내장형 리눅스를 위한 시스템 최적화 기술

김용운, 박정수, 김용진
한국전자통신연구원 표준연구센터

System Optimization Techniques for Embedded GNU/Linux

Yong-Woon KIM, Jung-Soo Park, and Yong-Jin KIM
Protocol Engineering Center, ETRI

요약

내장형 시스템(Embedded System)은 운영체제와 함께 사용자가 원하는 목적과 기능의 응용 프로그램을 내장하여 원하는 작업을 할 수 있도록 해주며, 각종 전자기기들이 디지털화 되면서 각종 기능들을 제어하기 위한 운영체제로서 내장형 운영체제가 주목을 받기 시작하였다. 데스크탑 시스템과 내장형 시스템의 하드웨어 특성과 요구사항은 서로 다르기 때문에 데스크탑 기반의 Unix 운영체제로 널리 쓰이고 있는 GNU/Linux를 내장형 시스템의 운영체제로 사용하기 위해서는 여러 가지 운영체제 구성 요소들에 대한 최적화가 뒤따라야 한다. GNU/Linux의 최적화를 위해 고려해 볼 수 있는 다섯 가지 방법을 설명하고, 실례로서 LRP에서 만든 결과를 분석해 보기로 한다.

1. 서 론

내장형 시스템은 운영체제와 함께 사용자가 원하는 목적과 기능의 응용 프로그램을 내장하여 원하는 작업을 할 수 있도록 해주며, 그런 점에서 PC 컴퓨터의 다른 형태라 할 수 있다. PC의 경우에는 ROM, RAM, CPU, 모니터, 키보드, 및 하드디스크가 주요한 구성 요소이고 응용 프로그램들과 운영체제가 하드 디스크에 저장되어 있다.

반면에 내장형 시스템은 PC 시스템과 비슷하게 구성되지만 운영체제와 응용 프로그램, 및 기타 파일들이 ROM에 저장되고, 하드 디스크에 저장되어 있는 파일 시스템을 이용하는 것이 아니라 ROM에 들어 있는 파일 시스템을 RAM에 로딩하여 사용하며, 주로 키보드 입력 장치를 사용하지 않고 시리얼 콘솔로 연결하여 가끔 일어나는 필요한 작업을 하고, 아무런 출력 장치를 사용하지 않는 경우가 많다. 그러나 필요에 따라 키보드를 입력장치로 활용하기도 하고, LCD 패널을 출력장치로 활용하기도 한다.

따라서 내장형 시스템은 PC 컴퓨터와 매우 유사한 기능을 한다고 할 수 있으나 하드웨어의 특성과 서비스 요구사항 및 목적이 다르다. PC 컴퓨터는 항상 전원이 들어와 있고 범용 목적으로 쓰이지만 내장형 시스템은 특정한 목적 전용으로 쓰이고 제한적 동작 환경에서 운용되는 경우가 많다. 로봇, 계측기, 등 산업현장 기기는 물론이고, 전자 시계, 프린터, TV용 인터넷 셋탑 박스, PDA, 디지털 카메라, 오디오, VCR 카메라 등이 예가 될 수 있고, 전자 제어 장치가 들어가는 모든 제품에 대해 원하는 서비스 기능을 제공하기 위해 내장형 시스템으로 구성될 수 있다.

따라서 내장형 시스템은 PC에 비해 시스템의 기능 구조는 비슷하지만, 제한된 목적으로 쓰이기 때문에 하드웨어 구조는 훨씬 더 단순할 수 있고, 더 다양하고 폭 넓은 응용 범위를 가진다. 특정 목적 전용의 시스템으로 만들어지고 그에 맞는 동작 환경에서 운용되기 때문에 시스템 하드웨어의 특성이 달라질 수밖에 없으며, PDA, 휴대용 PC, 셀룰러폰 등과 같이 건전지로 동작하는 장치의 경우에 절전이 매우 중요한 기능 목표가 되고, 내장형 시스템에 탑재되는 CPU 프로세서가 특정한 목적으로 개발된 것일 수도 있고, RAM이나 ROM의 용량도 비교적 소형일 경우가 많고, 키보드 입력과 LCD 패널 출력이 없는 형태로 구성될 수도 있다. 또한 사람의 안전과 직결되는 중요한 제어 시스템이 있을 수도 있으며, 이 경우에는 신뢰성과 안정성이 매우 중요한 고려사항이 되어야 한다.

그러므로 기존의 PC 운영체제를 내장형 시스템에 그대로 사용할 수가 없으며 설계된 하드웨어와 기능 요구사항에 따라 최적화된 운영체제가 사용되어야 하고, 이를 바탕으로 한 응용이 개발되어 탑재되어야 한다.

GNU/Linux가 장시간의 안정성과 신뢰성이 증명되면서 최근 내장형 시스템의 운영체제로 주목을 받고 있으며, 이것을 이용한 제품들이 속속 발표되고 있는 상황이다.

