4

2024년 9월 20일 금요일 오전 8:54

커널 구성과 빌드

커널은 운영체제 중 항상 메모리에 올라가 있는 운영체제의 핵심 부분으로써 하드웨어와 응용프로그램 사이에서 **인터페이스를 제공하는 역할**을 하며, **컴퓨터 자원들을 관리하는 역할**을 하며 최종 소프트웨어 빌드의 거의 모든 측면에 영향을 미침

커널은 무엇을 하는가?

리눅스는 1991년 리누스 토발즈가 인텔 386과 486 기반 개인용 컴퓨터를 위한 운영체제를 작성하던 무렵에 시작 앤드류 타넨바움이 작성한 미닉스 운영체제에서 영감

미닉스와 다른 리눅스의 특징?
-> 32비트 가상 메모리 커널이고 코드가 오픈 소스

리눅스 커널은 GNU 사용자 공간과 결합돼 데스크톱과 서버에서 실행되는 완전한 리눅스 배포판이 될 수 있음
BSD(Berkeley Software Distribution)운영체제는 커널, 툴체인, 사용자 공간이 하나의 코드 베이스로 결합되어 있음

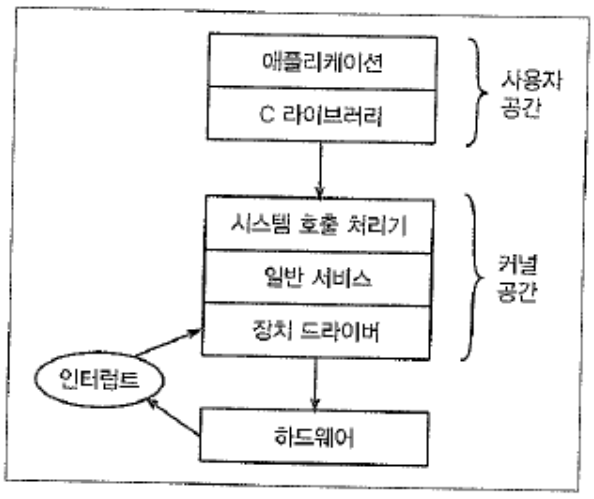


그림 4.1 사용자 공간, 커널 공간, 하드웨어

애플리케이션

- 낮은 CPU 특권 수준에서 실행되며, 라이브러리 호출 외에는 할 수 있는 일이 매우 적음

C 라이브러리

- POSIX에 정의된 것 같은 사용자 수준 함수들을 커널 시스템 호출로 변환
- 사용자 공간과 커널 공간 사이의 주된 인터페이스

시스템 호출 인터페이스

- 트랩이나 소프트웨어 인터럽트 같은 아키텍처 고유의 방법을 사용해 CPU를 낮은 특권 사용자 모드에서 높은 특권 커널 모드로 전환함으로써 모든 메모리 주소와 CPU 레지스터에 접근할 수 있도록 함

시스템 호출 처리기

- 호출을 적절한 커널 서브시스템으로 전달
- 하부 하드웨어로부터의 입력을 요구 시 장치 드라이버로 전달
- 경우에 따라 하드웨어 자체가 인터럽트를 발생 시켜 커널 함수를 호출

커널 선택하기

<커널 개발 주기>

1. 현재 커널 개발 전체 주기는 2주간의 머지 기간과 함께 시작 (리누스는 새로운 기능을 위한 패치를 받아들임)
2. 머지 기간이 끝나면 안정화 단계 시작 (리누스는 -rc1,-rc2 등으로 끝나는 릴리스 후보들을 만듦)
3. 후보들을 테스트하고 버그 보고서와 수정 사항 제출
4. 주요 버그들 수정 후 커널 릴리즈

<안정적 장기 지원 릴리스>

- 안정 릴리스
- 장기 릴리스

주류 커널은 릴리스 되고 나면 안정 트리로 이동
버그 수정은 안정 커널에 적용
주류 커널은 다음 개발 주기 시작

안정 커널의 버그 수정 릴리스는 세번째 번호로 표시 (버전 3 이전에는 4자리 릴리스 번호가 사용)

