The Linux GCC HOWTO

Daniel Barlow <dan@detached.demon.co.uk> v1.17, 28 February 1996

이 문서는 GNU C 컴파일러와 개발 라이브러리를 리눅스 상에서 어떻게 셋업하는지에 대해 다루고 있다. 그리고 리눅스 상에서 컴파일, 링킹, 실행, 디버깅을 어떻게 하는지에 대하여 개략적인 지식을 제공한다. 대부분의 내용은 Mitch D'Souza씨의 GCC-FAQ로부터 차용해온 것이며 (많은 부분 교체했다.) 또한 ELF-HOWTO로부터도 차용을 해온 것이다. (이것도 또한 대부분 바뀌게 될 것이다.) 이 문서는 첫번째 공개 버전이다. (버전 번호는 RCS 의 장난일 뿐이다.) 여러분의 의견을 환영한다.

제 1 절 시작하는 말

1.1 ELF vs. a.out

리눅스 개발은 지금 현재에도 끊임없는 변화 과정에 놓여 있다. 간단히 말해서, 리눅스의 측면에서 어떻게 실행해야 하는지 알고 있는 바이너리는 바로 이 2 가지 종류가 있다. 여러분의 시스템이 어떻게 구성되어 있는지에 따라 둘 다 가지고 있을수도 있다.

2 가지를 어떻게 구별하는가? `file`이라고 하는 유틸리티를 사용하면 된다. ELF프로그램에 대해서는 ELF 라고 어쩌구 저쩌구 말할 것이며, a.out 프로그램에 대해서는 `Linux/i386`이라는 단어가 들어가는 말로 얘기해줄 것이다.

둘 간의 차이는 문서 후반부에서 설명될 것이다. ELF 는 새로운 실행화일 형식이며, 일반적으로 더 뛰어나다고 여겨지고 있다.

1.2 책임(Admistrata)

저작권에 관련된 정보는 이 문서의 *마지막*을 참고하라. 또한, 후반부에서 이 글을 읽고, Usenet에 바보같은 질문(존재하지 않는 GCC의 버그를 발표하는 등)을 올리지 말라는 *경고*를 볼 수 있을 것이다.

제 2 절 필요한 것을 어디에서 얻을 수 있는가?

2.1 지금 이 문서

이 문서는 리눅스 하우투 문서 시리즈의 하나이다. 따라서 모든 리눅스 하우투 문서가 저장되어 있는 곳이라면 어디든 있다. 예를 들어서 `<http://sunsite.unc.edu/pub/linux/docs/HOWTO/>`와 같은 곳이 바로 그곳이다. HTML 버전은 `<http://ftp.linux.org.uk/~barlow/howto/gcc-howto.html>`에서 찾을 수 있으며 약간 버전이 높을 지도 모른다.

2.2 다른 문서들

gcc 에 대한 공식적인 문서는 소스 배포 화일에 들어있다. texinfo 화일, `.info`화일의 형식으로 들어있다. 네트워크 속도가 빠르다거나, 시디롬에 가지고 있거나, 또는 인내심이 많다고 생각될 때에는 그것을 untar 한 후에 해당 화일을 `/usr/info`디렉토리에 카피하도록 하자. 만약 없다면 *tsx-11* `<ftp://tsx-11.mit.edu:/pub/linux/packages/GCC/>`에 가서 자료를 찾아보자. 항상 최신 버전이 있는 것은 아닐 것이다.

libc 에 대한 문서는 2 가지가 있다. GNU libc 의 경우에는 info 화일들을 가지고 있는데 stdio 부분을 빼고는 아주 자세히 리눅스 libc 에 대해서 알려주고 있다. *맨페이지* <ftp://sunsite.unc.edu/pub/Linux/docs/>도 구할 수 있는데 시스템 호출(system call 섹션 2), 많은 libc 함수(섹션 3)에 대해 아주 상세히 설명하고 있다.

2.3 GCC

두 가지 답이 있다.

(a) 리눅스 GCC 의 공식적인 배포판은 <ftp://tsx-11.mit.edu:/pub/linux/packages/GCC/>에서 바이너리 형태로 구할 수 있다. 즉 이미 컴파일되어 있는 것을 말한다. 지금 글을 쓰고 있는 이 순간에 최신 버전은 2.7.2 로서 화일명은 `gcc-2.7.2.bin.tar.gz`이다. (편집자주: 이 문서의 작성시기가 96년도임을 상기하라!)

(b) FSF로부터의 최신 소스 버전은 GNU 프로그램 저장소인 *GNU archives* <ftp://prep.ai.mit.edu/pub/gnu/>에서 구할 수 있다. 소스 버전이 항상 공식배포판 바이너리 버전과 같은 것은 아니다. `configure` 스크립트를 이용해서 모든 설정을 할 수 있다. *tsx-11* <ftp://tsx-11.mit.edu:/pub/linux/packages/GCC/>도 마저 살펴보도록 하자. 패치화일이 필요할 지도 모르기 때문이다.

어떤 것이든 컴파일이라는 것을 하기 위해서는 다음이 필요하다.

2.4 C 라이브러리와 헤더 화일들

여기서 여러분에게 필요한 것은 일단 (1)여러분의 시스템이 a.out 인가? (2) 아니면 둘 다 있는 경우에 둘 중에 무엇을 택하고 싶은가? 에 따라 달라진다. 만약 여러분이 libc 4 에서 libc 5 로 업그레이드하려고 한다면 우선은 ELF-HOWTO문서를 봐야할 것이다.

tsx-11 <ftp://tsx-11.mit.edu:/pub/linux/packages/GCC/>에서 구할 수 있다.

`libc-5.2.18.bin.tar.gz`

— ELF 공유 라이브러리 이미지, 정적 라이브러리 그리고 C 라이브러리와 수학 라이브러리를 위한 헤더화일들

`libc-5.2.18.tar.gz`

— 위 라이브러리에 대한 소스. 여러분은 헤더 화일을 구해야 하기 때문에 위에 있는 바이너리배포판도 필요하다. 손수 컴파일을 할 것인지 아니면 그냥 바이너리를 사용할 것인지에 대한 답은 간단하다. 바이너리를 사용하라! 하지만 NYS나 셰도우 패스워드 기능을 원할 때는 손수 컴파일하는 수 밖에 없다.

`libc-4.7.5.bin.tar.gz`

— a.out 공유 라이브러리 이미지, 정적 라이브러리(C 함수, 수학 함수), 위에 있는 libc 5 와 공존할 수 있게끔 디자인되어 있다. 하지만 여러분이 a.out 프로그램을 아직도 갖고 있거나 개발하려고 할 때만 필요하다.

2.5 관련된 도구들 (as, ld, ar, strings 등등)

tsx-11 <ftp://tsx-11.mit.edu:/pub/linux/packages/GCC/>에서 구할 수 있으며, 현재 버전은 `binutils-2.6.x.x.bin.tar.gz`이다.

바이너리 유틸리티들은 오로지 ELF 만 있다는 사실에 유의하자. 현재 라이브러리는 ELF 로만 개발되고 있으며 a.out 라이브러리는 ELF 와 같이 쓸 때만 의미있다고 생각한다. C 라이브러리 개발은 ELF 쪽으로만 진행되고 있으며, a.out으로 해야할 커다란 이유 같은게 없다면 그에 따르는 것이 좋다.

제 3 절 GCC설치와 설정

3.1 GCC 버전

현재 사용 중인 gcc 의 버전을 알고 싶은 경우에는 `gcc -v`라고 셸 프롬프트에서 실행시키면 된다. 또한 이렇게 명령을 내리면 여러분의 시스템이 ELF로 세팅되어 있는지 아니면 a.out 으로 되어 있는지 확실하게 알아낼 수 있다. 필자의 시스템에서는 다음과 같이 나온다.

```
$ gcc -v
Reading specs from /usr/lib/gcc-lib/i486-box-linux/2.7.2/specs
gcc version 2.7.2
```

여기서 알아두어야 할 핵심적인 내용은 다음과 같다.

- `i486`. 이는 여러분이 486 프로세서 용으로 컴파일된 gcc를 사용하고 있다는 말이다. 이 부분은 다를 수 있는데 어떤 사람은 386, 586 에 따라 다를 수도 있다. 하지만 이 3 가지 칩에서 컴파일된 것들은 상관없이 서로 잘 실행된다. 차이점이라고 한다면 486 코드가 어디엔가 더해짐으로써 486 에서는 더욱 더 빨리 실행된다는 정도이다. 386 에서 실행하는데 해가 된다거나 하지는 않는다. 하지만 약간 바이너리가 커질 것이다.
- `box`. 이건 *전혀 중요한 부분이 아니다.* 예를 들어서 box라는 말 대신에 `slackware` 나 `debian` 등의 단어로 교체될 수도 있고 아예 이 부분이 없을 수도 있다. 보통은 `i486-linux` 이런 식일 것이다. 만약 gcc 를 컴파일해서 사용한다면 본인이 따로 i486-box-linux 라고 지정했듯이 gcc를 만들 때 정해줄 수 있다.
- `linux`. 이 단어 대신에 `linuxelf` 라든가 `linuxaout`이라는 단어가 들어갈 수도 있다. 또는 리눅스 커널 버전이 들어가도록 할 수도 있다. 암튼 리눅스용임을 잘 나타내고 있다. 간단히 결론을 말하자면, 이 단어의 뜻은 사용중인 GCC 버전에 따라 다르게 해석된다.
  - 2.7.0 이상의 버전에서는 그냥 `linux`이면 ELF 를 의미하고 a.out은 `linuxaout` 과 같은 이름을 갖는다.
  - 리눅스가 ELF 쪽으로 나아가면서 이름이 linux에서 밀려났다고도 할 수 있다. 따라서 2.7.0 그 이하에서는 `linuxaout` 이라는 말을 찾아볼 수 없을 것이다.
  - `linuxelf`라는 이름은 사라진 말이다. gcc 버전 2.6.3 시절에 ELF 실행화일을 만들기 위해서 지어졌던 이름이다. gcc 2.6.3 은 ELF 실행화일을 만드는데 버그가 있다고 알려져 있다. 업그레이드하기 바란다.
- `2.7.2` 이것은 버전 번호이다.

