리눅스용 소프트웨어 패키지를 컴파일하고 인스톨하기

Mendel Cooper <mailto:thegrendel@theriver.com> — *http://personal.riverusers.com/˜thegrendel/*
<http://personal.riverusers.com/˜thegrendel/> v1.91, 27 July 1999

번역: 박민석 mpark@hrl.hitachi.co.jp

이 글은 리눅스에서 "일반적인" 유닉스 소프트웨어 배포본을 컴파일하고 설치하는 방법에 대한 안내서이다. 아울러 "rpm"과 "deb" 형식으로 미리 패키지화된 바이너리들에 대해서도 조금 다룬다.

목 차

제 1 절 도입

유닉스와 리눅스용의 많은 소프트웨어 패키지가 압축된 소스 파일 상태로 배포된다. 똑같은 패키지가 서로 다른 종류의 컴퓨터에서 돌아가도록 컴파일될 수 있고, 따라서 소프트웨어 제작자는 여러 버전을 만드는 수고를 덜 수 있다. 이렇게 해서, 한 소프트웨어 패키지의 배포본이 인텔을 비롯해 DEC의 알파, RISC 워크스테이션에서 메인 프레임에 이르는 컴퓨터에서 컴파일되고 사용된다. 하지만 불행하게도 이런 방식에서는 사용자 – 바로 당신 – 에게 소프트웨어를 컴파일하고 설치할 책임을 져야 한다. 하지만, 이 과정을 이해하는 것은 보기보다 어렵거나 신비로운 일이 아니다. 이 안내서가 보여주듯이 말이다.

제 2 절 파일을 풀기

당신은 소프트웨어 패키지를 다운받는 등의 방법으로 구했다. 대개 그 패키지는 `.tar.gz` 이나 `.tgz` 이라는 형태로 한 파일로 모아서(*tarred*) 압축한(*gzipped*) 것이다. (흔히 "tarball"이라고 알려져있다) 우선 이 파일을 적당한 작업 디렉터리로 복사한다. 다음엔 압축을 풀고(*gunzip*) 원래의 여러 파일로 풀어놓는다(*untar*). **tar xzvf *파일이름*** 이라고 하면 된다. 물론 *파일이름* 이란 소프트웨어 파일의 이름이다. tar를 풀면 보통 하위 디렉터리들을 만들고 거기에 적당히 파일들을 설치한다. 혹시 패키지 파일의 확장자가 *.Z* 라고 해도 위의 방법을 쓸 수 있다. **uncompress**를 수행한 다음 **tar xvf** 라고 해도 된다. **tar tzvf 파일이름**이라고 하면 이 과정을 미리 볼 수 있다. 이 명령은 실제로 압축을 해제하거나 하지는 않고 패키지 안의 파일 목록을 보여준다.

"tarball"을 풀어내는 위 방법은 다음의 두 가지와 동등하다.

- gzip -cd filename | tar xvf -
- gunzip -c filename | tar xvf -

('-'는 *tar* 명령의 입력을 표준입력으로부터 읽게 한다.)

새로운 *bzip2*(`.bz2`) 형식의 소스 파일은 **bzip2 -cd 파일이름 | tar xvf -** 로 풀 수 있다. 또 tar에 적절한 패치를 했다면 (자세한 것은 `Bzip2HOWTO` 를 참고) 간단히 **tar xyvf 파일이름** 이라는 명령으로 할 수도 있다. 데비안 리눅스는 히로시 다케카와가 만든 다른 `tar` 패치를 쓰므로 *-I, –bzip2, –bunzip2* 같은 옵션을 사용한다.

[위 정보를 정정하고 업데이트하도록 해준 것에 대해 R. Lynn과 Fabrizio Stefani에게 감사한다.]

패키지의 설정 정보에 지정된 것 때문에, 어떤 때는 압축된 파일을 사용자의 홈 디렉터리나 /, `/usr/src`, `/opt` 등의 다른 디렉터리에 풀어 설치해야 한다. 압축을 풀 때(untar) 에러 메시지가 나온다면, 이 디렉터리 지정 때문일 수 있다. 패키지의 문서 파일, 특히 `README`와 혹시 있다면 `INSTALL` 파일을 읽어 보라. 그리고 설치 요령에 맞게 config 파일과 `Makefile`들을 필요에 따라 고치도록 하자. `Imake` 파일은 대개 건드려서는 안된다는 점에 주의하라. 겉으로 드러나지 않는 결과를 가지고 있을 수 있기 때문이다. 대부분의 소프트웨어 패키지는 `make install` 을 수행하면 바이너리 파일들을 시스템의 적절한 자리에 갖다두도록 하여 설치 디렉터리 문제를 자동적으로 처리한다.

- 인터넷의 뉴스그룹 같은 곳에서는 `shar` 파일이나 *shell archive* 형태의 소스파일들을 볼 수 있다. 이 형식들은 사람이 읽을 수 있다는 점 때문에 사용되고 있다. 뉴스그룹의 관리자가 소스파일을 읽고 정리하거

나, 부적절한 것일 경우에는 삭제할 수 있기 때문이다. 이 형식은 **unshar 파일이름.shar** 명령으로 풀어야 한다. 다른 것은 "tarball"과 같다.

- 어떤 소스 파일들은 *zip*, *arc*, *lha*, *arj*, *zoo*, *rar*, *shk*와 같이 DOS, Mac, Amiga 의 비표준적인 압축 유틸리티로 처리되어 있다. 다행히 이들 대부분을 처리할 수 있는 리눅스용 유틸리티가 *sunsite* `<http://metalab.unc.edu>` 등에 있다.

때로는 풀어놓은 소스 파일을 업데이트하거나 버그 수정사항을 적용할 필요가 있다. 이럴 때는 변경 사항을 나열한 `patch`나 `diff` 파일을 사용한다. 이것은 `README`나 기타 문서를 통해 알 수 있다. Larry Wall의 막강한 유틸리티 *patch*를 사용하는 보통 방법은 **patch < patchfile** 이다.

이제는 설치 과정 가운데 컴파일 단계로 넘어갈 수 있을 것이다.

제 3 절 make 쓰기

`Makefile`은 컴파일 과정의 열쇠다. 가장 간단한 형태로 보면, Makefile은 패키지의 실행 가능한 부분인 "바이너리 파일"을 컴파일하기 위한 하나의 스크립트다. Makefile은 소스 파일을 모두 다시 컴파일하지 않고 소프트웨어 패키지를 갱신하는 방법을 제공할 수도 있지만, 그것은 또 다른 이야기(혹은 또 다른 글)이다.

어느 시점에 Makefile은 `cc`나 `gcc`를 실행시킨다. `cc`나 `gcc`는 사실은 순서에 따라 호출된 전처리기와 C(혹은 C++) 컴파일러, 링커의 집합이다. 이 과정을 통해 소스는 실제로 실행시킬 수 있는 바이너리로 변환된다.

대개 **make** 라고 치기만 하면 *make*를 시작할 수 있다. 이렇게 하면 일반적으로 설치하려고 하는 패키지에 필요한 모든 실행 파일들을 컴파일한다. 하지만 make는 파일들을 적당한 디렉터리에 설치한다거나 (**make install**), 묵은 object 파일을 삭제하는 등(**make clean**), 다른 작업들도 할 수 있다. **make -n** 이라고 하면 make에 의해 수행될 모든 명령들을 실행하지는 않고 보여주기만 하므로 컴파일 과정을 미리 볼 수 있다.

일반적인 Makefile을 쓰는 것은 매우 간단한 소프트웨어 뿐이다. 좀더 복잡한 설치 과정에는 라이브러리와 include 파일들의 위치, 개별 컴퓨터의 자원 등에 따라 Makefile을 맞출 필요가 있다. 특히 컴파일에 `X11` 라이브러리가 필요할 경우가 이에 해당된다. 이 작업은 *Imake*와 *xmkmf*가 처리한다.

man 페이지에서는 `Imakefile`은 Makefile의 "템플릿"이라 하고 있다. imake 유틸리티는 Imakefile로부터 당신의 시스템에 맞는 Makefile을 구성한다. 하지만 대부분의 경우 당신은 imake 유틸리티의 프런트 엔드(front end)로서 imake를 불러내는 쉘 스크립트인 **xmkmf**를 수행시킬 것이다. 구체적인 지시는 소프트웨어 패키지에 들어있는 README나 INSTALL 파일을 읽어보아라. (혹시 소스 파일들의 압축을 푼 다음에 기본 디렉터리에 `Imake` 파일이 있다면, **xmkmf**를 수행하라는 얘기다.) 이 과정에 대한 더 자세한 분석은 `Imake`와 `xmkmf`의 man 페이지를 읽어보아라.