보통 안정 커널은 8~12주 뒤 다음 주류 릴리스가 나올 때 까지만 갱신
좀 더 오랜 기간동안 갱신되길 원하면서 버그가 발견되고 수정되는 것을 보장받고 싶은 사용자들을 위해 장기 커널 존재 (최소 2년이상 유지보수, long term으로 표시)

<라이선스>

리눅스 소스 코드는 GPL v2로 라이선스
(라이선스에 명시된 방법 중 하나에 따라 커널 소스를 공개해야 함)

커널의 실제 라이선스 문구는 COPYING 파일에 존재

커널 모듈은 단순히 실행 시에 커널과 동적으로 링크되는 코드 조각으로, 커널의 기능을 확장
GPL은 정적 링크과 동적 링크를 구별하지 않으므로 커널 모듈 소스도 GPL에 포함 되는 것처럼 보이지만 라이선스 적용되지 않음

커널 빌드하기

<소스 구하기>

주요 디렉터리

arch	아키텍처별 서브디렉터리
Documentation	커널 문서
drivers	드라이버 종류별 서브디렉터리
fs	파일시스템 코드
include	커널 헤더 파일들
init	커널 시작 코드
kernel	스케줄링, 잠금, 타이머, 전원관리, 디버그/추적코드 등 핵심 기능들
mm	메모리 관리
net	네트워크 프로토콜
scripts	여러가지 유용한 스크립트
tools	리눅스 성능 카운터 도구 등 여러가지 유용한 도구들

<커널 구성 이해하기: Kconfig>

구성 메커니즘을 Kconfig 라고 하고, Kconfig에 통합된 빌드 시스템을 Kbuild라고 함 (Documentation/Kbuild/)

```
source "arch/$(SRCARCH)/Kconfig"
```

Arch/Kconfig의 첫줄로, 활성화된 옵션에 따라 다른 Kconfig 파일들을 소스하는 아키텍처별 구성 파일 포함

- 아키텍처에게 역할을 맡기는 세가지 이유

1. 리눅스를 구성할 때 ARCH = [아키텍처]를 설정해 아키텍처를 명시해야 함
   - 그렇지 않으면 기본으로 로컬 기계 아키텍처로 설정
2. ARCH에 대해 설정한 값이 일반적으로 SRCARCH의 값을 결정, SRCARCH를 명시적으로 설정할 필요 없음
3. 최상위 수준 메뉴의 레이아웃은 아키텍처 마다 다름

Kconfig 파일은 menu와 endmenu 키워드로 구분되는 메뉴로 구성, 메뉴 항목은 config 키워드로 표시

```
menu "Character devices"
[…]
config DEVMEM
    bool "/dev/mem virtual device support"
    default y
    help
      Say Y here if you want to support the /dev/mem device.
      The /dev/mem device is used to access areas of physical
      memory.
      When in doubt, say "Y".
[…]
endmenu
```

이 구성 항목은 다른 모든 항목과 함께 .config라는 이름의 파일에 저장됨

자료형

bool	Y이거나 정의되지 않음
tristate	특정 기능이 커널 모듈로 빌드 되거나 주 커널 이미지에 내장될 수 있을 때 사용 모듈의 경우 m, 내장의 경우 y, 기능이 비활성화 되면 값이 정의되지 않음
int	10진법으로 표기된 정수
hex	16진법으로 표기된 부호 없는 정수
string	문자열의 값