따라서 종합해보면 필자는 지금 ELF 실행코드를 생성시키는 gcc 2.7.2 를 가지고 있다는 것이다.

3.2 도대체 내 gcc 가 어디에 있는건가?

그냥 아무 생각없이 gcc 를 설치했거나 배포판을 설치할 때 자동으로 설치하게 했다면, 도대체 리눅스 화일 시스템 상에서 어디에 위치하는지 알고 싶을 것이다. 대답은 이렇다.

- `/usr/lib/gcc-lib/`*target*`/`*version*`/` (그리고 모든 하위 디렉토리들)이 컴파일러의 대부분이 위치하는 장소이다. 컴파일을 수행하는 실행화일 그 자체와 gcc 버전에 따른 라이브러리와 헤더화일들이 들어 있다.

- /usr/bin/gcc는 컴파일러 운전사(Compiler Driver)역할을 한다. 커맨드 상에서는 gcc 라고만 명령한다. 만약 여러 버전의 컴파일러를 가지고 있다면 여러 버전과 함께 사용할 수 있다. gcc 가 사용하게 될 디폴트 버전의 컴파일러를 알아내기 위해서는 gcc -v라고 해보면 된다. 다른 버전으로 강제로 컴파일하게 하려면 gcc -V *version* 이런 식으로 사용하면 된다. 예를 들어서...

```
# gcc -v
Reading specs from /usr/lib/gcc-lib/i486-box-linux/2.7.2/specs
gcc version 2.7.2
# gcc -V 2.6.3 -v
Reading specs from /usr/lib/gcc-lib/i486-box-linux/2.6.3/specs
gcc driver version 2.7.2 executing gcc version 2.6.3
```

- /usr/*target*/(bin|lib|include)/. 여러분이 여러 개의 목표 형식을 가지고 있다면 (일단 ELF인가 a.out 인가 또는 여러 형태의 크로스 컴파일러 등) 디폴트 목표 형식용이 아닌 라이브러리, 바이너리 유틸리티 (as, ld 등...), 헤더 화일들도 찾아볼 수 있을 것이다. 오로지 한 종류의 gcc 를 가지고 있다 하더라도 매우 많은 것들이 그 디렉토리에 깔려있음을 확인할 수 있다. 그렇지 않다면 아마도 /usr/(bin|lib|include)에 있을 것이다.

- /lib/,/usr/lib 그리고 여타 라이브러리 디렉토리들이 기본 시스템을 위한 라이브러리 디렉토리이다. 여러분은 또한 상당히 많은 프로그램에 대하여 /lib/cpp를 가지고 있어야 한다. (X 가 실제로 많이 사용하고 있다.) /usr/lib/gcc-lib/*target*/*version*/에 있는 cpp 를 카피해놓던가? 아니면 심볼릭 링크를 해준다.

3.3 헤더 화일들은 어디에 있는가?

여러분이 손수 /usr/local/include에 설치한 것들 빼고 리눅스에는 3 가지 중요 헤더 디렉토리가 있다.

- 대부분의/usr/include/와 그 하부 디렉토리들은 H J Lu 의 libc 바이너리 배포판에 의해서 제공된다. 여기서 본인은 "대부분"이라는 표현을 썼는데, 그 이유는 다른 소스 (예를 들어 curses, dbm 라이브러리)에서 온 헤더화일들도 있기 때문이다. 특히나 최근 libc 배포판을 가져오면 그러한 헤더화일들은 없다. (예전에는 같이 달려서 왔지만)

- /usr/include/linux와 /usr/include/asm(<linux/*.h>화일과 <asm/*.h>에 의해 참조되는 헤더 화일들이 있는 장소)는 각각 커널 소스에서 linux/include/linux와 linux/include/asm을 가리키는 심볼릭 링크여야 한다. *뭔가 조금이라도 큰 작업*을 하려고 한다면 분명히 설치해야 한다. 커널 컴파일을 하기 위해서만 있는 것은 아니다.

  또한 커널 소스를 풀고 나서 make config라는 작업을 해주어야 할 것이다. 많은 화일들이 그 과정을 통해서 생겨나는 <linux/autoconf.h>라는 화일에 의존하기 때문이다. 그리고 어떤 버전의 커널에서는 asm 이라고 하는 것이 심볼릭 링크일 뿐, make config 할 때만 생기는 경우가 있다.

  asm 은 보통 asm-i386으로 링크되어 있다. 그전에는 오로지 인텔 머신용 헤더화일만이 있었기 때문에 asm 만이 있었지만 이제는 리눅스가 명실상부하게 멀티플랫폼 운영체제로 나아가고 있기 때문이다. asm-i386말고도 asm-alpha, asm-generic, asm-m68k, asm-mips, asm-ppc, asm-sparc등의 헤더 화일 디렉토리가 있는 것을 발견할 수 있다.

  따라서 /usr/src/linux라고 하는 디렉토리에 이미 소스를 풀어놓았다면...

```
$ cd /usr/src/linux
$ su
# make config
[answer the questions.  Unless you're going to go on and build the kernel
it doesn't matter _too_ much what you say]
# cd /usr/include
# ln -s ../src/linux/include/linux .
# ln -s ../src/linux/include/asm .
```

- <float.h>, <limits.h>, <varargs.h>, <stdarg.h> 그리고 <stddef.h> 등의 화일들은 컴파일러 버전마다 다를 것이다. 그리고 그들은 `/usr/lib/gcc-lib/i486-box-linux/2.7.2/include/`에 위치하고 있다.

3.4 크로스 컴파일러(Cross Compiler) 만들기

3.4.1 목표 플랫폼으로서의 리눅스

여러분이 지금 gcc 소스 코드를 가지고 있다고 생각하겠다. 보통은 GCC 에 대한 INSTALL 화일에서 지시하는 대로 따르면 된다. `configure --target=i486-linux --host=XXX` 이런 식으로 해주는데, `XXX`는 플랫폼을 말한다. 다음에는 `make` 과정을 거치면 된다. 리눅스 헤더화일, 커널 헤더화일이 필요하며, 크로스 컴파일러와 크로스 링커를 만들기 위해서도 필요하다. <ftp://tsx-11.mit.edu/pub/linux/packages/GCC/>에서 구할 수 있다.

3.4.2 소스 플랫폼으로서의 리눅스, 목표 플랫폼으로서의 MSDOS

흠. 소스를 리눅스에서 작성한 뒤에 도스에서 돌아가는 프로그램으로 컴파일하기 위해서는 `emx` 패키지나 `go extender`라는 것을 필요로 한다. <ftp://sunsite.unc.edu/pub/Linux/devel/msdos>에 가서 관련 화일을 찾아보기 바란다.

본인으로서는 테스트해본 적이 없으며, 쓸만하다고 단언하기는 힘들다.

제 4 절 포팅과 컴파일링

4.1 자동적으로 정의되는 심볼들

여러분은 여러분이 갖고 있는 버전의 gcc가 `-v` 옵션을 붙임으로써 어떠한 심볼을 자동적으로 정의하는지 알아낼 수 있다. 예를 들어 본인의 것은 다음과 같다.

```
$ echo 'main(){printf("hello world\n");}' | gcc -E -v -
Reading specs from /usr/lib/gcc-lib/i486-box-linux/2.7.2/specs
gcc version 2.7.2
 /usr/lib/gcc-lib/i486-box-linux/2.7.2/cpp -lang-c -v -undef
-D__GNUC__=2 -D__GNUC_MINOR__=7 -D__ELF__ -Dunix -Di386 -Dlinux
-D__ELF__ -D__unix__ -D__i386__ -D__linux__ -D__unix -D__i386
-D__linux -Asystem(unix) -Asystem(posix) -Acpu(i386)
-Amachine(i386) -D__i486__ -
```

만약 여러분의 코드가 리눅스에만 관계되는 코드라면, 다음과 같이 해주는 것이 좋다.

```
#ifdef __linux__
/* ... funky stuff ... */
#endif /* linux */
```

`__linux__`라는 이름을 사용하라. `linux`가 *아니다*. 후자가 정의되어 있기는 하지만 POSIX 규격에는 맞지 않기 때문이다.

4.2 컴파일러 부르기

컴파일러 스위치들에 대한 문서는 gcc info 페이지를 보면 된다. (여러분이 Emacs를 사용하고 있다면 `C-h i`그리고 나서 gcc 옵션을 선택하라) 여러분이 갖고 있는 배포판을 만든 사람이 gcc info 페이지를 넣어지 않았을 수도 있고, 또는 옛 버전의 것이 들어가 있을 수도 있다. 가장 좋은 방법은 `<ftp://prep.ai.mit.edu/pub/gnu>`나 또는 미러 사이트로 가서 gcc 소스 코드를 받아오는 것이다. 그리고 그 소스 안에서 카피해온다.

gcc 에 대한 맨페이지(`gcc.1`)는 일반적으로 시대에 뒤떨어져 있다고 말할 수 있다. 맨페이지를 보려고 하면 그러한 경고 문구를 볼 수 있다.

4.2.1 컴파일러 플래그(flag)

gcc를 사용할 때, `-O`n(여기서 n은 작은 양의 정수들, 생략해도 된다)을 커맨드 라인 옵션으로 넣어주면 출력 코드가 최적화된다. 여기서 사용되는 n 값 중에서 실제 의미를 갖는 값들은 gcc의 버전에 따라 다른데, 일반적으로 0 (최적화하지 않음)부터 시작해서 2(상당히 많이 최적화), 3(아주아주 많이 최적화)까지 쓰인다.

내부적으로 gcc는 이 옵션을 `-f`와 `-m` 이라는 옵션들로 바꾸어서 처리하게 된다. `-O`의 특정 레벨이 어떤 의미를 갖는지에 대해서는 gcc 실행시에 `-v`와 `-Q`(문서화되지 않았음)플래그를 붙여줌으로써 확인할 수 있다. 예를 들어 `-O2`는 다음과 같이 나타난다. (사람들마다 서로 다를 수 있다)

```
enabled: -fdefer-pop -fcse-follow-jumps -fcse-skip-blocks
-fexpensive-optimizations
         -fthread-jumps -fpeephole -fforce-mem -ffunction-cse -finline
         -fcaller-saves -fpcc-struct-return -frerun-cse-after-loop
         -fcommon -fgnu-linker -m80387 -mhard-float -mno-soft-float
         -mno-386 -m486 -mieee-fp -mfp-ret-in-387
```

여러분의 컴파일러가 지원하고 있는 최적화 레벨보다 큰 숫자를 사용한다면 (예를 들어 `-O6`), *그 컴파일러가 지원하는 최적의 레벨*로 최적화시켜준다. 이런 식으로 컴파일되도록 세팅되어 있는 코드를 배포하는 것은 별로 좋은 생각은 아닌 것 같다. 더 많은 최적화 레벨들이 차후 gcc 버전에 생긴다면, 잘못하면 여러분의 소스 코드가 엉뚱하게 컴파일되는 수도 있다.

