Program Library HOWTO

WheelerDavid A.

dwheeler (at) dwheeler.com

도운이

박상민

mskrap (at) kaist.ac.kr

고친 과정

고침 1.0022 March 2002

고침 0.602000-05 고친이 황진희

최초 번역

이 HOWTO문서는 프로그래머들이 리눅스 상에서 어떻게 라이브러리를 만들고 사용하는지에 대한 방법을 논의한다. 이 문서는 정적 라이브러리, 공유 라이브러리, 동적 적재 라이브러리를 다룬다.

1. 소개

이 HOWTO문서는 프로그래머들이 리눅스 상에서 GNU 프로그램을 사용해서 어떻게 라이브러리를 만들고 사용하는지에 대한 방법을 논의한다. "프로그램 라이브러리"는 나중에 본 프로그램에 통합될 컴파일된 코드(자료)를 포함하고 있는 파일이다. 프로그램 라이브러리는 프로그램이 더 모듈화되고, 컴파일하기 빠르게 하고, 업데이트 하기 쉽게 만들어준다. 프로그램 라이브러리는 3가지 타입으로 나눌수 있다: 정적 라이브러리(static library), 공유 라이브러리(shared library), 동적 적재(DL)라이브러리(dynamically loaded library)

이 문서는 처음으로 프로그램이 실행되기 전에 실행가능한 프로그램으로 설치되는 정적 라이브러리에 대해서 논의하겠다. 그리고 나서, 프로그램이 시작할때 적재되고, 프로그램사이에 공유가 가능한 공유 라이브러리에 대해서 논의한다. 마지막으로, 프로그램이 실행될때 언제든지 적재되고 사용될 수 있는 동적 적재 라이브러리에 대해서 논의한다. 동적적재(DL) 라이브러리는 다른 방식의 라이브러리 형식(format)과 다르지 않다. (정적, 공유 라이브러리도 DL 라이브러리로 사용될 수 있다); 그 대신에, DL 라이브러리는 프로그래머에게 사용되는 방법에 의해 차이가 있다. 이 HOWTO 문서는 더 많은 예제가 있는 단락, 더 많은 소스의 참고자료가 있는 단락을 포함한다.

라이브러리를 개발하고 있는 개발자들은 공유 라이브러리로 만들어야 한다. 왜냐하면, 공유 라이브러리는 그 라이브러리를 사용하고 있는 응용 프로그램으로부터 라이브러리를 개별적으로 업데이트 할 수 있기 때문이다. 동적 적재(DL) 라이브러리는 유용하다. 하지만, 사용하기 위해 더 많은 량의 작업을 필요하고 많은 프로그램들이 이 라이브러리가 제공하는 유연성을 필요로 하지 않는다. 반대로, 정적 라이브러리는 라이브러리를 업데이트 하는것이 더 어렵기 때문에 일반적인 목적으로 사용되는 프로그램에는 추천하기 힘들다. 그래도, 각각은 그들의 장점을 가지고 있고, 각각의 타입에 대한 장점은 그 타입을 설명하는 섹션들에서 논의될 것이다.

공유 라이브러리를 DLL(Dynamic Linked Library)라고 하거나, DL라이브러리로 사용되는 라이브러리에 대해 DLL이라고 하거나, 위의 두가지 상황에 맞는 라이브러리를 DLL이라고 하는 것은 의미 없는 일이다. 당신이 위의 것들중 어느 의미를 고르더라고, 이 HOWTO 문서는 리눅스 상에서 돌아가는 DLLs를 다룰 것이다.

이 HOWTO문서는 오늘날 대부분의 리눅스 배포판에서 사용되는 형식인 실행, 연결 형식(ELF : Executable and Linking Format)에 대해서 다룰것이다. GNU gcc는 사실 ELF이외에 더 많은 라이브러리를 다룬다; 특별히, 대부분의 리눅스 배포판은 아직까지도 퇴색한 a.out형식을 사용한다. 그러나, 그 형식은 이 문서의 범위를 벗어난다.

