -
fno-common -pipe -mpreferred-stack-boundary=2 -march=i686 -malign-
functions=4   -DUTS-MACHINE=’"i386"’ -c -o init/version.o init/version.c
make CFLAGS="-D--KERNEL-- -I/usr/src/linux-2.4.16/include -Wall -Wstrict-
prototypes -Wno-trigraphs -O2 -fomit-frame-pointer -fno-strict-aliasing -
fno-common -pipe -mpreferred-stack-boundary=2 -march=i686 -malign-
functions=4  " -C  kernel

<10124>
gcc -D--KERNEL-- -I/usr/src/linux-2.4.16/include -Wall -Wstrict-
prototypes -Wno-trigraphs -O2 -fomit-frame-pointer -fno-strict-aliasing -
fno-common -pipe -mpreferred-stack-boundary=2 -march=i686 -malign-
functions=4   -DUTS-MACHINE=’"i386"’ -c -o init/version.o init/version.c
make[1]: 들어감 `/usr/src/linux-2.4.16/kernel’ 디렉토리
make all-targets
make[2]: 들어감 `/usr/src/linux-2.4.16/kernel’ 디렉토리
rm -f kernel.o
ld -m elf-i386  -r -o kernel.o sched.o dma.o fork.o exec-
domain.o panic.o printk.o mod-
ule.o exit.o itimer.o info.o time.o softirq.o re-
source.o sysctl.o acct.o capability.o ptrace.o timer.o user.o sig-
nal.o sys.o kmod.o context.o uid16.o ksyms.o pm.o
make[2]: 나감 `/usr/src/linux-2.4.16/kernel’ 디렉토리
make[1]: 나감 `/usr/src/linux-2.4.16/kernel’ 디렉토리

<10125>
make CFLAGS="-D--KERNEL-- -I/usr/src/linux-2.4.16/include -Wall -Wstrict-
prototypes -Wno-trigraphs -O2 -fomit-frame-pointer -fno-strict-aliasing -
fno-common -pipe -mpreferred-stack-boundary=2 -march=i686 -malign-
functions=4  " -C  drivers
make[1]: 들어감 `/usr/src/linux-2.4.16/drivers’ 디렉토리
make[1]: 나감 `/usr/src/linux-2.4.16/drivers’ 디렉토리

<10126>
make CFLAGS="-D--KERNEL-- -I/usr/src/linux-2.4.16/include -Wall -Wstrict-
prototypes -Wno-trigraphs -O2 -fomit-frame-pointer -fno-strict-aliasing -
```

```
fno-common -pipe -mpreferred-stack-boundary=2 -march=i686 -malign-
functions=4  " -C  mm
make[1]: 들어감 `/usr/src/linux-2.4.16/mm’ 디렉토리
make all-targets
make[2]: 들어감 `/usr/src/linux-2.4.16/mm’ 디렉토리
rm -f mm.o
ld -m elf-i386  -r -o mm.o memory.o mmap.o filemap.o mpro-
tect.o mlock.o mremap.o vmalloc.o slab.o bootmem.o swap.o vmscan.o page-
io.o page-alloc.o swap-state.o swapfile.o numa.o oom-kill.o shmem.o
make[2]: 나감 `/usr/src/linux-2.4.16/mm’ 디렉토리
make[1]: 나감 `/usr/src/linux-2.4.16/mm’ 디렉토리

<10127>
make CFLAGS="-D--KERNEL-- -I/usr/src/linux-2.4.16/include -Wall -Wstrict-
prototypes -Wno-trigraphs -O2 -fomit-frame-pointer -fno-strict-aliasing -
fno-common -pipe -mpreferred-stack-boundary=2 -march=i686 -malign-
functions=4  " -C  fs
make[1]: 들어감 `/usr/src/linux-2.4.16/fs’ 디렉토리
rm -f fs.o
ld -m elf-i386  -r -o fs.o open.o read-write.o devices.o file-
table.o buffer.o super.o block-dev.o char-
dev.o stat.o exec.o pipe.o namei.o fcntl.o ioctl.o readdir.o se-
lect.o fifo.o locks.o dcache.o inode.o attr.o bad-
inode.o file.o iobuf.o dnotify.o filesystems.o namespace.o seq-file.o no-
quot.o binfmt-script.o binfmt-elf.o proc/proc.o parti-
tions/partitions.o ext2/ext2.o isofs/isofs.o nls/nls.o aut-
ofs4/autofs4.o devpts/devpts.o jfs/jfs.o
make[1]: 나감 `/usr/src/linux-2.4.16/fs’ 디렉토리

<10128>
make CFLAGS="-D--KERNEL-- -I/usr/src/linux-2.4.16/include -Wall -Wstrict-
prototypes -Wno-trigraphs -O2 -fomit-frame-pointer -fno-strict-aliasing -
fno-common -pipe -mpreferred-stack-boundary=2 -march=i686 -malign-
functions=4  " -C  net
make[1]: 들어감 `/usr/src/linux-2.4.16/net’ 디렉토리
make[1]: 나감 `/usr/src/linux-2.4.16/net’ 디렉토리

<10129>
make CFLAGS="-D--KERNEL-- -I/usr/src/linux-2.4.16/include -Wall -Wstrict-
prototypes -Wno-trigraphs -O2 -fomit-frame-pointer -fno-strict-aliasing -
fno-common -pipe -mpreferred-stack-boundary=2 -march=i686 -malign-
functions=4  " -C  ipc
make[1]: 들어감 `/usr/src/linux-2.4.16/ipc’ 디렉토리
make all-targets
make[2]: 들어감 `/usr/src/linux-2.4.16/ipc’ 디렉토리
rm -f ipc.o
ld -m elf-i386  -r -o ipc.o util.o msg.o sem.o shm.o
make[2]: 나감 `/usr/src/linux-2.4.16/ipc’ 디렉토리
make[1]: 나감 `/usr/src/linux-2.4.16/ipc’ 디렉토리
```

```
<10130>
make CFLAGS="-D--KERNEL-- -I/usr/src/linux-2.4.16/include -Wall -Wstrict-
prototypes -Wno-trigraphs -O2 -fomit-frame-pointer -fno-strict-aliasing -
fno-common -pipe -mpreferred-stack-boundary=2 -march=i686 -malign-
functions=4  " -C  lib
make[1]: 들어감 ‘/usr/src/linux-2.4.16/lib’ 디렉토리
make all-targets
make[2]: 들어감 ‘/usr/src/linux-2.4.16/lib’ 디렉토리
rm -f lib.a
ar  rcs lib.a errno.o ctype.o string.o vsprintf.o brlock.o cmdline.o bust-
spinlocks.o rbtree.o rwsem.o dec-and-lock.o
make[2]: 나감 ‘/usr/src/linux-2.4.16/lib’ 디렉토리
make[1]: 나감 ‘/usr/src/linux-2.4.16/lib’ 디렉토리
```

(10)

```
make CFLAGS="-D--KERNEL-- -I/usr/src/linux-2.4.16/include -Wall -Wstrict-
prototypes -Wno-trigraphs -O2 -fomit-frame-pointer -fno-strict-aliasing -
fno-common -pipe -mpreferred-stack-boundary=2 -march=i686 -malign-
functions=4  " -C  arch/i386/kernel
make[1]: 들어감 ‘/usr/src/linux-2.4.16/arch/i386/kernel’ 디렉토리
rm -f kernel.o
ld -m elf-i386  -r -o kernel.o process.o semaphore.o signal.o en-
try.o traps.o irq.o vm86.o ptrace.o i8259.o io-
port.o ldt.o setup.o time.o sys-i386.o pci-dma.o i386-
ksyms.o i387.o bluesmoke.o dmi-scan.o pci-i386.o pci-pc.o pci-
irq.o mtrr.o apm.o mpparse.o apic.o nmi.o io-apic.o acpitable.o
gcc -D--ASSEMBLY-- -D--KERNEL-- -I/usr/src/linux-2.4.16/include -
traditional -c head.S -o head.o
gcc -D--KERNEL-- -I/usr/src/linux-2.4.16/include -Wall -Wstrict-
prototypes -Wno-trigraphs -O2 -fomit-frame-pointer -fno-strict-aliasing -
fno-common -pipe -mpreferred-stack-boundary=2 -march=i686 -malign-
functions=4     -c -o init-task.o init-task.c
make[1]: 나감 ‘/usr/src/linux-2.4.16/arch/i386/kernel’ 디렉토리
```

(11)

```
make CFLAGS="-D--KERNEL-- -I/usr/src/linux-2.4.16/include -Wall -Wstrict-
prototypes -Wno-trigraphs -O2 -fomit-frame-pointer -fno-strict-aliasing -
fno-common -pipe -mpreferred-stack-boundary=2 -march=i686 -malign-
functions=4  " -C  arch/i386/mm
make[1]: 들어감 ‘/usr/src/linux-2.4.16/arch/i386/mm’ 디렉토리
make all-targets
make[2]: 들어감 ‘/usr/src/linux-2.4.16/arch/i386/mm’ 디렉토리
rm -f mm.o
ld -m elf-i386  -r -o mm.o init.o fault.o ioremap.o extable.o
make[2]: 나감 ‘/usr/src/linux-2.4.16/arch/i386/mm’ 디렉토리
make[1]: 나감 ‘/usr/src/linux-2.4.16/arch/i386/mm’ 디렉토리
```

(12)

```
make CFLAGS="-D--KERNEL-- -I/usr/src/linux-2.4.16/include -Wall -Wstrict-
prototypes -Wno-trigraphs -O2 -fomit-frame-pointer -fno-strict-aliasing -
fno-common -pipe -mpreferred-stack-boundary=2 -march=i686 -malign-
functions=4  " -C  arch/i386/lib
make[1]: 들어감 `/usr/src/linux-2.4.16/arch/i386/lib' 디렉토리
make all-targets
make[2]: 들어감 `/usr/src/linux-2.4.16/arch/i386/lib' 디렉토리
rm -f lib.a
ar  rcs lib.a checksum.o old-
checksum.o delay.o usercopy.o getuser.o memcpy.o strstr.o mmx.o
make[2]: 나감 `/usr/src/linux-2.4.16/arch/i386/lib' 디렉토리
make[1]: 나감 `/usr/src/linux-2.4.16/arch/i386/lib' 디렉토리
```

(13)

```
ld -m elf-i386 -T /usr/src/linux-2.4.16/arch/i386/vmlinux.lds -
e stext arch/i386/kernel/head.o arch/i386/kernel/init-
task.o init/main.o init/version.o \
--start-group \
arch/i386/kernel/kernel.o arch/i386/mm/mm.o ker-
nel/kernel.o mm/mm.o fs/fs.o ipc/ipc.o \
 drivers/acpi/acpi.o drivers/char/char.o drivers/block/block.o drivers/misc/misc.o drivers/
net.o drivers/pnp/pnp.o drivers/video/video.o drivers/md/mddev.o \
net/network.o \
/usr/src/linux-2.4.16/arch/i386/lib/lib.a /usr/src/linux-
2.4.16/lib/lib.a /usr/src/linux-2.4.16/arch/i386/lib/lib.a \
--end-group \
-o vmlinux
nm vmlinux | grep -v '\(com-
piled\)\|\(\.o$\)\|\( [aUw] \)\|\(\.\.ng$\)\|\(LASH[RL]DI\)' | sort > System.map
```

(14)

```
make[1]: 들어감 `/usr/src/linux-2.4.16/arch/i386/boot' 디렉토리
gcc -E -D--KERNEL-- -I/usr/src/linux-2.4.16/include -D--BIG-KERNEL-- -
traditional -DSVGA-MODE=NORMAL-VGA  bootsect.S -o bbootsect.s
as -o bbootsect.o bbootsect.s
bbootsect.s: Assembler messages:
bbootsect.s:257: Warning: indirect lcall without `*'
ld -m elf-i386 -Ttext 0x0 -s --oformat binary bbootsect.o -o bbootsect
```

(15)

```
gcc -E -D--KERNEL-- -I/usr/src/linux-2.4.16/include -D--BIG-KERNEL-- -D--
ASSEMBLY-- -traditional -DSVGA-MODE=NORMAL-VGA  setup.S -o bsetup.s
as -o bsetup.o bsetup.s
bsetup.s: Assembler messages:
bsetup.s:1716: Warning: indirect lcall without `*'
ld -m elf-i386 -Ttext 0x0 -s --oformat binary -e begtext -
o bsetup bsetup.o
```

(16)

```
make[2]: 들어감 `/usr/src/linux-2.4.16/arch/i386/boot/compressed' 디렉토리
tmppiggy=-tmp-$$piggy; \
rm -f $tmppiggy $tmppiggy.gz $tmppiggy.lnk; \
objcopy -O binary -R .note -R .comment -S /usr/src/linux-
2.4.16/vmlinux $tmppiggy; \
gzip -f -9 < $tmppiggy > $tmppiggy.gz; \
echo "SECTIONS { .data : { input-len = .; LONG(input-data-end - input-
data) input-data = .; *(.data) input-data-end = .; }}" > $tmppiggy.lnk; \
ld -m elf-i386 -r -o piggy.o -b binary $tmppiggy.gz -b elf32-i386 -
T $tmppiggy.lnk; \
rm -f $tmppiggy $tmppiggy.gz $tmppiggy.lnk
```

(17)

```
gcc -D--ASSEMBLY-- -D--KERNEL-- -I/usr/src/linux-2.4.16/include -
traditional -c head.S
```

(18)

```
gcc -D--KERNEL-- -I/usr/src/linux-2.4.16/include -Wall -Wstrict-
prototypes -Wno-trigraphs -O2 -fomit-frame-pointer -fno-strict-aliasing -
fno-common -pipe -mpreferred-stack-boundary=2 -march=i686 -malign-
functions=4  -c misc.c
```

(19)

```
ld -m elf-i386 -Ttext 0x100000 -e startup-32 -
o bvmlinux head.o misc.o piggy.o
make[2]: 나감 `/usr/src/linux-2.4.16/arch/i386/boot/compressed' 디렉토리
```

(20)

```
gcc -Wall -Wstrict-prototypes -O2 -fomit-frame-pointer -
o tools/build tools/build.c -I/usr/src/linux-2.4.16/include
```

(21)

```
objcopy -O binary -R .note -R .comment -
S compressed/bvmlinux compressed/bvmlinux.out
```

(22)

```
tools/build -b bbootsect bsetup compressed/bvmlinux.out CURRENT > bzImage
Root device is (3, 1)
Boot sector 512 bytes.
Setup is 4768 bytes.
System is 899 kB
make[1]: 나감 `/usr/src/linux-2.4.16/arch/i386/boot' 디렉토리
```

<10122> main.o

<10123> version.o

<10124> kernel

<10125> drivers

<10126> mm

<10127> fs

<10128> net

<10129> ipc

<10130> lib

(10) arch/i386/kernel

(11) arch/i386/mm

(12) arch/i386/lib

(13) vmlinux

(14) bbootsect

(15) bsetup

(16) arch/i386/boot/compressed/piggy.o

(17) arch/i386/boot/compressed/head.o

(18) arch/i386/boot/compressed/misc.o

(19) arch/i386/boot/compressed/bvmlinux

(20) build

(21) bvmlinux.out

(22) bzImage

위에 열거한 것과 같이 실제 컴파일에서의 순서가 명확하게 나왔다. drivers와 같은 단계에선 하위 디렉토리가 무척 많아 여러 디렉토리를 컴파일하는데 그런 것들은 모두 생략했다.

2.5. 단계별 자세한 분석

2.5.1. -Ttext 0x0의 의미

여기서 -Ttext 0x0에 대해 좀 알아보자.

```
.text

.global -start
-start:
test-val: .long test-data
nop
test-data: .word 0xaa55
```

위와 같은 코드를 'gcc -E -traditional -o test.s test.S'로 컴파일 하면

```
# 1 "test.S"
.text

.global -start
-start:
test-val: .long test-data
nop
test-data: .word 0xaa55
```

이렇게 되고 이를 다시 'as -o test.o test.s'로 어셈블하는데 -a를 사용해 중간 파일을 얻으면 다음과 같다.

```
GAS LISTING test.s  page 1


   1               # 1 "test.S"
   2               .text
   3
   4               .global -start
   5               -start:
   6 0000 05000000  test-val: .long test-data
   7 0004 90        nop
   8 0005 55AA      test-data: .word 0xaa55
   9 0007 90
AS LISTING test.s  page 2


DEFINED SYMBOLS
              test.s:5      .text:00000000 -start
              test.s:6      .text:00000000 test-val
              test.s:8      .text:00000005 test-data

NO UNDEFINED SYMBOLS
```

test-val에 저장된 값은 test-data의 .text의 시작점에서 부터의 offset이다. 프로그램의 시작인 0에서 부터 5번째에 있단 소리다.

최종적으로 'ld -m elf-i386 -s --oformat binary test.o -o test.1'한 결과를 hex로 살펴보면 다음과 같다.

```
00000000 05 80 04 08 90 55 aa 90
```

0x90은 nop(no operation)을 의미한다.

그리고 'ld -m elf-i386 -Ttext 0x0 -s --oformat binary test.o -o test.2'한 결과는 다음과 같다.

```
00000000 05 00 00 00 90 55 aa 90
```

둘 사이의 차이점은 -Ttext 0x0가 있고 없고의 차이다. 바이너리 포맷의 경우 .text를 지정해 주지 않으면 시작 번지를 맘대로 정해버리므로 .text를 지정하지 않은 test.1에서는 시작이 0x09048000으로 설정되어 있는 것을 알 수 있다. 엉뚱한 곳의 값을 사용하도록 만들기 때문에 바이너리를 사용할 땐 제대로된 주소가 들어가도록 .text를 필요한 곳으로 지정해줄 필요가 있다.

또 .text의 시작을 0x02로 했을 때의 바이너리는 다음과 같다.

```
00000000 90 90 09 00 00 00 90 55 aa 90
```

위에서 보듯이 0xaa55는 offset이 7이지만 실제 지정된 것은 9로 .text가 2부터 시작하기 때문이다. 만약 이 바이너리를 메모리에 그대로 올려 놓는다면 제대로 된 값을 읽지 못할 수도 있다. 이 경우엔 .text가 2에서 시작하는 것을 염두에 두고 메모리에 적재해야 제대로 동작할 수 있다.

쉽게 하기 위해선 .text를 0에서 시작하게 하면 바이너리가 메모리의 어느 위치에 있던 상관없이 잘 동작할 수 있게 된다.

2.5.2. 분석

2.4절에서 살펴본 것과 같은 각 단계마다 자세한 내용을 살펴 본다. 이 절이 끝나면 이제 리눅스 커널이 어떻게 만들어지고 어떤 구조를 갖는지 완전히 알 수 있을 것이다.

1 ~ 12 단계는 vmlinux를 만들기 위한 한계이고 커널 설정을 어떻게 했는가에 따라 달라지므로 여기서는 다루지 않는다. 또 17, 18 단계도 bvmlinux를 만들기 위해 필요한 단계이므로 생략한다. 필요한 내용은 2.4절을 참조하거나 각자 log를 만들어 살펴보기 바란다.

1. vmlinux

```
<10122>
ld -m elf-i386 -T /usr/src/linux-2.4.16/arch/i386/vmlinux.lds -
e stext arch/i386/kernel/head.o arch/i386/kernel/init-
task.o init/main.o init/version.o \
--start-group \
arch/i386/kernel/kernel.o arch/i386/mm/mm.o ker-
nel/kernel.o mm/mm.o fs/fs.o ipc/ipc.o \
 drivers/acpi/acpi.o drivers/char/char.o drivers/block/block.o drivers/misc/misc.o drive
net.o drivers/pnp/pnp.o drivers/video/video.o drivers/md/mddev.o \
net/network.o \
/usr/src/linux-2.4.16/arch/i386/lib/lib.a /usr/src/linux-
2.4.16/lib/lib.a /usr/src/linux-2.4.16/arch/i386/lib/lib.a \
--end-group \
-o vmlinux
<10123>
```

```
nm vmlinux | grep -v ’\(com-
piled\)\|\(\.o$\)\|\( [aUw] \)\|\(\.\.ng$\)\|\(LASH[RL]DI\)’ | sort > Sys-
tem.map
```

<10122> vmlinux는 커널 자체를 의미한다. 그래서 링크되는 오브젝트 들이 커널 설정에서 사용하겠다고 한 것들과 선택된 아키텍쳐에 관계된 것들이 뭉쳐져 하나의 파일을 만들어 낸다.

사용된 옵션은 다음과 같다.

- -m elf-i386

  어떤 포맷으로 출력물을 만들 것인지 지정

- -T /usr/src/linux-2.4.16/arch/i386/vmlinux.lds

  링킹하는데 필요한 스크립트 파일을 지정한다. 이 파일에 대한 내용은 아래 vmlinux.lds를 참조하기 바란다.

- -e stext

  프로그램의 시작점을 지정한다. 위 스크립트에 지정된 .stext를 사용한다.

- --start-group ... --end-group

  ...에 지정된 오브젝트를 서로간에 참조한 변수나 함수가 에러나지 않을 때까지 계속해서 검색한다.

- -o vmlinux

  출력물은 vmlinux로 지정

링크에 사용된 스크립트(vmlinux.lds)는 아래와 같다.

```
/* ld script to make i386 Linux kernel
 * Written by Martin Mares <mj@atrey.karlin.mff.cuni.cz>;
 */
<10122>
OUTPUT-FORMAT("elf32-i386", "elf32-i386", "elf32-i386")
OUTPUT-ARCH(i386)
ENTRY(-start)
SECTIONS
{
<10123>
  . = 0xC0000000 + 0x100000;
  -text = .; /* Text and read-only data */
<10124>
  .text : {
*(.text)
```

```
*(.fixup)
*(.gnu.warning)
} = 0x9090
  .text.lock : { *(.text.lock) } /* out-of-line lock text */

  -etext = .; /* End of text section */

  .rodata : { *(.rodata) *(.rodata.*) }
  .kstrtab : { *(.kstrtab) }

  . = ALIGN(16); /* Exception table */
  --start---ex-table = .;
  --ex-table : { *(--ex-table) }
  --stop---ex-table = .;

  --start---ksymtab = .; /* Kernel symbol table */
  --ksymtab : { *(--ksymtab) }
  --stop---ksymtab = .;

<10125>
  .data : { /* Data */
*(.data)
CONSTRUCTORS
}

  -edata = .; /* End of data section */

<10126>
  . = ALIGN(8192); /* init-task */
  .data.init-task : { *(.data.init-task) }

  . = ALIGN(4096); /* Init code and data */
  --init-begin = .;
  .text.init : { *(.text.init) }
  .data.init : { *(.data.init) }
  . = ALIGN(16);
  --setup-start = .;
  .setup.init : { *(.setup.init) }
  --setup-end = .;
  --initcall-start = .;
  .initcall.init : { *(.initcall.init) }
  --initcall-end = .;
  . = ALIGN(4096);
  --init-end = .;

<10127>
  . = ALIGN(4096);
  .data.page-aligned : { *(.data.idt) }
```

```
  . = ALIGN(32);
  .data.cacheline-aligned : { *(.data.cacheline-aligned) }

  --bss-start = .; /* BSS */
  .bss : {
*(.bss)
}
  -end = . ;

<10128>
  /* Sections to be discarded */
  /DISCARD/ : {
*(.text.exit)
*(.data.exit)
*(.exitcall.exit)
}

  /* Stabs debugging sections.  */
  .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) }
}
```

<10122> vmlinux의 포맷과 아키텍쳐를 지정하고 프로그램 시작점을 지정한다.

<10123> vmlinux의 시작 번지를 지정한다. 0x100000은 offset이고 앞의 0xc0000000은 gdt내에 들어갈 때 필요한 값으로 물리적으로는 0x100000 번지를 의미한다.

여기서 부터는 다른 부분과 달리 gdt등이 설정된 상태인 프로텍티드 모드에서 동작 하므로 메모리 관련된 것을 실제 어드레스를 사용하면 안된다.

<10124> 커널 코드가 위치할 곳이다. 0x9090은 빈공간에 채워넣기 할 때 0x9090을 사용하란 말이다.

<10125> 특별히 지정되지 않은 모든 데이타는 여기에 위치한다. CONSTRUCTOR는 C++ constructor 정보를 여기에 기록하란 말이다.

<10126> arch/i386/kernel/init-task.c에 지정되어있고 프로세스 스택을 다루는 방식 때문에 8192 bytes 단위로 정렬되야한다.

<10127> arch/i386/traps.c에 정의되어 있고 Pentium F0 0F 버그를 피하기 위한 간단한 방법으로 페이지 정렬을 사용한다(페이지는 4096 bytes를 의미한다).

<10128> 무시되고 사용되지 않는 섹션으로 vmlinux에 포함되지 않는다. 커널이 exit할 일은 없기 때문이다.

<10123> nm은 오브젝트 파일에서 심볼을 추출해 주는 프로그램이다. 커널 이미지 파일에서 모든 심볼을 추출해내고 이 중에 필요한 부분만을 추려 System.map을 만든다. grep에 사용된 -v는 뒤에 나오는 경우를 제외한 것 들을 찾아준다. 커널 컴파일이 끝난 후 'nm vmlinux > test.map' 만 한 결과와 System.map을 비교해 보면 grep에서 찾는 것들이 어떤 것인지 알 수 있을 것이다.

System.map은 처음 부팅 때 메모리에 읽혀 올려지고 드라이버등이 커널 심볼을 찾을 때 사용한다.

2. bbootsect

```
gcc -E -D--KERNEL-- -I/usr/src/linux-2.4.16/include -D--BIG-KERNEL-- -
traditional -DSVGA-MODE=NORMAL-VGA  bootsect.S -o bbootsect.s
as -o bbootsect.o bbootsect.s
ld -m elf-i386 -Ttext 0x0 -s --oformat binary bbootsect.o -o bbootsect
```

bbootsect.s는 bootsect.S를 컴파일해 만들되 --BIG-KERNEL--을 정의해 만든다. bootsect.S를 살펴보면 이와 관련된 곳이 한 군데 있는 것을 발견할 수 있다.

ld에 사용된 옵션은 다음과 같다.

- -m elf-i386

  elf-i386를 에뮬레이션

- -Ttext 0x0

  text 세그먼트의 시작을 0x0으로 지정

- -s

  모든 디버깅 정보를 없앤다

- -oformat binary

  bbootsect의 포맷은 바이너리

3. bsetup

```
gcc -E -D--KERNEL-- -I/usr/src/linux-2.4.16/include -D--BIG-KERNEL-- -
D--ASSEMBLY-- -traditional -DSVGA-MODE=NORMAL-VGA  setup.S -o bsetup.s
as -o bsetup.o bsetup.s
ld -m elf-i386 -Ttext 0x0 -s --oformat binary -e begtext -
o bsetup bsetup.o
```

bbootsect와 같은 방법을 만든다.

4. arch/i386/boot/compressed/piggy.o

```
tmppiggy=-tmp-$$piggy; \
rm -f $tmppiggy $tmppiggy.gz $tmppiggy.lnk; \
```

```
objcopy -O binary -R .note -R .comment -S /usr/src/linux-
2.4.16/vmlinux $tmppiggy; \
gzip -f -9 < $tmppiggy > $tmppiggy.gz; \
echo "SECTIONS { .data : { input-len = .; LONG(input-data-end - input-
data) input-data = .; *(.data) input-data-
end = .; }}" > $tmppiggy.lnk; \
ld -m elf-i386 -r -o piggy.o -b binary $tmppiggy.gz -b elf32-i386 -
T $tmppiggy.lnk; \
rm -f $tmppiggy $tmppiggy.gz $tmppiggy.lnk
```

piggy.o는 $(TOPDIR)/vmlinux를 압축해 만든다. 우선 vmlinux에서 심볼과 필요 없는 섹션을 없애고 바이너리 형태로 만든다음 gzip을 이용해 압축한다. 압축된 것을 다시 elf-i386 형태로 만들어 놓는다.

ld에 사용된 옵션은 다음과 같다.

- -m elf-i386

  elf-i386를 메뮬레이션

- -b binary $tmppiggy.gz

  $tmppiggy.gz은 바이너리 형식

- -b elf32-i386

  piggy.o는 elf32-i386 형식

- -T $tmppiggy.lnk

  $tmppiggy.lnk를 사용해 링크한다.

$tmppiggy.lnk의 내용은 다음과 같다.