만약 여러분이 지금 `-O3`이 최고 레벨이라는 가정하에서 `-O6`를 사용했다고 치자. 하지만 다음 버전(예를 들어서 2.7.3.?)에서 `-O8`까지 지원하게 된다면 `-O6`는 전혀 엉뚱한 의미를 가질 수도 있다.

gcc 버전 2.7.0 부터 2.7.2 까지의 사용자들은 `-O2` 최적화 플래그에 버그가 있다는 사실을 잘 알아두기 바란다. Strength Reduction이라고 하는 것이 제대로 작동하지 않는다. 이 문제를 해결할 수 있는 패치가 있고 다시 gcc 를 컴파일해야 할것이다. 또는 언제나 `-fno-strength-reduce` 라는 옵션을 주고 컴파일하기 바란다.

프로세서별 옵션 `-O` 옵션을 주어도 자동적으로 작동하지 않는 `-m` 플래그들이 있다. 하지만 이들은 상당히 유용하다. 중요한 것으로는 `-m386`과 `-m486`이 있다. 이 플래그들은 gcc더러 각각 386, 486중 어떤 것에 더 맞춰서 컴파일할 것인지를 알려주는 것이다. -m486으로 컴파일하였다고 하더라도 386 에서 실행되는데는 지장없다. 그러니 걱정할 필요없다. 486 코드가 조금 더 크지만 386 에서 느려지거나 하지는 않는다.

아직까지는 `-mpentium`이나 `-m586`과 같은 것은 없다. 리누스(Linus)는 486 코드옵티마이즈된 코드를 얻으면서도 펜티엄이 사용하지 않는 정렬방식과의 커다란 차이점이 없는 코드를 얻기 위해서는, `-m486 -malign-loops=2 -malign-jumps=2 -malign-functions=2`를 사용할 것을 제안하고 있다. Michael Meissner(Cygnus에 있는)는 다음과 같이 말하고 있다.

> 내 육감으로는 `-mno-strength-reduce`를 같이 쓰면 또한 x86 에서 더 빠른 코드를 얻어낼 수 있다는 것이다. (주의! 나는 지금 strength reduction 버그에 대해서 말하고 있는 것이 아니다.

> 그것은 전혀 다른 문제이다) 왜냐하면 x86은 다소 레지스터 숫자가 적기 때문이다. (그리고 다른 레지스터에 대하여 레지스터들을 그룹으로 묶어서 spill 레지스터 속으로 처리하는 GCC 의 처리방식은 전혀 도움이 되질 않는다) StrengthReduction은 전형적으로 곱셈을 덧셈으로 교체하기 위하여 다른 레지스터들을 사용하게 된다. `-fcaller-saves` 또한 이런 문제점이 있지 않나 생각하고 있다.
>
> 또 다른 예감은 이렇다. `-fomit-frame-pointer`는 도움이 될 수도 있고, 그렇지 않을 수도 있다는 것이다. 한 편으로는 또 다른 레지스터가 할당가능하다는 것을 의미할 수도 있고, 다른 한 편으로는 x86 이 연산지시(instruction)에 대하여 인코딩하는 방식으로서, 스택 상대적 주소가 프레임 상대적 주소보다도 더 많은 공간을 차지한다는 것을 의미하기도 한다. 이렇게 되면 프로그램에 사용될 수 있는 Icache이 약간 줄어든다. 또한 `-fomit-frame-pointer`는 컴파일러가 계속적으로 호출 후에도 스택 포인터를 조정해야 한다는 것을 뜻한다. 따라서 프레임을 갖는 경우, 몇 번의 호출만으로도 스택이 가득 차게 된다.

마지막 말은 리누스 또한 언급하고 있다.

> 만약 여러분이 최적화된 효율을 원한다면, 나를 믿지 말라. 실제로 테스트를 해봐야 한다. gcc 컴파일러의 옵션은 정말로 많다. 그리고 몇 개의 특정 조합이 가장 좋은 최적화를 이뤄줄 것이다.

4.2.2 `Internal compiler error: cc1 got fatal signal 11`

시그널 11번은 SIGSEGV, 즉 세그먼테이션 위반에 대한 시그널이다. 일반적으로 프로그램이 포인터를 잘못 썼다는 말이거나 자기가 소유하고 있지 않은 메모리에다 쓰기 작업을 하려고 할 때 발생한다. 그래서 이는 gcc의 버그일 수도 있다.

하지만 gcc는 대부분의 작업에서 매우 안정적이고 테스팅을 많이 거친 소프트웨어라는 사실을 기억하라. gcc는 또한 복잡한 자료 구조와 포인터를 엄청나게 많이 사용하고 있다. 간단히 말하자면 현재까지 소프트웨어 중에서 가장 뛰어난 램 테스팅 프로그램(RAM Tester)이라고 말할 수도 있다. *만약 매번 컴파일할 때마다 멈추는 위치가 다르다면* 이는 거의 대부분 여러분 하드웨어의 문제라고 봐도 된다. (CPU, 메모리, 마더보드나 캐쉬) 여러분의 컴퓨터가 파워 온 체킹을 거쳐서 잘 부팅되었고 그리고 윈도우즈 같은 것도 잘 돌아간다고 해서 **그것을 gcc의 버그로 돌리지는 말라**. 이러한 사실은 무의미하다. 그리고 커널 컴파일하면서 `make zImage`에서 꼭 멈춘다고 해서 gcc의 버그라고 말할 수는 없다. `make zImage`는 무려 200개 이상의 화일을 컴파일하고 있다. 그것보다는 좀 *작은 경우*를 찾아보도록 하자.

만약 계속적으로 버그가 똑같이 나타나고 자그마한 프로그램 컴파일에서도 그러하다면, FSF에다가 버그 리포트를 해도 되고, 또는 linux-gcc 메일링 리스트에 글을 올려도 된다. 그러기 위해서는 우선 gcc 문서를 읽어보고 어떤 절차가 필요한지 숙지한 다음 하기 바란다.

4.3 포팅(Portability)

요즘은 만약 그 소프트웨어가 리눅스로 포팅될 수 없다면 그 소프트웨어는 가치가 없는 프로그램이라고 말한다. :-)

진지하게 말하자면, 일반적으로 리눅스의 100% POSIX 호환성을 이루기 위해서는 아주 약간의 수정작업만이 필요하다. 또한 단지 make 라고만 하면 실행화일이 만들어질 수 있도록 하기 위하여 코드의 원저자에게 수정 코드를 보내는 것이야말로 가치있는 일이다.

4.3.1 BSDisms (`bsd_ioctl`, `daemon` 그리고 `<sgtty.h>`)

여러분은 여러분의 프로그램을 `-I/usr/include/bsd`를 넣어서 컴파일한 후, `-lbsd` 옵션을 넣고 링크할 수도 있다. (즉 Makefile 안에서 `-I/usr/include/bsd`를 `CFLAGS 변수`에 넣고, `-lbsd`를 `LDFLAGS`에 넣음으로

써) 이젠 BSD 타입의 시그널 행동을 얻어내기 위해서 `-D__USE_BSD_SIGNAL`를 *덧붙일 필요가 없다*. 왜냐하면 `-I/usr/include/bsd`라고 해주고 `<signal.h>`를 소스 안에서 포함하면 모든 일이 제대로 이루어진다.

4.3.2 없어진 시그널들 (SIGBUS, SIGEMT, SIGIOT, SIGTRAP, SIGSYS 등)

리눅스는 POSIX를 준수하고 있다. 이러한 시그널들은 POSIX 정의 시그널들이 아니다. 이는 ISO/IEC 9945-1:1990 (IEEE Std 1003.1-1990), paragraph B.3.3.1.1 에서 다음과 같이 말하고 있는 바이다.

> SIGBUS, SIGEMT, SIGIOT, SIGTRAP, 그리고 SIGSYS와 같은 시그널들은 POSIX.1으로부터 제외되었다. 왜냐하면 그들의 행동은 함축적이고 어떻게 부르느냐에 따라 다르기 때문에 적절하게 범주화시킬 수가 없다. 이러한 시그널들을 없애버리는 것이 규약 준수일 수도 있지만, 왜 그 시그널들을 제외해버렸는지에 대해서 문서화해야 한다. 그리고 그 시그널들을 어떻게 처리할 것인가에 대해서는 아무런 강제 규정도 없다.

이 문제를 해결할 수 있는 가장 간단한 방법은 이러한 시그널들을 모두 `SIGUNUSED`로 재정의하는 것이다. *바*른방법은 물론 이러한 시그널을 처리하는 부분을 `#ifdef` 문장을 써서 처리하도록 하는 것이다.

```
#ifdef SIGSYS
/* ... POSIX 규정이 아닌 SIGSYS 코드가 여기에 온다 .... */
#endif
```

4.3.3 K & R 코드

GCC는 ANSI 컴파일러이다. 하지만 아주 많은 코드들이 ANSI가 아니다. 이럴 때는 컴파일러 플래그에 `-traditional` 이라고만 붙여주면 된다고 할 수 있다. 물론 괴롭게 수작업을 해줘야 하는 부분도 많이 있다. gcc info 페이지를 살펴보기 바란다.

`-traditional`라는 옵션은 gcc 가 이용하려고 하는 C 언어 방식을 바꾸는 것 말고도 다른 효과를 지니고 있다. 예를 들어 그 옵션은 `-fwritable-strings`을 작동시키는데, 문자열 상수를 데이타 영역으로 보내는 역할을 한다. (텍스트 영역, 즉 그들이 쓸 수 없는 영역을 말한다) 이런 경우 프로그램의 메모리 사용흔적(footprint)이 증가하게 된다.