만약 당신이 프로그램을 많은 시스템에 포팅할것이라면, 라이브러리를 설치하고 만들기 위해서 리눅스 툴을 직접 사용하기보다 GNU libtool (http://www.gnu.org/software/libtool/libtool.html)을 사용하는 것을 고려하라. GNU libtool은 일관되고, 포팅가능한 인터페이스속에 공유라이브러리의 복잡성을 숨기는 역할(예를들어, 라이브러리를 만들고 설치하는 것)을 지원하는 일반적인 라이브러리 지원 스크립트이다. 동적 적재 라이브러리를 위한 포팅 인터페이스를 사용하기위해, 당신은 많은 포팅 툴을 사용할 수 있다. GNU libtool은 "libltdl"이라는 툴을 포함한다. 그외에도, 당신은 모듈의 동적 적재를 지원하는 glib라이브러리(glibc와 혼동하지마라)를 사용할 수 있다. glib에 대해서는 http://developer.gnome.org/doc/API/glib/glib-dynamic-loading-of-modules.html에서 더 많이 찾아볼 수 있다. 다시한번 말하지만, 리눅스에서는 이런 기능은 이 HOWTO에 기술된 구성개념으로 구현 될 수 있다. 만약 당신이 리눅스에서 코드를 개발하거나 디버깅 하고 있다면, 당신은 이 HOWTO의 정보를 원하고 있을 것이다.

이 HOWTO의 본래의 위치는 http://www.dwheeler.com/program-library이고, Linux Documentation Project (http://www.linuxdoc.org)에 올라가 있다. 저작권(Copyright)는 Copyright (C) 2000 David A. Wheeler이고, GPL 라이센스를 따른다; 더 많은 정보를 위해서 마지막 섹션을 참고하라.

2. 정적 라이브러리

정적 라이브러리는 단순히 보통의 목적파일(object file)의 모음이다. 관례적으로, 정적 라이브러리는 ".a"의 확장자로 끝난다. 이것은 ar(archiver)프로그램에 의해서 만들어진다. 정적 라이브러리는 나중에 설명될 공유 라이브러리의 이점들 때문에 예전만큼 많이 쓰이지는 않는다. 하지만, 아직도 이것들은 만들어지고, 역사적으로 처음으로 생겼고, 설명하기에 쉽다.

정적 라이브러리는 사용자들이 프로그램을 다시 컴파일하지 않아도 링크시킬수 있도록 해주기 때문에, 컴파일 시간을 줄여준다. 오늘날의 빠른 컴파일러를 보면 재 컴파일 시간은 그

다지 중요한 요소가 아니기 때문에, 이것은 예전만큼 이점이 되지 못한다. 정적 라이브러리는 프로그래머들이 라이브러리를 링크하게는 허락하고 싶은데, 소스코드를 공개하고 싶지 않을 때에 유용하다(이것은 라이브러리 판매자에게 유용할 수 있겠지만, 이 라이브러리를 사용하려는 프로그래머에게는 이점이 아니다). 이론적으로, 실행화일에 링크되는 정적 ELF라이브러리 코드는 공유 라이브러리나 동적 적재 라이브러리보다 1-5%정도 빠르게 동작해야 한다. 하지만, 실재적으로 다른 몇몇 요소들 때문에 이런 현상은 거의 나타나지 않게 된다.

정적 라이브러리를 만들거나, 존재하는 동적 라이브러리에 오브젝트 파일을 추가하기 위해서 다음과 같은 명령을 사용하라:

```
ar rcs my_library.a file1.o file2.o
```

이 예제 명령은 오브젝트 파일 file1.o, file2.o를 정적 라이브러리 my_library.a에 추가하고, 만약 my_library.a가 존재하지 않는다면 만드는 것이다. 정적 라이브러리를 만드는 것에 대해 더 많은 정보를 보려면 ar(1)을 참조하라.

당신이 정적 라이브러리를 만들었다면, 그것을 사용하고 싶어 할 것이다. 당신은 실행화일을 만들때, 컴파일, 링크 과정에서 정적 라이브러리를 부를 수 있다. 만약 실행화일을 만들기 위해 gcc(1)을 사용한다면, 라이브러리를 표시하기 위해 -l 옵션을 사용할 수 있다; 더 많은 정보를 위해 gcc를 찾아보라. 또한, -l, -L등의 옵션을 지원하는 ld(1) 링커를 사용할 수 있다; 그러나, 대부분의 경우 ld(1)을 사용하는 것보다 gcc(1)을 사용하는 것이 좋다. ld(1)이 더 많이 바뀌기 때문이다.