`xmkmf`와 `make`는 root가 실행시켜야 할 수도 있다는 점에 유의하라. 특히 바이너리 파일들을 `/usr/bin`이나 `/usr/local/bin` 디렉터리로 옮기기 위해서 **make install**을 실행시킬 때 그렇다. root의 권한이 없는 일반 사용자로서 make를 사용하면 *write access denied* 라는 에러 메시지를 보게 되기 쉽다. 시스템 디렉터리에 쓰기 허가권을 갖고 있지 않기 때문이다. 만들어진 바이너리 파일들이 당신과 다른 적절한 사용자들에게 적합한 실행 허가권을 갖고 있는지도 확인해야 한다.

xmkmf는 당신의 시스템에 맞는 새로운 Makefile을 만들기 위해서 `Imake` 파일을 사용한다. 보통 **xmkmf**는 **-a** 옵션과 함께 사용한다. *make Makefiles, make includes, make depend* 를 자동적으로 실행하기 위해서다. 이렇게 하면 컴파일러와 링커를 위해 변수들을 설정하고 라이브러리의 위치를 정의한다. 때로는 `Imake` 파일이 없고 대신 같은 역할을 하는 `INSTALL` 이나 `configure` 라는 쉘 스크립트가 있을 수도 있다. `configure`를 실행시키려면 **./configure** 라고 해야 한다는 점에 주의해야 한다. 현재 디렉터리의 정확한 `configure` 스크립트를 호출하기 위해서다. 대부분의 경우 배포본의 `README` 파일에 설치 과정에 대한 설명이 들어 있다.

보통 `xmkmf`나 설치 스크립트가 만들어낸 `Makefile` 속을 직접 살펴보는 것이 좋다. 대개 Makefile은 당신의 시스템에 적합하지만, 경우에 따라서는 Makefile을 변경하거나, 에러를 손으로 수정할 필요가 있을 수도 있다.

대개 루트로서 **make install**을 하면, 새로 만든 바이너리 파일들을 적절한 시스템 디렉터리에 설치하게 된다. 요즘의 리눅스 배포본의 경우, 시스템 전체용의 바이너리 파일들은 보통 /usr/bin, /usr/X11R6/bin, /usr/local/bin에 설치된다. 원래 설치했던 리눅스의 일부가 아니기 때문에 별도의 바이너리 파일로 남겨두기 위해서 새로운 패키지는 /usr/local/bin에 설치하는 것이 좋다.

원래 상업용 UNIX를 위해 만들어진 패키지는 /opt나 기타 낯선 디렉터리에 설치되도록 되어 있을 수도 있다. 이 경우, 지정된 디렉터리가 없다면, 당연히 설치 에러 메시지가 나오게 된다. 이 문제를 해결하는 가장 간단한 방법은 root 로서 /opt 디렉터리를 만들어서 패키지를 설치하고, 해당 디렉터리를 PATH 환경변수에 넣는 것이다. 또는 /usr/local/bin 디렉터리로 심볼릭 링크를 만들 수도 있다.

일반적인 설치 순서를 정리하면 다음과 같다.

- README 파일 및 그밖의 가능한 문서를 읽는다.
- **xmkmf -a**나, INSTALL, configure를 실행한다.
- Makefile을 확인한다.
- 필요하면, **make clean**, **make Makefiles**, **make includes**, **make depend**를 실행한다.
- **make**를 실행한다.
- 파일의 허가권을 확인한다.
- 필요하면 **make install**를 실행한다.

주의점:

- 보통 패키지의 컴파일은 root로서 하지 않는다. root로 **su**를 하는 것은 컴파일된 바이너리 파일들을 시스템 디렉터리에 설치할 때에만 필요하다.
- *make*와 그 사용법에 익숙해진 다음에는, 설치하려는 패키지에 포함된 혹은 새로 만들어진 표준 Makefile에 gcc를 위한 최적화 옵션을 추가하고 싶을 수도 있다. 흔한 옵션은 *-O2*, *-fomit-frame-pointer*, *-funroll-loops*, (펜티엄 cpu라면) *-mpentium* 등이다. *Makefile*을 변경할 때에는 주의해야 한다.
- *make*로 바이너리 파일을 만든 다음에는 **strip**을 하고 싶을 수가 있다. **strip**이란 바이너리 파일에서 심볼릭 디버깅 정보를 제거해서 그 크기를 상당히 줄여주는 명령이다. 물론 이렇게 하면 디버깅은 불가능하다.
- *Pack Distribution Project* <http://sunsite.auc.dk/pack/>는 별도의 *컬렉션 디렉터리*에 설치된 파일들 사이의 심볼릭 링크를 처리하도록 Python 스크립트로 작성된 도구들에 기반하는데, 압축된 소프트웨어 패키지를 만드는 또다른 방법을 제공한다. 이 압축 패키지들은 보통의 *tarball*이지만, /coll과 /pack 디렉터리에 설치된다. 이 배포본들을 쓰려면 위의 사이트에서 *Pack-Collection* 을 다운받아야 한다.

제 4 절 미리 패키지화된 바이너리 파일

4.1 rpms, 무엇이 잘못되었나?

소스로부터 수동으로 패키지를 컴파일하고 설치하는 것은 분명히 어떤 리눅스 사용자들에게는 매우 겁나는 일이다. 그래서 인기있는 *rpm*과 *deb* 혹은 더 새로운 Stampede *slp* 패키지 포맷을 사용하곤 한다. *rpm* 설치가 보통은 부드럽고 빠를 수 있다. 어느 악명높은 운영 체제에서 소프트웨어를 설치하는 것 만큼이나 말이다. 하지만 자동으로 설치되는 미리 패키지화된 바이너리 파일의 약점에 대해서도 분명히 생각해 보아야 한다.

첫째, 소프트웨어 패키지가 보통은 먼저 "tarball"로 배포되며, 미리 패키지화된 바이너리 파일은 며칠에서 몇주 심지어는 몇달 뒤늦게 나온다는 것을 알아야 한다. 현재의 *rpm* 패키지는 최신의 "tarball"에 비해 적어도

마이너 버젼 2 정도 늦는 것이 일반적이다. 따라서 소프트웨어의 첨단을 따라가고 싶은 사람이라면, *rpm*이나 *deb*가 나오기를 기다릴 수 없을 것이다. 덜 인기있는 패키지라면 아예 *rpm*으로 만들어지지 않을 수도 있다.

둘째, "tarball" 패키지가 보다 완전하며, 더 많은 옵션을 가지고 있고, 더 최적화시키기 쉽다. 바이너리 파일의 rpm 배포본은 원래 배포본의 기능 가운데 일부를 가지고 있지 않을 수도 있다. 소스 *rpm*은 전체 소스 코드를 포함하고 있으며, **rpm –recompile 패키지이름.rpm** 이나 **rpm –rebuild 패키지이름.rpm** 옵션으로 컴파일하고 설치하여야 한다.

셋째, 어떤 미리 패키지화된 바이너리 파일들은 제대로 설치되지 않으며, 설치가 되었다고 해도 정상적으로 종료되지 못하고 core-dump를 낼 수 있다. 이것은 의존하고 있는 라이브러리의 버젼이 당신의 시스템에 있는 것과 다르기 때문일 수도 있고, 제대로 패키지화되지 않았을 수도 있으며, 혹은 그저 실행에 실패한 것(plain broken)일 수도 있다. 어떤 경우건, *rpm*이나 *deb*를 설치했다면, 당신은 그 *rpm*이나 *deb* 패키지를 만든 사람의 숙련도를 믿어야만 한다.

끝으로, 소스를 만지고 그것으로부터 배우기 위해서는, 소스 코드를 손 위에 갖고 있는 편이 좋다. 소스 코드를 압축 파일로 갖고 있는 편이 별도의 소스 *rpm*으로 갖고 있는 편에 비해 훨씬 더 수월하다.

rpm 패키지를 설치하는 것이 꼭 아무 생각없이 가능한 것은 아니다. 의존성에 문제가 있으면 *rpm* 설치는 실패할 것이다. 비슷한 경우로, *rpm*이 당신 시스템에 있는 것과 다른 버젼의 라이브러리를 요구한다면, 설치는 제대로 되지 않을 것이다. 당신이 지금 있는 라이브러리에서 없는 라이브러리로의 심볼릭 링크를 만들어준다 해도 말이다. *rpm* 설치가 편리하기는 하지만, "tarball" 설치가 실패하는 것과 같은 이유로 실패하는 경우가 자주 있다.

당신은 필요한 쓰기 허가권을 갖기 위해서 *rpm*과 *deb*의 설치를 root로서 해야만 하는데, 이것은 심각한 잠재적 보안 문제를 야기한다. 당신이 생각없이 시스템의 바이너리 파일과 라이브러리들을 망쳐버리거나, 심지어 당신 시스템을 파괴할 *트로이 목마*를 설치할 수도 있기 때문이다. 따라서 *rpm*과 deb 패키지를 "믿을 수 있는 자료원"으로부터 얻는 것이 중요하다. 어떤 경우에나 당신은 *rpm* 패키지를 설치하기 전에, **rpm –cecksig 패키지이름.rpm** 명령으로 (MD5 checksum과 대조하여) '서명 확인'을 해야만 한다. 마찬가지로 **rpm -K –nopgp 패키지이름.rpm**을 수행할 것을 강력하게 권한다. *deb* 패키지에서 이에 해당하는 명령은 **dpkg -I | –info 패키지이름.deb** 와 **dpkg -e | –control 패키지이름.deb** 이다.