```
SECTIONS
{
.data : {
input-len = .;
LONG(input-data-end - input-data)
input-data = .;
*(.data)
input-data-end = .;
}
}
```

압축된 vmlinux는 .data에 들어가게 되고 *(.data)로 표시된 곳에 들어가게 된다. 그 전후에 LONG(input-data-end - input-data)로 압축된 커널의 크기를 저장한다.

5. arch/i386/boot/compressed/bvmlinux

```
ld -m elf-i386 -Ttext 0x100000 -e startup-32 -
o bvmlinux head.o misc.o piggy.o
```

bvmlinux는 압축된 커널과 head.o, misc.o를 합쳐 만든다. head.o는 메모리 세팅이라고 보면되고 misc.o는 압축을 풀기 위한 코드가 들어있다. ld에 사용된 옵션은 $(TOPDIR)/vmlinux를 만들 때와 거의 흡사하다. 단 text의 시작 번지는 0x100000이다.

부팅할 때 bvmlinux는 반드시 0x100000에 올려져서 실행되야한다. 그렇지 않으면 제대로 동작하지 않는다.

압축된 커널의 크기를 piggy.o에 저장해 놓았기 때문에 메모리의 어느 위치에서 piggy.o가 끝나는지 알수 있다. 이뒤에 압축을 풀어 놓고 압축이 풀린 커널을 다시 0x100000으로 옮겨와 실행한다.

6. build

```
gcc -Wall -Wstrict-prototypes -O2 -fomit-frame-pointer -
o tools/build tools/build.c -I/usr/src/linux-2.4.16/include
```

build는 2.3.5절에서와 같이 동작하도록 만들어진다.

7. bvmlinux.out

```
objcopy -O binary -R .note -R .comment -
S compressed/bvmlinux compressed/bvmlinux.out
```

bvmlinux에서 필요 없는 것을 제외하고 바이너리로 만든다.

8. bzImage

```
tools/build -
b bbootsect bsetup compressed/bvmlinux.out CURRENT > bzImage
Root device is (3, 1)
Boot sector 512 bytes.
Setup is 4768 bytes.
System is 899 kB
```

build를 사용해 bzImage를 만든다. 지정된 루트 디바이스, 부트섹터 크기, setup의 크기 그리고 커널의 크기를 표시해 준다. build의 동작은 2.3.5절을 참조 바란다.

주석

1. 부팅에 관한 좀더 자세한 정보는 LILO의 README를 참조하기 바란다.
2. Master Boot Record
3. zImage, bzImage 등에서 z가 의미하는 것이 gzip으로 압축됐단 것이다.
4. 하위 1M에 대한 것은 2.2.2절을 참조한다.

3장. 크로스 컴파일러 만들기

이 장에서는 임베디스 시스템 개발에 없어선 안될 툴체인 만드는 방법을 소개한다. 사용되는 타겟은 StrongARM SA1110을 사용하는 Assabet이란 인텔에서 발매하는 개발용 시험 보드다.

3.1. 크로스?

임베디드 시스템을 개발할 때 일반적으로 호스트와 타겟이란 말을 사용하는데 이 말에 대해 알아보자.

- 호스트

  임베디드 시스템을 개발하는 과정에서 프로그램을 개발하는 컴퓨터를 가리킨다. 예를 들어 ARM 프로 세서로 하드웨어를 꾸미고 여기에 리눅스를 OS로 사용하는 프로젝트를 생각해보자. 처음 하드웨어를 만들어 커널을 올리는 작업을 할 때 만들어진 하드웨어엔 아무 프로그램도 올라가 있지 않기 때문에 다른 곳에서 커널을 만들어 하드웨어에 심어줘야한다. 리눅스 커널을 만들어주는 시스템을 호스트라 고 한다. 보통은 PC에서 개발해 이식할 것이므로 PC가 호스트가 되겠다.

- 타겟

  임베디드 시스템의 개발에서 만들어지는 하드웨어를 타겟이라한다. 타겟 시스템 이라면 만들어진 임 베디드 시스템을, 타겟 프로세서라면 만들어진 임베디스 시스템의 프로세서를 말한다고 여기면 될 것이다.

그렇다면 크로스 컴파일러는 무엇일까? 위에서 말한 대로 호스트에서 타겟에서 돌아가는 프로그램을 만들어 이식해 주는데 호스트와 타겟에 사용되는 프로세서가 다르다면?

필자는 호스트로 Athlon CPU를 사용하는 PC를 사용하고 있다. 여기에서 실행되는 gcc는 386, 486, 586, K6, 686을 지원한다. 그러나 임베디스 시스템에 사용되는 프로세서는 지원하지 않고 있다. 그러므로 프로그램을 컴파일해도 실제 임베디드 시스템에 사용된 프로세서에서는 실행할 수 없게된다.

그러므로 실행은 호스트에서되지만 만들어지는 코드는 타겟 시스템에서 돌아갈 수 있는 컴파일러가 필요해 진다. 이 것이 바로 크로스 컴파일러다[1].

3.2. 툴체인

3.2.1. 배경

툴체인이란 여러 다른 컴포넌트로 이루어져 있다. 가장 중요한 것은 gcc 컴파일러 자체다. 또 gcc를 사 하기 위해 필요한 binutils도 포함된다. binutils는 바이너리를 다루는데 사용된다. 이둘로 C 라이브러리 를 사용하는 것을 제외한 나머지 대부분과 커널을 컴파일 할 수 있다. 보통은 컴파일러를 컴파일하는 경우 부트스트랩에 관계된 문제 때문에 툴체인을 만드는 것이 간단치 않다.

이 장에서는 툴체인을 어떻게 만드는지 알아본다. 툴체인 만들기로 시간을 낭비하기 싫거나 시도 하기 싫다면 이미 만들어진 것을 사용해도 좋다.

경험에 의하면 혼자 만드는 툴체인의 경우 조심하지 않으면 나중에 사용할 때 어떤 형태의 이상한 결과 를 만들어낼지 모르기 때문에 검증된 이미 만들어진 툴체인을 사용하는 것이 좋다고 감히 말할 수 있다. 또 만들어보면 알겠지만 상당한 노력과 주의를 기울여야하기 때문에 이미 만들어진 것을 사용하는게 좋을 것이다.

그러나 한번 쯤은 경험으로라도 컴파일러를 만들고 이 것으로 프로그램을 만들어 사용해보기 바란다.

ARM 프로세서에 관한 툴체인은 *http://www.arm.linux. org.uk* (http://www.arm.linux.org.uk) 혹은 *http://www.armlinux.org*에서 찾기 바란다.

3.2.2. 미리 만들어진 툴체인

Native Compile

ARM 상에서 도는 Native 컴파일러의 안정화 버전 바이너리는 아래 사이에서 얻을 수 있다.

- *안정화 버전(armv3l 이상) - 데비안 마스터 FTP 사이트* (ftp://ftp.debian.org/debian/dists/stable/main/binary-arm/devel/)
- *최신 버전(armv3l 이상) - 데비안 마스터 FTP 사이트* (ftp://ftp.debian.org/debian/dists/unstable/main/binary-arm/devel/)

arm.linux.org.uk 버전

공식 ARM 리눅스 사이트에서 배포하는 것이다.

- *cross-2.95.3.tar.bz2* (ftp://ftp.arm.linux.org.uk/pub/armlinux/toolchain/cross-2.95.3.tar.bz2)
- *cross-3.0.tar.bz2* (ftp://ftp.arm.linux.org.uk/pub/armlinux/toolchain/cross-3.0.tar.bz2)

Embedian - 필자는 데비안을 사용하지 않기 때문에 실제 이 툴이 어떤지는 잘 모른다.

Embedian 버전은 gcc 2.95.2, binutils 2.9.5.0.37과 glibc 2.1.3을 포함한다. 인스톨은 간단해 데비안 시스템을 사용하면 apt를 사용해 쉽게 설치할 수 있다. 그러나 /usr/bin에 설치되므로 이미 시스템에 깔린 것에 엎어 쓰지 않도록 주의를 요한다.

LART

LART 타르볼 (http://www.lart.tudelft.nl/lartware/compile-tools/)은 아래와 같은 것을 포함한다.

1. gcc 2.95.2
2. binutils 2.9.5.0.22
3. glibc 2.1.2

설치는 아래와 같이 한다.

```
mkdir /data
mkdir /data/lart
cd /data/lart
bzip2 -dc somewhere/arm-linux-cross.tar.bz2 | tar xvf -
```

그리고 /data/lart/cross/bin을 path에 추가해준다. C와 C++ 컴파일러는 arm-linux-gcc와 arm-linux-g++와 같이 명령을 사용해 실행할 수 있다.

Compaq - Itsy인가 하는 프로젝트에 있는데 사용해 보진 않았다.

컴팩 크로스 툴체인은 다음과 같은 것을 포함한다.

1. gcc 2.95.2
2. binutils 2.9.5.0.22
3. glibc 2.1.2 with international crypt library

이 툴체인은 호스트는 i386, 타겟은 armv4l이다.

설치는 아래와 같이 한다. 이 툴체인은 반드시 /skiff/local에 설치되야한다. 그리고 아래와 같이 심볼릭 링크를 만들어줘야한다.

```
ln -s /usr/src/linux/include/asm /skiff/local/arm-linux/include/asm
ln -s /usr/src/linux/include/linux /skiff/local/arm-
linux/include/linux
```

3.2.3. 툴체인 만들기

우선 크로스 컴파일러를 만들어보자. root로 시작하는 것이 좋겠다. 혹은 적어도 설치할 땐 root로 할 필요가 있다. binutils와 gcc를 다음 사이트에서 다운 받는다.

인텔 SA1100 Assabet 보드에 관한 리눅스 개발 사이트를 참조하면 보다 자세한 사항을 알 수 있다. *인텔 사이트* (http://www.intel.com/design/strong/applnots/sa1100lx/sa1100lx.htm).

- *binutils-2.11.2.tar.gz* (ftp://ftp.gnu.org/gnu/binutils/binutils-2.11.2.tar.gz)

- *gcc-2.95.3.tar.gz* (ftp://ftp.gnu.org/gnu/gcc/gcc-2.95.3.tar.gz)
- *linux-2.4.17.tar.bz2* (ftp://ftp.kernel.org/pub/linux/kernel/v2.4/linux-2.4.17.tar.bz2)
- *patch-2.4.17-rmk5.gz* (ftp://ftp.arm.linux.org.uk/pub/armlinux/source/kernel-patches/v2.4/patch-2.4.17-rmk5.gz)
- *glibc-2.1.3.tar.gz* (ftp://ftp.gnu.org/gnu/glibc/glibc-2.1.3.tar.gz)
- *glibc-linuxthreads-2.1.3.tar.gz* (ftp://ftp.gnu.org/gnu/glibc/glibc-linuxthreads-2.1.3.tar.gz)
- *glibc-linuxthreads-2.1.tar.gz* (ftp://ftp.gnu.org/gnu/glibc/glibc-crypt-2.1.tar.gz)

받은 파일이 /devel/arm/assabet에 들었다고 가정하고 그 디렉토리에서 다음과 같은 절차를 사용해 컴파일러를 만든다.

1. binutils

```
%cd /devel/arm/assabet
%tar xzf binutils-2.11.2.tar.gz
%cd binutils-2.11.2
%./configure --target=arm-linux --prefix=/usr/local/arm
%make
%make install
```

configure할 때의 --target은 만들어질 binutils가 arm-linux용이란 것을 나타내고, --prefix는 만들어진 binutils가 설치될 디렉토리를 나타낸다.

설치된 디렉토리의 리스트는 다음과 같이 될 것이다.

```
drwxr-xr-x    4 root     root           16  2월 26 09:58 arm-linux
drwxr-xr-x    2 root     root         4096  2월 26 09:58 bin
drwxr-xr-x    2 root     root           48  2월 26 09:58 include
drwxr-xr-x    2 root     root         4096  2월 26 09:58 lib
drwxr-xr-x    3 root     root            8  2월 26 09:58 man
drwxr-xr-x    2 root     root            1  2월 26 09:58 share
```

정상적으로 만들어 졌으면 다음 절차에서 사용되게 하기 위해 path에 임시로 추가해 준다. bash를 사용한다고 가정했다.

```
export PATH=/usr/local/arm/bin:$PATH
```

2. gcc

gcc를 만들기 위해선 리눅스 커널의 헤더 파일이 필요하다. 우선 커널 소스를 풀고 2개의 디렉토리를 링크해 준다. 사용된 커널은 원하는 버전을 받아 사용하기 바란다. 여기선 2.4.17을 사용했다.

```
%cd /devel/arm/assabet
%tar xvjf linux-2.4.17.tar.bz2
%mv linux linux-2.4.17
%cd /usr/local/arm/arm-linux
```

```
%mkdir include
%cd include
%ln -s /devel/arm/assabet/linux-2.4.17/include/asm-arm asm
%ln -s /devel/arm/assabet/linux-2.4.17/include/linux linux
%ls asm
%ls linux
%cd /devel/arm/assabet
%tar xzf gcc-2.95.3.tar.gz
%cd gcc-2.95.3
%./configure --target=arm-linux --prefix=/usr/local/arm
%make LANGUAGES="c"
```

make를 진행하는 중에 stdlib.h, unistd.h와 같은 몇 개의 파일이 없다고 에러가 날 것이다. ARM 리눅스 사이트에서 얻은 정보에 의하면, gcc/config/arm/t-linux란 파일을 수정하라고 되어있다.아래와 같이 수정한다.

```
%cd gcc/config/arm
%vi t-linux
```

제일 위에 있는 줄을 찾아보면 'TARGET_LIBGCC2_CFLAGS='란 줄이 있을 것이다. 이 줄의 끝에 '-Dinhibit_libc -D__gthr_posix_h'를 추가해 준다. 그리고 configure를 실행할 때 --disable-threads를 추가해 주고 다시 configure과 make를 실행한다.

```
%./configure --target=arm-linux --prefix=/usr/local/arm --disable-
threads
%make LANGUAGES="c"
```

컴파일 후 에러가 나는 것 처럼 보이지만 에러 나도 필요한 파일은 만들어졌으므로 아래 명령으로 확인 한다. 만약 gcc/xgcc가 없다면 make가 제대로 되지 않은 것이다. 아래 명령의 출력은 'arm-linux'가된다.

```
%./gcc/xgcc -dumpmachine
arm-linux
```

위와 같이 출력된다면 제대로 된 것이므로 인스톨한다.

```
%make LANGUAGES="c" install
```

3. glibc를 만드는 절차에선 커널의 version.h가 필요하다. version.h는 make dep를 하면 생성된다.

```
%cd /devel/arm/assabet
%cd linux-2.4.17
%zcat ../patch-2.4.17-rmk5.gz | patch -p1
```

patch까지 적용하고 나서 Makefile을 수정한다. /devel/arm/assabet/linux-2.4.17/Makefile을 열어 'ARCH := arm'으로 수정하고 'CROSS_COMPILE =/usr/local/arm/bin/arm-linux-'으로 수정한다.

```
%make assabet_config
%make config
```

그냥 enter만 쳐서 일단 default setting으로 저장한다.

```
%make dep
%find -name "version.h"
```

version.h가 include/linux에 없다면 에러!

glibc가 설치될 디렉토리를 만들어 준다.

```
%mkdir /usr/local/arm/glibc
%mkdir /usr/local/arm/glibc/arm-linux-glibc
%cd /devel/arm/assabet
%tar xzf glibc-2.1.3.tar.gz
%cd glibc-2.1.3
%tar xzf ../glibc-linuxthreads-2.1.3.tar.gz
%tar xzf ../glibc-crypt-2.1.tar.gz
%CC=arm-linux-gcc ./configure arm-linux --build=i686-pc-linux-gnu \
--prefix=/usr/local/arm/glibc/arm-linux-glibc --enable-add-ons \
--with-headers=/devel/arm/assabet/linux-2.4.17/include
%make
%make install
```

속도가 느린 호스트를 사용한다면 아마도 집에 퇴근하기 전에 make 실행해 놓고 집에 갔다 오면 에러가 나 있던지 제대로 컴파일이 되어 있던지 할 것이다. 전에 P-II 400MHz에서 약 1시간 걸린 기억이 있다. 현재 Athlon 1GHz에서 약 10분 걸린것 같다.

4. c++

c++ 컴파일러도 만들어보자.

```
%cd /devel/arm/assabet
%cd gcc-2.95.3
%./configure --host=i686-pc-linux-gnu --target=arm-linux \
--prefix=/usr/local/arm \
--with-headers=/usr/local/arm/glibc/arm-linux-glibc/include \
--with-libs=/usr/local/arm/glibc/arm-linux-glibc/lib
%make LANGUAGES="c c++"
%make LANGUAGES="c c++" install
```

5. 테스트

다 만들어진 컴파일러, 라이브러리를 테스트 해본다. 테스트 전에 /usr/local/arm/bin이 패스에 설정됐는지 확인. 없다면 넣어 주거나 아래 compiler 명령에서 적당히 path를 삽입해 줄것.

```
%cat > hello.c
/*
```

```
 * hello.c
 */
#include <stdio.h>

int main()
{
printf("hello.\n");
return 0;
}

%arm-linux-gcc -o hello hello.c
```

에러 없이 컴파일되고 hello가 만들어졌는지 확인할 것. 그리고 file 명령으로 만들어진 hello가 어떤 내용인지 확인해 본다. 아래 처럼 ELF이고 ARM용이면 OK.

```
%file hello
hello: ELF 32-bit LSB executable, ARM, version 1, dynami-
cally linked (uses shared libs), not stripped
```

여기까지 툴체인을 만들어 봤다. 커널이 잘 돌아간다면 만든 툴체인은 잘 동작한다고 봐도 될것이다.

주석

1. 크로스(cross)란 말이 호스트와 타겟이 다른단 것을 나타낸다.

4장. ARM 리눅스

이 장에선 인텔의 개발 보드인 Assabet을 이용해 ARM 리눅스 커널을 올리는 방법에 대해 알아본다. Assabet 보드는 StrongARM SA1110을 사용하고 LCD 등이 달려 있다. 이전에 SA1100을 사용하던 Brutus 보드와 마찬가지로 리눅스 커널에서 공식 지원하고 있는 보드다.

이미 많은 사람들이 이 보드에서 개발을 하고 자신 만의 플랫폼을 만들기 때문에 표준이라 할 수 있다. 게다가 인텔 사이트에서는 친절하게도 회로도를 공개하고 있기 때문에 많은 개발자가 별 수 없이(?) 이 프로세서와 함께 Assabet 보드를 사용한다.

4.1. ARM 프로세서 MMU(Memory Management Unit)

ARM 리눅스를 시작하기 전에 ARM 프로세서에 대해 충분히 알아야 쉽게 이해할 수 있을 것이다. 그러나 모든 것을 다루긴 힘들고 여기선 MMU에 대해 다루고 가상 어드레스를 어떻게 이해하면 되는지 정도를 습득하면 되겠다.

이 절의 그림과 글은 모두 ARM architectural Reference Manual(Dave Jaggar저)를 바탕으로 번역했다.

4.1.1. 개요

ARM프로세서의 MMU는 다음의 큰 두가지 일을 한다.

- 가상 어드레스를 물리 어드레스로 변환
- 메모리 접근 권한 제어

그리고 MMU는 위의 일을 하기위해 다음과 같은 하드웨어를 갖고 있다.

- 적어도 하나의 TLB(Translation Lookaside Buffer)
- 접근 제어 로직
- translation-table-walking 로직

4.1.1.1. TLB

TLB는 가상 어드레스를 물리 어드레스로 변환하는 것과 접근 권한을 캐싱하고 있다. 만약 TLB가 가상 어드레스에 대한 변환된 엔트리를 갖고 있다면 접근 제어 로직은 접근이 가능한지 판별한다. 접근이 허용된다면 MMU는 가상 어드레스에 대한 물리 어드레스를 출력해준다. 접근이 허용되지 않는 경우엔 MMU가 CPU에서 abort 시그널을 보낸다.

TLB가 없다면(가상 어드레스에 대한 변환된 엔트리를 갖고 있지 않다) 하드웨어를 움직이는 변환 테이블은 물리 메모리 내에 있는 변환 테이블에서 정보를 읽어온다. 일단 읽어온 후엔 그 정보가 TLB에 저장된다. 이 때 원래 있던 엔트리는 지워질 지워질 수도 있다.

4.1.1.2. 메모리 접근

MMU는 두 가지의 메모리 접근 방식을 지원한다.

- 섹션

  1MB 블럭 단위로 메모리 제어

- 페이지

  페이지 방식엔 또 두 가지 방식이 있다.

  - small page - 4kB 블럭 메모리
  - large page - 64kB 블럭 메모리

섹션과 큰페이지 방식은 TLB에 하나의 엔트리 만이 있을 때 큰 메모리 영역을 매핑하는데 사용된다. 추가로 작은 페이지 방식은 1kB 서브 페이지로 확장되고 큰 페이지 방식은 16kB로 확장 된다.

4.1.1.3. 변환 테이블

주 메모리 내의 변환 테이블은 두 가지 레벨을 갖고 있다.

- 1 레벨 테이블

  섹션 변환과 섹션 레벨 테이블에 대한 포인터를 갖고 있다.

- 2 레벨 테이블

  큰/작은 페이지 변환을 갖고 있다.

4.1.1.4. 도메인

MMU는 도메인이란 기능도 제공한다. 이 것은 개별적 접근 권한을 갖도록 정의된 메모리 영역을 말한다. DACR(Domain Access Control Register)를 사용해 16개 까지의 도메인을 지정할 수 있다.

MMU가 off 상태일 때(프로세서가 리셋된 직후가 이렇다) 가상 어드레스의 출력은 직접 물리 어드레스를 가리키고 메모리 접근 권한 검사는 하지 않는다.

두 TLB 엔트리가 중첩된 메모리 영역을 가리는 경우는 예측할 수 없는 일이 발생한다. 이런 경우는 다른 크기의 페이지로 재 매핑된 후 TLB를 갱신하지 않아 발생할 수 있다.

4.1.2. 변환 절차

MMU는 CPU에의해 만들어진 가상 어드레스를 외부 메모리에 접근하기 위한 물리 어드레스로 변환한다. 그리고 접근 권한 검사도 병행한다. 어드레스가 변환되는 방식은 섹션 방식인가 페이지 방식(페이지 방식엔 또 두 가지 크기의 페이지가 있다)인가에따라 3가지가 있다.

그러나 변환 절차는 언제나 같은 식으로 1레벨 읽기로부터 시작된다. 섹션 방식은 1레벨만 필요하고 페이지 방식은 2레벨도 사용해야한다.

4.1.3. 변환 테이블 베이스

변환 절차는 요청된 가상 어드레스에 대한 엔트리가 칩에 내장된 TLB에 없을 때부터 시작된다. TTBR(Translation Table Base Register)는 1레벨 테이블의 베이스를 가리키고 TTBR의 14에서 31 비트만이 사용된다. 나머지는 0이어야한다. 그러므로 1레벨 페이지 테이블은 반드시 16KB 단위로 정렬되야한다(2^{14}=16384).

4.1.4. 1레벨 읽기

TTBR의 비트 31:14는 그림 4-1에서 보듯이 30비트 어드레스를 만들기 위해 가상 어드레스의 31:20 비트와 연결된다. 이 어드레스는 섹션용 1레벨 디스크립나 2레벨 페이지 테이블 포인터용 디스크립터를 나타내는 4바이트 길이의 변환 테이블 엔트리를 선택한다.

그림 **4-1.** 변환 테이블 **1**레벨 디스크립터 접근

4.1.5. 1레벨 디스크립터

1레벨 디스크립터는 섹션 디스크립터나 2레벨 페이지 테이블 포인터가 될 수 있고 포맷을 그에 따라 변한다. 그림 4-2에서 비트 1:0이 디스크립터 타입을 나타낸다.

비트 1:0이 00인 디스크립터를 사용하면 변환 폴트를 발생한다. 비트 1:0이 11인 디스크립터는 어떻게 동작할지 모른다.

그림 **4-2. 1**레벨 디스크립터 포맷

4.1.6. 섹션 디스크립터와 섹션 변환

그림 4-3은 섹션 변환 전체를 나타낸다. 1레벨 디스크립터에 포함된 접근 권한은 물리 어드레스를 만들어내기 전에 먼저 체크된다.

그림 **4-3.** 섹션 변환

1레벨 디스크립터가 섹션 디스크립터인 경우 각 필드는 다음과 같은 의미를 갖는다.

표 **4-1.** 섹션 디스크립터 필드

비트 1:0	디스크립터의 타입을 나타냄(10은 섹션 디스크립터를 의미함)
비트 3:2	캐시가능, 버퍼가능 비트
비트 4	이 비트의 의미는 ’IMPLEMENTATION DEPENDENt’
비트 8:5	이 디스크립터에 의해 조정되는 모든 페이지에 대한 16개 까지의 도메인 지정
비트 9	사용되지 않음. SHOULD BE ZERO
비트 11:10	접근 권한
비트 19:12	사용되지 않음. SHOULD BE ZERO

비트 31:20	물리 어드레스의 상위 12비트를 구성하는 섹션 베이스 어드레스

4.1.7. 페이지 테이블 디스크립터

페이지 테이블 디스크립터를 1레벨에서 읽고나면 2레벨 디스크립터 읽기가 시작된다. 그림 4-4에 나타나 있다.

그림 **4-4.** 2레벨 디스크립터 접근

1레벨 디스크립터가 페이지 테이블 디스크립터인 경우 각 필드는 다음과 같은 의미를 갖는다.

표 **4-2.** 페이지 디스크립터 필드

비트 1:0	디스크립터의 타입을 나타냄(01은 페이지 디스크립터를 의미함)
비트 4:2	이 비트의 의미는 'IMPLEMENTATION DEPENDENt'
비트 8:5	이 디스크립터에 의해 조정되는 모든 페이지에 대한 16개 까지의 도메인 지정
비트 9	사용되지 않음. SHOULD BE ZERO
비트 31:10	페이지 테이블 베이스 어드레스는 2레벨 페이지 테이블 포인터이다. 2레벨 페이지 테이블은 1KB로 정렬되야한다($2_{10=1024}$)

4.1.8. 2레벨 디스크립터

2레벨 디스크립터는 큰페이지 혹은 작은 페이지로 정의된다.

그림 **4-5. 2**레벨 디스크립터 포맷

각필드는 다음과 같은 의미를 갖는다.

표 **4-3. 2**레벨 디스크립터 포맷

비트 1:0	디스크립터의 타입을 나타냄
비트 3:2	캐시가능, 버퍼 가능 비트
비트 11:4	접근 권한
비트 15:12	사용되지 않음. SHOULD BE ZERO
비트 31:12	작은 페이지 모드에서 물리 어드레스를 만드는데 사용됨
비트 31:16	큰 페이지 모드에서 물리 어드레스를 만드는데 사용됨

비트 11:4의 접근 권한은 다음과 같은 4가지의 서브 페이지로 나뉜다.

표 **4-4. 2**레벨 디스크립터 접근 권한

AP0	1서브 페이지에 대한 접근 권한
AP1	2서브 페이지에 대한 접근 권한
AP2	3서브 페이지에 대한 접근 권한
AP3	4서브 페이지에 대한 접근 권한

4.1.9. 큰 페이지 변환

그림 4-6은 큰 페이지 변환을 나타낸다. 페이지 인덱스의 상위 4비트와 2레벨 테이블 인덱스의 하위 4비트는 서로 겹치는데 큰 페이지에 대한 각 페이지 테이블 엔트리는 페이지 테이블

내에 16번 복사되야한다.

그림 **4-6.** 큰 페이지 변환

4.1.10. 작은 페이지 변환

그림 4-7은 작은 페이지 변환을 나타낸다.

그림 **4-7.** 작은 페이지 변환

4.1.11. 캐시와 쓰기 버퍼 제어

ARM 메모리 시스템은 각 가상 페이지 마다 개별적으로 선택 가능한 두가지 속성에 의해 제어된다.

표 **4-5.** 메모리 시스템 속성

Cacheable	이 속성은 페이지 내의 데이터가 캐시될 수 있음을 나타낸다. 이렇게하면 다음 이어지는 동작은 메인 메모리를 읽을 필요가 없게된다. 또 현재 실행점을 넘어 미리 명령을 읽을수 있음을 나타내기도 한다. 캐시는 write-back 혹은 write-through로 만들어질 수 있다(각 가상 페이지마다 개별적으로 설정할 수 있다).
Bufferable	페이지 내의 데이터가 쓰기 버퍼에 저장될 수 있음을 나타내고 이렇게하면 메인 메모리보다 빠른 동작을 할수 있게 된다. 쓰기 버퍼는 정확한 쓰기 명령을 보장하지 않는다. 그러므로 같은 위치에 대한 여러번의 쓰기 동작이 여러 번의 외부 쓰기 동작을 한다는 보장은 없다.

캐시와 쓰기 버퍼 비트 값은 다음과 같은 의미를 갖는다.

표 **4-6.** 캐시**,** 쓰기 버퍼 비트의 의미

C	**B**	의미
0	0	캐시 불가능, 쓰기 버퍼 불가능
0	1	캐시 불가능, 쓰기 버퍼 동작
1	0	캐시 동작, 쓰기 버퍼 불가능 혹은 write-through 캐시, 쓰기 버퍼 동작
1	1	캐시 동작, 쓰기 버퍼 동작 혹은 write-back 캐시, 쓰기 버퍼 동작

두가지 타입의 캐시를 모두 지원하려면 10으로 write-through, 11으로 writeback 캐시를 지정한다. 어느 한가지 타입의 캐시만을 지원하려면 C와 B 비트를 엄격히 적용해 캐시 가능과 쓰기 버퍼 가능이 각각 동작하도록 한다.

4.1.12. 접근 권한

섹션과 페이지 디스크립터 내의 접근 권한 비트는 해당 섹션이나 페이지의접근을 제어한다. 접근 권한은 시스템(S)과 롬(R) 제어 비트에 의해 변경된다. 테이블은 S, R 비트와 결합된 접근

권한 비트의 의미를 나타낸다. 필요한 권한 없이 메모리에 접근하면 퍼미션 폴트가 발생한다.

표 **4-7.** 접근 권한

AP	**S**	**R**	권한	
수퍼바이저	유저			
00	0	0	접근 불가	접근 불가
00	1	0	읽기만 가능	접근 불가
00	0	1	읽기만 가능	읽기만 가능
00	1	1	예측 불가능	
01	x	x	읽기/쓰기	접근 불가능
10	x	x	읽기/쓰기	읽기만 가능
11	x	x	읽기/쓰기	읽기/쓰기

4.2. Assabet 보드용 커널 컴파일

커널 컴파일에 필요한 기본 툴체인을 만들어 두거나 다운로드 받아서 설치해 둔후 진행하기 바란다. 그 외에 필요한 file은 아래의 것을 받기 바란다.

- *linux-2.4.17.tar.bz2* (ftp://ftp.kernel.org/pub/linux/kernel/v2.4/linux-2.4.17.tar.bz2)
- *patch-2.4.17-rmk5.gz* (ftp://ftp.arm.linux.org.uk/pub/armlinux/source/kernel-patches/v2.4/patch-2.4.17-rmk5.gz)
- *ramdisk_ks.gz* (ftp://ftp.netwinder.org/users/n/nico/ramdisk_ks.gz)
- *angelboot-1.10.nk.tar.gz* (ftp://ftp.netwinder.org/users/n/nico/angel/angelboot-1.10.nk.tar.gz)

1. 커널 컴파일

   우선 커널 소스를 풀고 필요한 패치 파일을 적용해 놓는다.

   ```
   %cd /devel/arm/assabet
   %tar xvjf linux-2.4.17.tar.bz2
   %mv linux linux-2.4.17
   %cd linux-2.4.17
   %zcat ../patch-2.4.17-rmk5.gz | patch -p1
   ```

   patch까지 적용하고 나서 Makefile을 수정한다. /devel/arm/assabet/linux-2.4.17/Makefile을 열어 ’ARCH := arm’으로 수정하고 ’CROSS_COMPILE =/usr/local/arm/bin/arm-linux-’으로 수정한다.