GNU/Linux는 세계적으로 수많은 개발자가 공동 참여하여 개발되고 있으므로 비교적 빠르게 새로운 기술이 접목되고 있으며, 10여년 동안 개발과 실제 운영이 지속적으로 같이 이루어져 발전한 것으로서 운영체제의 안정성과 신뢰성이 매우 높은 것으로 보고되고 있고, GNU 저작권 규정에 따라 별도의 저작권료를 낼 필요가 없어 많은 내장형 시스템에서 운영체제로 GNU/Linux가 선택되고 있는 상황이다.

GNU/Linux는 x86 PC 컴퓨터를 기반으로 만들어진 후에 MIPS, Alpha, PowerPC, SPARC, ARM 등과 같은 다양한 하드웨어 플랫폼에 이식되어 있으나, 범용 목적의 데스크탑 기반 컴퓨터에 운

용되도록 만들어져 있어 내장형 시스템의 운영체제로 사용하기 위해 내장형 시스템의 운용 목적에 맞는 최적화가 이루어져야 한다.

2절에서는 GNU/Linux에 적용할 수 있는 최적화 기술에 대해 설명하고, 3절에서는 LRP(Linux Router Project)에서 개발한 것을 바탕으로 최적화의 실제 예를 분석하고, 마지막으로 4절에서 결론을 맺도록 한다.

2. GNU/Linux 시스템 최적화 기술

GNU/Linux를 내장형 시스템용 운영체제로 최적화하는 데에 가능한 다섯 가지 방법을 제시하고, 이들 외에도 가능한 방법들이 있을 수 있다.

이러한 방법들은 운영체제의 크기를 최소화하여 ROM과 RAM의 크기를 최소화할 수 있도록 하는 것이며, 절전 기능이나 Real-time OS 등의 기능을 위해서는 별도의 최적화 또는 확장 작업이 이루어져야 한다. 통상 말하는 시스템 최적화는 운영체제의 규모와 크기를 최소화하는 것이다.

2.1. 시스템 유틸리티 최적화

필수 유틸리티 선택

많은 경우에 있어 내장형 시스템의 전체 소프트웨어 가운데 10%가 응용 프로그램이고 나머지 90%가 시스템 제어용 유틸리티와 시스템 커널이다. GNU/Linux에서의 시스템 유틸리티는 150여 개에 이르고 있으며 내장형 시스템에서 모두 필요한 것이 아니다.

따라서 내장형 시스템 제어에 필요한 유틸리티만 선택하고 불필요한 것들을 제거함으로써 최적화가 가능하다.

예를 들어, 파일 시스템의 디스크 용량을 알려주는 df, 디렉토리를 만들어주는 mkdir 등과 같은 것들은 내장형 시스템의 목적과 기능에 따라 필요없는 것이 될 수 있으므로 제거할 수 있다. 이러한 방법은 절대적인 것이 아니며 목적과 기능에 따라 결정되어야 한다.

BusyBox 활용

BusyBox는 소형 또는 내장형 시스템에서 최적의 유틸리티 환경을 제공해줄 수 있고, 다중호출(multi-call) 바이너리 형태의 소형 유닉스 유틸리티이다[1].

하나의 유틸리티는 고유한 목적의 기능 수행을 위한 코드 부분과 화면 입출력과 같은 다른 유틸리티와의 공통 코드 부분도 같이 포함하고 있다. 따라서 다수의 유틸리티에 있는 공통 코드와 각 개별 유틸리티의 고유 기능 코드를 한꺼번에 묶어서 다중호출 형태의 실행 프로그램 한 개로 만들게 되면 유틸리티 전체 크기를 대폭 줄이는 최적화 효과가 일어나게 된다. BusyBox는 이러한 방식으로 개발된 것이다.

일반적 사용 방법은 각 유틸리티 이름에 대해 BusyBox로 링크를 만들어두는 것이다. 예를 들어, ls 명령어에 대해,

```
$ ln s ./busybox ls
```

이와 같이 해두면, ./ls 명령어 실행에 대해 busybox가 ls로서 동작하여 해당하는 기능을 수행하게 된다. 또한,

```
$ ./busybox ls
```

이와 같은 명령어 실행도 같은 기능을 수행한다.