- `rpm --checksig gnucash-1.1.23-4.i386.rpm`

  `gnucash-1.1.23-4.i386.rpm: size md5 OK`

- `rpm -K --nopgp gnucash-1.1.23-4.i386.rpm`

  `gnucash-1.1.23-4.i386.rpm: size md5 OK`

진짜 편집증 환자(이 정도라면 편집광이라고 부르는 경우가 더 많다)라면, 패키지를 풀어서 그 구성요소를 확인하기위한 *unrpm*과 *rpmunpack* 유틸리티를 `Sunsiteutils/packagedirectory`의 utils/package 디렉터리에서 구할 수 있다.

Klee Diene <mailto:klee@debian.org>은 설치된 *.deb* 파일에 문제가 있는지 MD5 checksum과 비교하여 확인하는 *dpkgcert*라는 실험적인 패키지를 작성했다. `Debianftparchive`에서 구할 수 있다. 현재의 패키지 이름 및 버젼은 *dpkgcert_0.2-4.1_all.deb* 이다. "http://dpkgcert.jimpick.com" name="Jim Pick Software">사이트는 *dpkgcert*가 전형적인 데비안 시스템 내의 패키지들을 검증하도록 하는 실험적인 서버 데이터베이스를 운영하고 있다.

가장 단순하게는, **rpm -i 패키지이름.rpm** 과 **dpkg –install 패키지이름.deb** 명령으로 자동적으로 소프트웨어를 풀어서 설치할 수 있다. 하지만 이 명령을 맹목적으로 쓰면 당신 시스템을 해칠 수도 있으므로 주의해야 한다.

위의 경고는 정도는 덜 하지만 슬랙웨어의 *pkgtool* 설치 유틸리티에도 역시 적용된다. 모든 "자동화된" 소프트웨어 설치는 주의를 요하는 것이다.

martian `<http://www.people.cornell.edu/pages/rc42/program/martian.html>`와 *alien* `<http://kitenet.net/programs/alien/>` 프로그램은 *rpm*, *deb*, Stampede의 *slp*, *tar.gz* 패키지 형식을 서로 변환해준다. 이 프로그램들을 쓰면 이 패키지들을 모든 리눅스 배포본에서 사용할 수 있다.

더 자세한 정보가 필요하면 *rpm*과 *dpkg* 명령의 man 페이지를 주의해서 읽고, RPMHOWTO와 TFUG의 *Quick Guide to Red Hat's Package Manager* `<http://www.tfug.org/helpdesk/linux/rpm.html>`, *The Debian Package Management Tools* `<http://www.debian.org/doc/FAQ/debian-faq-7.html>`를 참조하라.

4.2 rpms의 문제: 한 가지 예

Jan Hubicka `<mailto:hubicka@paru.cas.cz>`는 *xaos*라는 매우 훌륭한 프랙탈 패키지를 만들었다. *그의 홈 페이지* `<http://www.paru.cas.cz/~hubicka/XaoS>`에서 .tar.gz과 rpm로 된 패키지들을 구할 수 있다. 편의를 위해서 "tarball" 보다는 rpm 버젼을 사용하기로 하자.

불행하게도 rpm 버젼의 *xaos*를 설치하는데 실패했다. 두 별개의 rpm 판이 모두 제대로 동작하지 않았다.

rpm -i –test XaoS-3.0-1.i386.rpm

```
error: failed dependencies:
        libslang.so.0 is needed by XaoS-3.0-1
        libpng.so.0 is needed by XaoS-3.0-1
        libaa.so.1 is needed by XaoS-3.0-1
```

rpm -i –test xaos-3.0-8.i386.rpm

```
error: failed dependencies:
        libaa.so.1 is needed by xaos-3.0-8
```

이상한 것은 `libslang.so.0`, `libpng.so.0`, `libaa.so.1`이 모두 시험된 시스템의 `/usr/lib` 디렉터리에 있었다는 것이다. *xaos*의 rpm들은 릴리즈 번호는 같아도 조금 다른 버젼의 라이브러리를 쓰도록 컴파일되었음이 틀림없다.

시험삼아 `xaos-3.0-8.i386.rpm`을 *–nodeps* 옵션을 주어 강제로 설치해 보자. *xaos*를 실행시켜 보지만, 작동하지 않는다.

```
xaos: error in loading shared libraries: xaos: undefined symbol: __fabsl
```

왜 이렇게 되는지 알아보기 위해, 계속 시도해 보기로 하자. *xaos* 실행파일이 어떤 라이브러리에 의존하고 있는지 찾아보기 위해 *ldd*를 실행시켜 보면, 필요한 공유 라이브러리가 모두 있다는 것을 보여준다. `/usr/lib/libaa.so.1` 라이브러리에 *nm*을 실행해서 그 심볼릭 레퍼런스 목록을 보면, 이 라이브러리에는 정말 *__fabsl*이 빠져있다는 것을 알 수 있다. 물론 이 빠져있는 레퍼런스는 다른 라이브러리에서 빠진 것일 수도 있지만... 라이브러리를 바꾸지 않는 한, 더 이상 어쩔 수가 없다.

rpm은 이정도로 충분하다. 이제 "tarball" 즉 `XaoS-3.0.tar.gz`을 홈 페이지나 *ftp 사이트* `<ftp://ftp.ta.jcu.cz/pub/linux/hubicka/XaoS/3.0>`에서 다운받는다. 이 패키지를 컴파일해보기로 하자. **./configure**, **make**를 실행시키고, 마지막으로 (루트로서) **make install**을 실행한다. 문제없이 작동한다.

이것은 미리 컴파일된 패키지가 그 장점 보다 더 많은 문제를 일으키는 많은 예 중의 하나일 뿐이다.

제 5 절 Termcap 과 Terminfo에 관련된 이슈

man 페이지에 의하면, *"terminfo는 스크린 지향적인 프로그램들에 의해 이용되는, 터미널을 기술한 데이터 베이스"*이다. terminfo는 텍스트를 터미널에 표시하기 위해 사용되는 제어 시퀀스 (escape code)의 일반적인 집

합을 정의하며, 특정한 드라이버 필요없이 서로 다른 터미널 하드웨어를 지원할 수 있도록 한다. *terminfo* 라이브러리는 현재의 리눅스 배포본에서 /usr/share/terminfo 에 위치한다.

terminfo 데이버 베이스는 이전의 *termcap*과 이젠 폐물이 된 *termlib*을 거의 밀어냈다. *termcap*을 요구하는 패키지를 다룰 때를 제외하고는, 프로그램을 설치하는데 이 문제까지 신경쓸 필요가 없다.

이제 대부분의 리눅스 배포본이 *terminfo*를 사용하고 있지만, 오래된 응용 프로그램들과의 호환성을 위해 오래된 termcap 라이브러리를 유지하고 있다. (/etc/termcap을 보라.) 가끔 termcap에 링크된 바이너리 파일을 쉽게 사용할 수 있도록 하기 위해 설치되어야 하는 특별한 호환성 패키지가 있다. 매우 드물게는 소스 파일에서 *#define termcap* 이라는 문장을 주석문 처리할 필요가 있을 수도 있다. 이에 대한 최종적인 정보는 당신이 사용하는 배포본의 적절한 문서 파일에서 확인하도록 하라.

제 6 절 a.out 바이너리 파일과의 호환성

매우 드문 경우이지만, a.out 바이너리 파일을 사용해야만 할 수도 있다. 소스 코드를 얻을 수 없다거나, 어떤 이유에서든 소스로부터 새로운 ELF 바이너리 파일을 만들 수 없기 때문이다.

이럴 때, ELF로 설치된 리눅스 시스템은 대개 /usr/i486-linuxaout/lib 디렉터리에 완벽한 a.out 라이브러리들을 가지고 있다. a.out 라이브러리는 ELF 라이브러리와의 혼란을 피하기 위해 ELF와 다르게 버젼을 붙인다. 따라서 a.out 바이너리 파일은 실행될 때 올바른 라이브러리를 찾을 수 있어야 하지만, 항상 이렇게 되지는 않는다.

커널이 a.out 지원 기능을 자체 내에나 모듈로서 갖도록 컴파일되었어야 한다는 것에 주의하라. 필요하면 커널을 다시 컴파일할 수도 있다. 어떤 리눅스 배포본은 특별한 호환용 유틸리티를 설치해야 하기도 한다. 예를 들어 a.out의 X 응용 프로그램을 위해서, 데비안에서는 xcompat를 설치해야 한다.