```
%cd /devel/arm/assabet
%cd linux-2.4.17
%make assabet_config
%make menuconfig
```

여기선 Assabet 보드에 사용할 커널을 가정했기 때문에 그냥 기본을 사용해도 되지만 각자에게 맞는 커널 설정을 한 후 컴파일하면 된다.

make menuconfig 후 기본 설정으로 동작시키려면 그냥 exit하면서 configuration만 저장하면 된다.

assabet_config 외에도 아래와 같은 다른 설정이 있다. 참조 바란다.

a. a5k_config

b. ebsa110_config

c. footbridge_config

d. rpc_config

e. brutus_config

f. victor_config

g. empeg_config

```
%make dep
%make modules
%make zImage
%make modules_install INSTALL_MOD_PATH=/devel/arm/assabet/modules
```

module을 설정 했다면 make modules가 있어야한다. 그리고 호스트에 설치할 것이 아니기 때문에 일단 /devel/arm/assabet/modules에 설치하고 ramdisk에 넣어주면 된다.

arch/arm/boot/zImage가 만들어졌는지 확인.

```
-rw-r--r--    1 root     root         3718  2월 26 11:39 Makefile
drwxr-xr-x    2 573      573            24 10월 12 01:04 bootp
drwxr-xr-x    2 573      573          4096  2월 26 14:27 compressed
-rw-r--r--    1 573      573          1350  1월 21  1998 install.sh
-rwxr-xr-x    1 root     root       728036  2월 26 14:27 zImage
```

2. 램디스크 설정

Assabet 보드에 다운로드될 램디스크 이미지는파일로 만들어져 있으므로 loopback device를 사용해 수정해야한다.

```
%cd /devel/arm/assabet
%mkdir ramdisk
%cd ramdisk
%mkdir rdisk
```

```
%cp ../ramdisk_ks.gz .
%gunzip ramdisk_ks.gz
%losetup /dev/loop0 ramdisk_ks
%mount /dev/loop0 rdisk
```

이렇게 하면 rdisk란 디렉토리에 램디스크 이미지가 마운트되므로 만들어진 module 등을 넣거나 사용자가 만든 프로그램을 넣어서 Assabet 보드에 다운로드 후 실행해 볼 수 있다.

3. 커널 테스트

angelboot를 컴파일해 실행 파일을 만들어 놓고 아래와 같은 내용의 파일을 만들어 둔다. minicom은 ttyS1/9600/8N1으로 맞춰 둔다. 시리얼 포트는 사용자에 따라 달리 변경하면된다.

```
%cd /devel/arm/assabet
%tar xzf angelboot-1.10.nk.tar.gz
%cd angelboot-1.10.nk
%make
%cd ..
%cat > opts
base 0xc0008000
entry 0xc0008000
r0 0x00000000
r1 0x00000019
device /dev/ttyS1
options "9600 8N1"
baud 115200
otherfile ramdisk_ks.gz
otherbase 0xc0800000
exec minicom

%./angelboot-1.10-nk/angelboot -f opts ./linux-
2.4.17/arch/arm/boot/zImage
```

커널은 0xc0008000에 올려지고 시작도 거기서 부터 시작된다. 램디스크는 0xc0800000에 올려진다. r0, r1의 값을 전달하는데 이 값은 커널 부팅에 사용되는 값이다.

r1은 아키텍쳐를 구분해 주는 번호인데 $(TOPDIR)/arch/arm/tools/mach-types에 정의되어 있다. Assabet 보드의 경우 25.

여기까지 실행되고 나면 Assabet 보드의 LCD에 펭귄이 보일 것이고 mincom엔 로그인 화면이 나올 것이다.

4.3. ARM 리눅스 Makefile 분석

2장에서 분석한 것과 같이 ARM 리눅스의 Makefile도 i386의 것과 비슷하다. 그러나 부팅하는 과정 등의 일부가 PC가 아닌 다른 시스템이기 때문에 많이 다르다. 그럼에도 불구하고 대부분 비슷한 방법으로 만들어지고 실행된다.

arch/arm 이하의 것만 제외한다면 나머지는 i386의 것과 동일하므로 2장을 참조하고 나머지 ARM 리눅스에 관련된 부분만 다룬다.

4.3.1. $(TOPDIR)/arch/arm/Makefile

Assabet 보드용 기본 설정에 해당하는 .Config file의 내용은 아래와 같다. 이를 참조해 아래 Makefile 분석을 좇아가기 바란다. 세팅된 것만 추리고 나머지는 버렸다.

```
#
# Automatically generated by make menuconfig: don't edit
#
CONFIG_ARM=y
CONFIG_UID16=y
CONFIG_RWSEM_GENERIC_SPINLOCK=y

#
# Code maturity level options
#
CONFIG_EXPERIMENTAL=y

#
# Loadable module support
#
CONFIG_MODULES=y

#
# System Type
#
CONFIG_ARCH_SA1100=y

#
# SA11x0 Implementations
#
CONFIG_SA1100_ASSABET=y
CONFIG_SA1100_USB=m
CONFIG_SA1100_USB_NETLINK=m

CONFIG_CPU_32=y
CONFIG_CPU_32v4=y
CONFIG_CPU_SA1100=y
CONFIG_DISCONTIGMEM=y

#
# General setup
```

```
#
# CONFIG_PCI is not set
CONFIG_ISA=y
CONFIG_CPU_FREQ=y
CONFIG_HOTPLUG=y

#
# PCMCIA/CardBus support
#
CONFIG_PCMCIA=y
CONFIG_PCMCIA_PROBE=y
CONFIG_PCMCIA_SA1100=y
CONFIG_NET=y
CONFIG_SYSVIPC=y
CONFIG_SYSCTL=y
CONFIG_FPE_NWFPE=y
CONFIG_KCORE_ELF=y
CONFIG_BINFMT_ELF=y
CONFIG_PM=y
CONFIG_CMDLINE="root=1f04 mem=32M"
CONFIG_LEDS=y
CONFIG_LEDS_TIMER=y
CONFIG_LEDS_CPU=y
CONFIG_ALIGNMENT_TRAP=y

#
# Memory Technology Devices (MTD)
#
CONFIG_MTD=y
CONFIG_MTD_PARTITIONS=y
CONFIG_MTD_REDBOOT_PARTS=y
CONFIG_MTD_CHAR=y
CONFIG_MTD_BLOCK=y

#
# RAM/ROM/Flash chip drivers
#
CONFIG_MTD_CFI=y
CONFIG_MTD_GEN_PROBE=y
CONFIG_MTD_CFI_ADV_OPTIONS=y
CONFIG_MTD_CFI_NOSWAP=y
CONFIG_MTD_CFI_GEOMETRY=y
CONFIG_MTD_CFI_B4=y
CONFIG_MTD_CFI_I2=y
CONFIG_MTD_CFI_INTELEXT=y

#
# Mapping drivers for chip access
#
```

```
CONFIG_MTD_SA1100=y

#
# Block devices
#
CONFIG_BLK_DEV_LOOP=m
CONFIG_BLK_DEV_RAM=y
CONFIG_BLK_DEV_RAM_SIZE=4096
CONFIG_BLK_DEV_INITRD=y

#
# Networking options
#
CONFIG_UNIX=y
CONFIG_INET=y

#
# Network device support
#
CONFIG_NETDEVICES=y

#
# Ethernet (10 or 100Mbit)
#
CONFIG_NET_ETHERNET=y

#
# PCMCIA network device support
#
CONFIG_NET_PCMCIA=y
CONFIG_PCMCIA_PCNET=y

#
# IrDA (infrared) support
#
CONFIG_IRDA=m
CONFIG_IRLAN=m

#
# Infrared-port device drivers
#
CONFIG_SA1100_FIR=m

#
# ATA/IDE/MFM/RLL support
#
CONFIG_IDE=y

#
```

```
# IDE, ATA and ATAPI Block devices
#
CONFIG_BLK_DEV_IDE=y
CONFIG_BLK_DEV_IDEDISK=y
CONFIG_BLK_DEV_IDECS=y

#
# Character devices
#
CONFIG_VT=y

#
# Serial drivers
#
CONFIG_SERIAL_SA1100=y
CONFIG_SERIAL_SA1100_CONSOLE=y
CONFIG_SA1100_DEFAULT_BAUDRATE=38400
CONFIG_SERIAL_8250=m
CONFIG_SERIAL_CORE=y
CONFIG_SERIAL_CORE_CONSOLE=y
CONFIG_UNIX98_PTYS=y
CONFIG_UNIX98_PTY_COUNT=32

#
# L3 serial bus support
#
CONFIG_L3=y
CONFIG_L3_ALGOBIT=y
CONFIG_L3_BIT_SA1100_GPIO=y
CONFIG_BIT_SA1100_GPIO=y

#
# Watchdog Cards
#
CONFIG_SA1100_RTC=y

#
# PCMCIA character devices
#
CONFIG_PCMCIA_SERIAL_CS=m

#
# File systems
#
CONFIG_FAT_FS=y
CONFIG_MSDOS_FS=y
CONFIG_JFFS2_FS=y
CONFIG_JFFS2_FS_DEBUG=0
CONFIG_TMPFS=y
```

```
CONFIG_PROC_FS=y
CONFIG_DEVPTS_FS=y
CONFIG_EXT2_FS=y

#
# Network File Systems
#
CONFIG_NFS_FS=y
CONFIG_SUNRPC=y
CONFIG_LOCKD=y

#
# Partition Types
#
CONFIG_PARTITION_ADVANCED=y
CONFIG_MSDOS_PARTITION=y
CONFIG_NLS=y

#
# Native Language Support
#
CONFIG_NLS_DEFAULT="iso8859-1"
CONFIG_NLS_CODEPAGE_437=y

#
# Console drivers
#
CONFIG_PC_KEYMAP=y

#
# Frame-buffer support
#
CONFIG_FB=y
CONFIG_DUMMY_CONSOLE=y
CONFIG_FB_SA1100=y
CONFIG_FBCON_CFB2=y
CONFIG_FBCON_CFB4=y
CONFIG_FBCON_CFB8=y
CONFIG_FBCON_CFB16=y
CONFIG_FBCON_FONTWIDTH8_ONLY=y
CONFIG_FBCON_FONTS=y
CONFIG_FONT_8x8=y

#
# Sound
#
CONFIG_SOUND=y
CONFIG_SOUND_SA1100=y
CONFIG_SOUND_UDA1341=y
```

```
CONFIG_SOUND_ASSABET_UDA1341=y

#
# Multimedia Capabilities Port drivers
#
CONFIG_MCP=y
CONFIG_MCP_SA1100=y
CONFIG_MCP_UCB1200=y
CONFIG_MCP_UCB1200_AUDIO=m
CONFIG_MCP_UCB1200_TS=y

#
# Kernel hacking
#
CONFIG_DEBUG_USER=y



#
# arch/arm/Makefile
#
# This file is subject to the terms and conditions of the GNU Gen-
eral Public
# License.  See the file "COPYING" in the main directory of this archive
# for more details.
#
# Copyright (C) 1995-2001 by Russell King

LINKFLAGS :=-p -X -T arch/arm/vmlinux.lds
GZFLAGS :=-9
CFLAGS +=-fno-common -pipe

ifneq ($(CONFIG_NO_FRAME_POINTER),y)
CFLAGS :=$(CFLAGS:-fomit-frame-pointer=)
endif

ifeq ($(CONFIG_DEBUG_INFO),y)
CFLAGS +=-g
endif

# Select CPU dependent flags.  Note that order of declara-
tion is important;
# the options further down the list override previous items.
#
# Note!  For APCS-26 YOU MUST HAVE AN APCS-26 LIBGCC.A
#
apcs-y :=-mapcs-32
apcs-$(CONFIG_CPU_26) :=-mapcs-26 -mcpu=arm3 -Os

<10122>
```

```
# This selects which instruction set is used.
arch-y :=
arch-$(CONFIG_CPU_32v3) :=-march=armv3
arch-$(CONFIG_CPU_32v4) :=-march=armv4
arch-$(CONFIG_CPU_32v5) :=-march=armv5

<10123>
# This selects how we optimise for the processor.
tune-y :=
tune-$(CONFIG_CPU_ARM610) :=-mtune=arm610
tune-$(CONFIG_CPU_ARM710) :=-mtune=arm710
tune-$(CONFIG_CPU_ARM720T) :=-mtune=arm7tdmi
tune-$(CONFIG_CPU_ARM920T) :=-mtune=arm9tdmi
tune-$(CONFIG_CPU_ARM922T) :=-mtune=arm9tdmi
tune-$(CONFIG_CPU_ARM926T) :=-mtune=arm9tdmi
tune-$(CONFIG_CPU_SA110) :=-mtune=strongarm110
tune-$(CONFIG_CPU_SA1100) :=-mtune=strongarm1100

CFLAGS_BOOT
:=$(apcs-y) $(arch-y) $(tune-y) -mshort-load-bytes -msoft-float
CFLAGS +=$(apcs-y) $(arch-y) $(tune-y) -mshort-load-bytes -msoft-float
AFLAGS +=$(apcs-y) $(arch-y) -mno-fpu -msoft-float

ifeq ($(CONFIG_CPU_26),y)
PROCESSOR  = armo
  ifeq ($(CONFIG_ROM_KERNEL),y)
    DATAADDR  = 0x02080000
    TEXTADDR  = 0x03800000
    LDSCRIPT  = arch/arm/vmlinux-armo-rom.lds.in
  else
    TEXTADDR  = 0x02080000
    LDSCRIPT  = arch/arm/vmlinux-armo.lds.in
  endif
endif

<10124>
ifeq ($(CONFIG_CPU_32),y)
PROCESSOR  = armv
TEXTADDR  = 0xC0008000
LDSCRIPT  = arch/arm/vmlinux-armv.lds.in
endif

ifeq ($(CONFIG_ARCH_ARCA5K),y)
MACHINE  = arc
endif

ifeq ($(CONFIG_ARCH_RPC),y)
MACHINE  = rpc
endif
```

```
ifeq ($(CONFIG_ARCH_EBSA110),y)
MACHINE  = ebsa110
endif

ifeq ($(CONFIG_ARCH_CLPS7500),y)
MACHINE  = clps7500
INCDIR  = cl7500
endif

ifeq ($(CONFIG_FOOTBRIDGE),y)
MACHINE  = footbridge
INCDIR  = ebsa285
endif

ifeq ($(CONFIG_ARCH_CO285),y)
TEXTADDR  = 0x60008000
MACHINE  = footbridge
INCDIR  = ebsa285
endif

ifeq ($(CONFIG_ARCH_FTVPCI),y)
MACHINE  = ftvpci
INCDIR  = nexuspci
endif

ifeq ($(CONFIG_ARCH_TBOX),y)
MACHINE  = tbox
endif

ifeq ($(CONFIG_ARCH_SHARK),y)
MACHINE  = shark
endif

<10125>
ifeq ($(CONFIG_ARCH_SA1100),y)
ifeq ($(CONFIG_SA1111),y)
# SA1111 DMA bug: we don’t want the kernel to live in precious DMA-
able memory
TEXTADDR  = 0xc0208000
endif
MACHINE  = sa1100
endif

ifeq ($(CONFIG_ARCH_L7200),y)
MACHINE  = l7200
endif

ifeq ($(CONFIG_ARCH_INTEGRATOR),y)
```

```
MACHINE  = integrator
endif

ifeq ($(CONFIG_ARCH_CAMELOT),y)
MACHINE          = epxa10db
endif

ifeq ($(CONFIG_ARCH_CLPS711X),y)
TEXTADDR  = 0xc0028000
MACHINE  = clps711x
endif

ifeq ($(CONFIG_ARCH_FORTUNET),y)
TEXTADDR  = 0xc0008000
endif

ifeq ($(CONFIG_ARCH_ANAKIN),y)
MACHINE  = anakin
endif

export MACHINE PROCESSOR TEXTADDR GZFLAGS CFLAGS_BOOT

# Only set INCDIR if its not already defined above
# Grr, ?= doesn’t work as all the other assignment opera-
tors do.  Make bug?
ifeq ($(origin INCDIR), undefined)
INCDIR := $(MACHINE)
endif

ifeq ($(origin DATAADDR), undefined)
DATAADDR := .
endif

<10126>
# If we have a machine-specific directory, then include it in the build.
MACHDIR := arch/arm/mach-$(MACHINE)
ifeq ($(MACHDIR),$(wildcard $(MACHDIR)))
SUBDIRS += $(MACHDIR)
CORE_FILES := $(MACHDIR)/$(MACHINE).o $(CORE_FILES)
endif

<10127>
HEAD := arch/arm/kernel/head-$(PROCESSOR).o \
   arch/arm/kernel/init_task.o
SUBDIRS += arch/arm/kernel arch/arm/mm arch/arm/lib arch/arm/nwfpe
CORE_FILES := arch/arm/kernel/kernel.o arch/arm/mm/mm.o $(CORE_FILES)
LIBS := arch/arm/lib/lib.a $(LIBS)

ifeq ($(CONFIG_FPE_NWFPE),y)
```

```
LIBS := arch/arm/nwfpe/math-emu.o $(LIBS)
endif

# Only include fastfpe if it is part of the kernel tree.
FASTFPE := arch/arm/fastfpe
ifeq ($(FASTFPE),$(wildcard $(FASTFPE)))
SUBDIRS += $(FASTFPE)
ifeq ($(CONFIG_FPE_FASTFPE),y)
LIBS := arch/arm/fastfpe/fast-math-emu.o $(LIBS)
endif
endif

ifeq ($(findstring y,$(CONFIG_ARCH_CLPS7500) $(CONFIG_ARCH_L7200)),y)
SUBDIRS += drivers/acorn/char
DRIVERS += drivers/acorn/char/acorn-char.o
endif

MAKEBOOT  = $(MAKE) -C arch/$(ARCH)/boot
MAKETOOLS  = $(MAKE) -C arch/$(ARCH)/tools

# The following is a hack to get 'constants.h' up
# to date before starting compilation

$(patsubst %,_dir_%, $(SUBDIRS)): maketools
$(patsubst %,_modsubdir_%,$(MOD_DIRS)): maketools

symlinks: archsymlinks

archsymlinks:
$(RM) include/asm-arm/arch include/asm-arm/proc
(cd include/asm-arm; ln -sf arch-$(INCDIR) arch; ln -sf proc-
$(PROCESSOR) proc)

vmlinux: arch/arm/vmlinux.lds

<10128>
arch/arm/vmlinux.lds: $(LDSCRIPT) dummy
@sed 's/TEXTADDR/$(TEXTADDR)/;s/DATAADDR/$(DATAADDR)/' $(LDSCRIPT) >$@

arch/arm/kernel arch/arm/mm arch/arm/lib: dummy
$(MAKE) CFLAGS="$(CFLAGS) $(CFLAGS_KERNEL)" $(subst $@, _dir_$@, $@)

bzImage zImage zinstall Image bootpImage install: vmlinux
@$(MAKEBOOT) $@

CLEAN_FILES += \
arch/arm/vmlinux.lds

MRPROPER_FILES += \
```

```
include/asm-arm/arch \
include/asm-arm/proc \
include/asm-arm/constants.h* \
include/asm-arm/mach-types.h

# We use MRPROPER_FILES and CLEAN_FILES now
archmrproper:
@/bin/true

archclean:
@$(MAKEBOOT) clean

archdep: scripts/mkdep archsymlinks
@$(MAKETOOLS) dep
@$(MAKEBOOT) dep

# we need version.h
maketools: checkbin include/linux/version.h
@$(MAKETOOLS) all

# Ensure this is ld "2.9.4" or later
NEW_LINKER
:= $(shell $(LD) --gc-sections --version >/dev/null 2>&1; echo $$?)

ifneq ($(NEW_LINKER),0)
checkbin:
@echo '*** ${VERSION}.${PATCHLEVEL} kernels no longer build cor-
rectly with old versions of binutils.'
@echo '*** Please upgrade your binutils to 2.9.5.'
@false
else
checkbin:
@true
endif

# My testing targets (that short circuit a few dependencies)
zImg:; @$(MAKEBOOT) zImage
Img:; @$(MAKEBOOT) Image
i:; @$(MAKEBOOT) install
zi:; @$(MAKEBOOT) zinstall
bp:; @$(MAKEBOOT) bootpImage

<10129>
#
# Configuration targets.  Use these to select a
# configuration for your architecture
%_config:
@( \
CFG=$(@:_config=); \
```

```
if [ -f arch/arm/def-configs/$$CFG ]; then \
  [ -f .config ] && mv -f .config .config.old; \
  cp arch/arm/def-configs/$$CFG .config; \
  echo "*** Default configuration for $$CFG installed"; \
  echo "*** Next, you may run 'make oldconfig'"; \
else \
  echo "$$CFG does not exist"; \
fi; \
)
```

<10122> ARM 프로세서의 종류에 따라 컴파일러에게 사용할 아키텍쳐를 알려준다. SA1110은 armv4를 사용한다.

ARM 아키텍쳐 (http://www.arm.com/armtech/ARM_Arch?OpenDocument)에서 아키텍쳐에(v6포함) 대한 자세한 정보를 얻기 바란다. 간단히 정리하면 다음과 같다.

- v3

  32 비트 어드레싱 시작, 아래와 같은 종류가 있다.

  - T

    Thumb 코드 실행

  - M

    long multiply 지원, 이것은 v4에서 기본이 됐다.

- v4

  halfword load, store 지원

- v5

  개선된 ARM, Thumb 동작. CLZ 명령 지원. 종류는

  - E

    개선된 DSP 명령

  - J

    JAVA 지원

<10123> 선정된 아키텍쳐에 속하는 변종들 중에 정확한 것을 지정한다.

<10124> SA1110의 경우 필요한 변수 들을 설정한다. .text를 0xC0008000에 맞추고 링크 스크립트를 그에 맞는 것을 사용하도록 한다.

<10125> MACHINE은 arch/arm 디렉토리 밑에 있는 많은 하위 디렉토리 중에 현재 선택된 시스템이 어떤 것인지에 따라 변경되는 내용을 담은 것을 선택하도록 한다.

<10126> MACHINE에서 선택된 것을 사용해 필요한 디렉토리를 선택한다.

<10127> ARM 프로세서의 종류에 따라 처음 실행되는 코드가 달라져야한다. 그 것을 선택한다.

<10128> 링크에 사용될 스크립트인 vmlinux.lds를 만들기 위해 vmlinux.*.lds.in에서 필요한 몇가지 변수를 조정해 vmlinux.lds를 만든다.

<10129> ARM 커널을 만들 때 'make assabet_config'를 실행한 것을 기억하는가? 보드에 따라 기본 설정을 세팅할 때 사용하는 명령에 따라 알맞은 설정을 복사하도록 한다.

arch/arm/def-config에 가능한 모든 정보가 들어있다.

4.3.2. $(TOPDIR)/arch/arm/vmlinux.lds

ARM 프로세서 커널의 링크에 사용되는 링크 스크립트 vmlinux.lds를 분석해보자. 4.3.1절에서 본 것 처럼 vmlinux.lds는 설정된 아키텍쳐에따라 만들어진 것이다.