- **항목 사이의 의존관계를 나타내는 방법**

의존 관계 : 특정 옵션이 활성화되기 위해 다른 옵션이 반드시 활성화되어야 함.
ex) depends on MODULES

역의존 관계 : 한 옵션이 활성화되면 자동으로 다른 옵션도 활성화됨.
ex) select MODULES

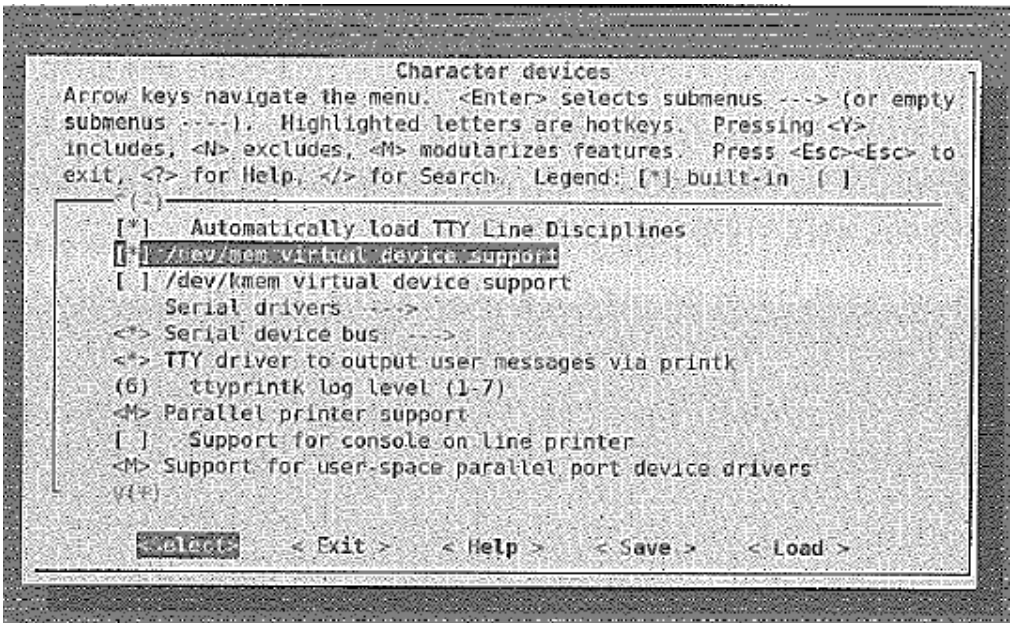
Menuconfig를 사용하려면 ncurses, flex, bison이 설치되어 있어야 함

```
user@raspberrypi:~ $ sudo apt install libncurese5-dev flex bison
```

-> 모든 필수 구성 요소를 설치

```
user@raspberrypi:~ $ make ARCH=arm menuconfig
```

-> menuconfig는 make 명령을 통해 실행, 커널의 경우 아키텍처를 지정해야 하므로 다음과 같이 실행



< * > -> 드라이버가 커널에 정적으로 빌드되도록 선택됐음을 의미
< M > -> 런타임에 커널에 삽입하기 위한 커널 모듈로 빌드되도록 선택됐음을 의미

make [구성 파일명] -> 구성 파일 선택 가능

```
$ make ARCH=arm multi_v7_defconfig
```

Oldconfig

- 기존 구성을 최신 커널 버전으로 옮길 때 사용
- 현재 디렉터리에 있는 `.config` 파일을 기반으로 하여 새로운 Kconfig 항목을 추가하거나 변경할 때, 기존의 설정을 최대한 유지
- 기존에 설정한 옵션은 그대로 사용되지만, 새로운 옵션에 대해서는 사용자에게 질문
- 이전 커널에서 새 소스 디렉터리로 .config를 복사하고 make ARCH=armoldconfig 명령을 실행해 최신 상태로 만듦

<LOCALVERSION 을 이용해 커널 식별하기>

- make kernelversion: 커널 버전 확인
- make kernelrelease: 커널 릴리스 확인

<언제 커널 모듈을 사용하는가?>

임베디드 커널은 모듈을 전혀 쓰지 않고 빌드하는 경우가 일반적임

임베디드 시스템에서 커널 모듈이 좋은 경우

1. 라이선스 이유로 인해 비공개 모듈이 있는 경우
2. 비필수 드라이버의 로딩을 연기해 부트 시간을 줄이기 위해
3. 로드할 수 있는 드라이브가 여러 개라서 정적으로 링크하면 너무 많은 메모리가 소요될 때
   (다양한 장치를 지원하는 USB 인터페이스를 갖고 있는 경우)