6.1 예

Jerry Smith는 몇 년 전에 *rolodex*라는 매우 편리한 프로그램을 짰다. 이 프로그램은 Motif 라이브러리를 사용하지만, 다행히도 정적으로 링크된 a.out 형식의 바이너리 파일을 구할 수 있다. 불행한 것은 *lesstif* 라이브러리를 써서 다시 컴파일 하려면 소스를 수없이 뜯어고쳐야 한다는 것이다. 더 큰 불행은 a.out이 어떤 ELF 시스템에서는 다음과 같은 에러 메시지를 터뜨린다는 것이다.

```
xrolodex: can't load library '//lib/libX11.so.3'
No such library
```

이 때, /usr/i486-linuxaout/lib에는 그런 라이브러리가 있지만 xrolodex는 그 라이브러리를 실행 중에 찾지 못한 것이다. 간단한 해결책은 /lib 라이브러리에 심볼릭 링크를 만들어주는 것이다.

```
ln -s /usr/i486-linuxaout/lib/X11.so.3.1.0 libX11.so.3
```

libXt.so.3과 libc.so.4 라이브러리도 비슷한 링크를 만들어주어야 한다. 물론 이런 일들은 root로서 해야 한다. 이때 이미 있는 라이브러리를 덮어쓰거나 버젼 번호에 충돌을 일으키는 것이 아닌지 아주 확실히 해야 한다는 점을 명심하라. 다행히도 새로운 ELF 라이브러리는 이런 문제를 예상하고 미리 a.out 라이브러리 보다 높은 버젼 번호를 가지고 있다.

위의 세 링크를 만들어주고 나면, *xrolodex*는 잘 실행된다.

xrolodex 패키지는 원래 *Spectro* <http://www.spectro.com/>에 올려졌지만, 그곳에서는 지워진 것 같다. 지금은 *Sunsite* <http://metalab.unc.edu/pub/Linux/apps/reminder/xrolodex.tar.z>에서 *tar.Z* 형식의 소스 파일[512k]로 다운받을 수 있다.

제 7 절 문제해결

에러 없이 *xmkmf*와 *make*에 성공했다면, 8(다음 절)로 넘어가도 좋다. 하지만 "실제 생활"에서는 첫번에 제대로 되는 일은 거의 없다. 이때가 당신의 재치를 시험할 때다.

7.1 링크 에러

- xmkmf를 실행했음에도 *make*가 다음과 같은 메시지를 내고 실패했다고 가정하자. `Ling error: -lX11: No such file or directory` 이 경우 *Imake* 파일이 제대로 설정되지 않았을 가능성이 크다. *Makefile*의 첫부분에 있는 다음과 같은 줄들을 확인해 보라.

  ```
  LIB=          -L/usr/X11/lib
  INCLUDE=      -I/usr/X11/include/X11
  LIBS=         -lX11 -lc -lm
  ```

  -L과 -I 스위치는 각각 컴파일러와 링커에게 *라이브러리*와 *include* 파일들을 어디에서 찾아야 하는지 알려주는 것이다. 이 예에서는 *X11 라이브러리*는 /usr/X11/lib 디렉터리에, *X11의 include 파일*들은 /usr/X11/include/X11 디렉터리에 있어야 한다. 당신의 컴퓨터는 이와 다르다면, *Makefile*에 필요한 수정을 가하고, *make*를 다시 해 보도록 하라.

- 다음과 같이 수학 라이브러리 함수에 대한 정의되지 않은 참조(undefined reference)가 있을 경우:

  ```
  /tmp/cca011551.o(.text+0x11): undefined reference to ‘cos’
  ```

  이 문제에 대한 해법은 Makefile 내의 *LIB*이나 *LIBS* 부분(위의 예를 보라)에 **-lm** 을 더함으로써 수학 라이브러리를 명시적으로 링크하는 것이다.

- *xmkmf*가 실패했을 때 해 볼 수 있는 또다른 것은 다음의 스크립트다.

  ```
  make -DUseInstalled -I/usr/X386/lib/X11/config
  ```

  이것은 *xmkmf*와 동등한 저수준 명령(bare bone)의 일종이다.

- 매우 드문 경우지만, *root*로서 *ldconfig*를 실행시키는 것이 해법이 될 수도 있다.

  # **ldconfig** 는 공유 라이브러리의 심볼릭 링크를 갱신한다. *이것은 반드시 필요한 것은 아닐 수도 있다.*

- 어떤 Makefile은 당신의 시스템에 있는 라이브러리에 대해 인식되지 않은(unrecognized) 별명(alias)를 사용하기도 한다. 예를 들어, 컴파일은 libX11.so.6을 요구하지만 /usr/X11R6/lib에는 그런 파일이나 링크가 없을 수 있다. 하지만 libX11.so.6.1은 있다. 해결책은 root로서 **ln -s /usr/X11R6/lib/libX11.so.6.1 /usr/X11R6/lib/libX11.so.6** 을 하는 것이다. 이 다음에는 **ldconfig**를 실행시켜야 할 수도 있다.

- 때로는 소스를 컴파일하기 위해 오래된 X11R5의 라이브러리가 필요할 수도 있다. /usr/X11R6/lib에 R5의 라이브러리가 있다면(처음 리눅스를 설치할 때 이것을 설치하도록 옵션을 주었다면), 소프트웨어를 컴파일하기 위해 필요한 링크가 제대로 있는지만 확인하면 된다. R5 라이브러리의 이름은 libX11.so.3.1.0, libXaw.so.3.1.0, libXt.so.3.1.0 이다. 보통 *libX11.so.3 -> libX11.so.3.1.0*과 같은 링크가 필요하다. 소프트웨어가 *libX11.so -> libX11.so.3.1.0* 형태의 링크를 필요로 할 수도 있다. 물론 "빠져있던" 링크를 만들려면, *root*로서 **ln -s libX11.so.3.1.0 libX11.so** 명령을 쓴다.

- 어떤 패키지는 라이브러리를 최신 버젼으로 설치하도록 요구하기도 한다. 예를 들어, StarDivision GmbH의 *StarOffice* 패키지 4.x 버젼들은 5.4.4 이상의 버젼을 가진 `libc`를 요구하기로 악명이 높았다. 그보다 뒤에 나온 *StarOffice* 5.0도 새로운 `glibc 2.1` 라이브러리를 설치하고 나면 작동하지 않는다. 다행히도, 그 다음에 나온 *StarOffice* 5.1은 이 문제를 해결했다. 오래된 버젼의 *StarOffice*를 실행시키려면, *root*로서 라이브러리들을 적절한 디렉터리에 복사하고, 오래된 라이브러리를 삭제한 다음 심볼릭 링크를 다시 설정해야 한다. (이에 대한 정보가 더 필요하면 최신 버젼의 `StarOffice miniHOWTO`를 보라.)

  주의: 실수하면 당신의 시스템이 작동하지 않도록 할 수도 있으므로, 이 과정에서는 극도로 주의하여야 한다. 갱신된 최신 라이브러리는 대개 Sunsite에서 찾을 수 있다.

  최신 라이브러리는 대개 `Sunsite`에서 찾을 수 있다.

7.2 다른 문제들

- 설치되어 있는 *Perl* 이나 그밖의 쉘 스크립트가 `No such file or directory`라는 에러 메시지를 낼 수 있다. 이 경우에는 파일이 실행 가능하게 되어 있는지 허가권을 확인하고, 스크립트가 호출하는 쉘이나 프로그램이 지정된 위치에 있는지 파일의 헤더 부분을 확인하도록 한다. 예를 들어 스크립트가 아래와 같이 시작한다고 하자.

  ```
  #!/usr/local/bin/perl
  ```

  *Perl*이 사실은 `/usr/local/bin`이 아니라 `/usr/bin` 디렉터리에 설치되어 있다면, 이 스크립트는 실행되지 않을 것이다. 이 문제를 해결하는 데에는 두 가지 방법이 있다. 스크립트 파일의 헤더를 `#!/usr/bin/perl`로 바꾸거나, **ln -s /usr/bin/perl /usr/local/bin/perl** 로 정확한 디렉터리로의 심볼릭 링크를 추가해 주면 된다.

- 어떤 X11 소프트웨어를 컴파일하려면 Motif 라이브러리가 필요하다. 표준적인 리눅스 배포본에는 Motif 라이브러리가 없으며, 현재 Motif는 100-200 달러다. (많은 경우 프리웨어인 *Lesstif* `<http://www.lesstif.org/>` 도 쓸 수 있지만.) 어떤 패키지를 컴파일하기 위해 Motif가 필요한데, Motif 라이브러리를 가지고 있지 않다면 *정적으로 링크된* 바이너리를 구할 수 있을 것이다. 정적 링크는 라이브러리 루틴들을 바이너리 자체에 넣는 것이다. 이렇게 하면 바이너리 파일은 훨씬 커지지만, 해당 라이브러리가 없는 시스템에서도 수행시킬 수 있다.

  컴파일을 위해 당신 시스템에 없는 라이브러리가 필요하다면, 그 패키지는 링크 에러(`undefined reference error`)를 일으킬 것이다. 그런 라이브러리는 비싼 것이거나, 다른 어떤 이유로 찾기 어려운 것일 수 있다. 그런 경우에는 *정적으로 링크된* 바이너리를 패키지 제작자나 리눅스 사용자 그룹에게서 구하는 것이 가장 쉬운 해결책이다.