BusyBox 0.45 버전에 포함되어 있는 유틸리티들은 다음과 같다.

ar, basename, cat, chgrp, chmod, chown, chroot, chvt, clear, cp, cut, date, dc, dd, deallocvt, df, dirname, dmesg, du, dutmp, echo, false, fbset, fdflush, find, free, freeramdisk, fsck.minix, grep, gunzip, gzip, halt, head, hostid, hostname, id, init, insmod, kill, killall, length, ln, loadacm, loadfont, loadkmap, logger, logname, ls, lsmod, makedevs, mkdir, mkfifo, mkfs.minix, mknod, mkswap, mktemp, more, mount, mt, mv, nc, nslookup, ping, poweroff, printf, ps, pwd, reboot, rm, rmdir, rmmod, sed, setkeycodes, sfdisk, sh, sleep, sort, swapoff, swapon, sync, syslogd, tail, tar, tee, telnet, test, touch, tr, true, tty, umount, uname, uniq, update, uptime, usleep, uudecode, uuencode, wc, which, whoami, yes, zcat, [

위와 같은 유틸리티들이 내장형 시스템에 모두 필요한 것은 아니기 때문에 내장형 시스템의 목적과 기능에 따라 적절히 선택하여야 하며, busybox에 포함시키거나 제외시키는 것은 busybox.def.h 파일에서 선택할 수 있도록 되어 있고, //를 이용하여 코멘트 표시를 하면 제외시킬 수 있다.

TinyLogin 활용

TinyLogin도 BusyBox와 비슷한 목적을 갖고 있다[2]. TinyLogin은 시스템 로그인, 사용자 인증 및 비밀번호 변경 등을 작업을 위해 필요한 몇 가지 유틸리티를 하나로 합친 것이다. TinyLogin은 시스템 보안을 향상시키는 방안으로 shadow password 기능을 제공하며, BusyBox를 보완하는 유틸리티로 활용할 수 있으나 BusyBox 없이 독립적으로 활용할 수도 있다. 하지만 대부분 경우에 같이 사용하곤 한다.

GNU/Linux 시스템에 있는 관련 실행 파일들의 크기와 TinyLogin의 실행 파일 크기를 비교해보면 다음과 같다.

```
$ du -ch 'which adduser deluser delgroup login
sulogin passwd getty'
60k     /usr/sbin/useradd
20k     /bin/login
16k     /sbin/sulogin
24k     /usr/bin/passwd
32k     /sbin/getty
152k    total
$ ls -sh ./tinylogin
36k     ./tinylogin
```

TinyLogin도 BusyBox와 마찬가지로 모듈 설계로 되어 있어 내장형 시스템의 목적과 기능에 따라 필요한 기능만을 선택적으로 포함시킬 수 있으며, tinylogin.def.h에서 //를 이용하여 코멘트 표시를 하면 제외시킬 수 있다.

Ash 활용

Unix 환경의 쉘 프로그램으로 bash가 여러 가지 향상된 기능들을 갖고 있어 많이 쓰이고 있는데 크기가 다소 크다. 반면에 ash는 매우 작은 크기로서 쉘이 가지는 기본적인 기능들을 다 가지고 있으며 POSIX 인터페이스 규격을 지원하고 있고, /bin/sh로 설치하여 사용할 수도 있다. 따라서 내장형 시스템의 쉘 프로그램으로 ash를 채택하게 되면 최적화에 도움이 될 수 있다. 참고로 두 가지 쉘의 크기를 살펴보면 다음과 같다[3].

```
$ ls -sh /bin/bash
376k /bin/bash
```

```
$ ls -sh /bin/ash
68k /bin/ash
```

2.2. 부팅 절차의 단순화

GNU/Linux 시스템에 전원이 공급되어 커널이 로딩되면 커널이 시스템 요소들에 대한 초기화 작업을 진행하고 이어서 시스템 최초의 프로세스로 init 프로그램을 실행시키고, init 프로세스는 프로세스 식별자로 '1'을 갖는다. 이렇게 되면 시스템 부팅 절차가 완료되었다고 할 수 있으며, init 프로세스는 시스템 콘솔을 열고 루트 파일 시스템을 마운트하는 등, 시스템 운영에 대한 초기화의 일부 역할을 담당하고, /etc/inittab의 정보를 참조하여 시스템 운영에 필요한 새로운 프로세스들을 생성시킨다.

Init이 생성시킨 새 프로세스들은 또 다른 프로세스들을 만들기도 하는데, 예를 들면, getty 프로세스는 사용자가 로그인을 시도할 때 login 프로세스를 만들기도 한다. 시스템 내에 있는 모든 프로세스들은 init 커널 프로세스의 자손들이다. 예를 들어, GNU/Linux 시스템 내의 프로세스 관계도 일부를 보면 다음과 같다. init은 모든 프로세스의 부모 프로세스이며, 그 외 프로세스들에 대한 부모와 자손 관계가 나타나있다.

```
$ pstree
init-+-apmd
     |-crond
     |-hanterm---bash---vi
     |-sh---hanterm-org---bash---su---bash
     |-syslogd
     `-xscreensaver---attraction
```

따라서 프로세스로 동작할 시스템 유틸리티를 최적화하는 것뿐만 아니라, inittab의 내용을 단순화시켜 내장형 시스템의 목적과 기능에 맞는 것만 수행시키도록 하여야 한다.

GNU/Linux 시스템에서 많이 사용하는 init 프로그램은 sysvinit 패키지 속에 있으며[4], 내장형 시스템에서도 그대로 사용할 수 있고, Debian 배포판에서 시스템 부팅을 더욱 빠르고 효과적으로 수행하도록 하기 위해 만든 start-stop-daemon 프로그램을 같이 사용하여 기능 향상을 도모할 수 있다. 이제는 sysvinit 패키지 속에 start-stop-daemon 프로그램도 같이 들어 있다.