컴파일하기 :Kbuild

커널 빌드 시스템 Kbuild는 .config 파일로부터 구성 정보를 취해서 의존 관계를 파악하고 커널 이미지를 만들기 위해 필요한 모든 것을 컴파일 하는 make 스크립트들도 이뤄져있음

커널 이미지는 정적으로 링크되는 모든 요소(장치 트리 바이너리와 하나 이상의 커널 모듈 등) 를 포함함

```
obj-y += mem.o random.o
obj-$(CONFIG_TTY_PRINTK) += ttyprintk.o
```

- **Obj-y 규칙**

무조건 파일을 컴파일 해서 타깃을 생성
mem.c와 random.c는 언제나 커널의 일부가 됨
Ttyprintk.c는 구성 매개변수에 의존

CONFIG_TTY_PRINTK=y	내장형으로 컴파일
CONFIG_TTY_PRINTK=m	모듈로 빌드

CONFIG_TTY_PRINTK가 정의되지 않으면 컴파일되지 않음.

<어떤 커널 이미지를 컴파일할지 알아내기>

커널 이미지를 빌드하려면 부트로더가 기대하는 것이 무엇인지 알아야함

U-Boot

- 전통적으로 U-Boot는 uImage를 요구하지만 근래의 버전은 bootz 명령을 통해 zImage파일을 로드할 수 있음

X86 타깃

- bzImage 파일을 요구

대부분의 부트로더

- zImage 파일을 요구

```
$ make -j 4 ARCH=arm CROSS_COMPILE=arm-cortex_a8-linux-gnueabihf- zImage
```

-j 4 옵션은 make에게 몇 개의 작업을 병렬로 실행할 것인지 알려줘서 빌드 시간을 줄여줌

uImage 형식이 다중 플랫폼 이미지와 호환되지 않음
커널을 부트하고자 하는 특정 SoC의 LOADADDR을 설저아면 대중 플랫폼 빌드를 통해 uImage 바이너리를 만들 수 있음

mach-[해당 SoC]/Makefile.boot를 살피고 zreladdr-y의 값을 기록하면 로드 주소를 찾을 수 있음
타깃 커널 이미지 형식에 관계없이, 부팅 가능한 이미지가 생성되기 전에 동일한 2개의 빌드 아티팩트가 먼저 생성됨

<빌드 아티팩트>

커널을 빌드하면 최상위 수준 디렉터리에 2개의 파일 생성

Vmlinux
ELF 바이너리로 이루어진 커널
ELF 바이너리 도구(예: `size`)를 사용하여 각 섹션의 길이를 측정할 수 있음

System.map
사람이 읽을 수 있는 형태의 심볼 테이블을 담고 있음

대부분의 부트로더는 ELF 코드를 직접 처리할 수 없으며, vmlinux를 처리해서 다양한 부트로더에 적합하도록 몇 몇 파일을 arch/$ARCH/boot에 위치 시키는 추가적인 단계가 필요함

Image	가공되지 않은 바이너리 형식으로 변환된 vmlinux
zImage	파워 pc 아키텍처의 경우 이 파일은 단순히 압축된 Image파일이며, 부트로더가 이 파일을 압축 해제해야함 다른 모든 아키텍처의 경우, 압축된 Image가 압축 해제 및 재배치 코드와 합쳐져있음
uImage	zImage에 64바이트 U-Boot 헤더가 합쳐져 있음

- **커널 빌드 실패 시 디버깅**
  빌드 실패 시 실제 실행된 명령어를 확인하려면, make 명령줄에 V=1을 추가

<장치 트리 컴파일하기>

다중 플랫폼 빌드의 경우 여러 장치 트리를 빌드해야 함
Dtbs 타깃은 arch/$ARCH/boot/dts/Makefile에 있는 규칙에 따라 해당 디렉터리 안의 소스 파일을 이용해서 장치 트리를 빌드

<모듈 컴파일하기>

일부 기능을 모듈로 빌드하도록 구성했다면, modules 타깃을 이용해 독립적으로 빌드할 수 있음

```
$ make -j 4 ARCH=arm CROSS_COMPILE=arm-cortex_a8-linux-gnueabihf-
modules
```