4.3.4 전처리기 심볼이 코드의 프로토타입과 충돌할 때

많이 발생하는 문제들 중에 하나가 바로 몇몇 함수들이 이미 리눅스 헤더화일들에 매크로로 정의되어 있고 전처리기가 코드 내에서 유사한 프로토타입에 대하여 처리 거부를 하는 경우이다. 보통 `atoi()`와 `atol()`인 경우가 많다.

4.3.5 `sprintf()`

`sprintf(string, fmt, ...)`이 많은 유닉스 시스템에서는 `문자열`에 대한 포인터를 반환하는 반면에 ANSI를 따르는 리눅스는 문자열에 삽입된 문자의 갯수를 반환한다. 이는 특히나 SunOS와 같은 것으로부터 포팅하는 경우에 더욱 주의해야 한다.

4.3.6 `FD_*` 같은 것들? `fcntl`과 그 비슷한 녀석들. 도대체 정의부분이 어디에 있는가?

`<sys/time.h>`에 있다. 만약 `fcntl`을 이용하고자 한다면 실제 프로토타입을 위하여 `<unistd.h>` 또한 포함시키고 싶을 것이다.

일반적으로 말하자면 어떤 함수에 대한 맨페이지를 보면 SYNOPSYS 부분에서 어떤 헤더화일을 `#include` 해야하는지 자세히 나타내주고 있으니 그것을 참고하기 바란다.

4.3.7 select()에서 타임아웃이 걸리고 프로그램이 계속 기다리기만 한다.

예전에는 select()에 대한 타임아웃 파라미터가 읽기전용으로만 사용되었다. 그리고 그 때에도 맨페이지에는 다음과 같은 경고가 있었다.

> select()는 아마도 적절한 곳에 있는 시간값을 변경함으로써 만약에 그러한 일이 발생한다면 원래의 타임아웃부터 남은 시간을 반환해야 할 것이다. 하지만 이 기능은 차기 버전에서나 구현될 것이다. 따라서 타임 아웃 포인터가 select() 호출에 의하여 수정되지 않을 것이라고 생각하는 것은 바람직하지 못하다.

바로 그 날이 왔다! 최소한 그것이 이루어지고 있다. select()호출로부터 돌아올 때, 타임아웃 인수는 데이터가 도착하지 않는다면 기다리려고 했던 잔류 시간으로 세팅된다. 만약 아무 데이터도 도착하지 않았었다면 이 값은 0(zero)이 되었을 것이다. 그리고 같은 타임아웃 구조체를 가지고 호출을 하게 되면 호출 즉시 되돌아올 것이다.

이 문제를 해결하기 위해서는 타임아웃 값을 매번 select()를 호출할 때마다 관련 구조체에 적어주어야 한다. 다음과 같은 코드가 있다면,

```
struct timeval timeout;
timeout.tv_sec = 1; timeout.tv_usec = 0;
while (some_condition)
      select(n,readfds,writefds,exceptfds,&timeout);
```

아래와 같이 바꾸도록 하라.

```
struct timeval timeout;
while (some_condition) {
      timeout.tv_sec = 1; timeout.tv_usec = 0;
      select(n,readfds,writefds,exceptfds,&timeout);
}
```

모자익(Mosaic)의 몇몇 버전이 한 때 이러한 문제로 떠들썩했었다. 회전하는 지구 애니매이션의 속도가 네트워크를 통해 들어오는 자료의 속도에 반비례하는 일이 벌어진 것이다!

4.3.8 시스템 호출이 인터럽트될 때

증상: 프로그램이 Ctrl+Z로 서스펜드되고 다시 시작되어 버린다. 또는 다른 때에는 Ctrl+C와 같은 시그널을 발생시키고 자식 프로세스들을 죽인다 등등... "interrupted system calls" 또는 "write: unknown error" 또는 그런 것 비슷한 에러를 낸다.

문제점: POSIX 시스템은 다른 구식 유닉스 체제에서보다 약간 더 많이 시그널에 대해서 체킹을 행한다. 리눅스는 시그널 핸들러들(signal handler)을 실행시킬 것이다.

- 타이머가 째깍댈 때마다 비동기적으로.
- 모든 시스템 호출 반환시에.
- 그리고 다음과 같은 시스템 호출 동안에도 그러하다: select(), pause(), connect(), accept(), 터미널 상에서의 read(), 소켓, 파이프나 라인 프린터, FIFO에 대한 open(), PTY나 시리얼 라인, 터미널에 대한 ioctl(), F_SETLKW 명령을 내리는 fcntl(), wait4(), syslog(), 모든 TCP 또는 NFS 작업

다른 운영체제의 경우에는 다음과 같은 시스템 호출에 대해서도 체크할 것이다. 위에서 말한 것 이외에도 다음과 같은 시스템 호출들: creat(), close(), getmsg(), putmsg(), msgrcv(), msgsnd(), recv(), send(), wait(), waitpid(), wait3(), tcdrain(), sigpause(), semop()

만약 시그널(프로그램에서 핸들러를 인스톨한 경우)이 시스템 호출 중에 발생한다면, 그에 대한 핸들러가 호출된다. 그리고 핸들러가 반환되면 (시스템 호출로), 시스템 호출은 중간에 가로채기를 당했는지 살펴보고 즉시 -1 값을 가지고 반환된다. 그리고errno 를 EINTR 로 세팅한다. 프로그램은 그러한 일이 있을 것이라고 예상하지 못하고 죽는 것이다.

여러분은 다음 2 가지 해결책 중에 하나를 고르면 된다.

(1) 여러분이 설치한 모든 시그널 핸들러에 대하여 SA_RESTART를 sigaction 플래그에 첨가한다. 다음과 같은 것이 있다면,

```
    signal (sig_nr, my_signal_handler);
```

를 다음과 같이 바꾼다.

```
    signal (sig_nr, my_signal_handler);
    { struct sigaction sa;
      sigaction (sig_nr, (struct sigaction *)0, &sa);
  #ifdef SA_RESTART
      sa.sa_flags |= SA_RESTART;
  #endif
  #ifdef SA_INTERRUPT
      sa.sa_flags &= ~ SA_INTERRUPT;
  #endif
      sigaction (sig_nr, &sa, (struct sigaction *)0);
    }
```

이 방법이 대부분의 시스템 호출에 적용되기는 하지만, read(), write(), ioctl(), select(), pause(), connect()에 대해서는 여러분 스스로 EINTR를 체크해주어야 한다. 다음을 살펴보자.

(2) 여러분이 직접 명시적으로 EINTR을 체크해준다.

read()를 사용하는 코드가 원래 이렇게 되어 있다고 치자.

```
int result;
while (len > 0) {
  result = read(fd,buffer,len);
  if (result < 0) break;
  buffer += result; len -= result;
}
```

이 코드를 다음과 같이 바꾸어주면 된다.

```
int result;
while (len > 0) {
  result = read(fd,buffer,len);
  if (result < 0) { if (errno != EINTR) break; }
  else { buffer += result; len -= result; }
}
```

이번에 이런 코드가 있다면,

```
int result;
result = ioctl(fd,cmd,addr);
```

그것은 또한 다음과 같이 바뀌어야 한다.

```
int result;
do { result = ioctl(fd,cmd,addr); }
while ((result == -1) && (errno == EINTR));
```

BSD 유닉스의 몇몇 버전에서는 시스템 호출을 재개하는 것이 기본 행동으로 되어 있는 경우도 있으므로 주의하자. 시스템 호출이 가로채기를 허용하기 위해서는 `SV_INTERRUPT` 또는 `SA_INTERRUPT` 플래그를 사용하도록 하자.

4.3.9 쓰기 가능 문자열 (프로그램이 랜덤하게 세그폴트를 낸다)

GCC는 gcc를 사용하는 사람들이 문자열 상수에 대하여 정확히 상수로서 계속 사용할 것이라고 낙관하고 있는 듯 하다. 따라서 그 문자열 상수를 프로그램의 텍스트 영역에 집어넣는다. 이렇게 함으로써 스왑 영역을 사용하는 것이 아니라 프로그램의 디스크 이미지로부터 페이지 인 & 아웃을 행할 수 있도록 해준다. 그러므로 문자열 상수에 대하여 다시 쓰기 작업을 하게 되면 세그멘테이션 폴트를 일으키게 되는 것이다.

예를 들어서 문자열 상수를 인수로 하여 `mktemp()`를 호출하는 옛날 프로그램들에서는 문제가 발생할 것이다. `mktemp()`는 주어진 인수에 다시 쓰려고 하기 때문이다.

이 문제를 고치기 위해서는 (a) `-fwritable-strings` 이라는 옵션을 주어서 컴파일한다. 이렇게 해주면 gcc는 문자열 상수를 데이타 영역에 넣게 된다. 또는 (b) 문제가 되는 부분을 수정해서 상수가 아니라 변수로 주어지게 만들고 호출 전에 strcpy 를 사용하여 데이터를 그곳으로 카피해준다.

4.3.10 왜 `execl()`호출이 실패하는가?

원인은 간단하다. 제대로 호출을 하지 않았기 때문이다. `execl`에 대한 첫번째 인수는 실행하고자 하는 프로그램이다. 그리고 두번째부터는 호출하는 프로그램에 전달할 `argv`배열이다. 기억하라! `argv[0]`는 전통적으로 아무런 인수 없이 실행되더라도 세팅이 된다는 사실을! 따라서 다음과 같이 코드를 써야한다.

```
execl("/bin/ls","ls",NULL);
```

절대로 다음과 같이 쓰면 안된다.

```
execl("/bin/ls", NULL);
```

아무런 전달인수 없이 실행시키는 경우에도 실행형식은 자신의 동적 라이브러리 의존성을 나타낼 수 있는 방식으로 구문을 맞춰준 형태라야 한다. 최소한도 a.out의 경우는 그러하다. ELF는 좀 다른 방식으로 작동한다.