BusyBox에는 가장 기본적인 형태의 init 기능도 포함되어 있다. 예를 들어, 여러 단계의 run level을 지원하지 않으며, inittab 파일이 없어도 다음과 같은 기본 설정으로 동작할 수도 있다.

```
::sysinit:/etc/init.d/rcS
::askfirst:/bin/sh
```

2.3. 라이브러리 최적화

라이브러리 최적화는 두 가지 방향에서 접근할 수 있다. 필요한 시스템 유틸리티의 수가 적다면 공유 라이브러리를 사용하는 것보다는 static으로 컴파일하여 라이브러리를 아예 사용하지 않도록 할 수 있다. 반면에 유틸리티의 수가 많다면 공유 라이브러리를 사용하는 것이 더 효과적으로 최적화될 수 있다.

공유 라이브러리를 사용하는 경우에 모든 라이브러리가 다 필요한 것은 아니므로 시스템 유틸리티가 필요로 하는 라이브러리만 포함시키도록 하고, 많은 기능을 갖고 있는 최신 라이브러리를 사용하는 것보다는 필요로 하는 기능이 지원되고 크기도 작다면 구형 버전의 라이브러리를 사용하는 것이 더 효율적일 것이다.

예를 들어, glibc-2.1.2 패키지에 의해 만들어진 libc-2.1.2.so 라이브러리의 크기는 다음과 같다.

```
$ ls -sh /lib/libc-2.1.2.so
3.9M /lib/libc-2.1.2.so
```

반면에 구형 버전은 훨씬 작은 크기이다.

```
$ ls -sh libc.so.6
340k libc.so.6
```

GNU/Linux 기반의 PC 시스템에는 보통 최신 라이브러리가 설치되어 있으며 libc.so.6를 libc-x.x.x.so로 링크해 두고 있다.

```
$ ls -al /lib/libc.so.6
lrwxrwxrwx  1 root  root  13 Apr 19 12:53
libc.so.6 -> libc-2.1.2.so
```

특히 BusyBox를 이용하는 경우에 컴파일된 busybox 유틸리티는 /lib/libc.so.6 라이브러리를 포함하고 있는데 /lib/libc-2.1.2.so 는 너무 큰 규모이므로 /lib/libc.so.6 구형 라이브러리를 사용해야 한다.

2.4. 커널 최적화

Linux 커널을 만드는 과정에 많은 선택 사항들이 있으며 이에 의해 특정 기능 모듈이 커널에 포함되어 만들어질 수 있고 제외될 수도 있으며 필요한 때만 선택적으로 커널로 로딩되었다가 필요 없을 때 제거되는 방식으로 만들어질 수 있다.

따라서 커널 최적화의 방법으로는 아예 필요 없는 기능을 제외하거나 대체 가능한 최소 기능 모듈을 선택하거나 커널 모듈의 선택적 로딩 등으로 나눌 수 있다.

예를 들어, 네트워킹이 필요하지 않은 내장형 시스템이라면 TCP/IP 네트워킹 모듈을 제외할 수 있고, SCSI 인터페이스를 사용하지 않고 HDD를 사용하지 않는다면 이 기능 또한 제거할 수 있고, 그 외의 많은 기능들에 대해서도 불필요한 것이라면 Linux 커널 컴파일 옵션에서 제외하도록 선택한다.

또한 GNU/Linux 시스템이 주로 사용하는 ext2 파일 시스템 대신에 소형인 minix 파일 시스템을 쓸 수도 있고, 키보드 콘솔 장치를 사용하지 않는 내장형 시스템이라면 Linux 콘솔 대신에 시리얼 콘솔을 사용하도록 할 수 있다.

앞으로 사용할 가능성이 있는 기능 모듈은 처음부터 커널에 포함시키는 것보다 모듈 로딩 방식으로 커널을 만들도록 하여 커널 내부에 포함시키지는 않지만 향후 필요에 따라 사용할 수 있도록 하는 것이 더 좋다.

2.5. 컴파일링 최적화

대부분의 프로그램과 라이브러리들은 기본적으로 특정한 CPU에 맞게 만들어지고, 또한 디버깅 심볼을 가지고 레벨 2로 (GCC 옵션에서 g와 O2) 컴파일되도록 하고 있다. 만약 목표로 하는 장치에서만 운용하고 다른 장치에서 사용할 계획이 없다면 컴파일 옵션을 바꾸어 목표 시스템에 맞게끔 최적화할 수 있다. 즉, 좀 더 높은 최적화 레벨을 적용하고, 디버깅 심볼을 포함시키지 않도록 하고, 목표 시스템의 하드웨어 플랫폼에 맞도록 컴파일하는 것이다[5].