- *configure* 스크립트를 수행시켰더니 당신이 컴파일하려는 패키지와는 관계없어 보이는 이상한 Makefile을 만드는 경우가 있다. 이것은 당신의 path 중의 다른 어딘가에서 발견된 엉뚱한 *configure*를 수행했다는 뜻이다. 이런 일을 피하려면 항상 **./configure**로 실행하도록 하자.

- 대부분의 리눅스 배포본은 옛날의 `lib 5`에서 `libc 6 /glibc 2` 라이브러리로 바뀌었다. 옛날 라이브러리와 함께 작동하도록 미리 컴파일된 바이너리라면 당신이 라이브러리를 업그레이드할 경우 문제를 일으킬 수 있다. 해법은 프로그램을 소스에서부터 다시 컴파일하든지, 미리 컴파일된 새로운 바이너리를 얻는 것이다. 혹시 당신이 시스템을 `libc 6`으로 업그레이드하는 중이고 이런 문제를 경험했다면 Eric Green의 *Glibc 2 HOWTO*를 참고하도록 하라.

  `glibc` 버젼들 사이에는 약간의 호환되지 않는 점이 있다는 것에 주의하라. 이 때문에 `glibc 2.1`에서 컴파일된 바이너리는 `glibc 2.0`과는 작동하지 않는다거나 그 반대의 경우가 있을 수 있다.

- 때로는 Makefile 안의 컴파일 옵션에서 *-ansi* 옵션을 제거할 필요가 있을 수도 있다. 이렇게 하면 gcc가 갖고 있는 ANSI 이외의 특징들을 활성화시키며, 이런 특징들을 요구하는 패키지들을 컴파일할 수 있게 된다. (이 사실을 지적해 준 Sebastien Blondeel에게 감사한다.)

- 어떤 프로그램들은 *root 권한*으로 실행되도록 하기 위해 *setuid root*를 해주어야 한다. 이렇게 하는 명령은 *root로서* **chmod u+s 파일이름** 하는 것이다. (프로그램이 그 이전에 미리 root 소유로 되어 있어야 한다는 점에 주의하라.) 이 명령은 파일 허가권 내의 *setuid* 비트를 설정하는 기능을 한다. 이 문제는 프로그램이 모뎀이나 cd-rom 드라이브 같은 시스템 하드웨어에 접근하거나, 어느 특히 악명높은 에뮬레이션 패키지 처럼 콘솔 모드에서 SVGA 라이브러리를 호출할 경우 일어난다. 프로그램이 root로서는 실행되지만, 일반 사용자에게는 *access denied* 에러 메시지를 내보낸다면 이 허가권 문제를 의심해보기 바란다.

  주의: root로의 *setuid*를 가진 프로그램은 시스템에 보안 상의 위험 요인이 될 수 있다. 이런 프로그램은 root의 권한을 갖고 실행되며 따라서 심각한 손상을 끼칠 잠재력을 갖고 있다. 따라서 *setuid* 비트를 주기 전에 프로그램이 무엇을 하는지, 가능하다면 소스를 살펴봄으로써, 확인하도록 하라.

7.3 변경(tweaking)과 미세한 조정(fine tuning)

당신의 시스템에 가장 좋은 컴파일 옵션들이 설정되어 있는지 확인하기 위해서 Makefile을 들여다보고 싶을 수도 있다. 예를 들어 *-O2* 옵션을 주면 최고 수준의 최적화를 선택하게 되고, *-fomit-frame-pointer* 옵션은 (디버깅은 불가능하게 되지만) 바이너리를 더 작게 만들어 준다. **자신이 무엇을 하고 있는지 모른다면 이런 것들을 건드리지 않도록 하라. 그리고 어떤 경우에든 시험삼아 그냥 컴파일해서 제대로 돌아가는 것을 확인한 다음에 하도록 하라.**

7.4 더 도움이 필요할 때 가 볼 곳

내 경험에 의하면, 응용 프로그램 가운데 그대로 문제없이 컴파일되는 것은 약 25% 정도였다. 50% 남짓은 간단하건 끔찍할 정도건 노력하면 컴파일할 수 있다. 즉 상당수의 패키지들은 아무리 해도 컴파일할 수가 없다는 뜻이다. 그렇다고 해도, 이 패키지들의 인텔 ELF 및 a.out 바이너리를 Sunsite나 TSX-11archive 사이트에서 찾을 가능성도 있다. *레드 햇* <http://redhat.com>과 *데비안* <http://www.debian.org>도 흔히 쓰이는 리눅스 소프트웨어의 대부분이 미리 패키지화된 바이너리로 저장되어 있다. 혹은 소프트웨어의 제작자가 당신의 특별한 취향의 컴퓨터를 위해 컴파일된 바이너리를 제공해 줄 수도 있다.

미리 컴파일된 바이너리를 얻었다면, 당신의 시스템과의 호환성을 확인하기 위해 다음 사항들을 점검해야 한다.

- 바이너리가 당신의 하드웨어(예를 들어 인텔 x86)에서 작동해야 한다.
- 바이너리가 당신의 커널과 호환되는 것이어야 한다.(a.out 이나 elf)
- 당신의 라이브러리가 최신의 것이어야 한다.
- 당신의 시스템에 적절한 설치 유틸리티(rpm이나 deb)가 있어야 한다.

만약 모든 시도가 실패한다면, comp.os.linux.x나 comp.os.linux.development 같은 적절한 뉴스그룹에서 도움을 찾을 수 있다.

혹시 모든 노력이 수포로 돌아갔다고 하더라도, 최소한 당신은 최선을 다했으며 그 과정에서 많은 것을 배웠을 테니 너무 낙심할 필요는 없다.

제 8 절 마지막 단계

(.bashrc나 .cshrc 안에) 어떤 환경 변수를 설정해야 하는지, 또 .Xdefaults와 .Xresources 파일을 고쳐야 하는지 결정하기 위해 소프트웨어 패키지의 문서를 읽어 보아라.

응용 프로그램의 기본 설정 파일이 있을 수도 있다. 원래의 배포본이 Xfoo 라면 보통 Xfoo.ad라는 이름이다. 이런 파일이 있으면 당신의 컴퓨터에 맞도록 Xfoo.ad 파일을 편집해서, Xfoo 로 바꾼 다음(**mv**), *root로서* /usr/lib/X11/app-defaults 디렉터리에 설치하도록 한다. 이렇게 하지 않으면 소프트웨어가 이상하게 작동하거나 아예 작동하지 않게 될 수 있다.

대부분의 소프트웨어 패키지에는 하나 이상의 미리 포맷된 man 페이지가 있다. *root로서* Xfoo.man 파일을 /usr/man, /usr/local/man, /usr/X11R6/man 의 적당한 디렉터리(man1 - man9)에 복사하고, 그에 맞춰 이름을 바꾸어 준다. 예를 들어 Xfoo.man 을 /usr/man/man4에 두었다면, Xfoo.4 로 이름을 고쳐야 한다. (mv Xfoo.man Xfoo.4) 관례에 따르면, 사용자가 쓰는 기본 명령은 man1, 게임은 man6, 관리용 패키지는 man8이다. (자세한 것은 *man docs*을 보라.) 물론 원한다면 당신 시스템에서는 이와 다르게 할 수도 있다.

어떤 패키지에는 Makefile에 *install* 옵션이 빠져 있어서, 바이너리 파일을 적절한 시스템 디렉터리에 설치하지 않는다. 이런 경우에는 수동으로 /usr/bin, /usr/local/bin, /usr/X11R6/bin 등의 시스템 디렉터리에 바이너리들을 복사해서 설치할 수 있다. 물론 *root로서*. 리눅스 배포본이 기본으로 설치하지 않는 바이너리는 /usr/local/bin이 권장된다는 점을 염두에 두자.

대부분의 경우, 위의 절차들의 일부 혹은 전부가 **make install**로 자동적으로 처리된다. 패키지에 따라 **make install.man** 이나 **make install_man** 이 있을 수도 있다. 이런 경우에는 README나 INSTALL 파일에 명시되어 있을 것이다.

제 9 절 첫번째 예: Xscrabble

Matt Chapman의 Xscrabble이라는 프로그램이 있으면 재미있겠다고 생각했다. 나는 자주 ScrabbleTM를 듣기 때문이다. 나는 이 프로그램을 다운받아서 압축을 풀고, README 파일에 있는 아래의 순서대로 컴파일했다.

```
xmkmf
make Makefiles
make includes
make
```

물론 이렇게 해서 제대로 된 건 아니지만...

```
gcc -o xscrab -O2 -O -L/usr/X11R6/lib
init.o xinit.o misc.o moves.o cmove.o main.o xutils.o mess.o popup.o
widgets.o display.o user.o CircPerc.o
-lXaw -lXmu -lXExExt -lXext -lX11 -lXt -lSM -lICE -lXExExt -lXext -lX11
-lXpm -L../Xc -lXc

BarGraf.o(.text+0xe7): undefined reference to `XtAddConverter'
BarGraf.o(.text+0x29a): undefined reference to `XSetClipMask'
BarGraf.o(.text+0x2ff): undefined reference to `XSetClipRectangles'
BarGraf.o(.text+0x375): undefined reference to `XDrawString'
BarGraf.o(.text+0x3e7): undefined reference to `XDrawLine'
etc.
etc.
etc...
```

나는 이에 대해 comp.os.linux.x 뉴스그룹에 물어보았고, 어떤 사람이 Xt, Xaw, Xmu, X11 라이브러리가 링크 단계에서 발견되지 않았음이 틀림없다고 친절하게 가르쳐 주었다. 흠...