```
/* ld script to make ARM Linux kernel
 * taken from the i386 version by Russell King
 * Written by Martin Mares <mj@atrey.karlin.mff.cuni.cz>
 */
<10122>
OUTPUT_ARCH(arm)
<10123>
ENTRY(stext)
SECTIONS
{
<10124>
. = 0xC0008000;
<10125>
.init : { /* Init code and data */
_stext = .;
__init_begin = .;
*(.text.init)

__proc_info_begin = .;
*(.proc.info)
__proc_info_end = .;
__arch_info_begin = .;
*(.arch.info)
__arch_info_end = .;
__tagtable_begin = .;
*(.taglist)
__tagtable_end = .;
```

```
*(.data.init)
. = ALIGN(16);
__setup_start = .;
*(.setup.init)
__setup_end = .;
__initcall_start = .;
*(.initcall.init)
__initcall_end = .;
. = ALIGN(4096);
__init_end = .;
}

<10126>
/DISCARD/ : { /* Exit code and data */
*(.text.exit)
*(.data.exit)
*(.exitcall.exit)
}

<10127>
.text : { /* Real text segment */
_text = .; /* Text and read-only data */
*(.text)
*(.fixup)
*(.gnu.warning)
*(.rodata)
*(.rodata.*)
*(.glue_7)
*(.glue_7t)
*(.got) /* Global offset table */

_etext = .; /* End of text section */
}

.kstrtab : { *(.kstrtab) }

. = ALIGN(16);
__ex_table : { /* Exception table */
__start___ex_table = .;
*(__ex_table)
__stop___ex_table = .;
}

__ksymtab : { /* Kernel symbol table */
__start___ksymtab = .;
*(__ksymtab)
__stop___ksymtab = .;
}
```

```
<10128>
. = ALIGN(8192);

.data : {
/*
 * first, the init task union, aligned
 * to an 8192 byte boundary.
 */
*(.init.task)

/*
 * then the cacheline aligned data
 */
. = ALIGN(32);
*(.data.cacheline_aligned)

/*
 * and the usual data section
 */
*(.data)
CONSTRUCTORS

_edata = .;
}

.bss : {
__bss_start = .; /* BSS */
*(.bss)
*(COMMON)
_end = . ;
}
/* Stabs debugging sections. */
.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) }
}
```

<10122> 최종 출력의 아키텍쳐를 지정한다.

<10123> stext는 arch/arm/kernel/head-armv.S에 정의되어 있다.

<10124> 그림 4-8을 참조하면 0xc0008000은 DRAM Bank 0의 일부분이다. 실제 Bank 0는 0xc0000000부터 시작하지만 앞부분 얼마는 Angel이 사용하기 때문에 여기서 부터 올려져 실행되도록 만들어진다.

그림 **4-8. SA-1110** 메모리 맵

<10125> 각종 초기화 코드만 따로 모은다. ENTRY가 stext이고 커널 이미지의 제일 앞이 .text.init부터 시작되므로 실행은 stext가 위치한 arch/arm/kernel/head-armv.S부터 실행된다.

<10126> 만들어지긴 하지만 실제 커널 이미지엔 포함되지 않는 코드다. 커널이 exit할 일은 없기 때문이다.

<10127> 진짜 text 세그먼트와 읽기 전용 데이터가 위치한다.

<10128> 한 프로세스의 스택이 8KB 단위로 작동되도록 만들어지므로 init task union은 8KB 단위로 정렬 되야 제대로 동작할 수 있다. init_task는 task_union이란 union type으로 만들어져 있고 $(TOPDIR)/include/linux/sched.h에 정의되어 있다.

```
union task_union {
struct task_struct task;
unsigned long stack[INIT_TASK_SIZE/sizeof(long)];
};
```

위에서 보듯이 task_union은 struct task_struct task;와 unsigned long stack[];으로 이뤄져 있다. 또 stack은 2048*sizeof(long)의 길이만큼을 차지하는데 SA1100에선 8192 바이트가 된다. 스택이 제대로 동작하려면 각 태스크는 적어도 8KB 단위로 정렬되야 제대로 동작할 수 있게 된다.

4.3.3. $(TOPDIR)/arch/arm/boot/compressed/vmlinux.lds

```
/*
 *  linux/arch/arm/boot/compressed/vmlinux.lds.in
 *
 *  Copyright (C) 2000 Russell King
 *
 * This program is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License version 2 as
 * published by the Free Software Foundation.
 */
OUTPUT_ARCH(arm)
<10122>
ENTRY(_start)
```

```
SECTIONS
{
<10123>
  . = 0xc0008000;
  _load_addr = .;

  . = 0xc0008000;
  _text = .;

<10124>
  .text : {
    _start = .;
    *(.start)
    *(.text)
    *(.fixup)
    *(.gnu.warning)
    *(.rodata)
    *(.rodata.*)
    *(.glue_7)
    *(.glue_7t)
    input_data = .;
    piggy.o
    input_data_end = .;
    . = ALIGN(4);
  }

  _etext = .;

  .data : {
    *(.data)
  }

  _edata = .;

  . = ALIGN(4);
  __bss_start = .;
  .bss : {
    *(.bss)
  }
  _end = .;

<10125>
  .stack : {
    *(.stack)
  }

  .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) }
}
```

<10122> 최종 커널의 실행 시작점은 _start가 된다.

<10123> 메모리에 올려지는 시작 위치는 0xc0008000. SA-1110의 메모리 맵에 의하면 램은 0xc0000000부터 시작하지만 angel이 앞부분 얼마를 사요하므로 여기에 일단 읽어 올려놓고 실행한다.

<10124> 엔트리로 지정된 _start의 위치를 지정한다. 즉 실행코드의 시작이되고 .start 부터 시작하므로 head.S의 start에서 부터 커널이 실행된다.

<10125> 스택의 뒤에 있는 exit 코드 들은 사실 필요 없으므로 무시되고 stack의 바로 뒤부터 해서 남은 메모리 영역에 커널의 압축이 풀린다.

4.3.4. Log 분석

아래 Log는 vmlinux가 만들어지는 과정을 생략하고 vmlinux의 ld가 실행되는 것과 그 이후의 과정만을 실었다. i386에서 추적했던 것과 비슷하게 만들어진다.

```
...

<10122>
/usr/local/arm/bin/arm-linux-ld -p -X -
T arch/arm/vmlinux.lds arch/arm/kernel/head-armv.o arch/arm/kernel/init-
task.o init/main.o init/version.o \
--start-group \
arch/arm/kernel/kernel.o arch/arm/mm/mm.o arch/arm/mach-
sa1100/sa1100.o kernel/kernel.o mm/mm.o fs/fs.o ipc/ipc.o \
 drivers/l3/l3.o drivers/serial/serial.o drivers/char/char.o drivers/block/block.o drivers/
net.o drivers/video/video.o \
net/network.o \
arch/arm/nwfpe/math-emu.o arch/arm/lib/lib.a /devel/arm/assabet/linux-
2.4.17/lib/lib.a \
--end-group \
-o vmlinux
/usr/local/arm/bin/arm-linux-nm vmlinux | grep -v '\(com-
piled\)\|\(\.o$\)\|\( [aUw] \)\|\(\.\.ng$\)\|\(LASH[RL]DI\)' | sort > System.map
make[1]: 들어감 `/devel/arm/assabet/linux-2.4.17/arch/arm/boot' 디렉토리
make[2]: 들어
감 `/devel/arm/assabet/linux-2.4.17/arch/arm/boot/compressed' 디렉토리
<10123>
```

```
/usr/local/arm/bin/arm-linux-gcc -D--ASSEMBLY-- -D--KERNEL-- -
I/devel/arm/assabet/linux-2.4.17/include -mapcs-32 -march=armv4 -mno-fpu -
msoft-float -traditional -c head.S
/usr/local/arm/bin/arm-linux-gcc -D--KERNEL-- -I/devel/arm/assabet/linux-
2.4.17/include -O2 -DSTDC-HEADERS -mapcs-32 -march=armv4 -
mtune=strongarm1100 -mshort-load-bytes -msoft-float -D--KERNEL-- -
I/devel/arm/assabet/linux-2.4.17/include  -c -o misc.o misc.c
<10124>
/usr/local/arm/bin/arm-linux-gcc -D--ASSEMBLY-- -D--KERNEL-- -
I/devel/arm/assabet/linux-2.4.17/include -mapcs-32 -march=armv4 -mno-fpu -
msoft-float   -c -o head-sa1100.o head-sa1100.S
<10125>
/usr/local/arm/bin/arm-linux-objcopy -O binary -R .note -R .comment -
S /devel/arm/assabet/linux-2.4.17/vmlinux piggy
gzip -9 < piggy > piggy.gz
/usr/local/arm/bin/arm-linux-ld -r -o piggy.o -b binary piggy.gz
rm -f piggy piggy.gz
<10126>
/usr/local/arm/bin/arm-linux-ld -p -X -T vmlinux.lds head.o misc.o head-
sa1100.o piggy.o /usr/local/arm/lib/gcc-lib/arm-linux/2.95.3/libgcc.a -
o vmlinux
make[2]: 나감 `/devel/arm/assabet/linux-2.4.17/arch/arm/boot/compressed' 디
렉토리
<10127>
/usr/local/arm/bin/arm-linux-objcopy -O binary -R .note -R .comment -
S compressed/vmlinux zImage
make[1]: 나감 `/devel/arm/assabet/linux-2.4.17/arch/arm/boot' 디렉토리
```

<10122> $(TOPDIR)/vmlinux와 System.map을 만든다.

<10123> i386과 마찬가지로 초기화를 담당하는 head와 압축을 풀어주는 misc를 컴파일한다.

<10124> SA1100에 관계된 특정 명령을 수행한다. 우선 command line으로 입력된 것을 0xc0000000에 복사하고 캐시 관계된 일을 처리하고 시리얼로 연결된 호스트에 터미널 프로그램이 뜰 동안 좀 기다려준다. 4.2절에서 처럼 커널과 램디스크 이미지를 다운로드한 다음에 터미널을 실행하는데 기다려주지 않으면 터미널로는 초기부팅 과정에 대한 것을 볼 수 없게 된다.

<10125> $(TOPDIR)/vmlinux를 바이너리로 만들고 압축한다. 그리고 다른 것과 링킹할 수 있도록 해놓는다.

<10126> 이제 head, misc, head-sa1100 piggy를 합쳐 커널 이미지를 만들어낸다.

<10127> 최종 커널 이미지를 만들어 낸다.

4.4. 소스 분석

$(TOPDIR)/vmlinux가 실행되기 전 까지의 소스 코드를 분석한다.

4.4.1. arch/arm/boot/compressed/head.S

커널 이미지의 제일 처음 실행되는 부분으로 head.S, misc.S, head-sa1100.S가 같이 실행된다. 이 중 head.S는 메모리 초기화에 해당하는 중요한 역할을 담당한다. 아래 소스 코드를 보고 분석해보자.

커널 이미지가 메모리에 어떤 위치에 올려지고 압축 풀린 것이 어디에 위치하고 다시 어디로 옮겨지는지 등에 관해 도식적으로 나타내봤다. 그림 4-9을 참조하면서 head.S를 분석한다.

그림 **4-9. ARM** 리눅스 커널 이미지 메모리 맵

```
/*
 *  linux/arch/arm/boot/compressed/head.S
 *
 *  Copyright (C) 1996-1999 Russell King
 *
 * This program is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License version 2 as
 * published by the Free Software Foundation.
 */
#include <linux/config.h>
#include <linux/linkage.h>

/*
 * Debugging stuff
 *
 * Note that these macros must not contain any code which is not
 * 100% relocatable.  Any attempt to do so will result in a crash.
 * Please select one of the following when turning on debugging.
 */
#ifdef DEBUG
#if defined(CONFIG_DEBUG_DC21285_PORT)
```

```
.macro loadsp, rb
mov \rb, #0x7c000000
.endm
.macro writeb, rb
strb \rb, [r3, #0x3f8]
.endm
#elif defined(CONFIG_ARCH_RPC)
.macro loadsp, rb
mov \rb, #0x03000000
orr \rb, \rb, #0x00010000
.endm
.macro writeb, rb
strb \rb, [r3, #0x3f8 << 2]
.endm
#elif defined(CONFIG_ARCH_INTEGRATOR)
.macro loadsp, rb
mov \rb, #0x16000000
.endm
.macro writeb, rb
strb \rb, [r3, #0]
.endm
#elif defined(CONFIG_ARCH_SA1100)
.macro loadsp, rb
mov \rb, #0x80000000 @ physical base address
#  if defined(CONFIG_DEBUG_LL_SER3)
add \rb, \rb, #0x00050000 @ Ser3
#  else
add \rb, \rb, #0x00010000 @ Ser1
#  endif
.endm
.macro writeb, rb
str \rb, [r3, #0x14] @ UTDR
.endm
#else
#error no serial architecture defined
#endif
#endif

.macro kputc,val
mov r0, \val
bl putc
.endm

.macro kphex,val,len
mov r0, \val
mov r1, #\len
bl phex
.endm
```

```
.macro debug_reloc_start
#ifdef DEBUG
kputc #’\n’
kphex r6, 8 /* processor id */
kputc #’:’
kphex r7, 8 /* architecture id */
kputc #’:’
mrc p15, 0, r0, c1, c0
kphex r0, 8 /* control reg */
kputc #’\n’
kphex r5, 8 /* decompressed kernel start */
kputc #’-’
kphex r8, 8 /* decompressed kernel end  */
kputc #’>’
kphex r4, 8 /* kernel execution address */
kputc #’\n’
#endif
.endm

.macro debug_reloc_end
#ifdef DEBUG
kphex r5, 8 /* end of kernel */
kputc #’\n’
mov r0, r4
bl memdump /* dump 256 bytes at start of kernel */
#endif
.endm

<10122>
.section ".start", #alloc, #execinstr
/*
 * sort out different calling conventions
 */
.align
start:
<10123>
.type start,#function
.rept 8
mov r0, r0
.endr

<10124>
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, #0 @ save r0
```

```
#ifndef __ARM_ARCH_2__
/*
 * Booting from Angel - need to enter SVC mode and disable
 * FIQs/IRQs (numeric definitions from angel arm.h source).
 * We only do this if we were in user mode on entry.
 */
mrs r2, cpsr @ get current mode
tst r2, #3 @ not user?
bne not_angel
mov r0, #0x17 @ angel_SWIreason_EnterSVC
swi 0x123456 @ angel_SWI_ARM
not_angel:
mrs r2, cpsr @ turn off interrupts to
orr r2, r2, #0xc0 @ prevent angel from running
msr cpsr_c, r2
#else
teqp pc, #0x0c000003 @ turn off interrupts
#endif

/*
 * Note that some cache flushing and other stuff may
 * be needed here - is there an Angel SWI call for this?
 */

/*
 * some architecture specific code can be inserted
 * by the linker here, but it should preserve r7 and r8.
 */

/*
 * ldmia에 의해 읽혀지는 값은 다음과 같다.
 * r2 : __bss_start
 * r3 : __end
 * r4 : _load_addr
 * r5 : _start
 * r6 : _usr_stack+4096
 */

.text
1: adr r2, LC0
ldmia r2, {r2, r3, r4, r5, sp}

mov r0, #0
1: str r0, [r2], #4 @ clear bss
str r0, [r2], #4
str r0, [r2], #4
str r0, [r2], #4
cmp r2, r3
blt 1b
```

```
mrc p15, 0, r6, c0, c0 @ get processor ID
bl cache_on

<10125>
mov r1, sp @ malloc space above stack
add r2, sp, #0x10000 @ 64k max

<10126>
teq r4, r5 @ will we overwrite ourselves?
moveq r5, r2 @ decompress after image
movne r5, r4 @ decompress to final location

<10127>
mov r0, r5
mov r3, r7
bl SYMBOL_NAME(decompress_kernel)

<10128>
teq r4, r5 @ do we need to relocate
beq call_kernel @ the kernel?

<10129>
add r0, r0, #127
bic r0, r0, #127 @ align the kernel length
/*
 * r0     = decompressed kernel length
 * r1-r3  = unused
 * r4     = kernel execution address
 * r5     = decompressed kernel start
 * r6     = processor ID
 * r7     = architecture ID
 * r8-r14 = unused
 */
add r1, r5, r0 @ end of decompressed kernel
adr r2, reloc_start
adr r3, reloc_end
1: ldmia r2!, {r8 - r13} @ copy relocation code
stmia r1!, {r8 - r13}
ldmia r2!, {r8 - r13}
stmia r1!, {r8 - r13}
cmp r2, r3
blt 1b

<10130>
bl cache_clean_flush
add pc, r5, r0 @ call relocation code
```

```
/*
 * _load_addr : 0xc0008000
 * _start : 0xc0008000
 * _user_stack+4096 : 총 4KB 만큼을 stack으로 할당하게 된다.
 */
.type LC0, #object
LC0: .word __bss_start
.word _end
.word _load_addr
.word _start
.word user_stack+4096
.size LC0, . - LC0

/*
 * Turn on the cache.  We need to setup some page tables so that we
 * can have both the I and D caches on.
 *
 * We place the page tables 16k down from the kernel execution address,
 * and we hope that nothing else is using it.  If we're using it, we
 * will go pop!
 *
 * On entry,
 *  r4 = kernel execution address
 *  r6 = processor ID
 *  r7 = architecture number
 *  r8 = run-time address of "start"
 * On exit,
 *  r0, r1, r2, r3, r8, r9 corrupted
 * This routine must preserve:
 *  r4, r5, r6, r7
 */
```

(10)

```
.align 5
cache_on: ldr r1, proc_sa110_type
eor r1, r1, r6
movs r1, r1, lsr #5 @ catch SA110 and SA1100
beq 1f
ldr     r1, proc_sa1110_type
eor r1, r1, r6
movs r1, r1, lsr #4
@ movne pc, lr
bne cache_off
```

(11)

```
1:
sub r3, r4, #16384 @ Page directory size
bic r3, r3, #0xff @ Align the pointer
bic r3, r3, #0x3f00
```

(12)

```
/*
```

```
 * Initialise the page tables, turning on the cacheable and bufferable
 * bits for the RAM area only.
 */
mov r0, r3
mov r8, r0, lsr #18
mov r8, r8, lsl #18 @ start of RAM
add r9, r8, #0x10000000 @ a reasonable RAM size
mov r1, #0x12
orr r1, r1, #3 << 10
add r2, r3, #16384
1: cmp r1, r8 @ if virt > start of RAM
orrge r1, r1, #0x0c @ set cacheable, bufferable
cmp r1, r9 @ if virt > end of RAM
bicge r1, r1, #0x0c @ clear cacheable, bufferable
str r1, [r0], #4 @ 1:1 mapping
add r1, r1, #1048576
teq r0, r2
bne 1b
/*
 * If ever we are running from Flash, then we surely want the cache
 * to be enabled also for our execution instance...  We map 2MB of it
 * so there is no map overlap problem for up to 1 MB compressed kernel.
 * If the execution is in RAM then we would only be duplicating the above.
 */
mov r1, #0x1e
orr r1, r1, #3 << 10
mov r2, pc, lsr #20
orr r1, r1, r2, lsl #20
add r0, r3, r2, lsl #2
str r1, [r0], #4
add r1, r1, #1048576
str r1, [r0]

mov r0, #0
mcr p15, 0, r0, c7, c10, 4 @ drain write buffer
mcr p15, 0, r0, c8, c7 @ flush I,D TLBs
mcr p15, 0, r3, c2, c0 @ load page table pointer
mov r0, #-1
mcr p15, 0, r0, c3, c0 @ load domain access register
mrc p15, 0, r0, c1, c0
orr r0, r0, #0x1000 @ I-cache enable
#ifndef DEBUG
orr r0, r0, #0x003d @ Write buffer, mmu
#endif
mcr p15, 0, r0, c1, c0
mov pc, lr


/*
```

```
 * This code is relocatable.  It is relocated by the above code to the end
 * of the kernel and executed there.  During this time, we have no stacks.
 *
 * r0     = decompressed kernel length
 * r1-r3  = unused
 * r4     = kernel execution address
 * r5     = decompressed kernel start
 * r6     = processor ID
 * r7     = architecture ID
 * r8-r14 = unused
 */
.align 5
reloc_start: add r8, r5, r0
debug_reloc_start
mov r1, r4
1:
.rept 4
ldmia r5!, {r0, r2, r3, r9 - r13} @ relocate kernel
stmia r1!, {r0, r2, r3, r9 - r13}
.endr

cmp r5, r8
blt 1b
debug_reloc_end

call_kernel: bl cache_clean_flush
bl cache_off
mov r0, #0
mov r1, r7 @ restore architecture number
mov pc, r4 @ call kernel

/*
 * Here follow the relocatable cache support functions for
 * the various processors.
 */

.type proc_sa110_type,#object
proc_sa110_type:
.word 0x4401a100
.size proc_sa110_type, . - proc_sa110_type

.type proc_sa1110_type,#object
proc_sa1110_type:
.word 0x6901b110
.size proc_sa1110_type, . - proc_sa1110_type

/*
 * Turn off the Cache and MMU.  ARMv3 does not support
 * reading the control register, but ARMv4 does.
```

```
 *
 * On entry,  r6 = processor ID
 * On exit,   r0, r1 corrupted
 * This routine must preserve: r4, r6, r7
 */
.align 5
cache_off:
#ifdef CONFIG_CPU_ARM610
eor r1, r6, #0x41000000
eor r1, r1, #0x00560000
bic r1, r1, #0x0000001f
teq r1, #0x00000600
mov r0, #0x00000060 @ ARM6 control reg.
beq __armv3_cache_off
#endif
#ifdef CONFIG_CPU_ARM710
eor r1, r6, #0x41000000
bic r1, r1, #0x00070000
bic r1, r1, #0x000000ff
teq r1, #0x00007000 @ ARM7
teqne r1, #0x00007100 @ ARM710
mov r0, #0x00000070 @ ARM7 control reg.
beq __armv3_cache_off
#endif
mrc p15, 0, r0, c1, c0
bic r0, r0, #0x000d
mcr p15, 0, r0, c1, c0 @ turn MMU and cache off
mov r0, #0
mcr p15, 0, r0, c7, c7 @ invalidate whole cache v4
mcr p15, 0, r0, c8, c7 @ invalidate whole TLB v4
mov pc, lr

__armv3_cache_off:
mcr p15, 0, r0, c1, c0 @ turn MMU and cache off
mov r0, #0
mcr p15, 0, r0, c7, c0 @ invalidate whole cache v3
mcr p15, 0, r0, c5, c0 @ invalidate whole TLB v3
mov pc, lr

/*
 * Clean and flush the cache to maintain consistency.
 *
 * On entry,
 *  r6 = processor ID
 * On exit,
 *  r1, r2, r12 corrupted
 * This routine must preserve:
 *  r4, r6, r7
 */
```

```
.align 5
cache_clean_flush:
ldr r1, proc_sa110_type
eor r1, r1, r6
movs r1, r1, lsr #5 @ catch SA110 and SA1100
beq 1f
ldr r1, proc_sa1110_type
eor r1, r1, r6
movs r1, r1, lsr #4
movne pc, lr
1:
bic r1, pc, #31
add r2, r1, #32768
1: ldr r12, [r1], #32 @ s/w flush D cache
teq r1, r2
bne 1b

mcr p15, 0, r1, c7, c7, 0 @ flush I cache
mcr p15, 0, r1, c7, c10, 4 @ drain WB
mov pc, lr

/*
 * Various debugging routines for printing hex characters and
 * memory, which again must be relocatable.
 */
#ifdef DEBUG
.type phexbuf,#object
phexbuf: .space 12
.size phexbuf, . - phexbuf

phex: adr r3, phexbuf
mov r2, #0
strb r2, [r3, r1]
1: subs r1, r1, #1
movmi r0, r3
bmi puts
and r2, r0, #15
mov r0, r0, lsr #4
cmp r2, #10
addge r2, r2, #7
add r2, r2, #’0’
strb r2, [r3, r1]
b 1b

puts: loadsp r3
1: ldrb r2, [r0], #1
teq r2, #0
moveq pc, lr
2: writeb r2
```

```
mov r1, #0x00020000
3: subs r1, r1, #1
bne 3b
teq r2, #'\n'
moveq r2, #'\r'
beq 2b
teq r0, #0
bne 1b
mov pc, lr
putc:
mov r2, r0
mov r0, #0
loadsp r3
b 2b

memdump: mov r12, r0
mov r10, lr
mov r11, #0
2: mov r0, r11, lsl #2
add r0, r0, r12
mov r1, #8
bl phex
mov r0, #':'
bl putc
1: mov r0, #' '
bl putc
ldr r0, [r12, r11, lsl #2]
mov r1, #8
bl phex
and r0, r11, #7
teq r0, #3
moveq r0, #' '
bleq putc
and r0, r11, #7
add r11, r11, #1
teq r0, #7
bne 1b
mov r0, #'\n'
bl putc
cmp r11, #64
blt 2b
mov pc, r10
#endif


reloc_end:
```

(13)

```
.align
.section ".stack"
user_stack: .space 4096
```

<10122> .section은 새로운 section을 정의하는 것이다. 어셈블러에 관계된 디렉티브 들은 모두 'info as'를 사용해 찾아보면 정확한 설명이 나와 있다. #alloc은 allocatable을 의미하고 #execinstr은 executable을 의미한다.

Assabet 보드에 올린 커널은 실행될 때 angelboot에 의해 실행되는데 angelboot로부터 r0=0, r1=0x19가 넘어온다. 즉 r1이 아키텍쳐 넘버가 된다. 4.2절 참조.

<10123> start가 function symbol 임을 나타낸다. 그리고 .rept 8은 .rept 부터 .endr 사이의 문장을 8번 반복하란 소리다. 즉 mov r0, r0를 8번 반복한다.

<10124> b 1f가 의미하는 것은 1이란 심볼로 점프하는 것을 의미하고 f는 forward를 의미해 현재 위치에서 앞으로 1이란 심볼을 찾아 점프한다. 반대 방향인 경우는 b를 사용한다.

<10125> stack 위의 메모리에 64KB를 할당하는데 stack은 4.3.3절에서 지정된 것 처럼 커널 이미지의 마지막에 위치한다. 즉, stack으로 할당한곳 뒤의 메모리는 사용하지 않는 영역이므로 여기를 할당해 사용한다. 새로 할당한 영역은 압축을 풀 때 사용하는 영역이다.

<10126> _load_addr과 _start의 위치가 같다면 압축을 푸는 시작위치를 바로 위에서 할당한 64KB 이후에서 부터 시작하고 같지 않다면 _load_addr에 압축을 풀어 놓는다.

<10127> decompress_kernel()은 misc.c에 정의되어 있다. 프로토 타입은

ulg decompress_kernel(ulg output_start, ulg free_mem_ptr_p, ulg free_mem_ptr_end_p, int arch_id);

head.S에서 r0에 넘긴 값이 output_start가 r1은 free_mem_ptr_p, r2는 free_mem_ptr_end_p 그리고 r3이 arch_id가 된다.

<10128> _load_addr과 _start가 서로 다른 경우는 call_kernel을 통해 커널을 실행하고 같은 경우는 압축 풀린 커널을 _load_addr로 옮긴다.

<10129> 압축 풀린 커널을 _load_addr로 옮기기 위해 리로케이션 하는 코드를 압축 풀린 커널의 뒤에 복사한다. 다시 말해 압축 풀린 커널을 그대로 _load_addr로 옮기면 현재 실행되고 있는 head.S를 엎어 쓰기 때문에 계속해서 제대로 실행되지 않는다. 그러므로 재배치 해주는 코드를 먼저 대피해 놓고 그 코드를 실행해 압축 풀린 커널을 옮긴다.

<10130> 최종적으로 재배치 코드를 실행한다.

(10) 정렬은 2^5=32 바이트로 한다.

(11) 1레벨 페이지 테이블은 반드시 16KB 단위로 정렬되야한다. 4.1.3절을 참조.

(12) 페이지 데이블 내용을 채워 넣는다. 램의 시작 점은 0xc0000000이지만 실제 커널이 올려지는 위치가 0xc0008000이므로 0xc0000000 ~ 0xc0008000 사이에 페이지 테이블을 만들고 섹션 디스크립터로 채워 넣는다.

각 섹션 디스크립터는 AP=11, IMP=1, Domain=00이란 속성을 갖는다. Assabet 보드의 경우 0xc0004000 ~ 0xc0008000에 페이지 테이블이 만들어진다.

4.4.2. arch/arm/kernel/head-armv.S

4.3.2절와 4.4.1절에서 살펴본 것 처럼 커널 이미지의 압축이 풀리고 재배치 후에 실행되는 코드는 stext로 $(TOPDIR)/arch/arm/kernel/head-armv.S에 정의되어 있다.