컴파일러와 라이브러리 선택

소스 코드를 컴파일하여 만들어지는 실행 프로그램의 크기는 어떤 종류의 컴파일러를 사용하느냐에 따라 달라질 수 있고, 같은 컴파일러를 사용하더라도 어떤 버전의 C 라이브러리를 사용하느냐에 따라 달라지기도 한다.

따라서 실행 코드 최적화를 위한 방안으로서 목표로 하는 시스템의 하드웨어 플랫폼에 최적화시킨 상용 컴파일러를 구입하는 것도 고려하여야 한다. 그러나 많은 경우에 GCC 범용 컴파일러를 이용한다.

GCC 컴파일러를 이용한다면 라이브러리 버전의 차이에 따른 크기 변화를 비교해 보고 선택하면 될 것이다.

디버깅 심볼 제거

이미 만들어진 바이너리 실행 프로그램으로부터 디버깅 심볼을 제거하기 위해서는 아래와 같은 strip 프로그램을 이용하면 된다. 여러 개의 실행 파일에서 디버깅 심볼을 제거하고자 할 때는 와일드 카드 (*) 표시를 이용할 수 있다.

```
$ strip strip-debug filename
```

또는

```
$ strip strip-debug /usr/bin/*
```

실행 프로그램을 만들기 위해 컴파일하는 경우라면 아예 디버깅 심볼을 포함시키지 않도록 컴파일하면 될 것이다. 따라서 GCC 컴파일 옵션에서 g 옵션을 빼도록 한다.

/lib와 /usr/lib에 있는 glibc와 gcc 파일들을 디버깅 심볼을 포함하는 것과 하지 않는 것을 비교하여 보면 그 차이가 더욱 뚜렷해서 87MB와 16MB라는 차이를 보이고 있다. 따라서 반드시 디버깅 심볼들을 포함하지 않도록 컴파일해야 한다.

컴파일 옵션 최적화

컴파일 최적화를 위해 사용하는 옵션은 O인데, O0, O1, O2, 및 O3 네 가지로 나뉜다. O0 옵션은 최적화를 하지 않는다는 뜻이고, 숫자가 높을수록 최적화 수준이 높아진다. 각각의 차이는 GCC 사용 설명서를 참조하면 된다. 또한 컴파일을 할 때 목표 시스템의 하드웨어 구조를 명시하여 최적화하는 것이 좋다.

이러한 최적화 기능을 수행하도록 하기 위해 기본 컴파일 옵션을 변경해야 하는데, 첫 번째 방법으로는 컴파일 대상 소스 코드 패키지 내에 있는 모든 Makefile에 들어 있는 CFLAGS (GCC 컴파일러 옵션) 및 CXXFLAGS (G++ 컴파일러 옵션) 변수를 적절히 수정하는 것이다.

그러나 모든 디렉토리를 찾아서 일일이 수정하는 것은 어려운 일이므로 CFLAGS와 CXXFLAGS 환경 변수를 만들어 Makefile에 있는 변수에 우선한다는 것을 알려주는 것이 좋다. 이것은 make에서 e 옵션을 사용하면 된다.

bash 쉘 환경에서 CFLAGS와 CXXFLAGS 환경 변수를 만들어 주는 방법은 다음과 같다.

```
$ export CFLAGS=-O3 mcpu=xx march=yy
```

또는

```
$ export CXXFLAGS=-O3 mcpu=xx march=yy
```

여기에서 xx와 yy는 적절한 CPU 식별자를 의미하는데, 예를 들어, 인텔 80686 CPU의 경우라면 i686이란 식별자를 사용하면 된다. 즉, 다음의 예와 같다.

```
$ export CFLAGS=-O3 mcpu=i686 march=i686
```

그러나 이러한 컴파일 옵션 대치 방식은 원래의 CFLAGS나 CXXFLAGS 내에 매크로나 다른 옵션을 포함하고 있을 때 문제를 일으킬 수 있다. 즉, 원래에 있던 컴파일 옵션이 제외될 가능성이 있는 것이다. 따라서 이러한 문제 가능성을 염두에 두고 유의하여 작업해야 한다. make와 make e 두 가지 컴파일에 대한 결과를 비교해 본다면 어떤 문제가 발생하는지 알 수 있을 것이다.

3. LRP 사례 분석

LRP는 1.44MB MS-DOS 형식 디스켓 한 장에 최소형의 GNU/Linux 시스템을 구성할 수 있는 커널, 시스템 유틸리티, 및 응용을 담고 있으며, 사실상 무용지물로 구석에 버려져 있는 인텔 80386, 80486과 같은 구형 PC 시스템을 가지고 소형 라우터로 만들 수 있도록 하기 위해 만든 것이다[6].

따라서 LRP(Linux Router Project)는 두 가지 관점에서 살펴볼 수 있는데, MS-DOS 형식의 디스켓으로부터 GNU/Linux 시스템 부팅을 할 수 있도록 하는 것과 1.44MB MS-DOS 디스켓 한 장에 최소형 GNU/Linux 시스템을 구성할 수 있도록 최적화하는 것이다.