컴파일된 모듈은 접미어 .ko가 붙고 소스 코드와 같은 디렉터리에 만들어짐
modules_install make 타깃을 이용해 올바른 장소에 설치 가능
루트 파일시스템의 스테이징 영역에 설치하려면 INSTALL_MOD_PATH를 이용해 경로 지정
-> /lib/modules/[커널버전]에 설치됨

<커널 소스 청소하기>

커널 소스 트리를 청소하기 위한 세가지 타깃

Clean	오브젝트 파일과 대부분의 중간 파일을 제거
Mrproper	모든 중간 파일 제거, 소스 트리를 복제하거나 압축 해제한 직후의 상태로 되돌릴 수 있음
Distclean	mrproper와 같지만 편집기 백업 파일과 패치 파일, 기타 소프트웨어 개발 부산물도 지움

<라즈베리 파이 4용 64비트 커널 빌드하기>

```
$ cd ~
$ wget https://developer.arm.com/-/media/Files/downloads/
gnu-a/10.2-2020.11/binrel/gcc-arm-10.2-2020.11-x86_64-aarch64-
none-linux-gnu.tar.xz
$ tar xf gcc-arm-10.2-2020.11-x86_64-aarch64-none-linux-gnu.
tar.xz
$ mv gcc-arm-10.2-2020.11-x86_64-aarch64-none-linux-gnu \
gcc-arm-aarch64-none-linux-gnu

$ sudo apt install subversion libssl-dev

$ git clone --depth=1 -b rpi-4.19.y https://github.com/
raspberrypi/linux.git
$ svn export https://github.com/raspberrypi/firmware/trunk/boot
$ rm boot/kernel*
$ rm boot/*.dtb
$ rm boot/overlays/*.dtbo

$ PATH=~/gcc-arm-aarch64-none-linux-gnu/bin/:$PATH
$ cd linux
$ make ARCH=arm64 CROSS_COMPILE=aarch64-none-linux-gnu- \
bcm2711_defconfig
$ make -j4 ARCH=arm64 CROSS_COMPILE=aarch64-none-linux-gnu
```

```
$ cp arch/arm64/boot/Image ../boot/kernel8.img
$ cp arch/arm64/boot/dts/overlays/*.dtbo ../boot/overlays/
$ cp arch/arm64/boot/dts/broadcom/*.dtb ../boot/
$ cat << EOF > ../boot/config.txt
enable_uart=1
arm_64bit=1
EOF
$ cat << EOF > ../boot/cmdline.txt
console=serial0,115200 console=tty1 root=/dev/mmcblk0p2
rootwait
EOF
```

1. 필요한 툴체인과 패키지 설치
2. 라즈베리 파이 재단 커널 포크의 rpi-4.19.y 브랜치를 linux 디렉터리에 복제
3. 라즈베리 파이 재단 펌웨어 저장소의 boot 서브디렉터리 내용을 boot 디렉터리로 export
4. Boot 디렉터리에서 기존 커널 이미지, 장치 트리 블롭, 장치 트리 오버레이를 삭제
5. Linux 디렉터리에서 라즈베리 파이4용 64비트 커널, 모듈, 장치 트리를 빌드
6. 새로 빌드된 커널 이미지, 장치 트리 블롭, 장치 트리 오버레이를 arch/arm64/boot/에서 boot 디렉터리로 복사
7. 라즈베리 파이 4의 부트로더가 읽고 커널에 전달할 수 있도록 config.txt와 cmdline.txt파일을 boot 디렉터리에 씀