(만약 이러한 라이브러리 정보를 원한다면 아주 간단한 인터페이스가 있다. 동적 로딩Dynamic Loading에 대한 섹션을 보거나 `ldd`에 대한 맨페이지를 참고하라)

제 5 절 디버깅과 Profiling

5.1 예방적인 관리(lint)

문제가 발생하고 나서 해결하는 것보다는 문제를 미연에 방지하는 것이 중요하지 않을까? 리눅스에 널리 쓰이는 lint는 없다. 아마도 대부분의 사람들이 gcc가 내놓는 자세한 경고 메세지에 만족하고 있기 때문인 것 같다. 아마도 가장 유용하쓰이는 것은 `-Wall` 스위치일 것이다. 이것이 의미하는 바는 "Warnings, all"로서 모든 경고 메세지를 발생시키라는 말이다. 또한 아주 자세하게 나온다.

Public Domain lint는 `<ftp://larch.lcs.mit.edu/pub/Larch/lclint>`에서 얻을 수 있다. 하지만 얼마나 괜찮은지 본인은 모른다.

5.2 디버깅

5.2.1 어떻게 하면 프로그램의 디버깅 정보를 알아낼 수 있는가?

그러기 위해서는 -g 옵션을 주고 컴파일/링크해야 한다. 그리고 -fomit-frame-pointer 스위치는 빼주어야 한다. 사실 모든 부분을 다시 컴파일할 필요는 없고, 여러분이 관심 갖고 있는 부분만을 그렇게 해주면 된다.

a.out에 있어서 공유라이브러리가 만약 -fomit-frame-pointer 스위치를 가지고 컴파일되었다면 gdb를 사용할 수 없을 것이다. -g 옵션을 주는 이유는 바로 정적 링크를 행하라는 말을 함축하게 된다.

만약 링커가 libg.a를 찾을 수 없다고 하면서 실패하게 된다면, 여러분이 /usr/lib/libg.a을 갖고 있지 않기 때문일 것이다. 그 화일은 특별한 라이브러리로서 디버깅 가능 C 라이브러리이다. libc 패키지에 포함되어 있거나 또는 libc 소스 코드를 받아서 컴파일하면 생긴다. 실제로 *그렇게 필요한 것은 아니고* 대충 /usr/lib/libc.a를 /usr/lib/libg.a로 링크시켜버려도 대부분 상관없을 것이다.

디버깅 정보를 어떻게 하면 다시 꺼낼 수 있는가? 아주 많은 GNU 소프트웨어들은 -g 옵션을 가지고 컴파일되어 있으므로 화일 크기가 매우 크다. (종종 정적 링크되어 있음) 그렇게 괜찮은 생각인 것 같지는 않다.

만약 프로그램이 autoconf에 의해 만들어진 configure를 가지고 있다면, 보통의 경우 Makefile을 건드림으로써 디버깅 정보를 넣지 않게 할 수 있다. 물론 ELF를 사용하고 있다면, 프로그램은 -g 세팅과는 상관없이 동적 링크되며, 그냥 쉽게 strip(디버깅 정보를 실행화일에서 빼버리는 행위)시킬 수 있다.

5.2.2 관련 소프트웨어

대부분의 사람들은 **gdb**를 사용하고 있다. gdb는 *GNU archive sites* <ftp://prep.ai.mit.edu/pub/gnu>에서 소스의 형태로, 아니면 *tsx-11* <ftp://tsx-11.mit.edu/pub/linux/packages/GCC>이나 선사이트에서 바이너리의 형태로 구할 수 있다. **xxgdb**는 gdb에 기초한 X 윈도우 디버거이다. 즉, 우선적으로 gdb를 이미 설치했어야 한다는 뜻이다. 그 소스는 <ftp://ftp.x.org/contrib/xxgdb-1.08.tar.gz>에서 찾을 수 있다.

또한 **UPS** 디버거가 Rick Sladkey씨에 의해 포팅되었다. X 윈도우에서도 잘 돌아간다. 하지만 xxgdb와 같이 텍스트 디버거인 gdb같은 것에 의존하는 형태는 아니다. 아주 훌륭한 기능들을 많이 가지고 있다. 따라서 여러분이 디버깅에 많은 시간을 할애하고 있다면, 우선적으로 UPS 디버거를 권한다. 리눅스용으로 컴파일된 바이너리나 소스 패치화일은 <ftp://sunsite.unc.edu/pub/Linux/devel/debuggers/>에서 구할 수 있고 오리지널 소스는 <ftp://ftp.x.org/contrib/ups-2.45.2.tar.Z>에서 찾을 수 있다.

디버깅에 쓰이는 또 다른 툴 하나를 들자면 **strace**를 들 수 있다. strace는 프로그램이 만들어내는 시스템 호출을 화면에 표시해준다. 이것 말고도 다방면으로 사용가능한데, 예를 들어 어떠한 패스명이 소스코드를 갖고 있지 않은 바이너리 화일 안에 컴파일되어 들어가있는지, 분명히 바이너리 안에 들어있는 조건들을 발견하고자 할 때, 일반적으로 일반적으로 어떻게 작동하고 있는지를 알아내고자 할 때 사용한다. 최신 strace 버전(현재 3.0.8)은 <ftp://ftp.std.com/pub/jrs/>에서 구할 수 있다.

5.2.3 백그라운드 (데몬) 프로그램

데몬 프로그램들은 전형적으로 fork()를 먼저 하고 나서, 부모 프로세스를 종료시켜 버린다. 이는 디버깅 세션에 대하여 공격적인 요소임이 분명하다.

이럴 때 가장 간단한 방법은 fork에 대하여 정지점(breakpoint)을 지정해주는 것이고 프로그램이 멈추면 다시금 그것을 0 으로 만들어주는 것이다.

```
(gdb) list
1       #include <stdio.h>
```

```
2
3       main()
4       {
5         if(fork()==0) printf("child\n");
6         else printf("parent\n");
7       }
(gdb) break fork
Breakpoint 1 at 0x80003b8
(gdb) run
Starting program: /home/dan/src/hello/./fork
Breakpoint 1 at 0x400177c4

Breakpoint 1, 0x400177c4 in fork ()
(gdb) return 0
Make selected stack frame return now? (y or n) y
#0  0x80004a8 in main ()
    at fork.c:5
5         if(fork()==0) printf("child\n");
(gdb) next
Single stepping until exit from function fork,
which has no line number information.
child
7       }
```

5.2.4 코어 화일(Core file)

보통 리눅스 부팅시에 코어 화일을 만들지 않도록 세팅되어 있다. 하지만 코어화일 생성을 가능케 하려고 한다면 그것을 다시 가능케 하는 셸의 내장 명령을 사용한다.

셸 호환 셸(예. tcsh)을 쓰고 있다면 다음과 같이 명령을 내린다.

```
% limit core unlimited
```

만약 본셸류(sh, bash, zsh, pdksh)를 사용하고 있다면,

```
$ ulimit -c unlimited
```

만약 코어 화일의 이름에 대하여 융통성을 가지고 싶다면, 커널 소스를 약간만 변경해주면 된다. 자, `fs/binfmt_aout.c`와 `fs/binfmt_elf.c`와 같은 화일을 찾아보자.

```
        memcpy(corefile,"core.",5);
#if 0
        memcpy(corefile+5,current->comm,sizeof(current->comm));
#else
        corefile[4] = '\0';
#endif
```

grep 같은 것을 가지고 이런 부분을 모두 찾은 후에 `0`이라고 되어 있는 것을 `1`이라고 모두 고쳐준다.

5.3 Profiling

Profiling이라고 하는 것은 프로그램의 어떤 부분이 제일 자주 호출되고 있는지 또는 많은 시간을 소요하고 있는지를 조사하는 것이다. 코드를 최적화시키고 시간이 가장 많이 소비되는 곳을 고쳐주는 좋은 방법이

다. 이렇게 하기 위해서는 -p 옵션을 주어서 시간 정보를 오브젝트 화일들이 가질 수 있도록 다시 컴파일해 주어야 한다. 또한 binutil 패키지에 있는 gprof 라는 것을 필요로 한다. 자세한 사항은 gprof 맨페이지를 참고하기 바란다.

제 6 절 링크

호환되지 않는 두 개의 바이너리 형식, 정적 라이브러리와 동적 라이브러리의 구분, 컴파일 과정 후에 일어나는 작업과 이미 컴파일을 마친 실행 프로그램이 실행될 때 일어나는 작업 둘 다에 대하여 "링크"라는 같은 말을 사용하여 생기는 혼란함(사실은 로드(load)한다라는 말에 대한 과부하라고 말할 수도 있다), 이런 모든 것에 대하여 다루므로 이번 섹션은 좀 복잡할 것이다. 말만 어려울 뿐이므로 크게 걱정할 필요는 없다.

이러한 혼란을 완화하기 위해서, 우리는 실행시(runtime)에 일어나는 일에 대하여 동적 로딩(Dynamic Loading)이라는 단어를 사용하겠다. 그리고 다음 섹션에 가서 다루고자 한다. 또는 동적 링킹(Dynamic Linking)이라는 단어로 표현되기도 한다. 이번 섹션에서는 오로지 컴파일 과정 바로 직후에 생기는 링크라는 작업에 대해서만 다루기로 한다.

6.1 정적 라이브러리 vs 공유 라이브러리

프로그램을 만드는 마지막 작업이 바로 링크(Link)라는 과정이다. 필요한 조각들을 모두 모으거나 어떤 부분이 빠져 있는지 알아보기 위한 과정이다. 분명히 프로그램들은 해야할 일이 많다. 이 모든 것을 일일이 다 짜주는 것은 아니다. 예를 들어 화일을 연다든지 하는 일인데 그러한 일들은 이미 여러분에게 라이브러리라는 형태로 주어져 있다. 평범한 리눅스 시스템에서는 /lib와 /usr/lib/에서 그러한 라이브러리들을 찾을 수 있다.

정적 라이브러리(Static Library)를 사용할 때, 링커는 프로그램이 필요로 하는 부분을 라이브러리에서 찾아서 그냥 실행화일에다 카피해버린다. 공유 라이브러리(또는 동적 라이브러리)의 경우에는 이렇게 하는 것이 아니라 실행화일에다가 단지 "실행될 때 우선 이 라이브러리를 로딩시킬 것"이라는 메세지만을 남겨놓는다. 당연히 공유 라이브러리를 사용하면 실행화일의 크기가 작아진다. 그들은 메모리도 또한 적게 차지하며, 하드 디스크의 용량도 적게 차지한다. 리눅스의 기본 행동은 일단 공유 라이브러리가 있으면 그것과 링크를 시키고, 그렇지 않으면 정적 라이브러리를 가지고 링크 작업을 한다. 공유 라이브러리를 쓴 실행화일을 얻고자 했는데, 우연찮게 정적 실행화일이 만들어졌다면 우선 공유 라이브러리가 제대로 있는지(a.out은 *.sa, ELF는 *.so)살펴보고 읽기 퍼미션이 주어져 있는지 알아본다.