패키지에는 두 개의 주된 Makefile이 있었고, 그 중 src 디렉터리에 있는 것이 내 주의를 끌었다. Makefile 내의 한 줄이 LOCAL_LIBS을 다음과 같이 정의하고 있었다. LOCAL_LIBS = $(XAWLIB) $(XMULIB) $(XTOOLLIB) $(XLIB) 링크가 찾지 못한 라이브러리들로의 참조가 여기 있었다.

LOCAL_LIBS 로의 다음번 참조를 찾다가, 나는 그 Makefile의 495 행에서 다음을 발견했다.

```
      $(CCLINK) -o $@ $(LDOPTIONS) $(OBJS) $(LOCAL_LIBS) $(LDLIBS)
$(EXTRA_LOAD_FLAGS)
```

이제 이 LDLIBS는 무엇이었을까?

```
      LDLIBS = $(LDPOSTLIB) $(THREADS_LIBS) $(SYS_LIBRARIES)
$(EXTRA_LIBRARIES)
```

SYS_LIBRARIES 은 다음과 같았다.

```
 SYS_LIBRARIES = -lXpm -L../Xc -lXc
```

그렇다! 여기 찾을 수 없던 라이브러리들이 있었다.

링커가 LOCAL_LIBS 전에 LDLIBS를 보아야 하는 것일 가능성이 있었다. 따라서 처음으로 시도해 볼 일은 495 행의 $(LOCAL_LIBS)와 $(LDLIBS)를 바꿔서 아래와 같이 하는 것이었다.

```
       $(CCLINK) -o $@ $(LDOPTIONS) $(OBJS) $(LDLIBS) $(LOCAL_LIBS)
$(EXTRA_LOAD_FLAGS)                       ^^^^^^^^^^^^^^^^^^^^^^^
```

나는 위와 같이 바꿔서 다시 *make*를 실행해보았고, 이번에는 제대로 실행되었다. 물론 Xscrabble은 아직 디렉터리의 이름을 바꾼다거나 소스 파일 중의 하나에 있는 몇몇 선언문을 주석 처리하는 등, 세부적인 조정과 변경이 필요했지만, 이 패키지는 그 뒤 오랫동안 나를 즐겁게 해주었다.

[새 버젼의 Xscrabble은 이제 rpm 형식으로 구할 수 있으며, 문제없이 설치된다.]

*Xscrabble*를 얻으려면 *Matt Chapman* <mailto:matt@belgarath.demon.co.uk>에게 e-mail을 하거나, 그의 홈 *페이지* <http://www.belgarath.demon.co.uk/programs/index.html>에서 다운받을 수 있다.

```
Scrabble은 Milton Bradley Co., Inc.의 등록된 상표이다.
```

제 10 절 두번째 예: Xloadimage

이 예의 문제 첫번째보다 쉽다. *xloadimage* 프로그램을 내 그래픽 도구 모음에 추가하면 좋겠다고 생각했다. 나는 xloadi41.gz 파일을 Mui와 Quercia가 쓴 16(X User Tools)라는 훌륭한 책의 부록 CD에 있는 소스 디렉터리에서 직접 복사했다. 기대한 대로, *tar xzvf*로 파일을 풀 수 있었다. 하지만 *make*는 고약해 보이는 에러를 내고는 중단되어 버렸다.

```
gcc -c -O -fstrength-reduce -finline-functions -fforce-mem
-fforce-addr -DSYSV  -I/usr/X11R6/include
-DSYSPATHFILE=\"/usr/lib/X11/Xloadimage\" mcidas.c

In file included from /usr/include/stdlib.h:32,
                 from image.h:23,
                 from xloadimage.h:15,
                 from mcidas.c:7:
/usr/lib/gcc-lib/i486-linux/2.6.3/include/stddef.h:215:
conflicting types for ‘wchar_t’
```

```
/usr/X11R6/include/X11/Xlib.h:74: previous declaration of
'wchar_t'
make[1]: *** [mcidas.o] Error 1
make[1]: Leaving directory
'/home/thegrendel/tst/xloadimage.4.1'
make: *** [default] Error 2
```

에러 메시지에 결정적인 단서가 들어있다.

image.h 파일의 23째 줄을 보면...

```
        #include <stdlib.h>
```

Aha, *xloadimage*의 소스 어딘가에서 *wchar_t*가 표준 include 파일인 stdlib.h에서 정의된 것으로 다시 정의되었다. 먼저 *stdlib.h*는 포함시킬 필요가 없을 듯 하므로 image.h의 23째 줄을 주석문으로 만들어 보자.

이제 설치 과정은 아무런 치명적인 에러 없이 진행된다. *xloadimage* 패키지는 지금 정확히 작동하고 있다.

제 11 절 세번째 예: Fortune

이 예는 C 프로그래밍에 대한 지식이 조금 필요하다. 유닉스나 리눅스용 소프트웨어의 많은 부분이 C로 쓰여져 있으므로, 소프트웨어 설치에 대해서 진지한 사람이라면 최소한 조금은 C를 배워두는 것이 좋다.

악명높은 *fortune* 프로그램은 리눅스가 뜰 때마다 "fortune cookie"라고 불리는 재미있는 격언을 보여준다. 불행하게도(unfortunately, 발음갖고 농담하는 거다), 2.0.30 커널을 사용하는 레드 햇 배포판에서 fortune을 컴파일하려고 하면 심각한 에러가 생긴다.

```
~/fortune# make all


gcc -O2 -Wall -fomit-frame-pointer -pipe   -c fortune.c -o
fortune.o
fortune.c: In function 'add_dir':
fortune.c:551: structure has no member named 'd_namlen'
fortune.c:553: structure has no member named 'd_namlen'
make[1]: *** [fortune.o] Error 1
make[1]: Leaving directory '/home/thegrendel/for/fortune/fortune'
make: *** [fortune-bin] Error 2
```

fortune.c를 살펴보면, 해당되는 줄은 다음과 같다.

```
  if (dirent->d_namlen == 0)
          continue;
      name = copy(dirent->d_name, dirent->d_namlen);
```

우리는 dirent라는 구조체를 찾아야 하지만, 이 구조체는 *fortune.c* 안에 선언되어 있지 않으며, **grep dirent**를 해봐도 다른 소스 파일들에서도 찾을 수 없다. 하지만, *fortune.c*의 제일 위에 다음과 같은 줄이 있다.

```
#include <dirent.h>
```

이것은 시스템 라이브러리의 include 파일로 보이므로, *dirent.h*를 찾을 논리적인 위치는 */usr/include* 다. 실제로 */usr/include*에는 *dirent.h* 파일이 있지만, 그 파일은 dirent 구조체의 정의를 포함하고 있지 않다. 이 파일은 그 대신 또다른 *dirent.h* 파일을 참조하고 있다.

```
#include <linux/dirent.h>
```

마침내, */usr/include/linux/dirent.h*에서, 바라던 구조체 선언을 찾았다.

```
struct dirent {
        long            d_ino;
        __kernel_off_t  d_off;
        unsigned short  d_reclen;
        char            d_name[256]; /* We must not include limits.h! */
};
```

물론 이 구조체 선언은 *d_namelen* 을 포함하고 있지 않지만, 그에 해당될 만한 "후보"는 두 개 있다. 가장 그럴 듯 한 것은 *d_reclen*인데, 이 구조체 멤버는 무엇인가의 길이를 나타내는 것 같고 또 short integer 형이기 때문이다. 또 하나의 가능성은, *d_ino*인데 그 이름과 자료형으로 보아 inode의 번호일 것 같다. 우리는 아마도 "directory entry" 구조체를 다루고 있는 듯 하다. 그 원소들은 파일의 속성, 그 이름, inode, 길이(블럭 수)를 나타낼 것이다. 이것도 (d_reclen 과 d_ino 에 대한) 우리의 추측을 확인해 주는 것으로 보인다.

`fortune.c` 파일을 편집해서, 551째줄과 553째줄의 *d_namelen*에 대한 참조를 *d_reclen*에 대한 참조로 고치도록 하자. 다시 *make all* 을 해 보자. **성공이다.** 에러 없이 컴파일되었다. 우리는 이제 fortune으로부터 싼 값에 스릴을 느낄 수 있다.