```
/*
 *  linux/arch/arm/kernel/head-armv.S
 *
 *  Copyright (C) 1994-1999 Russell King
 *
 * This program is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License version 2 as
 * published by the Free Software Foundation.
 *
 *  32-bit kernel startup code for all architectures
 */
#include <linux/config.h>
#include <linux/linkage.h>

#include <asm/assembler.h>
#include <asm/mach-types.h>
#include <asm/mach/arch.h>

#define K(a,b,c) ((a) << 24 | (b) << 12 | (c))

/*
 * We place the page tables 16K below TEXTADDR.  There-
fore, we must make sure
 * that TEXTADDR is correctly set.  Currently, we ex-
pect the least significant
 * "short" to be 0x8000, but we could probably relax this restriction to
 * TEXTADDR > PAGE_OFFSET + 0x4000
 *
 * Note that swapper_pg_dir is the virtual address of the page tables, and
 * pgtbl gives us a position-
independent reference to these tables.  We can
 * do this because stext == TEXTADDR
 *
 * swapper_pg_dir, pgtbl and krnladr are all closely related.
 */
#if (TEXTADDR & 0xffff) != 0x8000
#error TEXTADDR must start at 0xXXXX8000
#endif

.globl SYMBOL_NAME(swapper_pg_dir)
.equ SYMBOL_NAME(swapper_pg_dir), TEXTADDR - 0x4000

.macro pgtbl, reg, rambase
adr \reg, stext
```

```
sub \reg, \reg, #0x4000
.endm

/*
 * Since the page table is closely related to the kernel start address, we
 * can convert the page table base address to the base ad-
dress of the section
 * containing both.
 */
.macro krnladr, rd, pgtable, rambase
bic \rd, \pgtable, #0x000ff000
.endm

/*
 *  Kernel startup entry point.
 *
 * The rules are:
 *  r0      - should be 0
 *  r1      - unique architecture number
 *  MMU     - off
 *  I-cache - on or off
 *  D-cache - off
 *
 * See linux/arch/arm/tools/mach-types for the complete list of numbers
 * for r1.
 */
<10122>
.section ".text.init",#alloc,#execinstr
.type stext, #function
ENTRY(stext)
mov r12, r0
/*
 * NOTE!  Any code which is placed here should be done for one of
 * the following reasons:
 *
 *  1. Compatability with old production boot firmware (ie, users
 *     actually have and are booting the kernel with the old firmware)
 *     and therefore will be eventually removed.
 *  2. Cover the case when there is no boot firmware.  This is not
 *     ideal, but in this case, it should ONLY set r0 and r1 to the
 *     appropriate value.
 */
#if defined(CONFIG_ARCH_NETWINDER)
/*
 * Compatability cruft for old NetWinder NeTTroms.  This
 * code is currently scheduled for destruction in 2.5.xx
 */
.rept 8
mov r0, r0
```

```
.endr

adr r2, 1f
ldmdb r2, {r7, r8}
and r3, r2, #0xc000
teq r3, #0x8000
beq __entry
bic r3, r2, #0xc000
orr r3, r3, #0x8000
mov r0, r3
mov r4, #64
sub r5, r8, r7
b 1f

.word _stext
.word __bss_start

1:
.rept 4
ldmia r2!, {r6, r7, r8, r9}
stmia r3!, {r6, r7, r8, r9}
.endr
subs r4, r4, #64
bcs 1b
movs r4, r5
mov r5, #0
movne pc, r0

mov r1, #MACH_TYPE_NETWINDER @ (will go in 2.5)
mov r12, #2 << 24 @ scheduled for removal in 2.5.xx
orr r12, r12, #5 << 12
__entry:
#endif
#if defined(CONFIG_ARCH_L7200)
/*
 * FIXME -
No bootloader, so manually set 'r1' with our architecture number.
 */
mov r1, #MACH_TYPE_L7200
#endif

<10123>
mov r0, #F_BIT | I_BIT | MODE_SVC @ make sure svc mode
msr cpsr_c, r0 @ and all irqs disabled
bl __lookup_processor_type
teq r10, #0 @ invalid processor?
moveq r0, #'p' @ yes, error 'p'
beq __error
bl __lookup_architecture_type
```

```
teq r7, #0 @ invalid architecture?
moveq r0, #'a' @ yes, error 'a'
beq __error
bl __create_page_tables
<10124>
adr lr, __ret @ return address
add pc, r10, #12 @ initialise processor
@ (return control reg)

.type __switch_data, %object
__switch_data: .long __mmap_switched
.long SYMBOL_NAME(compat)
.long SYMBOL_NAME(__bss_start)
.long SYMBOL_NAME(_end)
.long SYMBOL_NAME(processor_id)
.long SYMBOL_NAME(__machine_arch_type)
.long SYMBOL_NAME(cr_alignment)
.long SYMBOL_NAME(init_task_union)+8192

<10125>
.type __ret, %function
__ret: ldr lr, __switch_data
mcr p15, 0, r0, c1, c0
mov r0, r0
mov r0, r0
mov r0, r0
mov pc, lr

/*
 * This code follows on after the page
 * table switch and jump above.
 *
 * r0  = processor control register
 * r1  = machine ID
 * r9  = processor ID
 */
.align 5
__mmap_switched:
adr r3, __switch_data + 4
ldmia r3, {r2, r4, r5, r6, r7, r8, sp}@ r2 = compat
@ sp = stack pointer
str r12, [r2]

mov fp, #0 @ Clear BSS (and zero fp)
1: cmp r4, r5
strcc fp, [r4],#4
bcc 1b

str r9, [r6] @ Save processor ID
```

```
str r1, [r7] @ Save machine type
#ifdef CONFIG_ALIGNMENT_TRAP
orr r0, r0, #2 @ ...........A.
#endif
bic r2, r0, #2 @ Clear ’A’ bit
stmia r8, {r0, r2} @ Save control register values
<10126>
b SYMBOL_NAME(start_kernel)



<10127>
/*
 * 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.
 *
 * We only map in 4MB of RAM, which should be sufficient in
 * all cases.
 *
 * r5 = physical address of start of RAM
 * r6 = physical IO address
 * r7 = byte offset into page tables for IO
 * r8 = page table flags
 */
__create_page_tables:
<10128>
pgtbl r4, r5 @ page table address

<10129>
/*
 * Clear the 16K level 1 swapper page table
 */
mov r0, r4
mov r3, #0
add r2, r0, #0x4000
1: str r3, [r0], #4
str r3, [r0], #4
str r3, [r0], #4
str r3, [r0], #4
teq r0, r2
bne 1b

<10130>
/*
 * Create identity mapping for first MB of kernel to
 * cater for the MMU enable.  This identity mapping
 * will be removed by paging_init()
 */
```

```
krnladr r2, r4, r5 @ start of kernel
add r3, r8, r2 @ flags + kernel base
str r3, [r4, r2, lsr #18] @ identity mapping
```

(10)

```
/*
 * Now setup the pagetables for our kernel direct
 * mapped region.  We round TEXTADDR down to the
 * nearest megabyte boundary.
 */
add r0, r4, #(TEXTADDR & 0xff000000) >> 18 @ start of kernel
bic r2, r3, #0x00f00000
str r2, [r0] @ PAGE_OFFSET + 0MB
add r0, r0, #(TEXTADDR & 0x00f00000) >> 18
str r3, [r0], #4 @ KERNEL + 0MB
add r3, r3, #1 << 20
str r3, [r0], #4 @ KERNEL + 1MB
add r3, r3, #1 << 20
str r3, [r0], #4 @ KERNEL + 2MB
add r3, r3, #1 << 20
str r3, [r0], #4 @ KERNEL + 3MB
```

(11)

```
/*
 * Ensure that the first section of RAM is present.
 * we assume that:
 *  1. the RAM is aligned to a 32MB boundary
 *  2. the kernel is executing in the same 32MB chunk
 *     as the start of RAM.
 */
bic r0, r0, #0x01f00000 >> 18 @ round down
and r2, r5, #0xfe000000 @ round down
add r3, r8, r2 @ flags + rambase
str r3, [r0]

bic r8, r8, #0x0c @ turn off cacheable
@ and bufferable bits
#ifdef CONFIG_DEBUG_LL
/*
 * Map in IO space for serial debugging.
 * This allows debug messages to be output
 * via a serial console before paging_init.
 */
add r0, r4, r7
rsb r3, r7, #0x4000 @ PTRS_PER_PGD*sizeof(long)
cmp r3, #0x0800
addge r2, r0, #0x0800
addlt r2, r0, r3
orr r3, r6, r8
```

```
1: str r3, [r0], #4
add r3, r3, #1 << 20
teq r0, r2
bne 1b
#if defined(CONFIG_ARCH_NETWINDER) || defined(CONFIG_ARCH_CATS)
/*
 * If we're using the NetWinder, we need to map in
 * the 16550-type serial port for the debug messages
 */
teq r1, #MACH_TYPE_NETWINDER
teqne r1, #MACH_TYPE_CATS
bne 1f
add r0, r4, #0x3fc0
mov r3, #0x7c000000
orr r3, r3, r8
str r3, [r0], #4
add r3, r3, #1 << 20
str r3, [r0], #4
1:
#endif
#endif
#ifdef CONFIG_ARCH_RPC
/*
 * Map in screen at 0x02000000 & SCREEN2_BASE
 * Similar reasons here - for debug.  This is
 * only for Acorn RiscPC architectures.
 */
add r0, r4, #0x80 @ 02000000
mov r3, #0x02000000
orr r3, r3, r8
str r3, [r0]
add r0, r4, #0x3600 @ d8000000
str r3, [r0]
#endif
(12)
mov pc, lr



/*
 * Exception handling.  Something went wrong and we can't
 * proceed.  We ought to tell the user, but since we
 * don't have any guarantee that we're even running on
 * the right architecture, we do virtually nothing.
 * r0 = ascii error character:
 * a = invalid architecture
 * p = invalid processor
 * i = invalid calling convention
 *
```

```
 * Generally, only serious errors cause this.
 */
__error:
#ifdef CONFIG_DEBUG_LL
mov r8, r0 @ preserve r0
adr r0, err_str
bl printascii
mov r0, r8
bl printch
#endif
#ifdef CONFIG_ARCH_RPC
/*
 * Turn the screen red on a error - RiscPC only.
 */
mov r0, #0x02000000
mov r3, #0x11
orr r3, r3, r3, lsl #8
orr r3, r3, r3, lsl #16
str r3, [r0], #4
str r3, [r0], #4
str r3, [r0], #4
str r3, [r0], #4
#endif
1: mov r0, r0
b 1b

#ifdef CONFIG_DEBUG_LL
err_str: .asciz "\nError: "
.align
#endif
```

(13)

```
/*
 * Read processor ID register (CP#15, CR0), and look up in the linker-
built
 * supported processor list.  Note that we can't use the abso-
lute addresses
 * for the __proc_info lists since we aren't running with the MMU on
 * (and therefore, we are not in the correct address space).  We have to
 * calculate the offset.
 *
 * Returns:
 * r5, r6, r7 corrupted
 * r8  = page table flags
 * r9  = processor ID
 * r10 = pointer to processor structure
 */
__lookup_processor_type:
adr r5, 2f
```

(14)

```
ldmia r5, {r7, r9, r10}
sub r5, r5, r10 @ convert addresses
add r7, r7, r5 @ to our address space
add r10, r9, r5
mrc p15, 0, r9, c0, c0 @ get processor id
```

(15)

```
1: ldmia r10, {r5, r6, r8} @ value, mask, mmuflags
and r6, r6, r9 @ mask wanted bits
teq r5, r6
moveq pc, lr
add r10, r10, #36 @ sizeof(proc_info_list)
cmp r10, r7
blt 1b
mov r10, #0 @ unknown processor
mov pc, lr
```

(16)

```
/*
 * Look in include/asm-arm/procinfo.h and arch/arm/kernel/arch.[ch] for
 * more information about the __proc_info and __arch_info structures.
 */
2: .long __proc_info_end
.long __proc_info_begin
.long 2b
.long __arch_info_begin
.long __arch_info_end
```

(17)

```
/*
 * Lookup machine architecture in the linker-build list of architectures.
 * Note that we can’t use the absolute addresses for the __arch_info
 * lists since we aren’t running with the MMU on (and therefore, we are
 * not in the correct address space).  We have to calculate the offset.
 *
 *  r1 = machine architecture number
 * Returns:
 *  r2, r3, r4 corrupted
 *  r5 = physical start address of RAM
 *  r6 = physical address of IO
 *  r7 = byte offset into page tables for IO
 */
__lookup_architecture_type:
adr r4, 2b
ldmia r4, {r2, r3, r5, r6, r7} @ throw away r2, r3
sub r5, r4, r5 @ convert addresses
add r4, r6, r5 @ to our address space
add r7, r7, r5
1: ldr r5, [r4] @ get machine type
```

```
teq r5, r1
beq 2f
add r4, r4, #SIZEOF_MACHINE_DESC
cmp r4, r7
blt 1b
mov r7, #0 @ unknown architecture
mov pc, lr
2: ldmib r4, {r5, r6, r7} @ found, get results
mov pc, lr
```

<10122> 압축 풀린 커널의 실행 시작 위치로 0xc0008000이 된다. 바로 밑의 #if define으로 된 두 부분은 Assabet 보드의 경우엔 해당 사항 없기 때문에 무시되고 넘어간다.

<10123> Assabet 보드의 실제 실행 코드 시작점으로 인터럽트를 불가능으로 만들고 프로세서 타입 아키텍져 타입을 알아낸 후 필요한 페이지 테이블을 만들고 프로세서를 설정한다.

<10124> 페이지 테이블을 만든 후 무언가를 실행하는데 돌아오는 위치는 __ret가 되도록 한다. 실행되는 그 무엇은 r10 + #12로 Assabet보드의 경우 __sa1100_setup 이란 곳이된다. 이곳은 $(TOPDIR)/arch/arm/mm/proc-sa110.S에 정의되어 있다.

<10125> __sa1100_setup이 실행된 후 돌아오는 곳이 이곳인데 다시 __switch_data란 곳으로 분기할 준비를 한다. 즉 __mmap_switched를 실행하게 된다.

<10126> 이제 모든 준비가 끝났다. 리눅스 커널을 시작한다. start_kernel은 $(TOPDIR)/init/main.c에 정의되어 있다.

<10127> 초기 페이지 테이블을 설정한다. 여기서는 커널이 실행될 만큼만 설정하고 커널이 초기화 되면서 설정된다. 4MB 정도면 커널이 실행되는덴 충분하므로 이정도만 설정한다.

<10128> 페이지 테이블의 시작 어드레스를 계산한다. r4=0xc0004000이 된다.

<10129> 이미 설정되어 있던 페이지 테이블을 모두 지운다.

<10130> 페이지 테이블에 들어갈 디스크립터 값을 설정한다. krnladr은 커널의 시작이 위치한 곳을 1MB 단위로 끊어준 값이다. 여기선 페이지 테이블을 섹션 디스크립터로 채우기 때문에 1MB 단위로 끊을 필요가 있는 것이다. r2=0xc0000000이 된다(램의 시작점).

섹션 디스크립터의 값은 AP=11, Domain=0, IMP=0, C=1, B=1의 속성을 갖는다(0xC0E).

[r4, r2, lsr #18]이 의미하는 것은 r2를 18bit Left Shift하고 r4에 더한다는 뜻으로, 최종 값은 0xc0007000이 된다.

이 값이 의미하는 것은 페이지 테이블 내의 커널 시작 어드레스를 가리키는 위치가 된다. 섹션 디스크립터는 1MB단위로 메모리를 관리하고 32bit ARM 프로세서의 최대 가능 용량인 4GB를 다루기 위해선 4G/1M=4K 만큼의 디스크립터가 필요하다. 또 각 디스크립터는 4Bytes로 구성되므로 페이지 테이블은 4K*4Bytes=16KB 만큼의 크기가 필요하다.

그렇다면 커널이 시작하는 곳의 디스크립터는 16KB내의 어디에 위치하는 것일까? 커널의 시작이 포함된 1MB 단위의 램 시작점이 0xc0000000 이므로 0xc0000000/1M*4=0x3000이 된다. 페이지 테이블의 시작이 0xc0004000이므로 여기에

0x3000을 더한 0xc0007000에 위치한 디스크립터가 바로 커널의 시작 위치를 가리키는 디스크립터가 된다.

(10) 위에서 말한대로 필요한 4MB 만을 위한 디스크립터를 설정한다.

(11) 램이 32MB 로 정렬됐고 커널이 같은 32MB 내에서 실행된다고 가정하고 이에 해당하는 디스크립터를 조정한다.

(12) 모든 페이지 테이블 설정을 마치고 되돌아간다.

(13) 프로세서 타입을 알아낸다.

(14) 첫 부분에 ldmia로 값을 읽어 내는데 r7=__proc_info_end, r9=__proc_info_begin, r10=2f의 어드레스가 된다.

(15) 프로세서에 대한 정보는 SA-110, SA-1100, SA-1110의 정보가 연속해 존재하므로 SA-110 부터 시작해 현재 실행되고 있는 프로세서와 비교해 맞는 것을 골라 정보를 얻는다.

(16) __proc_info_end 등은 $(TOPDIR)/arch/arm/vmlinux.lds에 정의되어 있다.

그리고 이 값은 $(TOPDIR)/arch/arm/mm/proc-sa110.S에 다음과 같이 정의 되어 있다.

```
__sa1110_proc_info:
.long 0x6901b110
.long 0xfffffff0
.long 0x00000c0e
b __sa1100_setup
.long cpu_arch_name
.long cpu_elf_name
.long HWCAP_SWP | HWCAP_HALF | HWCAP_26BIT | HWCAP_FAST_MULT
.long cpu_sa1110_info
.long sa1100_processor_functions
.size __sa1110_proc_info, . - __sa1110_proc_info
```

(17) 아키텍쳐 타입에 대한 정보를 얻어낸다. $(TOPDIR)/arch/arm/mach-sa1100/assabet.c에 다음과 같이 정의 되어 있다.

```
MACHINE_START(ASSABET, "Intel-Assabet")
BOOT_MEM(0xc0000000, 0x80000000, 0xf8000000)
BOOT_PARAMS(0xc0000100)
FIXUP(fixup_assabet)
MAPIO(assabet_map_io)
INITIRQ(sa1100_init_irq)
MACHINE_END
```

위 정의에 대한 매크로는 $(TOPDIR)/include/asm-arm/mach/arch.h에 다음과 같이 정의 되어 있다.

```
/*
 * Set of macros to define architecture features.  This is built into
 * a table by the linker.
 */
#define MACHINE_START(_type,_name) \
const struct machine_desc __mach_desc_##_type \
```

```
 __attribute__((__section__(".arch.info"))) = { \
nr: MACH_TYPE_##_type, \
name: _name,

#define MAINTAINER(n)

#define BOOT_MEM(_pram,_pio,_vio) \
phys_ram: _pram,                 \
phys_io: _pio,                   \
io_pg_offst: ((_vio)>>18)&0xfffc,

#define BOOT_PARAMS(_params) \
param_offset: _params,

#define VIDEO(_start,_end) \
video_start: _start, \
video_end: _end,

#define DISABLE_PARPORT(_n) \
reserve_lp##_n: 1,

#define BROKEN_HLT /* unused */

#define SOFT_REBOOT \
soft_reboot: 1,

#define FIXUP(_func) \
fixup: _func,

#define MAPIO(_func) \
map_io: _func,

#define INITIRQ(_func) \
init_irq: _func,

#define MACHINE_END \
};
```

5장. 리눅스 커널 부팅

이 장에서는 리눅스 커널의 압축이 풀린 후 실행되는 start_kernel() 부터 init가 실행될 때 까지의 절차를 추적해 보고 필요한 것들을 분석해 본다.

5.1. 커널 시작

4.4.2절의 <10126>에서 start_kernel이 불리는데 여기 부터가 일반 적인 커널의 시작이라고 생각하면 된다.

start_kernel 전 까지는 리눅스 커널이 실행되기 위한 기본 적인 초기화 등을 해놓은 상태라고 생각하면 된다. 아래에 start_kernel()만을 발췌해 놨다. 또 커널 부팅 중 남은 기록은 5.8절를 참조 바란다.

```
/*
 * Activate the first processor.
 */

asmlinkage void __init start_kernel(void)
{
char * command_line;
unsigned long mempages;
extern char saved_command_line[];
/*
 * Interrupts are still disabled. Do necessary setups, then
 * enable them
 */
<10122>
lock_kernel();
<10123>
printk(linux_banner);
<10124>
setup_arch(&command_line);
<10125>
printk("Kernel command line: %s\n", saved_command_line);
<10126>
parse_options(command_line);
<10127>
trap_init();
<10128>
init_IRQ();
<10129>
sched_init();
<10130>
softirq_init();
(10)
```

```
time_init();

/*
 * HACK ALERT! This is early. We're enabling the console before
 * we've done PCI setups etc, and console_init() must be aware of
 * this. But we do want output early, in case something goes wrong.
 */
(11)
console_init();
#ifdef CONFIG_MODULES
(12)
init_modules();
#endif
if (prof_shift) {
unsigned int size;
/* only text is profiled */
prof_len = (unsigned long) &_etext - (unsigned long) &_stext;
prof_len >>= prof_shift;

size = prof_len * sizeof(unsigned int) + PAGE_SIZE-1;
prof_buffer = (unsigned int *) alloc_bootmem(size);
}

(13)
kmem_cache_init();
(14)
sti();
(15)
calibrate_delay();
#ifdef CONFIG_BLK_DEV_INITRD
if (initrd_start && !initrd_below_start_ok &&
initrd_start < min_low_pfn << PAGE_SHIFT) {
printk(KERN_CRIT "initrd overwritten (0x%08lx < 0x%08lx) - "
    "disabling it.\n",initrd_start,min_low_pfn << PAGE_SHIFT);
initrd_start = 0;
}
#endif
(16)
mem_init();
(17)
kmem_cache_sizes_init();
pgtable_cache_init();

mempages = num_physpages;

(18)
fork_init(mempages);
(19)
proc_caches_init();
```

```
(20)
vfs_caches_init(mempages);
buffer_init(mempages);
page_cache_init(mempages);
#if defined(CONFIG_ARCH_S390)
ccwcache_init();
#endif
signals_init();
#ifdef CONFIG_PROC_FS
proc_root_init();
#endif
#if defined(CONFIG_SYSVIPC)
(21)
ipc_init();
#endif
(22)
check_bugs();
printk("POSIX conformance testing by UNIFIX\n");

/*
 * We count on the initial thread going ok
 * Like idlers init is an unlocked kernel thread, which will
 * make syscalls (and thus be locked).
 */
(23)
smp_init();
(24)
rest_init();
}
```

<10122> 5.2절 참조

<10123> linux_banner는 init/version.c에 다음과 같이 정의되어 있다.

```
const char *linux_banner =
"Linux version " UTS_RELEASE " (" LINUX_COMPILE_BY "@"
LINUX_COMPILE_HOST ") (" LINUX_COMPILER ") " UTS_VERSION "\n";
```

이 내용은 부팅할 때 아래와 같이 출력되고 /var/log/dmesg의 첫 줄에 기록된다.

```
Linux version 2.4.16 (root@localhost) (gcc version 2.95.3 20010315 (re-
lease)) #22 2002. 02. 27. (수) 13:30:14 KST
```

<10124> 5.3절 참조

<10125> 화면에 command line option을 출력한다.

<10126> command line option을 해석한다. 여기에서 해석되는 것은 모두 커널 내부적으로 사용되는 것이고 init로 보내지진 않는다.

해석된 옵션은 ’=’가 있으면 환경 변수로 취급되고 없으면 옵션을 처리되 환경 변수는 envp_init[]에 담기고 옵션은 argv_init[]에 담긴다.

<10127> 5.4절 참조

<10128> 5.5절 참조

<10129> 5.6절 참조

<10130> 소프트웨어 인터럽트를 기본 시스템을 초기화 한다. bh에 대한 초기화가 이뤄지기도 한다.

(10) CMOS에서 시간을 읽고 CPU의 속도를 얻어낸다. /var/log/dmesg에 "Detected 1009.016 MHz processor."라고 출력되는 부분이기도 하다.

(11) 콘솔 디바이스를 초기화 한다. 모든 초기화를 수행하는 것은 아니고 초기에 필요한 정도만 하고 나머지는 나중에 한다. dmesg에 ’Console: colour VGA+ 132x43’를 출력한다.

(12) 모듈의 초기화를 하지만 i386, ARM에선 아무 것도 하지 않는다. ia64만 뭔가를 한다.

(13) 대부분의 슬랩 배정자(slab allocator)를 초기화 한다.

(14) 인터럽트를 가능하게 한다.

(15) BogoMIPS를 계산한다. dmesg에 ’Calibrating delay loop... 2011.95 BogoMIPS’라고 출력된다.

(16) max_mapnr, totalram_pages, high_memory를 계산하고 dmesg에 ’Memory: 512920k/524208k available (1213k kernel code, 10900k reserved, 482k data, 228k init, 0k highmem)’라고 출력한다.

(17) 슬랩 초기화를 마친다.

(18) uid_cache를 만들고 사용 가능한 메모리의 양에 따라 max_threads를 초기화하고 RLIMIT_NPROC을 max_threads/2로 정한다.

(19) procfs가 사용하는 데이터 스트럭쳐를 초기화 한다.

(20) VFS, VM, buffer cache 등에 대한 슬랩 캐시를 만든다.

(21) System V IPC가 지원되는 커널이라면 IPC 하부 시스템을 초기화 한다. 세마포어, 메시지큐, 공유메모리를 초기화 한다.

(22) 아키텍쳐에 따른 버그를 검사한다. 예를 들어 ia32의 "f00f 버그"다.

(23) 멀티 프로세서 시스템이 가능한 아키텍쳐의 경우에만 해당하는 내용으로 SMP를 초기화 한다.

(24) rest_init() 자체는 무척 간단하다. 우선 init 프로세스를 실행해 준다. 그리고 start_kernel()의 첫 부분에서 lock 했던 커널을 unlock 해주고 idle 상태로 들어간다. idle 상태로 들어가도 이미 init 프로세스가 생성된 후기 때문에 상관 없이 커널의 부팅은 진행된다. idle 프로세스는 0번의 프로세스 번호를 갖는다.

나머지에 대해선 5.7절 참조

5.2. lock_kernel()

lock_kernel()은 각 아키텍쳐 마다 하나씩 따로 정의되어 있다. 보통 $(TOPDIR)/include/asm-*/smplock.h 에 함수로 정의되어 있고 spark64, sh, cris 아키텍쳐는 매크로로 정의되어 있다.

5.2.1. Lock이 왜 필요하지?

리눅스는 CPU를 여러 개 가진 시스템에서도 실행되고 이를 지원하고 있다. 만약 여러 개의 CPU가 동시에 같은 변수의 값을 조정하고 읽는 경우가 생긴다면 어떻게 되겠는가? 아래의 간단한 예를 보면서 얘기해 보자.