커널 부팅하기

리눅스 부팅은 매우 장치 의존적

<라즈베리 파이 4 부팅하기>

- 라즈베리 파이는 U-Boot 대신 브로드컴에서 제공하는 독점 부트로더를 사용
- 라즈베리 파이4의 부트로더는 온보드 SPI EEPROM에 존재 (이전 버전은 마이크로 SD카드)
- 커널 빌드 아티팩트를 저장할만큼 충분히 큰 FAT32 부팅 파티션이 있는 마이크로 SD 카드가 필요
- Boot파티션은 마이크로 SD카드의 첫번째 파티션이어야 함 (파티션 크기는 1GB면 충분)
- 마이크로 SD카드를 카드 리더에 넣고 boot 디렉터리의 전체 내용을 boot 파티션에 복사
- 카드를 마운트 해제하고 라즈베리 파이 4에 넣기
- USB-to-TTL 시리얼 케이블을 40핀 GPIO 헤더의 GND, TXD, RXD 핀에 연결
- Gtkterm과 같은 터미널 에뮬레이터를 시작
- 라즈베리 파이 4의 전원을 켜면 시리얼 콘솔에 다음과 같은 출력 표시

<커널 패닉>

커널이 복구할 수 없는 에러를 만났을 때 발생
기본 설정으로는 콘솔로 메시지를 출력한 다음 멈춤
Panic 명령줄 매개변수를 설정해서 패닉이 일어났을 때 몇초 기다린 뒤 리부트하도록 설정 가능

램디스크나 대용량 저장 장치에 루트 파일시스템을 제공함으로써 사용자 공간 제공

```
[ 1.886379] Kernel panic - not syncing: VFS: Unable to mount root fs on
unknown-block(0,0)
[ 1.895105] ---[ end Kernel panic - not syncing: VFS: Unable to mount root
fs on unknown-block(0, 0)
```

<초기 사용자 공간>

커널 초기화에서 사용자 공간으로 이행하기 위해, 커널은 루트 파일시스템을 마운트하고 루트 파일시스템에 있는 프로그램 실행
이는 램디스크를 통하거나 블록 장치 상의 실제 파일시스템을 마운트함으로써 시작
이를 처리하는 모든 코드는 init/main.c에 있고 rest_init()의 코드를 실행
이 함수는 PID가 1인 첫번째 스레드를 만들고 kernel_init()의 코드를 실행
램디스크가 있다면 프로그램/init을 실행하려고 할 것이며, 이 프로그램은 계속해서 사용자 공간을 설정하는 작업을 수행할 것임
커널이 /init 을 찾아서 실행하는데 실패하면, init/do_mounts.c안의 함수 prepare_namespace()를 불러 파일 시스템을 마운트 하려고 할 것

```
root=/dev/<disk name><partition number>
```

마운트 할 때 쓸 블록 장치 이름을 제공

또는 SD카드와 eMMC의 경우

```
root=/dev/<disk name>p<partition number>
```

성공적으로 마운트되면, 다음 경로의 프로그램을 순차적으로 실행하려고 시도

- /sbin/init
- /etc/init
- /bin/init
- bin/sh

프로그램은 명령 줄에서 다른 프로그램으로 바꿀 수 있음

- 램디스크의 경우 rdinit= 사용
- 파일 시스템의 경우 init= 사용

<커널 메시지>

메시지는 중요도에 따라 분류, 가장 높은 중요도: 0

수준	값	뜻
KERN_EMERG	0	시스템이 사용 불능이다.
KERN_ALERT	1	즉시 조치를 취해야 한다.
KERN_CRIT	2	위급 상태
KERN_ERR	3	에러 상태
KERN_WARNING	4	경고 상태
KERN_NOTICE	5	정상이지만 중요한 상태
KERN_INFO	6	정보
KERN_DEBUG	7	디버그 수준 메시지