제 12 절 네번째 예: Hearts

이번에는 80년대에 Bob Ankeney가 유닉스 시스템을 위해 작성했고, 1992년에 Mike Yang이 고쳤으며, 지금은 *Jonathan Badger* `<mailto:badger@phylo.life.uiuc.edu>`가 관리하고 있는 고색창연한 옛 게임인 Hearts다. 그 선조는 Oregon 소프트웨어의 Don Backus가 쓴 훨씬 더 오래된 파스칼 프로그램으로 나중에 Jeff Hemmerling이 개정했다. 원래는 다중 사용자 클라이언트를 염두에 둔 것이지만, 컴퓨터를 상대로 한 단일 사용자 모드도 가능하다. 세련된 면이 부족하고 컴퓨터 상대가 별로 강하지 않지만, 그래픽은 훌륭하다. 유닉스와 리눅스에서 가능한 것은 지금으로서도 친절한 Hearts 게임 밖에는 없는 듯 하다.

그 나이와 혈통 때문에, 이 패키지는 특히 리눅스 시스템에 설치하기 어렵다. 설치를 위해서는 길고 골치아픈 일련의 퍼즐을 풀어야 한다. 이 과정은 인내와 결의의 훈련이라 할 수 있다.

시작하기 전에, `motif` 나 `lesstif` 라이브러리가 설치되어 있는지 확인하도록 하라.

-

xmkmf

make

```
client.c: In function ‘read_card’:
client.c:430: ‘_tty’ undeclared (first use in this function)
client.c:430: (Each undeclared identifier is reported only once
client.c:430: for each function it appears in.)
client.c: In function ‘scan’:
client.c:685: ‘_tty’ undeclared (first use in this function)
make: *** [client.o] Error 1
```

`client.c`에 범인이 있다.

```
#ifndef SYSV
        (buf[2] != _tty.sg_erase) && (buf[2] != _tty.sg_kill)) {
```

```
 #else
        (buf[2] != CERASE) && (buf[2] != CKILL)) {
#endif
```

•

client.c의 39째 줄에

```
#define SYSV
```

를 더한다. 이렇게 하면 *_tty*로의 참조를 무시한다.

make

```
client.c:41: sys/termio.h: No such file or directory
make: *** [client.o] Error 1
```

•

리눅스 시스템에서는 termio.h 파일이 /usr/include 에 있다. 더 오래된 유닉스에서는 /usr/include/sys에 있다. 따라서, clinet.c의 41째 줄을

```
#include <sys/termio.h>
```

에서

```
#include <termio.h>
```

로 바꾼다.

make

```
gcc -o hearts -g      -L/usr/X11R6/lib client.o hearts.o select.o connect.o
sockio.o start_dist.o  -lcurses -ltermlib
/usr/bin/ld: cannot open -ltermlib: No such file or directory
collect2: ld returned 1 exit status
make: *** [hearts] Error 1
```

•

요즘의 리눅스 배포본은 구식의 *termlib* 데이터베이스 보다는 *terminfo*나 *termcap*을 사용한다.

Makefile의 655째 줄,

```
CURSES_LIBRARIES = -lcurses -ltermlib
```

를

```
CURSES_LIBRARIES = -lcurses -ltermcap
```

로 바꾼다.

make

```
gcc -o xmhearts -g      -L/usr/X11R6/lib xmclient.o hearts.o select.o
connect.o sockio.o start_dist.o gfx.o  -lXm_s -lXt -lSM -lICE -lXext -lX11
-lPW
/usr/bin/ld: cannot open -lXm_s: No such file or directory
collect2: ld returned 1 exit status
```

•

lesstif 의 주 라이브러리는 `libXm_s`가 아니라 `libXm`이다. 따라서 `Makefile`를 고친다.

653째 줄:

```
XMLIB = -lXm_s $(XTOOLLIB) $(XLIB) -lPW
```

다음과 같이 한다.

```
XMLIB = -lXm $(XTOOLLIB) $(XLIB) -lPW
```

make

```
gcc -o xmhearts -g      -L/usr/X11R6/lib xmclient.o hearts.o select.o
connect.o sockio.o start_dist.o gfx.o  -lXm -lXt -lSM -lICE -lXext -lX11 -lPW
/usr/bin/ld: cannot open -lPW: No such file or directory
collect2: ld returned 1 exit status
make: *** [xmhearts] Error 1
```

•

늘 하던 의심을 해보자

`PW`라이브러리가 없다. `Makefile`를 고친다.

653째 줄,

```
XMLIB = -lXm $(XTOOLLIB) $(XLIB) -lPW
```

를 다음과 같이 한다.

```
XMLIB = -lXm $(XTOOLLIB) $(XLIB) -lPEX5
```

(The `PEX5` lib comes closest to `PW`.)

make

```
rm -f xmhearts
gcc -o xmhearts -g      -L/usr/X11R6/lib xmclient.o hearts.o select.o
connect.o sockio.o start_dist.o gfx.o  -lXm -lXt -lSM -lICE -lXext -lX11 -lPEX5
```

드디어 `make`에 성공했다. (만세!)

•

설치:

root로서 다음과 같이 한다.

```
[root@localhost hearts]# make install
install -c -s  hearts /usr/X11R6/bin/hearts
install -c -s  xmhearts /usr/X11R6/bin/xmhearts
install -c -s  xawhearts /usr/X11R6/bin/xawhearts
install in . done
```

•

시험삼아 돌려보자.

rehash

(우리는 tcsh 쉘을 쓰고 있다.)

xmhearts

```
localhost:~/% xmhearts
Can't invoke distributor!
```

•

heats 패키지의 README 파일에 다음과 같이 적혀 있다.

> heartsd, hearts_dist와 hearts.instr를 local.h에 정의된 HEARTSLIB 디렉터리 안에 두고, 이 파일들에 누구나 접근할 수 있도록 하라.

local.h 파일 내용:

```
/* where the distributor, dealer and instructions live */

#define HEARTSLIB "/usr/local/lib/hearts"
```

이것은 RTFM의 고전적인 경우다.

*root*로서 다음과 같이 한다.

cd /usr/local/lib

mkdir hearts

cd !$

설치될 파일들을 이 디렉터리에 복사한다.

cp /home/username/hearts/heartsd .

cp /home/username/hearts/hearts_dist .

cp /home/username/hearts/hearts.instr .

•

다시 한 번 시험삼아 돌려보자.

xmhearts

한동안 돌아가다가 dealer died! 라는 메시지를 내며 죽는다.

•

"distributor"와 "dealer"는 하드웨어의 포트들을 조사(scan)한다. 따라서 우리는 이 프로그램들이 root의 권한을 필요로 하는지 의심해야 한다.

*root*로서 다음과 같이 해보자,

chmod u+s /usr/local/lib/heartsd

chmod u+s /usr/local/lib/hearts_dist

(앞서 이야기했듯이, *suid*된 바이너리는 보안 상의 허점을 만들 수도 있다는 것에 주의하라.)

xmhearts

드디어 돌아간다!

*Hearts*는 에서 구할 수 있다.

제 13 절 다섯번째 예: XmDipmon

```
Bullwinkle: Hey Rocky, watch me pull a rabbit out of my hat.
Rocky:      But that trick never works.
Bullwinkle: This time for sure.
            Presto!
            Well, I'm gettin' close.
Rocky:      And now it's time for another special feature.
            --- "Rocky and His Friends"
```

XmDipmon 은 인터넷 연결 상태를 보여주는 버튼을 표시하는 작고 재미있는 응용 프로그램이다. 낡은 전화선에서는 흔히 연결이 끊어지곤 하는데, 그런 경우에는 반짝거리면서 경고음을 낸다. 불행히도, XmDipmon은 *dip* 과만 작동하는데, 이때문에 인터넷 연결에 *chat*를 쓰는 대부분의 사람들에게는 쓸모가 없다.

XmDipmon 을 컴파일하는 것은 문제가 아니다. XmDipmon은 *Motif* 라이브러리에 링크되어 있지만, *Lesstif*와도 잘 컴파일되고 잘 실행된다. 이번 도전은 *chat*를 쓸 때도 작동되도록 패키지를 고치는 것이다. 이 작업은 실제로 소스 코드를 땜질하는 것까지 포함하고 있으며, 따라서 프로그래밍에 대한 지식이 어느 정도 필요하다.

```
"실행되면, xmdipmon 은 /etc/dip.pid 라는 파일을 확인한다. (-pidfile 의
명령행 인자를 써서 다른 파일을 찾도록 할 수도 있다.) 이 파일은 dip
데몬의 PID 를 포함하고 있다. (dip 는 일단 연결이 이루어지면 자신을
데몬 모드로 전환한다.)"
            --- XmDipmon README 파일로부터 인용.
```

-pidfile 옵션을 주어 실행하면, XmDipmon이 *chat* 로그인이 성공했을 때에만 존재하는 다른 파일을 확인하도록 할 수 있다. 명백한 후보는 모뎀의 lock 파일이다. 따라서 **xmdipmon -pidfile /var/lock/LCK..ttyS3** 로 프로그램을 실행시켜 보자. (이 명령은 모뎀이 4번 com 포트, 즉 ttyS3에 있다고 가정한 것이다.) 하지만 이것은 문제의 일부를 푼 것일 뿐이다. 프로그램은 계속 *dip 데몬*을 감시하며, 따라서 우리는 *dip 데몬* 대신 *chat*나 *ppp*와 관련된 프로세스를 감시하도록 바꿔야 한다.