리눅스에서 정적 라이브러리는 libname.a 과 같은 식의 이름을 갖는다. 그에 비해 공유 라이브러리는 libname.so.x.y.z 라는 식의 이름을 갖는데 x.y.z는 버전을 뜻한다. 또한 공유 라이브러리는 종종 링크되어 있다. (아주 중요) libname.so.x 그리고 libname.so라는 식의 링크를 갖는다. 표준 라이브러리들은 이 둘을 모두 가지고 있다.

여러분은 ldd라는 것을 사용함으로써 특정 프로그램이 어떤 공유 라이브러리를 원하는지 알 수 있다. (ldd = List Dynamic Dependencies)

```
$ ldd /usr/bin/lynx
        libncurses.so.1 => /usr/lib/libncurses.so.1.9.6
        libc.so.5 => /lib/libc.so.5.2.18
```

위 결과는 본인의 시스템에서 텍스트용 웹 브라우저로 사용하고 있는 lynx 라는 프로그램에 대하여 의존성 체크를 해본 결과이다. libc.so.5 (C 라이브러리)와 libncurses.so.1 (터미널 제어에 사용되는 라이브러리)를 필요로 하고 있다고 출력하고 있다. 아무런 공유 라이브러리도 필요없으면 그냥 'statically linked' 또는 'statically linked (ELF)' 라고만 출력한다.

6.2 라이브러리 들여다보기 (도대체 sin()은 어디에 들어있는가?)

nm *libraryname* 이라고 실행시키면 라이브러리 내의 모든 심볼을 출력해준다. 이는 공유 라이브러리와 정적 라이브러리 둘 다 적용된다. 만약 tcgetattr()이라는 함수를 찾고 싶다면 다음과 같이 해주면 된다.

```
$ nm libncurses.so.1 |grep tcget
        U tcgetattr
```

U가 뜻하는 바는 "undefined" 즉 ncurses 라이브러리가 사용하고는 있지만 아직 정의는 하지 않고 있다는 뜻이다.

이렇게도 할 수 있다.

```
$ nm libc.so.5 | grep tcget
00010fe8 T __tcgetattr
00010fe8 W tcgetattr
00068718 T tcgetpgrp
```

'W'는 "weak" 즉 심볼이 정의는 되어있으나 다른 라이브러리에 의해 재정의될 수 있는 형태라는 의미이다. 일반적으로 정상적인 경우에는 'T'라고 씌여진다.

sin()이 어디에 있는가라는 질문에 대한 가장 짧은 답은 libm.(so|a)이다. <math.h>에 정의되어 있는 모든 함수들은 바로 이 수학 라이브러리에 들어있다. 그것을 사용하기 위해서는 링크시에 -lm 옵션을 주어야 한다. using any of them.

6.3 화일 찾기

ld: Output file requires shared library 'libfoo.so.1'

컴파일을 하다보면 위와 같은 메세지가 종종 나오는 것을 볼 수 있을 것이다. ld 그리고 유사한 프로그램들이 화일을 찾는 방식은 버전에 따라 다르지만 기본적으로 /usr/lib를 찾게 된다. 이 곳 말고도 다른 곳에 라이브러리를 가지고 있고 그것을 ld 에게 알려주기 위해서는 gcc 나 ld 에게 라이브러리가 잇는 디렉토리를 -L 옵션을 줘서 알린다.

-L 옵션을 주어도 안된다면, ld 가 원하는 화일이 적절한 장소에 가 있는지 확인해보라. a.out 에 대해서는 -lfoo 라고 하면 ld는 libfoo.sa (공유 라이브러리)를 찾게 된다. 만약 그것을 찾는데 실패하면 libfoo.a (정적 라이브러리)라는 화일을 찾는다. ELF에 한해서는 libfoo.so를 찾고 나서 libfoo.a를 찾는다. libfoo.so는 libfoo.so.x에 대한 링크이다.

6.4 여러분만의 라이브러리 만들기

6.4.1 버전 관리

다른 모든 프로그램과 마찬가지로 라이브러리 또한 계속적으로 버그를 잡아가야 한다. 또는 새로운 기능을 도입하거나 현재 있는 것을 더 효율적인 것으로 교체한다든지 그리고 필요없는 것은 없애버린다든지 하는 일이 필요하다. 이런 경우 변화하는 라이브러리를 가지고 프로그래밍하는 것은 문제가 아닐 수 없다. 만약 사라져버린 옛 기능에 의존하는 프로그램이라면?

그래서 우리는 라이브러리 버전이라고 하는 것을 도입한다. 그리고 라이브러리의 변화를 마이너 또는 메이저 변화 이렇게 분류하고 마이너 업그레이드는 기존의 프로그램들과 충돌이 없는 변화를 지칭하게 한다. 라이브러리의 버전은 화일명을 보면 알 수 있다. (사실 엄밀히 말하자면, ELF에 대해서는 거짓말이다. 왜 그러한지는 계속 읽어보면 나올 것이다) libfoo.so.1.2는 메이저 버전 1 이고 마이너 버전2 이다. 마이너 버전도 다소 중요한 것이 될 수도 있다. libc의 경우에는 마이너버전에다 패치레벨을 집어넣는다. 따라서

libc.so.5.2.18과 같은 이름이 생긴다. 숫자 말고도 문자, 언더스코어문자(_), 또는 프린트 가능한 문자를 넣어도 좋다.

ELF와 a.out 형식의 커다란 차이점 중에 하나가 바로 공유 라이브러리를 만드는 방식에 있다. 우선은 ELF를 알아보기로 하자. 왜냐하면 더 쉽기 때문이다.

6.4.2 ELF? 도대체 그게 무엇인가?

ELF (Executable and Linking Format)이라고 하는 것은 원래 USL(UNIX System Laboratories)라고 하는 곳에서 개발한 바이너리 형식이다. 그리고 현재는 솔라리스와 SVR4에서 사용 중이다. 리눅스가 사용해왔던 오래된 a.out보다 더욱 더 좋은 유연성 때문에 GCC와 C 라이브러리 개발자들은 지난 해 리눅스 표준 바이너리 형식과 마찬가지로 ELF로 이동하기로 결정하였다.

다시 한 번 더? 이번 섹션은 '/news-archives/comp.sys.sun.misc' 문서로부터 나오는 내용이다.

> ELF ("Executable Linking Format)라고 하는 것은 "새롭고 향상된" 오브젝트 화일 형식으로서 SVR4 에 도입되었다. ELF는 그냥 COFF 방식보다 더욱 강력하다. 왜냐하면 사용자 확장성이 있기 때문이다. ELF는 오브젝트 화일을 임의의 길이를 갖는 섹션들의 리스트라고만 생각한다. 그것은 고정된 크기의 객체을 갖는 배열과는 다르다. 이러한 섹션은 COFF와는 달리 특정 위치에 있을 필요도 없고, 또한 특수한 순서대로 놓여있을 필요도 없다. 사용자들은 원한다면 새로운 섹션을 첨가할 수 있다. ELF는 또한 DWARF(Debugging With Attribute Record Format)라고 하는 아주 아주 강력한 디버깅 포맷을 가지고 있다. - 리눅스에서는 아직 완벽히 구현되고 있지는 않다. 하지만 작업이 진행 중이다 DWARF DIE들(또는 Debugging Information Entries) ELF 에서 .debug 섹션을 형성한다. 고정된 크기의 작은 정보들 대신에 DWARF DIE들은 각각 임의의 길이를 갖는 복잡한 속성들을 포함하고 있으며 영역별로 프로그램 데이타의 트리구조로 씌여져 있다. DIE는 COFF .debug 섹션보다 많은 양의 정보를 잡아낼 수 있다.(COFF의 경우에는 C++ 계승 그래프와 같은 것들을 잡아낼 수 없다.)
>
> ELF 화일들은 SVR4(솔라리스 2.0 ?)의 ELF 접근 라이브러리를 통해서 접근할 수 있다. 그 라이브러리는 ELF에 대하여 쉽고 빠른 인터페이스를 제공하고 있다. ELF 접근 라이브러리를 쓰면서 생기는 중요한 잇점중의 하나는 ELF 화일을 유닉스 화일로서 볼 필요가 전혀 없다는 것이다. 그것은 단지 Elf * 로서 접근가능하다. elf_open() 호출을 하면 그 다음부터 가능하다. 그 후에 elf_foobar()와 같은 작업을 한다. 이는 예전의 COFF 방식에서 실제 디스크 상의 이미지를 가지고 작업했던 것과는 전혀 다른 것이다.

ELF에 대한 찬성/반대, 그리고 현재의 a.out 시스템을 ELF 지원 시스템으로 업그레이드해야 할 필요성들은 ELF하우투 문서에서 다루고 있으며 본인은 그것을 여기에 적고자 하지는 않는다.

ELF 공유 라이브러리 libfoo.so라는 공유 라이비르러를 만들기 위한 기본적인 절차는 다음과 같다.

```
$ gcc -fPIC -c *.c
$ gcc -shared -Wl,-soname,libfoo.so.1 -o libfoo.so.1.0 *.o
$ ln -s libfoo.so.1.0 libfoo.so.1
$ ln -s libfoo.so.1 libfoo.so
$ LD_LIBRARY_PATH=`pwd`:$LD_LIBRARY_PATH ; export LD_LIBRARY_PATH
```

이렇게 하면 libfoo.so.1.0이라는 공유 라이브러리가 만들어질 것이다. 그리고 ld (libfoo.so 필요)와 동적 링커(libfoo.so.1 필요)에 필요한 적절한 링크가 만들어진다. 그것을 테스트해보기 위해서 우리는 LD_LIBRARY_PATH에다 현재 디렉토리를 첨가한다.