3.1. GNU/Linux 시스템 부팅

모든 PC 시스템은 전원을 켠 후에 ROM BIOS의 코드를 실행하는 것으로부터 동작을 시작하고, 주어진 시스템 점검을 마친 후에 부트 드라이브의 헤드0, 실린더 0, 섹터 1의 자리에 있는 부트 섹터를 RAM으로 로딩(loading)한다. 그런 후에 프로그램 실행은 로딩된 부트 코드로 넘어가게 된다[7][8].

대부분의 GNU/Linux 시스템 부트 드라이브 헤드0, 실린더 0, 섹터 1의 자리에는 다음 두 가지 경우 가운데 하나가 들어 있는데, LILO와 같은 부트 로더가 있거나, Linux 운영체제 커널의 시작점이 있다.[1]

만약 Linux 커널을 디스켓이나 하드 드라이브의 첫 번째 섹터로부터 로딩해서 부팅을 시키게 되면 BIOS 코드가 실행된 후에 곧바로 커널 부팅이 시작되게 된다. 그러나 이런 방식은 사용자가 운영체제를 자유로이 선택할 수 없는 제한을 만들게 되므로 부트 로더 (boot loader)를 이용하여 부팅 절차를 제어할 수 있도록 한다.

부트 로더는 같은 하드웨어 시스템에서 단지 다시 부팅만 함으로써 원하는 운영체제를 선택하여 다른 운영체제로 전환할 수 있도록 한다. GNU/Linux 시스템에서 가장 많이 쓰이는 부트 로더는 LILO이며, 커널이 있는 위치를 알고 있어 커널 로딩을 하고 실행시키도록 함으로써 정상적인 부팅을 시작하도록 한다. LILO 부트 로더는

1 부트 섹터는 512 바이트로 이루어져 있으며, 부트 섹터에 부트 로더 전체가 들어가 있을 수도 있고, 부트 로더의 크기가 커서 부트 섹터 밖의 데이터 영역까지 자리할 수 있다. 후자의 경우와 마찬가지로 Linux와 같은 운영체제 커널도 부트 섹터 1의 자리부터 커널이 시작되어 데이터 영역까지 자리잡을 수 있다. 참고로 Linux 커널의 앞부분 512바이트에는 부트 로더가 들어 있다. 따라서 섹터 1번지에 부트 로더를 설치하지 않고, Linux 커널을 두더라도 정상적인 부팅이 가능하다.

부트 섹터에 저장되어 있고 lilo.conf 파일에 부트 로더에 대한 환경 설정을 하도록 하고 있다. lilo.conf 파일을 수정하여 lilo 프로그램을 실행시키면, 수정된 LILO 부트 로더가 부트 섹터에 다시 저장된다.

전원을 켜고 부팅이 시작되면 BIOS에 의해 LILO 부트 로더가 로딩되어 부트 로더 코드가 실행된다. LILO 부트 로더는 운영체제와 부팅에 관련된 옵션 파라미터를 선택할 수 있는 인터페이스를 제공하며,[2] 커널을 로딩하거나 다른 운영체제의 부트 섹터를 로딩하도록 선택할 수 있다. 이어서 로딩된 커널이나 새로운 부트 코드가 실행되는데, Linux 커널이 로딩된 경우에 커널은 각종 하드웨어 장치들에 대한 점검과 초기화를 하고 루트 파일 시스템을 마운트한다.[3]

루트 파일 시스템이 마운트되면, 커널은 파일 시스템 내에 있는 init 프로그램을 찾아 실행시킨다. 이로써 시스템 최초의 프로세스가 생성되고, init 프로세스는 inittab 설정 파일에 따라 지정된 작업을 수행한다.

이러한 작업 가운데 하나로서 sysinit 스크립트를 찾아서 실행시킨다. 이 스크립트는 시스템 기본 서비스를 제공할 수 있도록 하는 셸 명령어들의 집합으로 이루어져 있는데, 모든 디스크에 대해 fsck를 실행시키고, 필요한 커널 모듈들을 로딩하고, 스와핑과 네트워크 초기화 작업을 수행시키고, fstab에 기록되어 있는 디스크 마운팅 작업을 수행시키고, 그 외의 다른 스크립트들도 동작시킨다.

sysinit 스크립트가 작업을 끝내면 시스템 제어 권한을 다시 init 프로세스에게 돌려주고, init 프로세스는 inittab 파일의 initdefault에 표시되어 있는 default runlevel로 진입한다. 통상의 경우에 runlevel은 시스템 콘솔과 tty 사이의 통신을 담당하는 getty를 나타내기 위해 쓰이는데, getty는 login: 프롬프트를 표시하고, 사용자 인증 및 세션을 관리하는 login 프로그램을 실행시킨다. 이로써 부팅 절차는 완료되고 사용자 로그인을 위한 대기 상태로 있게 된다.