```
very_important_count++;
```

두개의 CPU가 동시에 같은 코드를 실행했다면 아래와 같은 표 처럼 실행되야 맞다고 가정해 보자.

표 **5-1.** 예상 결과

Instance1	**Instance2**
read very_important_count (5)	
add 1 (6)	
write very_important_count (6)	
	read very_important_count (6)
	add 1 (7)
	write very_important_count (7)

그러나 이건 아마도 이런 식으로 실행될 수도 있다.

표 **5-2.** 가능한 결과

Instance1	**Instance2**
read very_important_count (5)	
	read very_important_count (5)
add 1 (6)	
	add 1 (6)
write very_important_count (6)	
	write very_important_count (6)

결과 적으로 예상은 Instance1이 실행됐을 때 값이 6이되고 Instance2가 실행됐을 때 7이 되길

바란 것이지만 표 5-2 처럼 서로 실행이 중첩되버리면 결과 값이 6으로 엉망이 될 수도 있다.

이런 일을 막기 위해 보통 락(lock)이란 것을 쓰는데 쉽게 말해 서로 중첩되서 실행되지 않도록 보장 하는 것이라고 이해하면 된다.

시스템을 초기화 하는 동안에 이런 일이 발생하면 시스템 초기화는 엉망이 되고 커널은 제대로 부팅할 수 없게 된다. 그래서 초기화 동안 락을 걸고 나중에 초기화가 다 끝나면 락을 풀어주게 된다.

5.2.2. Lock - 기초적 설명

멀티태스킹 OS에서 한 변수를 여러 개의 프로세스가 공유하고 이를 동시에 사용한다면 여러 프로세스 간의 연계된 시간에 따라 이 변수를 사용하는 것이 중첩되는데 이를 보통 레이스 컨디션(race condition) 이라고 부른다. 그리고 동시 발생 문제를 다루는 코드를 크리티컬 리전(critical region)이라 부른다. 리눅스는 SMP 상에서 동작하므로 이런 문제가 커널 디자인에 있어서 중요한 문제중 하나다.

위에서 말한 동시 접근과 같은 문제의 해결은 락(lock)을 이용하는 것이고 락은 한번에 하나의 접근만이 크리티컬 리전에 들어가도록 해서 해결한다.

락엔 두가지 타입이 있다. 하나는 스핀락(spinlock)으로 include/asm/spinlock.h에 정의되어 있다. 이 타입은 싱글홀더락(single-holder lock)이고 매우 작고 빠르고 아무데서나 사용할 수 있다.

두번째 타입은 세마포어로 include/asm/semaphore.h에 정의되어 있다. 세마포어는 보통 싱글홀더락 (mutex)으로 사용되지만 한번에 여러 홀더를 가질 수 있다. 사용자가 세마포어를 얻지 못하면 사용자의 프로세스는 큐에 넣어지고 세마포어가 사용 가능해질 때 깨어나게된다. 이 말은 프로세스가 기다리는 동안 CPU가 다른 뭔가를 한다는 것이다. 그러나 많은 경우에 그냥 기다릴 수 없을 때가 있다. 이 경우엔 스핀락을 대신 사용해야한다.

커널을 설정할 때 SMP 지원을 체크하지 않았다면 스핀락은 존재하지 않게 된다. 이런 결정은 아무도 동시에 실행하지 않고 락을 걸 이유가 없는 경우 아주 좋은 커널 디자인이라 말할 수 있다. 세마포어는 언제나 존재하는데 이는 사용자 프로세스간에 동기를 맞추기위해 필요하기 때문이다.

5.2.3. i386, ARM의 스핀락

i386 계열은 SMP 시스템이 존재 하기 때문에 스핀락이 정의되어 있고 사용되지만 ARM의 경우 SMP 시스템이 없기 때문에 스핀락이 정의되어 있지 않다. 그래서 ARM의 asm/smplock.h는 기본으로 정의된 것이 사용된다.

기본 정의된 스핀락은 include/linux/spinlock.h에 다음과 같이 정의되어 있다.

```
#define spin_lock(lock) (void)(lock) /* Not "unused variable". */
```

이에 반해 i386은 include/asm-i386/spinlock.h에 다음과 같이 정의되어 있다.

```
static inline void spin_lock(spinlock_t *lock)
{
#if SPINLOCK_DEBUG
__label__ here;
here:
if (lock->magic != SPINLOCK_MAGIC) {
printk("eip: %p\n", &&here);
BUG();
}
#endif
<10122>
__asm__ __volatile__(
spin_lock_string
:"=m" (lock->lock) : : "memory");
}
```

<10122> 인라인 어셈블리에 대해선 부록 C를 참조해 무슨 내용인지 확인 하기 바란다.

5.3. setup_arch()

setup_arch()는 arch/*/kernel/setup.c에 각 아키텍쳐에 따른 정의가 되어 있다.

여기서는 아키텍쳐(좀더 정확히는 타겟 보드에 따라)에 따른 설정을 한다. i386에서는 아래와 같은 정보를 수집하 거나 초기화 해 놓는다. CPU가 초기화 되면서 /var/log/dmesg에 "Initializing CPU#0"를 출력한다.

- 기본 루트 디바이스 선택
- 시스템에 연결되어 있는 드라이브 정보 수집
- 화면 정보 수집
- APM 정보 수집
- 시스템 정보 수집
- 램디스크 플래그 설정
- 메모리 영역 설정
- 메모리 매니저 변수 초기화
- 커맨드 라인 명령 해석
- 부팅할 때 사용하는 메모리 초기화
- 페이징 시스템 초기화
- 전원 관리 초기화

- 표준 롬 초기화

ARM 의 경우 i386과는 달리 프로세서 종류가 몇 가지 되므로 프로세서와 아키텍쳐 타입에 따른 설정을 마친 후 커맨드 라인 명령을 해석한다. 이어 메모리 설정을 초기화하고 페이지 설정도 한다.

5.4. trap_init()

트랩은 인터럽트와는 달리 정해진 곳으로 분기하도록 되어 있고 번호로 정해져 있다. 아래는 i386에서 정해져 있는 트랩의 일부를 열거한 것이다.

- 0 - divide_error
- 1 - debug
- 2 - nmi
- 3 - int3
- 4 - overflow
- 5 - bounds
- 6 - invalid_op
- 7 - device_not_available
- 8 - double_fault
- 9 - coprocessor_segment_overrun
- 10 - invalid_TSS
- 11 - segment_not_present
- 12 - stack_segment
- 13 - general_protection
- 14 - page_fault
- 15 - spurious_interrupt_bug
- 16 - coprocessor_error
- 17 - alignment_check
- 18 - machine_check
- 19 - simd_coprocessor_error

trap_init()에선 시스템 콜을 위한 초기화도 실행해 0x80을 시스템 콜에 사용하도록 해놓는다.

그리고 CPU를 초기화 한다. CPU 초기화에선 페이지, gdt, ldt, idt, tss 등이 설정되고 이를 사용할 수 있는 상태로 만들어 본격적인 커널 실행에 들어간다. /var/log/dmesg의 <10127>에 출력된 한 줄이 CPU의 초기화를 의미한다.

i386에서 trap을 초기화하는 함수인 _set_gate()는 C.2.2절를 참조 하기 바란다.

ARM 프로세서의 trap은 arch/arm/kernel/entry-armv.S나 arch/arm/kernel/entry-armo.S에 정의되어 있고 내용은 다음과 같다. 앞의 값은 vector의 offset을 말한다.

- 0x00000000 - reset
- 0x00000004 - Undefined instruction
- 0x00000008 - Software Interrupt(SWI)
- 0x0000000C - Prefetch Abort(Instruction fetch memory abort)
- 0x00000010 - Data Abort(Data Access memory abort)
- 0x00000018 - IRQ(Interrupt)
- 0x0000001C - FIQ(Fast Interrupt)

5.5. init_IRQ()

i386의 PC 계열에선 ISA 혹은 APIC를 지원하는 시스템인 경우에 따라 인터럽트 설정을 하고 타이머 인터럽트를 동작시킨다. 아직은 인터럽트가 사용가능하지 않으므로 인터럽트가 동작하진 않는다. 0x20 ~ 0x2f 까지의 벡터는 ISA 인터럽트용 벡터이고 0xf0 ~ 0xff는 SMP 시스템용 인터럽트 벡터로 사용된다. 나머지 0x30 ~ 0xee는 APIC가 사용한다. 단 0x80은 시스템 콜이 사용하므로 제외한다.

ARM Assabet 보드에 사용된 SA-1100 CPU의 경우 arch/arm/mach-sa1100/assabet.c에 정의된 것에 따라 sa1100_init_irq가 불리게 된다. 다음과 같다. ARM의 경우 cpu가 같아도 플랫폼이 다르거나 CPU의 종류도 많으므로 각 CPU나 시스템의 타입에 따라 다른 함수를 사용할 수 있도록 만들어져 있다.

```
MACHINE_START(ASSABET, "Intel-Assabet")
BOOT_MEM(0xc0000000, 0x80000000, 0xf8000000)
BOOT_PARAMS(0xc0000100)
FIXUP(fixup_assabet)
MAPIO(assabet_map_io)
INITIRQ(sa1100_init_irq)
MACHINE_END
```

5.6. sched_init()

init 태스크가 사용하는 CPU 번로를 할당해 주고 pid hash table을 초기화 한다. 이어 타이머 인터럽트 벡터를 초기화 한다.

인터럽트 처리 루틴은 되도록이면 간결하고 빨라야할 필요가 있다. 프로그램이 실행 중에 인터럽트가 걸리면 프로그램의 실행을 멈추고 인터럽트를 처리하므로 인터럽트 처리 시간이 많이 걸린다면 다른 프로그램의 실행에 영향을 미치게된다. 리눅스에선 긴 처리 시간을 필요로 하는 인터럽트 루틴 의 문제를 해결하기 위해 인터럽트 루틴을 둘로 나눠 이 문제를 해결한다.

이 둘을 top-half, bottom-half라고 부른다. top-half는 request_irq로 등록되는 부분이고 bottom-half(줄여서 bh)는 나중에 시간이 충분할 때 실행되도록 top-half에 의해 스케쥴 된다.

top-half와 bh의 차이라면 bh가 실행되는 동안엔 다른 모든 인터럽트가 가능 상태인 것이다. 즉 top-half는 인터럽트가 걸리면 처음 실행되고 디바이스의 데이터를 특정 버퍼에 저장해 놓고 자신의 bh에 표시를 한다음 빠져나간다. 이렇게 하면 top-half는 매우 빠르게 실행되기 때문에 다른 것에 영향을 주지 않게 된다.

그러나 만약 top-half가 동작하는 중에 다른 인터럽트가 걸리면 이 것은 무시된다. 왜냐면 top-half가 실행되는 동안엔 인터럽트 컨트롤러의 IRQ 라인이 불가능 상태이기 때문이다.

가장 대표적인 인터럽트 루틴인 네트워크 인터럽트 루틴은 새로운 패킷이 도착하면 핸들러가 도착한 데이터만을 읽어 프로토콜 레이어에 전달하고 실제의 처리는 나중에 bh에 의해 나중에 실행된다.

스케쥴러의 초기화에선 가장 근본적인 3개의 bh를 초기화한다. TIMER_BH, TQUEUE_BH, IMMEDIATE_BH의 3개이다.

5.7. init()

커널의 부팅에 필요한 기본 초기화(CPU, 메모리 등)가 끝나면 init 프로세스가 만들어지고 시스템에 존재하는 다른 하드웨어 등을 초기화 한다음 루트 디바이스를 찾아 나머지 부팅을 시작한다. init 프로세스는 1번 프로세스 번호를 갖는다.

```
static int init(void * unused)
{
lock_kernel();
<10122>
do_basic_setup();

<10123>
prepare_namespace();

/*
 * Ok, we have completed the initial bootup, and
 * we’re essentially up and running. Get rid of the
 * initmem segments and start the user-mode stuff..
 */
<10124>
free_initmem();
unlock_kernel();
```

```
<10125>
if (open("/dev/console", O_RDWR, 0) < 0)
printk("Warning: unable to open an initial console.\n");

(void) dup(0);
(void) dup(0);

/*
 * We try each of these until one succeeds.
 *
 * The Bourne shell can be used instead of init if we are
 * trying to recover a really broken machine.
 */

if (execute_command)
execve(execute_command,argv_init,envp_init);
<10126>
execve("/sbin/init",argv_init,envp_init);
execve("/etc/init",argv_init,envp_init);
execve("/bin/init",argv_init,envp_init);
execve("/bin/sh",argv_init,envp_init);
<10127>
execve("/sbin/init",argv_init,envp_init);
panic("No init found.  Try passing init= option to kernel.");
}
```

<10122> 여기 까지의 상태는 시스템이 이제 사용 가능한 상태까지는 왔지만 붙어 있는 다른 모든 디바이스에 대해선 초기화가 되지 않은 상태다. CPU 하부 시스템, 메모리 그리고 프로세스 관리는 동작하는 상태다.

이제 할 것은 나머지 디바이스 들을 모두 초기화 하는 일을 하는 것이다. 초기화 하는 목록은 다음과 같다. 이외에 더 있으나 중요한 것만 조금 간추렸다.

mtrr

현재는 i386에서만 존재 하는 기능으로 MTRR(Memory Type Range Register)를 말한다. PCI나 AGP 비디오 카드를 좀더 빨리 쓸 수 있도록 해준다.

sysctrl

proc file system을 사용하도록 설정 되어 있으면 이를 초기화 해준다.

pci

PCI 시스템을 초기화 한다. PCI의 루트 디바이스를 초기화 하고 이어 PCI 버스에 연결된 모든 다른 하드웨어를 찾아 리스트에 등록한다.

isapnp

ISA 버스에 물려 있는 PnP 디바이스를 초기화한다.

socket

사용되는 프로토콜을 초기화 한다. 소켓 용 슬랩도 초기화 하고 netlink, netfileter 등도 초기화 한다.

context thread

keventd를 kernel thread로 실행한다.

pcmcia

PCMCIA 디바이스 초기화 한다.

<10123> 무엇을 어디서 마운트할 지 결정한다. 루트 디바이스를 마운트하고 램디스크를 읽어들이는 일도 한다.

<10124> 바로 전까지 실행되면 이제 커널이 완전히 부팅한 것으로 봐도 된다. 커널 부팅에 사용된 메모리 중 필요 없는 것을 반환한다.

<10125> 초기 콘솔을 열고 stdin, stdout, stderr을 open 한다.

<10126> 이제 마운트된 루트 파일 시스템에서 init를 찾아 실행해 준다.

<10127> init를 찾지 못하면 여기와서 커널의 부팅이 멈춘다. 여기 까지 온다는 것은 아마도 루트 파일 시스템을 마운트하지 못했거나 루트 파일 시스템에 init가 없기 때문일 것이다.

5.8. dmesg 정리

/var/log/dmesg는 부팅하는 동안 커널의 기록을 남겨 놓은 파일이다. 이 파일의 출력을 구분지어 어느 단계에서 어떤 메시지가 출력되는지 보자.

단계를 구분지어 놓으면 start_kernel()을 분석하는데 많은 도움이 될 것이고 커널 부팅 중에 에러가 났다면 어느 단계에서 에러 났는지 범위를 좁히고 찾아내는데 많은 도움이 될 것이다.

```
<10122>
Linux version 2.4.16 (root@halite) (gcc version 2.95.3 20010315 (re-
lease)) #22 2002. 02. 27. (수) 13:30:14 KST
<10123>
BIOS-provided physical RAM map:
 BIOS-e820: 0000000000000000 - 000000000009fc00 (usable)
 BIOS-e820: 000000000009fc00 - 00000000000a0000 (reserved)
 BIOS-e820: 00000000000f0000 - 0000000000100000 (reserved)
 BIOS-e820: 0000000000100000 - 000000001ffec000 (usable)
 BIOS-e820: 000000001ffec000 - 000000001ffef000 (ACPI data)
 BIOS-e820: 000000001ffef000 - 000000001ffff000 (reserved)
 BIOS-e820: 000000001ffff000 - 0000000020000000 (ACPI NVS)
```

```
 BIOS-e820: 00000000ffff0000 - 0000000100000000 (reserved)
<10124>
On node 0 totalpages: 131052
zone(0): 4096 pages.
zone(1): 126956 pages.
zone(2): 0 pages.
<10125>
Local APIC disabled by BIOS -- reenabling.
Found and enabled local APIC!
<10126>
Kernel command line: BOOT_IMAGE=linux ro root=301 mem=nopentium hdd=ide-
scsi
ide_setup: hdd=ide-scsi
<10127>
Initializing CPU#0
<10128>
Detected 1009.016 MHz processor.
<10129>
Console: colour VGA+ 132x43
<10130>
Calibrating delay loop... 2011.95 BogoMIPS
(10)
Memory: 512920k/524208k available (1213k kernel code, 10900k re-
served, 482k data, 228k init, 0k highmem)
Checking if this processor honours the WP bit even in supervi-
sor mode... Ok.
(11)
Dentry-cache hash table entries: 65536 (order: 7, 524288 bytes)
(12)
Inode-cache hash table entries: 32768 (order: 6, 262144 bytes)
(13)
Mount-cache hash table entries: 8192 (order: 4, 65536 bytes)
(14)
Buffer-cache hash table entries: 32768 (order: 5, 131072 bytes)
(15)
Page-cache hash table entries: 131072 (order: 7, 524288 bytes)
(16)
CPU: Before vendor init, caps: 0183fbff c1c7fbff 00000000, vendor = 2
Intel machine check architecture supported.
Intel machine check reporting enabled on CPU#0.
CPU: L1 I Cache: 64K (64 bytes/line), D cache 64K (64 bytes/line)
CPU: L2 Cache: 256K (64 bytes/line)
CPU: After vendor init, caps: 0183fbff c1c7fbff 00000000 00000000
CPU:     After generic, caps: 0183fbff c1c7fbff 00000000 00000000
CPU:             Common caps: 0183fbff c1c7fbff 00000000 00000000
CPU: AMD Athlon(tm) Processor stepping 02
Enabling fast FPU save and restore... done.
Checking 'hlt' instruction... OK.
(17)
```

```
POSIX conformance testing by UNIFIX
(18)
enabled ExtINT on CPU#0
ESR value before enabling vector: 00000000
ESR value after enabling vector: 00000000
(19)
Using local APIC timer interrupts.
(20)
calibrating APIC timer ...
..... CPU clock speed is 1009.0421 MHz.
..... host bus clock speed is 201.8084 MHz.
(21)
cpu: 0, clocks: 2018084, slice: 1009042
CPU0<T0:2018080,T1:1009024,D:14,S:1009042,C:2018084>
(22)
mtrr: v1.40 (20010327) Richard Gooch (rgooch@atnf.csiro.au)
mtrr: detected mtrr type: Intel
(23)
PCI: PCI BIOS revision 2.10 entry at 0xf1180, last bus=1
(24)
PCI: Using configuration type 1
(25)
PCI: Probing PCI hardware
(26)
Unknown bridge resource 0: assuming transparent
(27)
PCI: Using IRQ router VIA [1106/0686] at 00:04.0
PCI: Found IRQ 10 for device 00:0b.0
PCI: Sharing IRQ 10 with 00:11.0
PCI: Found IRQ 5 for device 00:0d.0
PCI: Sharing IRQ 5 with 00:04.2
PCI: Sharing IRQ 5 with 00:04.3
PCI: Disabling Via external APIC routing
(28)
isapnp: Scanning for PnP cards...
isapnp: No Plug & Play device found
(29)
Linux NET4.0 for Linux 2.4
Based upon Swansea University Computer Society NET3.039
(30)
Initializing RT netlink socket
(31)
apm: BIOS version 1.2 Flags 0x03 (Driver version 1.15)
(32)
Starting kswapd
JFS development version: $Name:  $
(33)
ACPI: APM is already active, exiting
(34)
```

```
pty: 256 Unix98 ptys configured
```

(35)

```
Serial driver version 5.05c (2001-07-
08) with MANY_PORTS SHARE_IRQ SERIAL_PCI ISAPNP enabled
ttyS01 at 0x02f8 (irq = 3) is a 16550A
```

(36)

```
block: 128 slots per queue, batch=32
```

(37)

```
Uniform Multi-Platform E-IDE driver Revision: 6.31
```

(38)

```
ide: Assuming 33MHz system bus speed for PIO modes; over-
ride with idebus=xx
```

(39)

```
VP_IDE: IDE controller on PCI bus 00 dev 21
VP_IDE: chipset revision 16
VP_IDE: not 100% native mode: will probe irqs later
VP_IDE: VIA vt82c686a (rev 22) IDE UDMA66 controller on pci00:04.1
    ide0: BM-DMA at 0xd800-0xd807, BIOS settings: hda:DMA, hdb:DMA
    ide1: BM-DMA at 0xd808-0xd80f, BIOS settings: hdc:DMA, hdd:DMA
```

(40)

```
PDC20265: IDE controller on PCI bus 00 dev 88
PCI: Found IRQ 10 for device 00:11.0
PCI: Sharing IRQ 10 with 00:0b.0
PDC20265: chipset revision 2
PDC20265: not 100% native mode: will probe irqs later
    ide2: BM-DMA at 0x8000-0x8007, BIOS settings: hde:DMA, hdf:DMA
    ide3: BM-DMA at 0x8008-0x800f, BIOS settings: hdg:DMA, hdh:pio
```

(41)

```
hda: Maxtor 4W080H6, ATA DISK drive
hdb: IC35L040AVER07-0, ATA DISK drive
hdc: QUANTUM FIREBALLlct15 20, ATA DISK drive
hdd: LG CD-RW CED-8080B, ATAPI CD/DVD-ROM drive
```

(42)

```
ide0 at 0x1f0-0x1f7,0x3f6 on irq 14
ide1 at 0x170-0x177,0x376 on irq 15
hda: 160086528 sec-
tors (81964 MB) w/2048KiB Cache, CHS=9964/255/63, UDMA(33)
hdb: 80418240 sec-
tors (41174 MB) w/1916KiB Cache, CHS=5005/255/63, UDMA(33)
hdc: 39876480 sectors (20417 MB) w/418KiB Cache, CHS=39560/16/63, UDMA(33)
```

(43)

```
Partition check:
 hda: hda1 hda2
 hdb: hdb1
 hdc: [PTBL] [2482/255/63] hdc1 hdc2 hdc3
```

(44)

```
Floppy drive(s): fd0 is 1.44M
FDC 0 is a post-1991 82077
```

(45)

```
Linux agpgart interface v0.99 (c) Jeff Hartmann
agpgart: Maximum main memory to use for agp memory: 439M
agpgart: Detected Via Apollo Pro KT133 chipset
agpgart: AGP aperture is 128M @ 0xe0000000
[drm] AGP 0.99 on VIA Apollo KT133 @ 0xe0000000 128MB
[drm] Initialized mga 3.0.2 20010321 on minor 0
(46)
Linux Kernel Card Services 3.1.22
  options:  [pci] [cardbus] [pm]
(47)
NET4: Linux TCP/IP 1.0 for NET4.0
IP Protocols: ICMP, UDP, TCP, IGMP
IP: routing cache hash table of 4096 buckets, 32Kbytes
TCP: Hash tables configured (established 32768 bind 32768)
NET4: Unix domain sockets 1.0/SMP for Linux NET4.0.
(48)
ds: no socket drivers loaded!
(49)
request_module[nls_EUC-KR]: Root fs not mounted
Unable to load NLS charset EUC-KR
(50)
VFS: Mounted root (jfs filesystem) readonly.
(51)
Freeing unused kernel memory: 228k freed
```

<10122> start_kernel()

<10123> setup_memory_region()/setup_arch()/start_kerenel()

<10124> free_area_init_core()/free_area_init()/paging_init()/setup_arch()/start_kernel()

<10125> detect_init_APIC()/init_apic_mappings()/setup_arch()/start_kernel()

<10126> start_kernel()

<10127> cpu_init()/trap_init()/start_kernel()

<10128> time_init()/start_kernel()

<10129> con_init()/console_init()/start_kernel()

<10130> calibrate_delay()/start_kernel()

(10) mem_init()/start_kernel()

(11) dcache_init()/vfs_caches_init()/start_kernel()

(12) inode_init()/vfs_caches_init()/start_kernel()

(13) mnt_init()/vfs_caches_init()/start_kernel()

(14) buffer_init()/start_kernel()

(15) page_cache_init()/start_kernel()

(16) identify_cpu()/check_bugs()/start_kernel()

(17) start_kernel()

(18) setup_local_APIC()/APIC_init_uniprocessor()/smp_init()/start_kernel()

(19) setup_APIC_clocks()/APIC_init_uniprocessor/smp_init()/start_kernel()

(20) calibrate_APIC_clock()/setup_APIC_clocks()/APIC_init_uniprocessor/smp_init()/start_kernel()

(21) setup_APIC_timer()/setup_APIC_clocks()/APIC_init_uniprocessor/smp_init()/start_kernel()

(22) mtrr_setup()/mtrr_init()/do_basic_setup()/init()/rest_init()/start_kernel()

(23) check_pcibios()/pci_find_bios()/pcibios_config_init()/pcibios_init()/pci_init()/do_basic_setup()/init()/rest_init()/start_kernel()

(24) pci_check_direct()/pcibios_config_init()/pcibios_init()/pci_init()/do_basic_setup()/init()/rest_init()/start_kernel()

(25) pcibios_init()/pci_init()/do_basic_setup()/init()/rest_init()/start_kernel()

(26) pci_read_bridge_bases()/pcibios_fixup_bus()/pci_do_scan_bus()/pci_scan_bus()/pcibios_init()/pci_init()/do_basic_setup()/init()/rest_init()/start_kernel()

(27) pirq_find_router()/pcibios_irq_init()/pcibios_init()/pci_init()/do_basic_setup()/init()/rest_init()/start_kernel()

(28) isapnp_init()/do_basic_setup()/init()/rest_init()/start_kernel()

(29) sock_init()/do_basic_setup()/init()/rest_init()/start_kernel()

(30) rtnetlink_init()/sock_init()/do_basic_setup()/init()/rest_init()/start_kernel()

(32) kswapd_init()/do_initcalls()/do_basic_setup()/init()/rest_init()/start_kernel()

(33) acpi_init()/do_initcalls()/do_basic_setup()/init()/rest_init()/start_kernel()

(34) pty_init()/do_initcalls()/do_basic_setup()/init()/rest_init()/start_kernel()

(35) show_serial_version()/rs_init()/do_basic_setup()/init()/rest_init()/start_kernel()

(36) blk_dev_init()/do_basic_setup()/init()/rest_init()/start_kernel()

(37) ide_init()/do_basic_setup()/init()/rest_init()/start_kernel()

(38) ide_system_bus_speed()/ide_init()/do_basic_setup()/init()/rest_init()/start_kernel()

(39) ide_scan_pcidev()/ide_scan_pcibus()/probe_for_hwifs()/ide_init_builtin_drivers()/ide_init()/do_basic_setup()/init()/rest_init()/start_kernel()

(40) ide_setup_pci_device()/ide_scan_pcidev()/ide_scan_pcibus()/probe_for_hwifs()/ide_init_builtin_drivers()/ide_init()/do_basic_setup()/init()/rest_init()/start_kernel()

(41) do_identify()/actual_try_to_identify()/try_to_identify()/do_probe()/probe_for_drive()/probe_hwif()/ideprobe_init()/init_module()/do_basic_setup()/init()/rest_init()/start_kernel()

(42) init_irq()/hwif_init()/ideprobe_init()/init_module()/do_basic_setup()/init()/rest_init()/start_kernel()

(43) check_partition()/grok_partitions()/idedisk_revalidate/init_module()/do_basic_setup()/init()/rest_init()/start_kernel()

(44) config_types()/floppy_init()/do_basic_setup()/init()/rest_init()/start_kernel()

(45) agp_init()/do_basic_setup()/init()/rest_init()/start_kernel()

(46) init_pcmcia_cs()/do_basic_setup()/init()/rest_init()/start_kernel()

(47) inet_init()/do_basic_setup()/init()/rest_init()/start_kernel()

(48) init_pcmcia_ds()/do_basic_setup()/init()/rest_init()/start_kernel()

(49) request_module()/load_nls()/init_nls_euc_kr()/do_basic_setup()/init()/ rest_init()/start_kernel()

(50) mount_root()/prepare_namespace()/init()/rest_init()/start_kernel()

(51) free_initmem()/init()/rest_init()/start_kernel()

커널의 초기화 동안 등록된 드라이버들이 자동으로 실행되고 초기화 되도록 하는데, 커널을 어떻게 설정하는가에 따라 커널에 포함되는 것이 다르므로 일일히 기록하는 대신에 자동으로 커널 컴파일하는 동안 포함되도록 할 수 있다. 필요한 모듈에 __init란 속성을 사용하고 module_init()를 사용해 이런 일이 가능하도록 한다.

module_init()는 include/linux/init.h에 다음과 같이 정의되어 있다.