만약 라이브러리가 제대로 작동한다는 것을 확인하면, 그 라이브러리를 /usr/local/lib로 이동시킨다. 그리고 다시 링크를 만들어준다. libfoo.so.1로부터 libfoo.so.1.0에 이르는 링크는 ldconfig라고 하는 프로그램에 의해 항상 최신 정보로 관리된다. 보통은 부팅과정에서 알아서 해준다. 하지만 libfoo.so는 수동으로 해주어야 한다. 여러분이 한번에 한 라이브러리의 모든 부분들(예를 들어 헤더화일도 해당) 꼼꼼히 업그레이드해주려고 한다면 libfoo.so -> libfoo.so.1이라는 링크를 만들어 주면 된다. 그렇게 되면 ldconfig가 알아서 링크를 관리해준다. *만약에 이런 것까지 모두 여러분 스스로 모두 행하려고 한다면 나중에 문제가 생길 수도 있다.* 분명히 말해두었다.

```
$ su
# cp libfoo.so.1.0 /usr/local/lib
# /sbin/ldconfig
# ( cd /usr/local/lib ; ln -s libfoo.so.1 libfoo.so )
```

버전 번호 붙이기, soname 그리고 심볼릭 링크 각 라이브러리는 *soname*이라는 것을 가지고 있다. 링커가 찾고 있는 라이브러리 안에서 이러한 이름을 발견하게 되면, 실제 화일명(libfoo.so와 같은 이름)이 아니라 soname이라고 하는 것을 실행 바이너리에 표시해둔다. 실행시에는 동적 로더가 soname을 갖는 화일을 찾게 된다. 이 역시 화일명이 아니다. 이는 무엇을 의미하는가? 하면 libfoo.so 화일명을 가진 라이브러리는 libbar.so라는 soname을 가질 수도 있고 그곳에 링크된 모든 프로그램은 결국 libbar.so를 찾는다는 것이다.

이것은 상당히 무의미한 기능처럼 보이는데 사실은 이것이야말로 같은 라이브러리의 서로 다른 버전이 어떻게 한 시스템에서 공존할 수 있는가를 이해하는데 있어 핵심적인 부분이다. 리눅스에서 라이브러리 이름짓는 사실상의 표준은 라이브러리를 libfoo.so.1.2 이런 식으로 부르고 libfoo.so.1이라는 soname을 부여하는 것이다. 만약 표준 라이브러리 디렉토리(예를 들어/usr/lib)에 추가되면 ldconfig는 libfoo.so.1 -> libfoo.so.1.2라는 링크를 만들어 줄 것이다. 그렇게 함으로써 실행시에 적절한 이미지가 선택되도록 해준다. 여러분은 또한 libfoo.so -> libfoo.so.1이라는 심볼릭 링크도 필요하다. 왜냐하면 ld 가 링크할 때 정확한 soname 을 찾게 하기 위해서이다.

따라서 라이브러리의 버그를 고칠 때 또는 새로운 기능을 첨가할 때(기존의 프로그램에 악영향을 주지 않는 변화들), 다시 라이브러리를 만들고 같은 soname을 주고 화일명은 바꾸도록 한다. 만약 여러분의 라이브러리와 링크되어 있는 기존의 프로그램들과 충돌하게 되는 라이브러리로 변화할 때는 soname의 숫자를 하나 늘리면 된다. 이러한 경우 새로운 버전의 라이브러리는 libfoo.so.2.0이 될테고, soname은 libfoo.so.2가 될 것이다. 그리고 이번에는 libfoo.so를 새로운 버전의 라이브러이에 심볼릭 링크시키도록 하자.

여러분이 *꼭 이런 식으로 라이브러리 이름을 지어줄 필요는 없다*. 하지만 그것은 괜찮은 관습이다. ELF는 여러분에게 라이브러리 이름짓기에 있어 유연성을 주고 있지만 그렇다고 해서 꼭 그렇게만 하라는 것은 아니다.

요약하자면, 여러분이 호환성을 깨는 것이 메이저 업그레이드이고 그렇지 않은 것이 마이너 업그레이드라는 전통을 준수한다면 다음과 같이 하라.

```
gcc -shared -Wl,-soname,libfoo.so.major -o libfoo.so.major.minor
```

모든 것이 제대로 될 것이다.

6.4.3 a.out 전통적인 형식

공유 라이브러리 만들기의 용이함은 ELF로의 업그레이드에 대한 중요한 이유이다. a.out으로 가능하기는 하다. <ftp://tsx-11.mit.edu/pub/linux/packages/GCC/src/tools-2.17.tar.gz>를 받아오자. 그리고 그 화일을 풀어서 나오는 20 페이지짜리 문서를 읽어본다. 남들에게 뻔히 보이는 열성지지자가 되고 싶지는 않다. 하지만 나는 나 자신을 귀찮게 하고 싶지는 않다. :-)

ZMAGIC vs QMAGIC QMAGIC 이라고 하는 것은 예전의 a.out(ZMAGIC 이라고 알려져 있다)과 마찬가지로 실행 화일의 형식이다. 하지만 첫번째 페이지는 매핑하지 않는 바이너리이다. 0-4096 까지 어떠한 매핑도 존재하지 않기 때문에 이렇게 함으로써 NULL 디레퍼런시 트래핑(deference trapping)을 아주 쉽게 할 수 있다. 부차적인 효과로서 여러분의 실행화일은 약 1K 정도 작아지게 된다.

구식 링커들은 오로지 ZMAGIC 만을 지원한다. 약간 덜 구식의 링커들은 둘 다 지원하면, 최신 버전들은 오로지 QMAGIC 만을 지원하고 있다. 이것은 별로 중요하지 않다. 왜냐하면 커널 자체가 두 가지를 모두 실행시킬 수 있기 때문이다.

file 명령을 주면 그것이 QMAGIC인지 판별할 수 있을 것이다.

화일 위치(File Placement) a.out(DLL) 공유 라이브러리는 2 개의 실제적인 화일 그리고 하나의 링크로 구성 되어 있다. 이 문서 전체를 통해서 계속 사용해온 이름인 foo 라는 라이브러리에 대하여 예를 들어 알아보자. foo 에 대하여 `libfoo.sa`, `libfoo.so.1.2` 그리고 `libfoo.so.1` 이라는 링크로 구성되어 있다. 링크는 `libfoo.so.1.2`를 가리킨다. 이것들 모두 무엇인가?

컴파일할 때 `ld`는 `libfoo.sa`를 찾는다. 이것이야말로 라이브러리에 대한 그루터기 화일이 된다. 그리고 링크과정에 대한 모든 외부 데이타와 함수에 대한 포인터를 지니고 있다.

하지만 실행시에는 동적 로더가 `libfoo.so.1`을 찾는다. 이는 실제 화일이 아니라 심볼릭 링크이다. 그 이유는 앞서와 마찬가지로 라이브러리가 기존의 어플리케이션과의 충돌없이, 더 새로운, 버그가 잡힌 새로운 버전으로 교체될 수 있도록 하기 위해서이다. 새로운 버전이 나오면(예를 들어 `libfoo.so.1.3`)이라고 하자. ldconfig를 실행시키면 자동으로 `libfoo.so.1 --> libfoo.so.1.3` 링크 작업을 해 줄 것이다. 구버전을 쓰는 프로그램도 아무 이상이 없을 것이다.

DLL 라이브러리(동어반복이라는 사실은 알고 있다. 역자 주 :DLL 에 이미 라이브러리라는 말이 들어있다)는 종종 정적 라이브러리보다 크다. DLL은 미래의 확장성을 위해서 뻥 뚫린 구멍의 형태로 자리를 유보해 둔다. 하지만 그 자리는 디스크 영역을 차지하지는 않도록 할 수 있다. 간단한 `cp`나 `makehole`이라는 프로그램으로 이렇게 하는 것이 가능하다. 이미 고정된 위치에 주소들이 있으므로 라이브러리 생성 후에 strip 할 수 있다. **하지만 ELF 라이브러리에 대해서는 strip하지 말라**.

libc-lite 란 무엇인가? libc-lite 라고 하는 것은 libc 에 대한 소규모 버전이라고 할 수 있다. 하나의 플로피 안에 들어가고 유닉스의 자잘한 많은 업무들에 충분한 정도만으로 구성된 라이브러리이다. 그것은 curses 나 dbm, termcap 등의 코드를 *포함하고 있지 않다*. 만약 여러분의 `/lib/libc.so.4`가 lite 버전의 라이브러리에 링크되어 있다면 즉시 완전한 libc 버전으로 교체하기 바란다.

보통 슬랙웨어의 루트 디스켓을 마운트해보면 이 lite 버전의 C 라이브러리가 들어있음을 알 수 있을 것이다. 설치 준비와 설치에 필요한 만큼의 작은 C 라이브러리이다.

6.4.4 링크하기 : 일반적인 문제들

여러분의 링크 문제를 내게 보내달라! 그러면 그것에 대해서 나는 아무 일도 하지 않을 것이다. 하지만 많이 쌓이는 문제에 대해서는 글을 쓰겠다.

공유 라이브러리와 링크되길 바라는데 정적 라이브러리와 링크되고 있다.

> 우선은 `ld`가 공유라이브러리를 제대로 찾을 수 있도록 링크가 알맞게 되어 있는지 점검한다. ELF에 대해서라면 이것은 `libfoo.so` 심볼릭 링크를 말하며 a.out의 경우에는 `libfoo.sa`화일을 말하는 것이다. ELF binutil 2.5 버전에서 2.6 버전으로 업그레이드한 많은 사람들이 겪고 있는 문제이다. 전 버전이 공유 라이브러리에 대하여 오히려 더 똑똑하게 찾아냈는데, 그 사람들은 모든 링크를 제대로 만들지 않았던 것이다. 지적인 행동양식을 다른 모든 설계방식과의 호환성을 위해서 신버전에서 제거되었다. 지적 행동양식은 잘못된 가정을 갖게 되고 오히려 더 많은 문제를 낳기 때문에 그렇게 한 것이다.

DLL 툴인 mkimage 가 libgcc를 찾는데 실패한다.

`libc.so.4.5.x`와 그 이상의 버전에 관하여 libgcc는 더 이상 공유 라이브러리가 아니다. 따라서 여러분은 '`-lgcc`'와 같은 라인을 모두 '`gcc -print-libgcc-file-name`'로 바꿔주어야 한다. (주의할 것은 바로 백쿼우트문자(')의 사용이다. 꼭 이 문자만을 사용하라.)