메시지는 먼저 버퍼(__log_buf)에 기록되는데, 그 크기는 2의 CONFIG_LOG_BUF_SHIFT승

기본 콘솔 로그 수준은 7, 수준 6 이하 메시지는 표시되지만 수준 7인 KERN_DEBUG는 필터링됨

콘솔 로그 수준을 바꾸는 방법
Loglevel=<수준>, dmesg -n <수준> 등

<커널 명령줄>

U-Boot의 경우 부트로더가 bootargs 변수를 통해 커널에게 전달하는 문자열
장치 트리에 정의돼 있을수도 있고, 커널 구성의 일부로 CONFIG_CMDLINE에 설정 될수도 있음

debug	콘솔 로그 수준을 가장 높은 수준(8)으로 설정해 모든 커널 메시지를 콘솔에서 볼 수 있도록 함
init=	마운트된 루트 파일 시스템에서 실행되는 init 프로그램. 기본 설정은 /sbin/init임
lpj=	loops_per_jiffy를 주어진 상수로 설정, 이 목록 다음 문단에 이 설정의 중요성에 대한 설명이 있음
panic	커널 패닉 시 동작 0보다 크면 해당 초만큼 기다렸다가 리부트 0이면 영원히 기다림 (기본 설정) 0보다 작으면 기다리지 않고 리부트
quiet	콘솔 로그 수준을 silent로 설정해 긴급 메시지를 뺀 모든 메시지를 보이지 않게 함 대부분의 장치는 시리얼 콘솔을 사용하므로 이들 문자열 모두를 출력하려면 시간이 걸림 이 옵션을 이용해 메시지 수를 줄이면 부트 시간이 줄어듦
rdinit=	램 디스크로부터 실행되는 init 프로그램. 기본 설절은 /init
ro	루트 장치를 읽기 전용으로 마운트. 언제나 읽고 쓸 수 있는 램디스크에는 효과가 없음
root	루트 파일시스템을 마운트할 장치
rootdelay=	루트 장치를 마운트하기 전에 기다릴 초의 수. 장치가 하드웨어를 검색할 때 시간이 걸리는 경우 유용
rootfstype=	루트 장치 파일 시스템 종류. 많은 경우에는 마운트 도중 자동 탐지 되지만 gffs2파일시스템의 경우 수동 설정이 필요
rootwait	루트장치가 탐지되기를 무한정 기다림. 보통 MMC 장치에 필요

rw	루트 장치를 읽기 전용으로 마운트

- **lpj=??**
  커널이 루프를 수행하는 데 필요한 시간을 조정하여 부팅 시간을 최적화

```
Calibrating delay loop... 996.14 BogoMIPS (lpj=4980736)
```

lpj 값은 CPU의 클록 주파수에 따라 달라지며, 특정 하드웨어에 맞춰 최적화된 값을 설정

리눅스를 새 보드에 이식하기

<새로운 장치 트리>

1. 보드의 장치 트리 생성, 보드상에 추가되거나 변경된 하드웨어를 기술하도록 수정

2. Am335x-boneblack.dts 를 nova.dts로 복사하고 nova.dts 안의 보드 이름을 변경

```
/dts-v1/;

#include "am33xx.dtsi"
#include "am335x-bone-common.dtsi"
#include "am335x-boneblack-common.dtsi"

/ {
        model = "Nova";
        compatible = "ti,am335x-bone-black", "ti,am335x-bone",
"ti,am33xx";
};
[…]
```

3. NOVA 디바이스 트리 바이너리를 명시적으로 빌드

```
$ make ARCH=arm nova.dtb
```

4. AM33xx타깃이 선택될 때마다 make ARCH=arm dtbs 를 통해 Nova 디바이스 트리가 컴파일되길 바라면 arch/arm/boot/dts/Makefile 에 의존관계를 추가

```
[…]
dtb-$(CONFIG_SOC_AM33XX) +=
    nova.dtb
[…]
```

Nova 디바이스 트리를 이용해 비글본 블랙을 부팅한 효과를 볼 수 있음

<보드의 compatible 프로퍼티 설정하기>

보드 설정은 루트 노드의 compatible 프로퍼티로 제어

커널은 노드를 파싱 시, compatible 프로퍼티의 각 값과 일치하는 기계를 왼쪽에서 시작해 일치할 때까지 찾음
각 기계는 DT_MACHINE_START와 MACHINE_END 매크로로 구분된 구조체로 정의
DT_MACHINE_START와 MACHINE_END 사이의 구조체에는 문자열 배열의 포인터와 보드 설정 함수의 함수 포인터가 있음