```
#define module_init(x)  __initcall(x);
```

__initcall(x)는 module로 지정됐는가 아닌가에 따라 정의가 달라지는데 어찌됐든 .initcall.init란 섹션에 속하게 된다. 여기에 속하면 커널의 링킹 동안 모아진 .initcall.init가 do_basic_setup() 혹은 do_initcalls()에 의해 자동으로 불려지게 된다(vmlinux.lds를 보면 .initcall.init를 모아주는 부분이 있다).

6장. 디바이스 드라이버

임베디스 시스템 개발자가 커널의 포팅 이후에 필요한 작업이 자신이 만든 새로운 시스템에 존재하는 많은 디바이스를 사용하는 것이다. 예를 들어 일반적이지 않는 통신 포트를 하나 갖고 있는 시스템을 만들었고 이미 커널은 포팅되서 동작한다고 가정하면 이 통신 포트를 사용하기 위한 일을 해줘야한다.

보통은 이 포트에 대한 디바이스 드라이버를 만들어 사용하면 될 것이다. 그렇다면 리눅스에선 어떤 식으로 디바이스 드라이버가 만들어지는지 알아보자.

6.1. 디바이스 번호

처음 유닉스접했을 때(사실 유닉스라기 보다는 리눅스가 맞겠다) /dev 디렉토리에 들어 있는 것들이 무엇인지 궁금한 적이 있었다. 사운드 블래스터 카드에 붙이는 CD-ROM 드라이브를 사용하기 위해 How-to 문서를 뒤적여 mknod란 것도 처음 사용해 보곤 했지만 /dev 디렉토리 내의 파일이 갖는 정확 한 의미를 알지는 못했다.

나중에 리눅스에서 프로그래밍을 하면서 깨닫게 됐지만 /dev 디렉토리가 의미하는 것은 device의 약 자이고 여기에 들어있는 것들은 어떤 물리적인 디바이스를 나타낸다는 것을 알았다.

예를 들어o/dev/ttyS0와 /dev/fd0는 다음과 같다. 참고로 ttyS0는 시리얼 포트 1번을 의미하고(일반 적으로 PC에서 COM1으로 불린다) fd0는 플로피 디스크 드라이브 첫 번째 것으로 a:를 의미한다.

```
crw-rw----    1 root     uucp       4,  64  4월 30 11:23 /dev/ttyS0
brw-rw----    1 root     floppy     2,   0  4월 30 11:23 /dev/fd0
```

ttyS0는 속성에 보면 ’c’가 처음에 시작하는데 이 문자가 의미하는 것은 ’character device’ 즉 문자 디바이스를 의미한다. 이에 반해 fd0는 ’b’로 시작하고 ’block device’ 즉 블럭 디바이스를 의미한다. 문자/블럭 디바이스에 대한 내용은 다음을 참조 하라.

문자 디바이스

> 문자 디바이스는 하나 혹은 수십 내지 수백 개의 가변 크기의 버퍼를 사용해 디바이스에 일고 쓰기를 한다.

블럭 디바이스

> 블럭이라 불리는 일정 크기의 버퍼(512, 1K Bytes등, 장치 의존적)단위로 데이터의 읽기 쓰기가 행해진다.

그리고 나오는 것이 owner, group인데 이 것은 각 디바이스에 따라 다르므로 각 디바이스의 속석을 참조하기 바란다.

그리고 여기서 얘기해야할 가장 중요한 부분이 나오는데 일반적인 파일의 경우 'ls -l'로 보면 그룹을 나타내는 곳 뒤에 파일의 크기가 나오지만 /dev 내의 것은 크기가 아니라 두 개의 숫자가 나온다.

이 숫자가 의미하는 것은 디바이스의 번호로 리눅스 시스템에서 혹은 유닉스 시스템에서는 정해진 유일한 번호를 갖는다. 즉 ttyS0는 무조건 4, 64의 번호를 가져야 이 디바이스가 첫번째 시리얼 포 트를 나타내게 된다. 번호를 4, 64를 갖고 이름이 다른 경우라도 상관 없이 첫번째 시리얼 포트를 나타낸다.

4, 64에서 첫번째 것은 주(major) 디바이스 번호고 두번째 것은 부(minor) 디바이스 번호다. 주 번호가 의미하는 것은 이 디바이스가 무엇인지를 나타내고 부 번호가 나타내는 것은 이 디바이스의 몇 번째 것 혹은 여러 종류 중의 구분을 의미한다. 예를 들어 PC엔 시리얼 포트가 적어도 2개가 있고 많게는 4개 까지도 존재한다. 이런 경우라면 여러개 모두 시리얼 포트이기 때문에 주 번호는 시리얼 포트란 것을 나태내도록 통일해주고(4번이 시리얼 포트를 의미한다) 부 번호를 사용해 각각의 시리얼 포트를 구분해 주게된다(64번이 COM1, 65번이 COM2).

리눅스 내의 모든 디바이스는 반드시 주/부 디바이스 번호를 사용해 구분되야한다. 번호가 같은 디 바이스는 이름이 다를지라도 같은 디바이스를 의미한다. 아래는 필자의 PC에 사용중인 다비이스들 을 나열한 것이다. 이 정보는 /proc/device를 읽어 보면 알 수 있다. 각 디바이스의 이름 앞 번호는 디바이스의 주 번호를 의미한다.

```
Character devices:
  1 mem
  2 pty
  3 ttyp
  4 ttyS
  5 cua
  7 vcs
 10 misc
 13 input
 14 sound
 29 fb
116 alsa
119 vmnet
128 ptm
136 pts
162 raw
180 usb
226 drm

Block devices:
  2 fd
  3 ide0
  8 sd
  9 md
 11 sr
 22 ide1
 65 sd
 66 sd
```

리눅스에 등록된 디바이스 리스트는 커널 소스 디렉토리의 Documents/devices.txt에 있거나 *http://www.lanana.org/docs/device-list/* (http://www.lanana.org/docs/device-list/)에서 확인할 수 있다.

주/부 번호 모두 255까지 가능하다. 위의 리스트에 등록됐다고 하지만 실제 많이 쓰이지 않는 디바이스에 대한 것은 /dev에 존재하지 않는 경우도 있고 새로 필요해 만든 디바이스 드라이버에게 번호를 할당하기 위해선 필요 없는 것에서 번호를 선택하거나 아니면 현재 시스템을 검색해 사용하지 않는 번호를 할당할 수도 있다.

리눅스에서 사용 가능한 주번호는 60~63, 120~127, 240~254 까지고 여기엔 아무런 것도 할당되어 있지 않은 상태다.

6.2. 샘플 디바이스 드라이버

디바이스 드라이버를 만드는 방법에 대해 알아보자. 우선 대부분의 문서에 나오는 간단한 예제를 이용해보자.

```
/* hello.c */
<10122>
#ifndef __KERNEL__
#define __KERNEL__
#endif
#ifndef MODULE
#define MODULE
#endif

<10123>
#define __NO_VERSION__
#include <linux/module.h>
#include <linux/version.h>
#include <linux/fs.h>

<10124>
struct file_operations Fops = {
NULL, /* owener */
NULL, /* llseek */
NULL, /* read */
NULL, /* write */
NULL, /* readdir */
NULL, /* poll */
NULL, /* ioctl */
NULL, /* mmap */
NULL, /* open */
NULL, /* flush */
NULL, /* release */
NULL, /* fsync */
NULL, /* lock */
```

```
NULL, /* readv */
NULL, /* writev */
NULL, /* sendpage */
NULL /* get_unmapped_area */
};

<10125>
int init_module()
{
if (register_chrdev(213, "hello", &Fops) < 0)
return -EIO;

printk("hello.o start\n");

return 0;
}

<10126>
void cleanup_module()
{
unregister_chrdev(213, "hello");
printk("hello.o end\n");
}
```

컴파일은 다음과 같이 한다.

```
gcc -o hello.o -c -D__KERNEL__ -DMODULE -O -Wall -I/usr/include hello.c
```

컴파일 할 때 컴파일러에게 커널에 해당하는 프로그램임과 이 것이 MODULE 임을 알려준다. 일반 적인 프로그램과는 달리 -c 옵션을 사용해 링킹을 하진 않아야한다.

에러 없이 컴파일 됐으면 다음과 같은 명령으로 만들어진 디바이스 드라이버를 등록해보자.

```
insmod hello.o
```

화면에 뭔가 출력되는가? 아무 것도 출력되지 않으면 dmesg 명령을 사용해 커널에서 뿌린 메시지의 마지막에 'hello.o start'가 찍혔는지 확인한다. 또 lsmod 명령으로 정상적으로 hello.o가 등록됐느지 확인해 보자. 아래 것은 필자의 리눅스 시스템에 올라간 모듈들을 본것이다. 제일 위에 hello.o가 등록된 것이 보일 것이다. 비록 아무 것도 사용하지 않는다고 나와 있지만 처음 만들어본 디바이스 드라이버가 등록된 것을 확인할 수 있다.

```
Module                  Size  Used by
hello                    592   0  (unused)
smbfs                  31376   4  (autoclean)
sd_mod                 10640   2  (autoclean)
vfat                    9520   1  (autoclean)
fat                    29696   0  (autoclean)  [vfat]
```

```
sr_mod                 12176   0  (autoclean)
tuner                   8176   0  (autoclean) (unused)
i2c-core               13024   0  (autoclean) [tuner]
vmnet                  17984   6
parport_pc             19648   0  (unused)
parport                14176   0  [parport_pc]
vmmon                  18784   0  (unused)
3c59x                  24960   1
ide-scsi                7648   0
ide-cd                 26656   0
cdrom                  29056   0  [sr_mod ide-cd]
md                     44224   0  (unused)
snd-pcm-oss            35792   0  (unused)
snd-mixer-oss           8528   1  [snd-pcm-oss]
snd-card-fm801          7296   1
snd-pcm                47744   0  [snd-pcm-oss snd-card-fm801]
snd-mpu401-uart         2656   0  [snd-card-fm801]
snd-rawmidi            11968   0  [snd-mpu401-uart]
snd-ac97-codec         22832   0  [snd-card-fm801]
snd-opl3                5264   0  [snd-card-fm801]
snd-timer               9584   0  [snd-pcm snd-opl3]
snd-hwdep               3376   0  [snd-opl3]
snd-seq-device          3744   0  [snd-rawmidi snd-opl3]
snd                    23632   0  [snd-pcm-oss snd-mixer-oss snd-card-
fm801 snd-pcm snd-mpu401-uart snd-rawmidi snd-ac97-codec snd-opl3 snd-
timer snd-hwdep snd-seq-device]
hid                    19152   0  (unused)
input                   3360   0  [hid]
usb-storage            26400   1
scsi_mod               88400   4  [sd_mod sr_mod ide-scsi usb-storage]
usb-uhci               21408   0  (unused)
usbcore                49632   1  [hid usb-storage usb-uhci]
rtc                     5600   0  (autoclean)
```

이어 'rmmod hello'란 명령을 실행해 보자. insmod 때와 마찬가지로 화면에 아무 것도 출력되지 않으면 dmesg를 사용해 또 확인해 보자. 'hello.o end'란 말이 출력됐는가? 이어 lsmod를 사용해 hello.o가 해제됐는지 확인해 보기 바란다.

hello.c를 간단하게 분석해 보자.

<10122> 모듈로 만들어지는 디바이스 드라이버는 __KERNEL__과 MODULE이 반드시 정의되야만 한다.

<10123> 필요한 헤더를 읽어 들인다.

<10124> file_operations 라는 구조체로 모든 모듈엔 반드시 존재해야한다. hello.c는 아무 동작도 하지 않기 때문에 여기에 아무 것도 채워 넣지 않았지만 다른 디바이스 드라이버를 만들 땐 알맞는 항목을 채워 넣어 동작 하도록 해줘야한다.

<10125> init_module()은 insmod 명령 등을 이용해 디바이스 드라이버를 커널에 등록할 때 무조건 처음 실행되는 함수다. 다시 말해 모든 모듈엔 init_module()과 cleanup_module()이 존재 해야한다. 보통은 init_module()에서 디바이스 드라이버를 주/부 번호를 사용해 등록하는 함수를 실행한다.

<10126> cleanup_module()은 모듈을 제거할 때 커널에 의해 무조건 불리는 함수로 init_module()과는 반대로등록된 모듈을 해제하는 함수를 부른다.

만약 위의 예제를 X-window 상에서 insmod/rmmod 한다면 화면에 아무 것도 나오지 않을 것이다. 이는 printk의 출력이 가상 터미널 7번에 출력되기 때문이다. 그러므로 xterm에 옵션을 주지 않고 그냥 연 창엔 출력되지 않으므로 demsg를 사용해 확인해야한다. 대신 xterm -C로 연 xterm에선 바로 확인이 가능할 것이다.

예제는 그 크기가 작기 때문에 하나의 파일에 모두 들어가지만 일반적인 경우 한개의 파일에 모듈 전체의 내용이 들어가지 않을 경우엔 소스 코드 여러개에 나눠 쓰게 된다. 이런 경우엔 각각의 파일에 모듈에 필요한 정의를 하고 컴파일 후 링커를 사용해 합쳐주면 된다.

```
gcc -D__KERNEL__ -DMODULE -Wall -O -c -o start.o start.c
gcc -D__KERNEL__ -DMODULE -Wall -O -c -o stop.o stop.c
ld -m elf_i386 -r -o hello.o start.o stop.o
```

start.c에 init_module()이 들어있고 stop.c에 cleanup_module()이 들어있다면 각각을 컴파일 한 후 ld를 사용해 하나로 묶어준다.

6.3. 모듈 동작의 이해

모듈은 등록될 때 디바이스 번호를 등록하고 이와 함께 file_operations 라는 구조체를 커널에 알려준다. file_operations는 include/linux/fs.h에 정의되어 있고 다음과 같다.

```
/*
 * NOTE:
 * read, write, poll, fsync, readv, writev can be called
 *   without the big kernel lock held in all filesystems.
 */
struct file_operations {
struct module *owner;
loff_t (*llseek) (struct file *, loff_t, int);
ssize_t (*read) (struct file *, char *, size_t, loff_t *);
ssize_t (*write) (struct file *, const char *, size_t, loff_t *);
int (*readdir) (struct file *, void *, filldir_t);
unsigned int (*poll) (struct file *, struct poll_table_struct *);
int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned long);
int (*mmap) (struct file *, struct vm_area_struct *);
int (*open) (struct inode *, struct file *);
int (*flush) (struct file *);
int (*release) (struct inode *, struct file *);
```

```
int (*fsync) (struct file *, struct dentry *, int datasync);
int (*fasync) (int, struct file *, int);
int (*lock) (struct file *, int, struct file_lock *);
ssize_t (*readv) (struct file *, const struct iovec *, un-
signed long, loff_t *);
ssize_t (*writev) (struct file *, const struct iovec *, un-
signed long, loff_t *);
ssize_t (*send-
page) (struct file *, struct page *, int, size_t, loff_t *, int);
unsigned long (*get_unmapped_area)(struct file *, unsigned long, un-
signed long, unsigned long, unsigned long);
};
```

모든 디바이스 드라이버는 사용자가 file_operations를 사용해 등록해준 표준화되어 있는 인터페이스 를 사용해 입/출력 등의 일을 하게된다. 유닉스에서는 디바이스나 네트웍이나 모두 하나의 파일 처럼 동작하도록 되어 있는데 이에 따른 함수들이 등록되어 있다. 예를 들어 디바이스로부터 읽기 동작을 원한다면 file_operations에 등록된 read 함수를 사용해 읽기를 한다.

file_operations는 모두 채울 필요는 없다. 필요하거나 지원되야하는 것을 채워 넣으면 된다. 그러나 범용적인 디바이스를 만든다면 되도록 모든 함수를 다 채워넣어 주는 것이 좋을 것이다.

그러나 file_operations에 존재하는 함수의 개수에 제약이 있으므로 디바이스에 대해 file_operations 외의 다른 기능 혹은 함수를 원하는 경우엔 ioctl을 사용한다. ioctl의 C 라이브러리 내의 정의는 다음과 같이 되어 있다.

```
#include <sys/ioctl.h>

int ioctl(int d, int request, ...)
```

ioctl 함수를 사용할 때 request란 숫자를 전달해 주는데 이 것이 ioctl에 의해 불리는 함수의 인덱스가 된다. 즉 ioctl로 불리는 함수는 switch 문과 같은 것을 이용해 request로 전달된 값을 비교해 해당 함수를 다시 호출해 주게 된다. 다음 소스는 하드 디스크의 시리얼 번호를 읽어 내는 기능을 하는 디바이스 드라이버를 만들어 본 것이다.

```
/* hddinfo.c */
#ifndef __KERNEL__
#define __KERNEL__
#endif
#ifndef MODULE
#define MODULE
#endif

#define __NO_VERSION__
#include <linux/module.h>
#include <linux/version.h>
#include <linux/errno.h>
```

```
#include <linux/fs.h>
#include <linux/ide.h>
#include <asm/uaccess.h>

#define HDA 0x0300
#define HARDDISK HDA

char kernel_version[] = UTS_RELEASE;

static int hddinfo_open(struct inode *node, struct file *f)
{
return 0;
}

static int hddinfo_release(struct inode *node, struct file *f)
{
return 0;
}

static ssize_t hd-
dinfo_read(struct file *f, char *buf, size_t nbytes, loff_t *ppos)
{
return nbytes;
}

static ssize_t read_serial(char *dst)
{
ide_drive_t *drv;

drv = get_info_ptr(HARDDISK);
if (drv)
copy_to_user(dst, drv->id->serial_no, 20);
else
{
;//PDEBUG("HDDINFO : Cannot get the drive information\n");
return 0;
}

return 20;
}

int hddinfo_ioctl(struct inode *node, struct file *f, un-
signed int ioctl_num, unsigned long ioctl_param)
{
switch (ioctl_num)
{
case 0 :
read_serial((char *)(ioctl_param));
break;
```

```
}

return 0;
}

struct file_operations Fops = {
NULL,
NULL,
hddinfo_read,
NULL,
NULL,
NULL,
hddinfo_ioctl,
NULL,
hddinfo_open,
NULL,
hddinfo_release
};

int init_module()
{
if (register_chrdev(212, "hddinfo", &Fops) < 0)
{
//PDEBUG("HDDINFO : Unable to register driver.\n");
return -EIO;
}

return 0;
}

void cleanup_module()
{
if (unregister_chrdev(212, "hddinfo") < 0)
;//PDEBUG("HDDINFO : Unable to unregister\n");
}
```

hddinfo.c에서 정의된 file_operations는 read/ioctl/open/release 만을 사용한다. open과 release는 이 디바이스를 open/close할 때 불리므로 디바이스를 사용하기 전에 초기화 해야하거나 혹은 사용을 중지하기 전에 또 필요한 작업을 해야하는 경우 이 함수들에 필요한 기능을 넣으면 된다. hddinfo.c 에서는 open/close에 따른 작업을 할 필요가 없어 아무것도 넣지 않고 그냥 0을 리턴하는 기능을 넣어 예제로 올렸다. read를 사용해 하드디스크의 시리얼 번호를 읽도록 해도 되지만 여기선 ioctl의 사용을 보기위해 일부러 read에서 할 읽을 ioctl로 뽑아 만들어 봤다.

이 모듈의 사용은 아래와 같은 프로그램으로 동작 시킨다.

```
#include <stdio.h>
#include <unistd.h>
#include <sys/ioctl.h>
```

```
#include <stdlib.h>
#include <fcntl.h>

int main()
{
int fd;
char buf[256];

fd = open("/dev/hdd_info", O_RDWR);
if (fd < 0)
{
printf("Device open error.\n");
return -1;
}

ioctl(fd, 0, buf);

printf("buf : %s\n", buf);

close(fd);

return 0;
}
```

test.c를 동작 시키기 전에 /dev/hdd_info를 만들어 줘야한다. 'mknod c 212 0 /dev/hdd_info'로 만들어 준다. 일반적으로 디바이스를 사용하기 위해 디바이스 파일을 사용해 디바이스를 open 한다. 여기에서 얻어지는 핸들을 사용해 디바이스에 읽고 쓰기 동작 등을 한 후 다 사용했으면 close로 사용을 중지해 준다. 이 절차가 전형 적인 절차에 해당한다.

test.c에서도 open 후 ioctl의 0번 함수를 호출해 hddinfo.c의 read_serial()을 불러 하드디스크의 시리얼 번호를 읽어온다. 하드디스크의 시리얼 번호는 커널 부팅할 때 이미 얻어진 하드디스크에 대한 정보를 갖고 있는 구조체에서 복사한다.

6.4. 알아야할 것 들

디바이스 드라이버를 만들기 위한 아주 기본 적인 것은 이미 알았을 것이다. 이제는 좀더 복잡하지만 알아야만 하는 것들에 대해 얘기해보자.

버전

> 모듈을 만드는 것은 커널의 버전과 밀접한 관계가 있고 커널의 버전이 변경되면서 디바이스 드라이버의 구조 자체도 조금씩 변하므로 여러 버전의 커널에서 동시에 사용될 수 있도록 만들기 위해선 커널의 버전을 구분해 컴파일되고 동작되도록 해줘야한다.

리눅스에선 현재 커널의 버전을 LINUX_VERSION_CODE로 나타낸다. 그리고 KERNEL_VERSION이란 매크로가 있어 이 것을 사용하면 LINUX_VERSION_CODE와 비교할 수 있게된다.

디바이스 번호 동적 할당

디바이스 번호가 이미 정해진 것이 많기 때문에 그 값을 정해 쓸 필요가 없는 경우엔 현재 시스템에서 사용하지 않는 번호를 찾아 사용하면 되기 때문에 모듈을 등록하는 당시에 비어 있는 번호를 동적으로 알아내 그 것을 사용한다. 사용은 init_module()에서 다음 함수를 사용해 주번호를 얻어 온다.

```
#define DEVICE_NAME "char_dev"

static int Major;

...

int init_module()
{
Major = module_register_chrdev(0, DEVICE_NAME, &Fops);

...
}

...
```

module_register_chrdev() 함수의 처음 값인 주번호에 0을 넘겨주면 동적으로 할당해 준다. 리턴되어 오는 값이 음수인 경우는 에러가 있는 것이고 양수인 경우는 그 것을 그대로 사용 하면 된다. cleanup_module()에선 얻어져 저장되어 있는 Major 변수의 값을 사용하면 된다.

Use Count

lsmod 명령으로 현재 시스템에 등록된 모듈에 대해 열거해보면 세번째 항목이 ’Used’인데 이 것은 이 모듈이 다른 모듈에 의해 멀마나 사용되는 가를 나타낸다. 이 값을 위한 매크로가 준비되어 있는데 MOD_INC_USE_COUNT, MOD_DEC_USE_COUNT 이고 각각을 open과 release에 넣어 주면 lsmod를 사용해 값을 알 수 있게 된다.

/proc

리눅스 커널은 /proc 이란 파일 시스템이 존재한다. /proc엔 커널의 내부에 존재하는 정보를 얻을 수 있거나 혹은 커널이나 모듈 프로세스로 정보를 전달하고 읽을 때 사용할 수 있다. 예를 들어 ’cat /proc/interrupt’을 해보면 현재 시스템의 인러텁드에 대한 정보를 알 수 있는데 이 내용은 모두 커널 내부에 들어 있는 것들이다.

/proc을 사용하기 위해선 init_module()에서 특정 정보를 등록해 주고 cleanup_module()에서 해제해 주면된다. 그러나 /proc으로는 Use Count를 알 수 없고 특히 파일은 열리고 모

듈은 제거됐으면 결과는 예측할 수 없게 된다. /proc을 위한 구조체의 정의는 include/linux/proc_fs.h에 정의된 proc_dir_entry다.

```
struct proc_dir_entry {
unsigned short low_ino;
unsigned short namelen;
const char *name;
mode_t mode;
nlink_t nlink;
uid_t uid;
gid_t gid;
unsigned long size;
struct inode_operations * proc_iops;
struct file_operations * proc_fops;
get_info_t *get_info;
struct module *owner;
struct proc_dir_entry *next, *parent, *subdir;
void *data;
read_proc_t *read_proc;
write_proc_t *write_proc;
atomic_t count; /* use count */
int deleted; /* delete flag */
kdev_t rdev;
};
```

실제 사용하는 것은 다음과 같다. Ori Pomerantz가 지은 ' 리눅스 커널 모듈 프로그래밍 안내서' 에서 발췌 했다.

```
int procfile_read(char *buffer, char **buffer_location, off_t off-
set, int buffer_length, int zero)
{
int len;

static char my_buffer[80];
static int count = 1;

if (offset > 0)  return 0;

len = sprintf(my_buffer, "For the %d%s time, go away!\n",
                 count,
                 (count % 100 > 10 && count % 100 < 14) ? "th" :
                 (count % 10 == 1) ? "st" :
                 (count % 10 == 2) ? "nd" :
                 (count % 10 == 3) ? "rd" : "th" );

count++;

*buffer_location = my_buffer;
```

```
return len;
}

struct proc_dir_entry Our_Proc_File =
{
0,
4,
"test",
S_IFREG | S_IRUGO,
1,
0, 0,
80,
NULL,
procfile_read,
NULL
};

int init_module()
{
return proc_register(&proc_root, &Our_Proc_File);
}

void cleanup_module()
{
proc_unregister(&proc_root, Our_Proc_File.low_ino);
}
```

변수

리눅스에서 Ethernet Card나 Sound Card 설정을 해본 사람은 이 카드들에게 특정 경우 IO 어드레스나 IRQ 등을 지정하기 위해 insmod 등을 사용하면서 같이 파라미터를 넘겨준 경험이 있을 것이다.

커널 모듈은 argc, argv를 받을 수 없기 때문에 대신 전역 변수를 사용해 값을 넘겨 줄 수 있도록 되어 있다. 변수는 전역으로 설정하고 특정 매크로를 사용해 준다.