소스 파일은 하나 밖에 없으며, 다행히도 주석문이 잘 달려있다. `xmdipmon.c` 파일을 보면, 헤더가 아래와 같이 되어 있는 *getProcFile* 함수를 찾을 수 있다.

```
/*****
* Name:                getProcFile
* Return Type:  Boolean
* Description:  tries to open the /proc entry as read from the dip pid file.
<snip>
*****/
```

결정적인 단서다. 함수 내부를 찾아보자.

```
/* we watch the status of the real dip deamon */
sprintf(buf, "/proc/%i/status", pid);
procfile = (String)XtMalloc(strlen(buf)*sizeof(char)+1);
strcpy(procfile, buf);
procfile[strlen(buf)] = '\0';
```

범인은 2383째 줄이다.

```
sprintf(buf, "/proc/%i/status", pid);
             ^^^^^^^^^^^^^^^^^^^^^
```

이 함수가 dip 데몬 프로세스가 실행되고 있는지 검사하는 것이다. 그러면, 대신 pppd 데몬을 감시하도록 하려면 이 부분을 어떻게 고치면 될까?

pppd 의 맨페이지를 보자.

```
FILES
       /var/run/pppn.pid (BSD or Linux), /etc/ppp/pppn.pid (others)
                     Process-ID for pppd process on ppp interface unit n.
```

`xmdipmon.c`의 2383째 줄을 아래와 같이 고치자.

```
sprintf(buf, "/var/run/ppp0.pid" );
```

고친 패키지를 다시 컴파일하자. 문제없이 컴파일된다. 이제 새로운 명령행 인자를 써서 시험해 보자. 아마 신통하게 잘 실행될 것이다. 인터넷 서비스 제공자(ISP)로 ppp 연결이 이루어져 있을 때에는 작고 파란 버튼이 표시되고, 연결이 끊어지면 반짝거리면서 경고음을 낸다. 이제 우리는 완벽하게 작동하는 *chat* 모니터를 갖게 되었다.

XmDipmon는 `RipleyLinuxTools`에서 다운받을 수 있다.

제 14 절 소스 모음을 찾을 수 있는 곳

당신의 시스템에 유틸리티나 다른 프로그램들을 추가하기 위해 새로 얻은 지식을 쓰고 싶다면, 소스 모음을 *Linux Applications and Utilities Page* `<http://www.redhat.com/linux-info/linux-app-list/linapps.html>`이나 *Red Hat* `<http://www.redhat.com/>`, *InfoMagic* `<http://www.infomagic.com>`, *Linux Systems Labs* `<http://www.lsl.com>`, *Cheap Bytes* `<http://www.cheapbytes.com>` 등의 매우 저렴한 가격의 시디롬을 통해서 구할 수 있다.

소스 코드를 잘 정리해 둔 곳으로는 `compsourcesUNIXarchive`가 있다.

많은 유닉스 소스 코드들이 `alt.sources` 뉴스그룹에 게재되어 있다. 특정한 패키지의 소스 코드를 찾고 있다면, 관련된 `alt.sources.wanted` 뉴스그룹에 게시물을 올리면 된다. `comp.os.linux.announce` 뉴스그룹도 확인해 볼 만한 곳이다. `Unixsources` 메일링 리스트에 등록하려면, *subscribe* 메시지를 보내도록 하라.

`alt.sources` 뉴스그룹의 게시물 모음은 다음의 ftp 사이트에 있다.

- `ftp.sterling.com/usenet/alt.sources/`
- `wuarchive.wustl.edu/usenet/alt.sources/articles`
- `src.doc.ic.ac.uk/usenet/alt.sources/articles`

제 15 절 마지막 이야기

한 마디로, 고집이 모든 차이를 만드는 것이다. (그리고 포기할 선을 높이 두는 것이 틀림없이 도움이 된다.) 모든 도전이 그렇듯이, 실수로부터 배우는 것이 지극히 중요하다. 각각의 실수, 모든 실패가 **소프트웨어 설치 기술**의 달인이 되기 위해 필요한 지식을 더해 줄 것이다.

제 16 절 참고 문헌과 더 읽을 꺼리

BORLAND C++ TOOLS AND UTILITIES GUIDE, Borland International, 1992, pp. 9-42.
[볼랜드 C++ 3.1 버전과 함께 배포된 매뉴얼 중의 하나. 도스를 위해 절름발이로 구현된 볼랜드 C++을 이용하여, 구문(syntax)과 개념 작성에 대해 상당히 좋은 소개를 제공한다.]

DuBois, Paul: SOFTWARE PORTABILITY WITH IMAKE, O'Reilly and Associates, 1996, ISBN 1-56592-226-3.
[나는 이 글을 완성하도록 읽어본 적이 없었지만, imake 레퍼런스의 결정판이라는 평판이다.]

Frisch, Aeleen: ESSENTIAL SYSTEM ADMINISTRATION (2nd ed.), O'Reilly and Associates, 1995, ISBN 1-56592-127-5.
[각별히 탁월한 시스템 관리 핸드북이지만, 소프트웨어 설치에 대해서는 간단한 스케치 정도 만으로 다루고 있다.]

Hekman, Jessica: LINUX IN A NUTSHELL, O'Reilly and Associates, 1997, ISBN 1-56592-167-4.
[리눅스 명령어 전반에 대한 훌륭한 레퍼런스.]

Lehey, Greg: PORTING UNIX SOFTWARE, O'Reilly and Associates, 1995, ISBN 1-56592-126-7.

Mayer, Herbert G.: ADVANCED C PROGRAMMING ON THE IBM PC, Windcrest Books, 1989, ISBN 0-8306-9363-7.
[고급 C 프로그래머라면 즉각 응용할 수 있는 아이디어로 가득한 책. 알고리즘에 대한 탁월한 해설, 프로그래밍 언어에 대한 풍자, 재미. 안타깝만 절판이다.]

Mui, Linda and Valerie Quercia: X USER TOOLS, O'Reilly and Associates, 1994, ISBN 1-56592-019-8, pp. 734-760.

Oram, Andrew and Steve Talbott: MANAGING PROJECTS WITH MAKE, O'Reilly and Associates, 1991, ISBN 0-937175-90-0.

Peek, Jerry and Tim O'Reilly and Mike Loukides: UNIX POWER TOOLS, O'Reilly and Associates / Random House, 1997, ISBN 1-56592-260-3.
[환상적인 아이디어의 원천. 그리고 이 글에서 설명된 방법을 써서 소스 코드로부터 설치할 수 있는 유틸리티가 잔뜩 있다.]

Stallman, Richard M. and Roland McGrath: GNU MAKE, Free Software Foundation, 1995, ISBN 1-882114-78-7.
[읽도록 요구되는 글.]

Waite, Mitchell, Stephen Prata, and Donald Martin: C PRIMER PLUS, Waite Group Press, ISBN 0-672-22090-3,.

```
[아마도 C 프로그래밍에 대한 최고의 입문서. 초보자를 위한 자세한 해설. 새로운
판도 지금 구할 수 있다.]

Welsh, Matt and Lar Kaufman: RUNNING LINUX, O'Reilly and Associates,
1996, ISBN 1-56592-151-8.
[몇몇 분야에서는 깊이가 부족하지만, 여전히 리눅스 전반에 대한 최고의 레퍼런스.]


dpkg, gcc, gzip, imake, ldconfig, ldd, make, nm, patch,
rpm, shar, strip, tar, termcap, terminfo, xmkmf의 맨 페이지.

David Fetter의 BZIP2 HOWTO.

Eric Green의 Glibc2 HOWTO.

Daniel Barlow의 LINUX ELF HOWTO.

Donnie Barnes의 RPM HOWTO.

Matthew Borowski의 StarOffice miniHOWTO.
```

[이 HOWTO들은 당신 시스템의 /usr/doc/HOWTO나 /usr/doc/HOWTO/mini 디렉터리에 있을 것이다. 텍스트, HTML, SGML 포맷의 개정 버젼들은 *LDP 사이트* <http://metalab.unc.edu/LDP/HOWTO> 및 해당 저자의 홈 페이지에서 구할 수 있다.]

제 17 절 감사의 글

이 HOWTO 의 저자는 아래의 사람들이 보내준 제안과 수정, 격려에 대해 감사한다.

- R. Brock Lynn
- Michael Jenner
- Fabrizio Stefani

이 HOWTO 를 이탈리아어와 일본어로 번역한 좋은 친구들에게도 역시 영예를 돌린다.

그리고 물론 *Linux Documentation Project* <http://metalab.unc.edu/LDP/>의 Greg Hankins 와 Tim Bynum 에게도 감사와 찬양과 축복을 보낸다. LDP가 이 모든 것을 가능하게 했기 때문이다.