3.2. LRP의 부팅

LRP를 위해 사용하는 디스켓의 파일 시스템은 이미 손에 익숙하고 손쉽게 파일 관리를 할 수 있는 MS-DOS 형식을 사용하고 있다.

MS-DOS 플로피 디스켓으로부터 GNU/Linux 운영체제를 로딩하기 위해 SYSLINUX 부트 로더를 사용한다[9].

LRP 디스켓에는 다음과 같은 파일들이 들어 있으며, lrp 확장자를 가진 파일들은 tar로 묶여져 gzip으로 압축되어 있는 형식으로 이루어져 있다. 즉, *.lrp는 *.tar.gz와 완전히 똑같다.

```
$ mount t msdos o rw /dev/fd0 /mnt/fdd
$ cd /mnt/fdd
$ ls
total 1222
        7168    Jan  1  1970 ./
        4096    Jun 23 16:42 ../
       32889    May 29  1999 etc.lrp
        5476    May 29  1999 ldlinux.sys
      362995    May 29  1999 linux
         488    May 29  1999 local.lrp
         628    May 29  1999 log.lrp
       52302    Jun 23 16:46 modules.lrp
      782604    May 29  1999 root.lrp
         179    Jun 23 16:52 syslinux.cfg
         515    Jun 23 16:51 syslinux.dpy
```

LRP에 들어 있는 syslinux.cfg의 내용을 보면 다음과 같다.

```
DISPLAY syslinux.dpy
TIMEOUT 0
DEFAULT linux
APPEND=load_ramdisk=1 initrd=root.lrp
initrd_archive=minix \
ramdisk_size=4096 root=/dev/ram0
boot=/dev/fd0,msdos \
LRP=etc,log,local,modules
```

위와 같은 syslinux.cfg를 가지고 부팅을 시켜보면 다음과 같이 동작한다. 전원을 켜고 BIOS가 실행되면 플로피 드라이브로부터 부트 섹터와 ldlinux.sys를 읽어 들여 SYSLINUX 부트 로더를 로딩하고, 이때 부트 로더는 syslinux.cfg 파일을 참조하여 작업을 수행한다. 다음과 같은 사항들이 정의되어 있다[10][11][12].

- 부팅되는 동안 syslinux.dpy 파일을 화면에 보여 준다. (DISPLAY syslinux.dpy)
- 로딩 대상 커널을 선택하기 위해 제공하는 boot: 프롬프트에서 사용자 입력 대기 시간을 주지 않도록 한다. 즉, 사용자 입력 없이 다음 단계로 넘어간다는 뜻이다. (TIMEOUT 0)
- 디폴트로 로딩할 커널의 이름은 linux이다. (DEFAULT linux)
- RAM 디스크를 사용하고 크기는 4096KB이다. (load_ramdisk=1 ramdisk_size=4096)
- 루트 파일 시스템은 RAM 디스크를 사용하고, 부팅할 때 RAM 디스크로 초기화 로딩할 압축 파일은 root.lrp 이고, RAM 디스크에 있는 루트 파일 시스템이 사용할 파일 시스템의 종류는 minix이다. (root=/dev/ram0 initrd=root.lr initrd_archive=minix)
- LRP 시스템을 구성하기 위해 루트 파일 시스템으로 로딩할 파일들은 etc.lrp, log.lrp, local.lrp, modules.lrp 이다.
- 시스템 부팅 위치는 플로피 디스크 드라이브이며, MS-DOS 파일 시스템으로 이루어져 있다. 이 위치를 바꾸면 하드 디스크로부터 부팅을 시작하도록 할 수 있다. 즉, 플로피 디스켓으로부터 SYSLINUX 부트 로더를 읽어 들인 후에 boot 변수에 지정된 장소에서 커널과 파일 시스템, 및 그 외 필요한 파일들을 로딩하도록 하는 것이다. (boot=/dev/fd0,msdos)

syslinux.cfg 파일에서 GNU/Linux 시스템이 사용할 파일 시스템으로 minix를 선택하였기 때문에 /etc/fstab의 내용에도 이것을 표시하는 내용이 아

---

2 부트 로더가 커널을 로딩하면서 각종 선택 변수들을 전달해 주는데, LILO의 경우에 lilo.conf 파일에 그 내용이 저장된다. 다양한 선택 변수들에 대한 설명이 http://www.linuxdoc.org/H-OWTO/BootPrompt-HOWTO.html 문서에 있다.

3 디스켓으로부터 부팅되는 경우와 같은 상황에서 종종 루트 파일 시스템은 RAM 디스크 장치로 로딩되곤 한다. 이때 파일 시스템은 마치 하드 디스크에 있는 것처럼 RAM을 이용한다. 파일 시스템을 RAM 디스크로 로딩하는 데는 두 가지 이유가 있는데, 첫 번째는 플로피 디스크보다 RAM이 매우 빠르기 때문에 시스템 동작으로 빠르게 할 수 있으며, 두 번째는 커널이 플로피 디스크로부터 압축된 형태의 파일 시스템을 로딩하여 RAM 디스크로 풀어서 로딩할 수 있으므로 디스켓에 보다 많고 큰 크기의 파일들을 담아둘 수 있기 때문이다.