```
char *str1, *str2;

#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
MODULE_PARM(str1, "s");
MODULE_PARM(str2, "s");
#endif
```

위에서 str1과 str2가 전역 변수로 선언되고 모듈 파라미터로 정의됐다. 나중에 insmod를 실행할 때 'insmod str1=abc str2=def'와 같이 하면되다.

실수

모듈은 커널과 같은 레벨에서 실행되므로 표준 라이브러리는 사용할 수 없다. 사용할 수 있는 것은 커널 함수이고 /proc/ksyms에서 확인할 수 있다.

인터럽트를 사용하는 경우 처리하는 동안 다른 인터럽트가 걸리지 않도록 하기 위해 인터럽트를 막아 놓았다면 처리가 다 끝나고 나서는 반드시 인터럽트를 가능하도록 해줘야한다. 그러지 않으면 시스템은 먹통이 될 것이다.

부록 A. SEGA DreamCast Linux

세가사의 드림캐스트 게임기엔 히타치의 SH4란 프로세서가 사용된다. 물론 SH4 프로세서가 그대로 사용되는 것은 아니고 게임기에 필요한 여러 기능이 합쳐진 형태의 프로세서를 사용한다. 필자도 새로운 임베디드 시스템에 사용할 프로세서를 수배하다가 이 것을 써볼 요량으로 드림캐스트를 구입하게 됐다. 중고로 구입 했지만 사실 리눅스 보다는 게임을 더 많이 했다.

SH 프로세서는 종류가 여러 가지인데 SH 뒤의 숫자가 버전 정도의 의미를 갖는다. ARM 프로세서는 ARM7, ARM9 등과 같이 말하는데 이와 비슷한 개념이다. SH는 SuperH의 약자로 현재 SH1 부터 SH4 까지 나와 있는 상태다. ARM과 비슷한 리스크 프로세서로 카시오의 초기 PDA 등에 사용된 적이 있다.

A.1. LinuxSH

SuperH 프로세서에 포팅하는 리눅스에 대한 정보는 SourceForge에 있는 *http://linuxsh.sourceforge.net/* (http://linuxsh.sourceforge.net/)에서 정보를 얻을 수 있다. 개발자는 주로 일본 사람들이며(프로세서가 일본제니 어쩔수 없나?) 꽤 많이 진척되어 있고 최근의 커널 버전을 지원한다. CVS를 통해 최근의 커널 소스 코드를 얻을 수 있다.

필자가 가장 많이 찾는 사이트는 *http://www.m17n.org/linux-sh/* (http://www.m17n.org/linux-sh/)로 드림캐스트에 대한 정보를 많이 얻을 수 있고 RedHat에서 만든 RedBoot 등을 얻을 수 있다. 또 Debian GNU/Linux on SuperH에 대한 정보도 얻을 수 있다. 드림캐스트에서 돌아가는 리눅스의 모습을 보려면 *http://www.m17n.org/linux-sh/dreamcast/* (http://www.m17n.org/linux-sh/dreamcast/)를 보기 바란다.

이런 것에 앞서 드림캐스트에서 프로그래밍을 하려는 사람은 *http://mc.pp.se/dc/* (http://mc.pp.se/dc/)를 참조 바란다. 드림캐스드에서 사용되는 cd-rom을 만들기 위한 준비나 만드는 방법 등이 설명되어 있다.

필자의 드림캐스트에서 돌고 있는 리눅스의 모습은 *http://ruby.medison.co.kr/~halite* (http://ruby.medison.co.kr/~halite)에서 확인할 수 있다. 드림캐스트가 PC와는 달리 저장 공간도 없고 메모리도 작기 때문에 뭔가를 하려면 항상 cd-rom을 읽기 때문에 무척 느리긴 하다. 그러나 SuperH를 사용하는 사람이고 또 리눅스를 포팅하려는 사람이라면 충분히 좋은 테스트 기계가 될 것이다.

A.2. 드림캐스트에서 리눅스 실행해 보기

이미 만들어진 이미지를 사용해 드림캐스트에서 실행되는 리눅스를 맛 볼수 있다. *ftp://ftp.m17n.org/pub/super-h/dreamcast/* (ftp://ftp.m17n.org/pub/super-h/dreamcast/dreamcast-linux-010605.tar.bz2)에서 미리 만들어진 이미지를 받아 사용하면 된다. 만드는 방법은 *http://www.m17n.org/linux-sh/dreamcast/distribution* (http://www.m17n.org/linux-sh/dreamcast/distribution)을 참조하고 미리 IP.BIN 이란 것을 만들어

둬야한다. IP.BIN은 *http://mc.pp.se/dc/files/makeip.tar.gz* (http://mc.pp.se/dc/files/makeip.tar.gz)을 사용해 만들기 바란다.

만들 때 CD-R을 사용하는데 cdrecord란 프로그램에서 제대로 지원하지 못하는 경우도 있기 때문에 cdrecord 사이트에서 지원하는 것인지 확인 한 후 진행하기 바란다. 필자는 LG CED-8080B를 가지고 있고 지원하는 드라이브라고 나와 있지만 제대로 구어지지 않아 여러 장의 CD-R을 버리고 나서 SONY VAIO GR9/K에 있는 CD-RW/DVD 콤보로 굽는데 성공 했다.

부록 B. 리눅스에 시스템 콜 만들어 넣기

커널을 임베디스 시스템에 포팅해 넣다보면 가끔 나만의 시스템 콜을 사용할 때가 있는데 직접 시스템 콜을 하나 만들어 넣고 이를 통해 커널의 소스 구조를 조금 알아보자.

B.1. 시스템 콜의 흐름

리눅스 내에서 시스템 콜이 발생하면 진행되는 흐름은 다음과 같다.

1. 사용자 프로세스
2. libc.a
   - 아규먼트 스택에 넣음
   - 시스템 콜 번호 저장
   - 트랩(trap) 발생
3. system_call()
   - IDT에 의해 트랩을 시작
   - 진짜 핸들러 실행
   - sys_call_table 사용
4. 진짜 시스템 콜 핸들러

B.2. IDT(Interrupt Descriptor Table)

시스템 콜을 호출하면 최종적으로 트랩을 발생시키는데 이 것은 소프트웨어 인터럽트라고 이해하면된다. i386에선 IDT를 통해 모든 인터럽트가 관리되는데 시스템 콜은 0x80 번의 인터럽트를 사용한다. 아래에 IDT의 구조가 나와 있다.

그림 **B-1. IDT** 구조

B.3. 시스템 콜 테이블

0x80 트랩 핸들러는 모든 시스템 콜에의해 불려지고 이 핸들러는 불려질 당시의 시스템 콜 번호를 가지고 해당 시스템 콜을 시스템 콜 테이블에서 찾아 실행해 준다.

모든 시스템 콜의 번호는 $(TOPDIR)/include/asm/unistd.h에 정의되어 있다.

- #define __NR_exit 1
- #define __NR_fork 2
- #define __NR_read 3
- ...
- #define __NR_vfork 190

시스템 콜 테이블은 $(TOPDIR)/arch/i386/kernel/entry.S에 정의되어 있고 각 시스템 콜의 주소가 연속되게 적혀져 있다.

- .long SYMBOL_NAME(sys_ni_syscall)
- .long SYMBOL_NAME(sys_exit)
- .long SYMBOL_NAME(sys_fork)
- .long SYMBOL_NAME(sys_read)
- ...
- .long SYMBOL_NAME(sys_vfork)
- .rept NR_syscalls-190

그림 B-2은 fork()가 실행될 때의 흐름을 나타낸 그림으로 이해에 도움을 줄 수 있을 것이다.

그림 **B-2. fork()**가 실행될 때의 흐름

B.4. 시스템 콜 추가

시스템 콜을 추가해보자. 우선 새로운 시스템 콜이 들어있는 파일을 만들자($TOPDIR)/kernel/mysyscall.c

```
/* $(TOPDIR)/kernel/mysyscall.c */
#include <linux/linkage.h>

asmlinkage int sys_mysyscall()
{
printk("My First System Call.\n");
}
```

$(TOPDIR)/include/asm/unistd.h에 새로운 시스템 콜을 위한 번호를 추가한다.

```
...
#define __NR_vfork 190
#define __NR_mysyscall 191
```

$(TOPDIR)/i386/kernel/entry.S에 있는 시스템 콜 테이블에 등록한다.

```
ENTRY(sys_call_table)
...
.long SYSBOL_NAME(sys_mysyscall)
.rept NR_syscalls-191
```

커널을 컴파일하는데 위에서 만든 mysyscall.c를 Makefile에 등록해 준다. 간단히 ’O_OBJS=’이란 줄에 mysyscall.o라고 추가해주면 된다.

커널 컴파일이 끝나면 새로운 커널을 설치하고 재부팅한 다음 아래와 같은 테스트 프로그램을 만들어 실행해 보자.

```
/* test.c */
#include <linux/unistd.h>

_syscall0(int, mysyscall);

int main()
{
int i;

i = mysyscall();

return i;
}
```

테스트 프로그램을 실행했을 때 화면에 'My First System Call'이라고 출력되면 다행인데 아무런 출력도 없다면 dmesg를 사용해 커널 출력을 확인해 보자. 제일 끝에 문장이 제대로 찍혔는가?

부록 C. Inline Assembly

인라인 어셈블리는 어셈블리로 짜는 소스 코드가 아닌 다른 언어 내에 들어가는 어셈블리 형태의 코드를 말한다. 예를 들어 많이 사용하는 C 내에서 C로는 할 수 없는 일이나 속도가 아주 빠른 작업을 원할 때 직접 어셈블리 코드를 입력해 사용할 수 있다.

시스템 프로그래밍이나 성능을 높여야 하는 경우에 보통은 어셈블리를 사용한다. 즉 프로그램의 대부분을 C나 C++등으로 만들고 고급언어로는 할 수 없거나, 그 부분에 어셈블리를 써서 속도 향상이 가능할 때 그 부분만을 어셈블리로 만들어 C나 C++로 만든 나머지 부분과 같이 쓰게 된다.

C로 된 부분과 어셈블리로 된 부분을 같이 동작시키는 데는 크게 두 가지 방법이 있다. 하나는 어셈블리로 쓴 부분을 독립된 함수로 만들어 따로 어셈블 한 후에 오브젝트 화일을 링크시키는 방법이 있고 나머지 하나는 인라인 어셈블리를 쓰는 방법이 있다.

어셈블리 파일을 따로 만드는 경우엔 어셈블리 함수가 C의 함수 호출 방식을 따르도록 해주면 C와 사용하기 쉬우므로 크기가 큰 경우라면 이 방법을 쓰는 것이 좋다.

그러나 일부분에서만 어셈블리가 필요하거나 특히 고급 언어가 사용하지 못하는 프로세서의 특정한 기능을 쓰기위해 어셈블리를 쓸 때는 많아도 이삼십개 정도의 명령 만을 어셈블리로 만들면 되는 경우가 대부분이고 이를 위해서 따로 함수를 만들어 링크하는 것은 번거로운데다가 자주 호출되는 경우라면 성능에도 영향을 미친다. 이런 경우에 인라인 어셈블리를 쓰게된다.

C.1. 인라인 어셈블리 기초

C.1.1. 알아야할 것 들

인라인 어셈블리를 사용할 땐 다음과 같은 것을 명시해 줘야한다. 물론 빼고 사용할 수도 있다.

- 어셈블리 코드
- output 변수
- input 변수
- 값이 바뀌는 레지스터

그리고 사용되는 문법의 형태는 다음과 같다.

```
__asm__ __volatile__ (asms : output : input : clobber);
```

__asm__

다음에 나오는 것이 인라인 어셈블리 임을 나타낸다. ANSI엔 __asm__ 으로만 정의되어 있으므로 asm 과 같은 키워드는 사용하지 않는 것이 바람직하다.

__volatile__

이 키워드를 사용하면 컴파일러는 프로그래머가 입력한 그래도 남겨두게 된다. 즉 최적화 나 위치를 옮기는 등의 일을 하지 않는다. 예를 들어 output 변수중 하나가 인라인 어셈블리엔 명시되어 있지만 다른 곳에서 사용되지 않는다고 판단되면 컴파일러는 이 변수를 알아서 잘 없애주기도 한다. 이런 경우 이런 것을 고려해 프로그램을 짰다면 상관 없겠지만 만에 하나 컴파일러가 자동으로 해준 일 때문에 버그가 발생할 수도 있다. 그러므로 __volatile__ 키워드를 사용해 주는 것이 좋다.

asms

따옴표로 둘러싸인 어셈블리 코드. 코드 내에서는 %x과 같은 형태로 input, output 파라미터를 사용할 수 있으며 컴파일 하면 파라미터가 치환된 대로 어셈블리 코드로 나타난다.

output

변수들을 적어 주고 각각은 쉼표고 구분된다. 결과 값을 출력하는 변수를 적는다.

input

output과 같은 방식으로 사용하고 인라인 어셈블리 코드에 넘겨주는 파라미터를 적는다.

clobber

output, input에 명시되어 있진 않지만 asms를 실행해서 값이 변하는 것을 적어 준다. 각 변수는 쉼표로 구분되고 각각을 따옴표로 감싸준다.

asms는 반드시 있어야하지만 output, input, clobber는 각각 없을 수도 있다. 만약 clobber가 없는 경우 라면 clobber와 바로 앞의 콜론(:)을 같이 쓰지 않아도 된다. 마찬가지로 input, clobber가 없다면 output까지만 쓰면 된다.

그러나 output, clobber는 있고 input이 없는 경우엔 다음과 같이 input 만을 제외한 나머지는 반드시 써줘야한다.

```
__asm__ __volatile__ (asms : output : : clobber);
```

중간에 있는 것이 없는 경우엔 해당 항목만을 없애고 콜론은 그대로 내버려둬야 다음 필드가 어떤 것을 의미하는지 나타내게 된다.

인라인 어셈블리가 사용된 예를 들어보자. include/asm-i386/bitops.h에 정의되어 있는 함수다.

```
/**
 * test_and_set_bit - Set a bit and return its old value
 * @nr: Bit to set
 * @addr: Address to count from
 *
```

```
 * This operation is atomic and cannot be reordered.
 * It also implies a memory barrier.
 */
static __inline__ int test_and_set_bit(int nr, volatile void * addr)
{
int oldbit;

__asm__ __volatile__( LOCK_PREFIX
"btsl %2,%1\n\tsbbl %0,%0"
:"=r" (oldbit),"=m" (ADDR)
:"Ir" (nr) : "memory");
return oldbit;
}
```

C.1.2. 어셈블리

인라인 어셈블리 중 asms에 해당하는 실제 코드를 적는 부분은 AT&T 어셈블리 문법을 따르고 여기에 적인 그대로가 컴파일 후 gasm에 넘겨지기 때문에 gasm의 문법을 따라야한다.

명령의 구분은 세미콜론(;)이나 개행문자(\n)으로 한다.

그리고 gasm의 문법에서 주의할 것은 레지스터를 %ax과 같은 식으로 쓴다는 것과 인텔 어셈블리와는 달리 destination이 뒤에 나온다는 것이다. 그러므로 인텔 문법에 익숙한 사람은 사고의 전환이 필요할 것이다.

인라인 어셈블리에선 %0, %1등을 사용해 input, output 오퍼랜드를 나타낸다. output에서 부터 시작해 input에 나열된 변수들의 순서 대로 %0, %1, ... 으로 번호가 매겨진다.

모든 코드는 따옴표 안에 있어야하기 때문에 많은 수의 명령을 한줄로 쓰면 보기도 않좋기 때문에 명령 수가 많아지면 각 명령을 따옴표로 감싸고 뒤에 \t\n을 넣고 다음 줄에 다시 명령을 따옴표로 적으면 된다. 아래의 예를 보면 이해가 쉬울 것이다.

```
static __inline__ int find_first_zero_bit(void * addr, unsigned size)
{
int d0, d1, d2;
int res;

if (!size)
return 0;
/* This looks at mem-
ory. Mark it volatile to tell gcc not to move it around */
__asm__ __volatile__(
"movl $-1,%%eax\n\t"
"xorl %%edx,%%edx\n\t"
"repe; scasl\n\t"
"je 1f\n\t"
"xorl -4(%%edi),%%eax\n\t"
"subl $4,%%edi\n\t"
```

```
"bsfl %%eax,%%edx\n"
"1:\tsubl %%ebx,%%edi\n\t"
"shll $3,%%edi\n\t"
"addl %%edi,%%edx"
:"=d" (res), "=&c" (d0), "=&D" (d1), "=&a" (d2)
:"1" ((size + 31) >> 5), "2" (addr), "b" (addr));
return res;
}
```

바로 위의 예에서 %eax가 아니라 %%eax라고 씌어진 것이 있는데 %%는 gasm에 넘겨질 때 %로 해석되 넘겨진다. 즉 output, input에 레지스터를 직접 지정할 때 이렇게 쓴다. 그러나 output, input에 아무 것도 지정되어 있지 않다면 %%는 %로 바뀌지 않는다. 그러므로 %eax와 같이 써야만 한다.

C.1.3. Output/Input

이전의 예들에서 보면 output, input에 지정된 것이 무척 어렵게 되어 있는데 output, input은 constraints와 변수 이름이 쉼표로 구분된 리스트로 이루어져 있다.

constraints는 의미를 나타내는 문자와 몇가지 modifier를 조합해 만들어진다. 자세한 내용은 'info gcc' 를 해서 ::Constraints 항목에서 찾길 바란다. 아래 열거된 것은 몇 가지만을 간추린 것이다.

C.1.3.1. Constraints

'm'

아키텍쳐가 지원하는 모든 종류의 메모리 어드레스를 사용하는 오퍼랜드

'o'

옵셋화 가능한 어드레스를 사용하는 메모리 오퍼랜드

'V'

옵셋화 불가능한 어드레스를 사용하는 메모리 오퍼랜드

'<'

자동 감소(미리 감소하거나 나중에 감소한다) 어드레스용 메모리 오퍼랜드

'>'

자동 증가(미리 증가하거나 나중에 증가한다) 어드레스용 메모리 오퍼랜드

'r'

일반 레지스터 사용 오퍼랜드

’d’, ’a’, ’f’, ...

시스템에 따른 레지스터를 나타내는 다른 오퍼랜드로 d, a, f는 각각 68000/68020에서 데이터, 어드레스, 플로팅포인트 레지스터를 나타낸다.

’i’

immediate 정수 값을 나타내는 오퍼랜드. 심볼로된 상수도 여기에 해당한다.

’n’

immediate 정수 값으로 알려진 정수 값을 나타낸다. 많은 시스템이 어셈블할 때 한 워드 이하의 오퍼랜드용 상수를 지원하지 않으므로 ’i’보단 ’n’을 사용하는 것이 바람직하다.

’I’, ’J’, ’K’, ... ’P’

시스템에 따라 특정 범위 내의 값을 나타내는 오퍼랜드. 68000에선 ’I’가 1에서 8까지의 값을 나타낸다. 이것은 시프트 명령에서 허용된 시프트 카운트의 범위다.

’E’

immediate 플로팅 오퍼랜드로 호스트와 같은 타겟 플로팅 포인트 포맷인 경우에만 사용가능.

’F’

immediate 플로팅 오퍼랜드.

’G’, ’H’

특정 범위 내의 값을 나타내는 플로팅 오퍼랜드로 시스템에 따라 다르다.

’s’

값이 명확히 정해지지 않은 immediate 정수를 나타내는 오퍼랜드

’s’

값이 명확히 정해지지 않은 immediate 정수를 나타내는 오퍼랜드. ’s’를 ’i’? 대신 쓰는 이유는 좀더 좋은 코드를 만들어낼 수도 있기 때문이다.

’g’

특수 레지스터를 제외한 일반 레지스터, 메모리 혹은 immediate 정수 중 아무것이나 나타내는 오퍼랜드.

’0’, ’1’, ’2’, ... ’9’

같이 사용된 오퍼랜드의 번호를 나타냄.

’p’

올바른 메모리 어드레스를 나타내는 오퍼랜드. "load address"와 "push address" 명령을 위한 것.

’Q’, ’R’, ’S’, ... ’U’

Q에서 U까지의 문자는 시스템에 따라 변하는 여러 다른 오퍼랜드를 의미한다.

C.1.3.2. Modifier

’=’

오퍼랜드가 쓰기 전용임을 나타냄. 이전 값은 없어지고 새로운 값으로 교체됨.

’+’

읽기, 쓰기 모두 가능. ’=’는 보통 output용 ’+’는 input/output 모두에 사용 가능하다. 나머지 다른 모든 오퍼랜드는 input 전용으로 간주된다.

’&’

"earlyclobber" 오퍼랜드를 나타내고 input 오퍼랜드를 사용하는 명령이 끝나기 전에 변경된다는 것을 의미한다. 그래서 input 오퍼랜드나 메모리 어드레스의 일부을 나타내는 레지스터엔 못 쓴다.

gcc는 input 변수가 다 사용되고 나면 output에 사용된다고 가정하기 때문에 input에 사용된 변수가 output과 같게 되고 또 output이 input 보다 먼저 사용되는 경우가 발생할 수 있다. 이런 경우를 막기 위해 output에 사용된 변수가 input이 모두 사용되기 전에 변경될 수도 있다고 알려줘야만 input과 output이 같아져 생기는 에러를 막을 수 있다.

’%’

%뒤에 따라오는 오퍼랜드로 대체 가능함을 나타낸다. 직접 레지스터를 명시하고 사용할 때 %%eax 등과 같이 하는 것을 기억하는가?

’#’

이후의 쉼표가 나올 때 까지의 모든 문자를 constraints로 취급하지 않는다.

C.1.3.3. ARM Family Constraints

’f’

플로팅 포인트 레지스터

’F’

0.0, 0.5, 1.0, 2.0, 3.0, 4.0, 5.0, 10.0 중의 하나를 나타내는 플로팅 포인트 상수

’G’

음수 값인 경우의 ’F’

’I’

데이터 프로세싱 명령에서 유효한 immediate 정수 값 오퍼랜드. 0에서 255사이의 2의 배수 값을 나타낸다.

’J’

-4095에서 4095 사이의 정수

’K’

’I’를 만족하는 값을 1의 보수 취한 것

’L’

’I’를 만족하는 값을 음수로 취한 값(2의 보수)

’M’

0에서 32 사이의 정수 값

’Q’

한 레지스터에 담겨있는 정확한 어드레스를 나타내는 메모리

’R’

constalt pool 내의 아이템

’S’

현재 파일의 텍스트 세그먼트 내의 심볼

C.1.3.4. i386 Family Constraints

’q’

’a’, ’b’, ’c’, ’d’ 레지스터

’A’

’a’ 또는 ’d’ 레지스터 (64비트 정수 용)

’f’

플로팅 포인트 레지스터

’t’

첫번째(스택의 최상위) 플로팅 포인트 레지스터

’u’

두번째 플로팅 포인트 레지스터

’a’

’a’ 레지스터

’b’

’b’ 레지스터

’c’

’c’ 레지스터

’d’

’d’ 레지스터

’D’

’di’ 레지스터

’S’

’si’ 레지스터

’I’

0에서 31 사이의 상수(32비트 시프트용)

’J’

0에서 63 사이의 상수(64비트 시프트용)

’K’

’0xff’

’L’

’0xffff’

’M’

0, 1, 2, 3 (lea 명령을 위한 시프트)

’N’

0에서 255 사이의 값(out 명령 용)

’G’

80387 플로팅 포인트 상수를 나타냄

C.2. 사례 분석

리눅스 커널에 이미 사용된 수 많은 예를 통해 어떤 식으로 인라인 어셈블리가 사용됐는지 알아보자.

C.2.1. strcpy()

아래 소스 코드는 include/asm-i386/string.h에 있는 strcpy() 함수를 가져와 컴파일 해보기 위해 조금 추가한 코드다.

```
/* test.c */
static inline char * strcpy(char * dest,const char *src)
{
int d0, d1, d2;
__asm__ __volatile__(
"1:\tlodsb\n\t"
"stosb\n\t"
"testb %%al,%%al\n\t"
"jne 1b"
: "=&S" (d0), "=&D" (d1), "=&a" (d2)
:"0" (src),"1" (dest) : "memory");
return dest;
}

int main()
{
char a[] = "1234";
char b[] = "4567";

strcpy(a, b);

return 0;
}
```

컴파일은 'gcc -S -c test.c'라고 한다. 그러면 test.s가 생길 것이다. test.s는 다음과 같다.

```
.file "test.c"
.version "01.01"
gcc2_compiled.:
.section .rodata
.LC0:
.string "1234"
.LC1:
.string "5678"
.text
.align 4
.globl main
```

```
.type  main,@function
main:
pushl %ebp
movl %esp,%ebp
subl $24,%esp
leal -8(%ebp),%eax
movl .LC0,%edx
movl %edx,-8(%ebp)
movb .LC0+4,%al
movb %al,-4(%ebp)
leal -16(%ebp),%eax
movl .LC1,%edx
movl %edx,-16(%ebp)
movb .LC1+4,%al
movb %al,-12(%ebp)
addl $-8,%esp
leal -16(%ebp),%eax
pushl %eax
leal -8(%ebp),%eax
pushl %eax
call strcpy
addl $16,%esp
xorl %eax,%eax
jmp .L3
.p2align 4„7
.L3:
movl %ebp,%esp
popl %ebp
ret
.Lfe1:
.size  main,.Lfe1-main
.align 4
.type  strcpy,@function
strcpy:
pushl %ebp
movl %esp,%ebp
subl $28,%esp
pushl %edi
pushl %esi
pushl %ebx
movl 12(%ebp),%esi
movl 8(%ebp),%edi
#APP
1: lodsb
stosb
testb %al,%al
jne 1b
#NO_APP
movl %esi,%ecx
```

```
movl %edi,%edx
movl %ecx,%ebx
movl %ebx,-4(%ebp)
movl %edx,%edx
movl %edx,-8(%ebp)
movl %eax,%eax
movl %eax,-12(%ebp)
movl 8(%ebp),%eax
jmp .L2
.L2:
leal -40(%ebp),%esp
popl %ebx
popl %esi
popl %edi
movl %ebp,%esp
popl %ebp
ret
.Lfe2:
.size   strcpy,.Lfe2-strcpy
.ident "GCC: (GNU) 2.95.3 20010315 (release)"
```

인라인 어셈블리는 #APP와 #NO_APP사이에 존재한다.

: "=&S" (d0), "=&D" (d1), "=&a" (d2)

> output의 구성을 나타낸다. "=&S" (d0)는 d0를 'si' 레지스터에 저장하는 것이고 "=&D" (d1)은 d1을 'di' 레지스터에 저장하란 것이고 "=&a" (d2)는 d2를 'a' 레지스터에 저장하란 것이다.
>
> test.s에 의하면 어셈블리 코드가 실행된 후 output으로 d0, d1, d2가 있는데 #NO_APP 바로 밑의 3줄이 이 역할을 한다. d2는 %ebx에 할당됐음을 알 수 있다.

:"0" (src),"1" (dest)

> input의 구성을 나타낸다. "0" (src)는 src가 0번째 오퍼랜드와 같은 위치를 점유하란 말로 %0인 d0를 의미한다. 또 d0가 si를 사용하므로 결국 si의 초기 값이 src가된다. dest는 %1인 di에 입력된다.
>
> test.s에 의하면 #APP 바로 전의 두줄이 input에 해당하고 %esi와 %edi에 src, dest를 입력해 준다.

: "memory"

> clobber에 지정된 "memory"는 컴파일러에게 어셈블리코드가 메모리의 어딘가를 변경한다고 가르쳐 주는 것이다. 이 것을 사용하지 않으면 어셈블리코드에서 메모리의 내용을 변경하는 것을 컴파일러는 전혀 알 수 없다. 잘 못하면 어셈블리에서 고친 값과 다른 값을 컴파일러 는 사용하고 있을 가능성도 있다. "memory"를 명시해 주면 컴파일러는 어셈블리 코드를 실 행하기 전/후에 레지스터에 저장되어 있는 모든 변수의 값을 갱신하도록 한다.

"1:\tlodsb\n\t"

1:은 label을 의미한다. loadsb 명령으로 al 레지스터에 es:esi의 내용을 읽어 온다. 여기서 src의 내용을 읽어 온다. 명령 실행후 esi는 자동으로 1이 증가한다(바이트 단위로 읽기 때문).

"stosb\n\t"

al의 값을 es:edi에 저장한다. edi도 명령 실행 후 1 증가한다.

"testb %%al,%%al\n\t"

al의 내용이 0인지 테스트한다. 스트링을 복사할 땐 NULL 캐릭터가 나올 때 까지 복사하기 때문에 0인지 판별한다.

"jne 1b"

0이 아닌 경우, 즉 NULL 캐릭터가 아닌 경우 계속해서 복사한다.

C.2.2. _set_gate()

arch/i386/kernel/trap.c에 있는 _set_gate()의 내용을 가져다 컴파일 하기 위해 약간 변경한 것이다.

```
/* sg.c */

#define __KERNEL_CS 0x10

#define _set_gate(gate_addr,type,dpl,addr) \
do { \
  int __d0, __d1; \
  __asm__ __volatile__ ("movw %%dx,%%ax\n\t" \
"movw %4,%%dx\n\t" \
"movl %%eax,%0\n\t" \
"movl %%edx,%1" \
:"=m" (*((long *) (gate_addr))), \
 "=m" (*(1+(long *) (gate_addr))), "=&a" (__d0), "=&d" (__d1) \
:"i" ((short) (0x8000+(dpl<<13)+(type<<8))), \
 "3" ((char *) (addr)),"2" (__KERNEL_CS << 16)); \
} while (0)

int main()
{
_set_gate(0, 1, 2, 3);
return 0;
}
```

'gcc -S -c sg.c'로 컴파일한 것은 다음과 같다.

```
.file "sg.c"
.version "01.01"
gcc2_compiled.:
.text
.align 4
.globl main
.type  main,@function
main:
pushl %ebp
movl %esp,%ebp
subl $24,%esp
nop
.p2align 4„7
.L3:
movl $3,%edx
movl $1048576,%ecx
movl %ecx,%eax
#APP
movw %dx,%ax
movw $-16128,%dx
movl %eax,0
movl %edx,4
#NO_APP
movl %eax,%ecx
movl %ecx,-4(%ebp)
movl %edx,%eax
movl %eax,-8(%ebp)
.L5:
jmp .L4
.p2align 4„7
.L6:
jmp .L3
.p2align 4„7
.L4:
xorl %eax,%eax
jmp .L2
.p2align 4„7
.L2:
movl %ebp,%esp
popl %ebp
ret
.Lfe1:
.size  main,.Lfe1-main
.ident "GCC: (GNU) 2.95.3 20010315 (release)"
```

:"i" ((short) (0x8000+(dpl<<13)+(type<<8))), "3" ((char *) (addr)),"2" (__KERNEL_CS << 16));

input으로 정의된 것 들이다. "3", "2"는 각각 %3(__d0), %2(__d1)로 대응되도록 한다. $APP 전의 3줄 중 윗 2줄이 "3", "2"에 해당하는 것들이다.

:"=m" (*((long *) (gate_addr))), "=m" (*(1+(long *) (gate_addr))), "=&a" (__d0), "=&d" (__d1)

output으로 정의된 것 들. %0은 값이 0이되고(main에서 _set_gate(0, 1, 2, 3)으로 했기 때문에) %1은 4가 된다.