또한 모든 `/usr/lib/libgcc*` 화일들을 삭제하라. 이것이 중요하다.

__NEEDS_SHRLIB_libc_4도 마찬가지 문제이다.

DLL 생성시에 "Assertion failure" 메시지

이 메시지는 여러분이 가지고 있는 jump table 슬롯이 원래의 `jump.vars`화일에 너무 적은 공간 밖에 예약되지 않았기 때문에 오버플로우로 인해 생기는 문제이다. 여러분은 tools-2.17.tar.gz 패키지에 들어 있는 '`getsize`' 명령을 사용하여 그 범인을 찾아낼 수 있다. 아마도 유일한 해결책은 메이저 번호의 증가 밖에 없는 것 같다. 단지 이전 버전과 호환되도록 고려하면서 말이다.

`ld: output file needs shared library libc.so.4`

이러한 문구는 보통 libc가 아닌 라이브러리들 (즉, X 윈도우 라이브러리들...)하고 링크하려고 할 때 발생한다. `-static`을 함께 사용하지 않고 링크 시에 `-g` 옵션을 주었을 때이다.

공유 라이브러리에 대한 `.sa` 화일은 보통 정의되지 _NEEDS_SHRLIB_libc_4 라는 심볼을 가지고 있는데 나중에 `libc.sa`에서 해결된다. 하지만 `-g` 옵션을 주게 되면 `libg.a` 또는 `libc.a`와 링크되게 되므로 그 심볼은 해결이 되지 않게 되고 위와 같은 에러 메세지가 뜨게 되는 것이다.

결론적으로 `-g` 플래그로 컴파일할 때는 `-static` 이라는 옵션을 함께 주기 바란다. 또는 `-g`로 컴파일하지 않으면 된다. *링크할 것 없이* 원하는 부분만 `-g` 옵션을 주고 컴파일해도 충분한 디버깅 정보를 얻을 수 있다.

제 7 절 동적 로딩(Dynamic Loading)

이번 섹션은 지금 현재로선 아주 적은 내용만을 가지고 있다. ELF 하우투 문서를 발췌함으로써 그 내용이 계속적으로 늘어나게 될 것이다.

7.1 개념 잡기

리눅스는 공유 라이브러리를 가지고 있다. 이 글 전체를 읽는 동안 이제는 이런 말 듣는 것도 질렸을 것이다. 전통적으로 프로그램 링크 과정에서 행한 작업은 로딩 과정에서 그 반대 과정을 거쳐야 한다.

7.2 에러 메세지

`can't load library: /lib/libxxx.so, Incompatible version`

a.out 에서만 일어나는데, 이 말은 여러분의 라이브러리 메이저 버전이 틀리다는 말이다. 다른 버전을 가지고 있다고 해서 눈가림식으로 심볼릭 링크하는 것으로 안된다. 된다 할지라도 결국엔 세그폴트를 일으킬 것이다. 새로운 버전을 가져오라.ELF에서도 비스한 메세지가 나온다.

```
ftp: can't load library 'libreadline.so.2'
```

`warning using incompatible library version xxx`

a.out의 경우이다. 프로그램 컴파일한 사람보다 낮은 마이너 버전의 라이브러리를 갖고 있기 때문에 발생하는 경고 메세지이다. 프로그램이 실행되기는 할 것이다. 업그레이드하는 것이 어떨까?

7.3 동적 로더의 작동 제어하기

많은 환경 변수들이 동적 로더에 관계한다. 대부분은 일반 사용자보다는 `ldd`에게 유용하다. `ldd`에 다양한 스위치를 줌으로써 쉽게 세팅할 수 있다.

- `LD_BIND_NOW` — 일반적으로 함수가 호출되기 전까지는 라이브러리에서 찾아보지 않는다. 이 플래그를 세팅해주면 라이브러리 적재시에 모든 체크를 하게 되고 시작은 상당히 느리게 된다. 이것은 여러분이 만든 프로그램이 모든 것들과 제대로 링크가 되었는지 시험해볼 때 유용하다.
- `LD_PRELOAD` — overriding 함수 정의를 가지고 있는 화일에 세팅될 수 있다. 예를 들어서 메모리 할당 방법을 테스팅하려고 하며, malloc를 교체하려고 할 때는 여러분이 원하는 루틴으로 만든 후에 교체할 수가 있다. `malloc.o` 라는 이름으로 컴파일한 후 다음과 같이 해보자.

  ```
  $ LD_PRELOAD=malloc.o; export LD_PRELOAD
  $ some_test_program
  ```

  `LD_ELF_PRELOAD`와 `LD_AOUT_PRELOAD` 이 둘은 비슷하다. 하지만 각각 특정 형태에만 관계한다. 만약 `LD_ELF_PRELOAD`와 `LD_PRELOAD`가 둘 다 사용되었다면 좀 더 자세히 지정한 전자 `LD_ELF_PRELOAD`가 사용된다.
- `LD_LIBRARY_PATH` — 이것은 공유 라이브러리를 찾을 때 참고할 디렉토리를 콜론(:)을 분리자로 써서 표현한 리스트이다. 그것은 ld에 *영향을 주지는 못한다.* 단지 실행시에만 관계한다. 또한 setuid나 setgid를 갖는 프로그램에 대해서는 무용지물이다. 마찬가지로 `LD_ELF_LIBRARY_PATH`와 `LD_AOUT_LIBRARY_PATH`는 각각의 바이너리 형식에만 적용되도록 하고 있다. `LD_LIBRARY_PATH`는 정상적인 경우 그렇게 필요하진 않다. 대신에 `/etc/ld.so.conf/`에 디렉토리를 추가하고 ldconfig를 다시 한 번 실행시키는게 좋다.
- `LD_NOWARN` — 이는 a.out에만 적용된다. 예를 들어 다음과 같이 세팅하면 `LD_NOWARN=true; export LD_NOWARN`) 마이너 버전이 다르다든지 하는, 크게 심각하지 않는 경고를 표시하지 않도록 한다.
- `LD_WARN` — 이는 ELF에만 해당된다. 세팅되면 일반적으로 "Can't find library"와 같은 심각한 에러를 경고로 바꾸어준다. 별로 필요없는 옵션이다.
- `LD_TRACE_LOADED_OBJECTS` — ELF에만 적용된다. 프로그램으로 하여금 `ldd` 하에서 실행되고 있다고 생각하게끔 만든다.

  ```
  $ LD_TRACE_LOADED_OBJECTS=true /usr/bin/lynx
          libncurses.so.1 => /usr/lib/libncurses.so.1.9.6
          libc.so.5 => /lib/libc.so.5.2.18
  ```

7.4 동적 로딩을 사용하는 프로그램 만들기

이는 솔라리스 2.x의 동적 로딩 지원이 이뤄지는 방식과 매우 흡사하다. H J Lu의 ELF 프로그래밍 문서에 자세히 나와 있으며 `dlopen(3)` 맨페이지에 아주 잘 나와 있다. 맨페이지는 ld.so 패키지에 들어있다. 다음 프로그램을 `-ldl` 옵션을 주고 링크하라.

```
#include <dlfcn.h>
#include <stdio.h>

main()
{
  void *libc;
  void (*printf_call)();

  if(libc=dlopen("/lib/libc.so.5",RTLD_LAZY))
```

```
    {
      printf_call=dlsym(libc,"printf");
      (*printf_call)("hello, world\n");
    }

  }
```

제 8 절 개발자와 연락하기(이하는 번역되어 있지 않습니다.)

8.1 Bug reports

Start by **narrowing the problem down**. Is it specific to Linux, or does it happen with gcc on other systems? Is it specific to the kernel version? Library version? Does it go away if you link static? Can you trim the program down to something **short** that demonstrates the bug?

Having done that, you'll know what program(s) the bug is in. For GCC, the bug reporting procedure is explained in the info file. For ld.so or the C or maths libraries, send mail to `linux-gcc@vger.rutgers.edu`. If possible, include a short and self-contained program that exhibits the bug, and a description both of what you want it to do, and what it actually does.

8.2 Helping with development

If you want to help with the development effort for GCC or the C library, the first thing to do is join the `linux-gcc@vger.rutgers.edu` mailing list. If you just want to see what the discussion is about, there are list archives at `<http://homer.ncm.com/linux-gcc/>`. The second and subsequent things depend on what you want to do!

제 9 절 The Remains

9.1 The Credits

> Only presidents, editors, and people with tapeworms have the right to use the editorial "we".

(Mark Twain)

This HOWTO is based very closely on Mitchum DSouza's GCC-FAQ; most of the information (not to mention a reasonable amount of the text) in it comes directly from that document. Instances of the first person pronoun in this HOWTO could refer to either of us; generally the ones that say "I have not tested this; don't blame me if it toasts your hard disk/system/spouse" apply to both of us.

Contributors to this document have included (in ASCII ordering by first name) Andrew Tefft, Axel Boldt, Bill Metzenthen, Bruce Evans, Bruno Haible, Daniel Barlow, Daniel Quinlan, David Engel, Dirk Hohndel, Eric Youngdale, Fergus Henderson, H.J. Lu, Jens Schweikhardt, Kai Petzke, Michael Meissner, Mitchum DSouza, Olaf Flebbe, Paul Gortmaker, Rik Faith, Steven S. Dick, Tuomas J Lukka, and of course Linus Torvalds, without whom the whole exercise would have been pointless, let alone impossible :-)

Please do not feel offended if your name has not appeared here and you have contributed to this document (either as HOWTO or as FAQ). Email me and I will rectify it.

9.2 Translations

At this time, there are no known translations of this work. If you wish to produce one, please go right ahead, but do tell me about it! The chances are (sadly) several hundred to one against that I speak the language you wish to translate to, but that aside I am happy to help in whatever way I can.

9.3 Feedback is welcomed. Mail me at `dan@detached.demon.co.uk`. My PGP public key (ID 5F263625) is available from my *web pages* `<http://ftp.linux.org.uk/~barlow/>`, if you feel the need to be secretive about things.

9.4 Legalese

제 10 절 Index

Entries starting with a non-alphabetical character are listed in ASCII order.