래와 같이 들어가야 한다.

```
# /etc/fstab: static file system info.
#
/dev/ram0 /       minix   rw
proc      /proc   proc    noauto  0  0
```

3.3. LRP의 GNU/Linux 최적화

LRP는 GNU/Linux 시스템에 대한 응급 복구용이 아니라 일반 PC를 라우터 시스템으로 운용하고자 하는 목적이기 때문에 많은 시스템 유틸리티가 필요 없어 /bin과 /sbin 디렉토리에 작은 수의 시스템 유틸리티를 포함하고 있으며, 부가적으로 네트워크 관리용 유틸리티들을 갖고 있다. 만약 GNU/Linux 시스템의 응급 복구용이라면 시스템 관리를 위한 /bin과 /sbin의 많은 시스템 유틸리티들을 포함해야 할 것이다.

따라서 LRP에서는 BusyBox를 이용할 필요가 없는 것으로 보인다.

시스템 로그인을 위해 TinyLogin을 이용한 최적화는 하지 않고 있으나 다음과 같은 최소형의 로그인 환경을 만들어 사용하고 있다.

```
$ ls /bin/login /usr/bin/passwd /sbin/getty
22968     /bin/login
12156     /sbin/getty
15        /usr/bin/passwd -> ../../bin/login
```

라우터 시스템에 필요없는 많은 라이브러리를 제외시켜 라이브러리 최적화를 하고있으나 LRP 라우터의 시스템 유틸리티에 필요한 라이브러리를 추가적으로 사용하고 있어 그 수는 많은 편이다.

LRP의 커널은 Linux 2.0.36pre15를 사용하여 많은 기능들을 모듈로 추가하여 쓸 수 있도록 만들어진 것으로서 커널 내에 내장하는 것보다는 훨씬 크기가 작다. 이에 따라 모듈에 대한 파일이 있는지 여부에 따라 해당 기능에 대한 지원 여부가 결정된다.

```
$ ls -sh linux
360k linux
```

따라서 GNU/Linux 시스템의 최적화는 LRP 사례에서와 같이 목적과 기능에 따라 다양한 방법을 통해 이루어진다는 것을 알 수 있다.

4. 결론

내장형 시스템을 위한 GNU/Linux의 최적화 방법은 시스템 유틸리티 및 라이브러리들을 가능한 한 작은 크기로 만드는 데에 핵심이 있다.

이에 따라 2절에서 시스템 유틸리티 최적화, 단순화된 부팅, 커널 최적화, 라이브러리 최적화, 컴파일러 최적화 등의 방법들을 살펴보았다.

이러한 최적화의 방법들은 일률적으로 똑같이 적용되는 것이 아니라 목표로 하는 시스템의 목적과 기능 특성에 따라 좌우되는 것이므로 보다 효과적으로 최적화하기 위해서는 시스템 특성을 먼저 고려하는 것이 좋다.

그 결과에 따라 무의미한 최적화가 발견될 수 있어 생략할 수도 있고, 모든 최적화 기술을 다 동원해야 할 수도 있고, 상호 대칭적이어서 선택해야 하는 것도 있을 수 있다.

이러한 최적화 기술을 통해 거대하기만 했던 GNU/Linux 시스템이 매우 제한적인 하드웨어 동작 환경 속에서도 운용될 수 있도록 만들 수 있게 된다.

참고문헌

[1] BusyBox, http://busybox.lineo.com/

[2] TinyLogin, http://tinylogin.lineo.com/

[3] Ash, http://www.debian.org/Packages/unstable/shells ash.html

[4] Sysvinit, ftp://ftp.cistron.nl/pub/people/miquels/sysvinit/

[5] Gerard Beekmans, Compiler optimizations, http://www.linuxfromscratch.org/view/

[6] LRP Linux Router Project, http://www.linuxrouter.org/

[7] Tom Fawcett, The Linux Bootdisk HOWTO, http://www.linuxdoc.org/HOWTO/Bootdisk-HO WTO/index.html

[8] 김용운, 김용진, GNU/Linux 시스템의 부팅 과정, 2000. 7. 4.

[9] H. Peter Anvin, SYSLINUX A boot loader for Linux using MS-DOS floppies

[10] Werner Almesberger, LILO User's guide, ftp://metalab.unc.edu/pub/Linux/system/boot/li

[11] Paul Gortmaker, The Linux BootPrompt-HowTo, http://www.linuxdoc.org/HOWTO/BootPrompt-H OWTO.html

[12] Using the RAM disk block device with Linux, /usr/src/linux/Documentation/ramdisk.tx