3. 공유 라이브러리

공유 라이브러리는 프로그램이 시작할때 적재되는 라이브러리이다. 공유 라이브러리가 제대로 설치된다면, 그다음에 시작하는 모든 프로그램은 자동적으로 새 공유 라이브러리를 사용한다. 이것은 훨씬 더 유연성 있고, 발전된것이다. 왜냐하면, 리눅스에서 사용하는 방법은 당신에게 다음과 같은 것들을 허용하기 때문이다:

- 라이브러리를 업데이트해도 프로그램이 예전 버전의 라이브러리를 사용할수 있도록 지원한다.
- 어떤 특정한 프로그램을 실행시킬때, 라이브러리 내의 특정한 라이브러리나 함수를 오버라이드 할 수 있다.
- 이 모든것들을 존재하고있는 라이브러리에서 프로그램이 실행되는 동안에 가능하다.

3.1. 관례들(Conventions)

이 원하는 모든 기능을 지원하는 공유 라이브러리에 대해, 많은 관례와 지침이 따라주어야 한다. 당신은 라이브러리의 이름의 차이를 알아야 한다. 특별히, 불리는 이름“soname”과 실제 이름“real name”에 대해서 알아야 한다(그리고 이것들이 어떻게 상호작용하는지를). 당신은 또한 이것들이 파일시스템의 어느부분에 위치하는지 알아야 한다.

3.1.1. 공유 라이브러리 이름들

모든 공유 라이브러리들은 "불리는 이름"이라 불리는 특별한 이름을 가지고 있다. 그 불리는 이름은 접두사 "lib", 그 라이브러리 이름, ".so"와 인터페이스가 바뀜에 따라 증가되는 기간과 버전넘버로 이루어진다(특별 케이스로, 저 수준의 C라이브러리는 "lib"으로 이름이 시작하지 않는다). 충분히 검증받은(fully-qualified) 불리는 이름은 그 라이브러리가 속해있는 디렉토리를 접두사로 한다; 실제 시스템에서 충분히 검증받은 불리는 이름은 "실제이름"의 심볼릭 링크가 된다.

모든 공유 라이브러리들은 "실제이름"이라 불리는 실제 라이브러리 코드를 포함하는 파일 이름을 가지고 있다. 실제 이름은 불리는 이름에다가 기간, 마이너 숫자, 또다른 기간, 출시 숫자를 포함한다. 마지막의 기간과 출시 숫자는 옵션이다. 마이너 숫자와 출시 숫자는 당신이 라이브러리의 어떤 버전을 쓰고 있는지 정확하게 알게 함으로서 형상관리를 할 수 있도록 도와준다. 이 숫자들이 사용을 쉽게 한다 하더라도, 라이브러리 문서에 쓰인것과 같은 숫자가 아닐 수 있다는 점에 유의하라.

게다가, 컴파일러가 라이브러리를 요구할 때 사용하는 다른 이름이 있다(나는 그것을 "링크 이름(linker name)"이라 명칭할 것이다). 그것은 단지 불리는 이름에서 숫자를 없앤 이름이다.

공유 라이브러리를 다루는 방법은 이 이름들의 구분을 하는 것이다. 프로그램이 그들이 원하는 공유 라이브러리를 내부적으로 나열한다면 그것들은 그 라이브러리들의 불리는 이름만 나열해야 한다. 반대로, 당신이 공유 라이브러리를 만든다면, 당신은 그 라이브러리를 특정한 파일이름으로 만들어야 한다(더 자세한 버전 정보들과 함께). 당신이 새로운 버전의 라이브러리를 설치한다면, 당신은 그것을 새로운 특별한 디렉토리에다 설치하고 ldconfig(8)을 사용해서 프로그램을 돌린다. ldconfig은 존재하는 파일을 조사하고, /etc/ld.so.cache의 캐시 파일을 설정하면서(잠시후에 언급 될 것이다) 실제 이름에다가 불리는 이름으로 심볼릭 링크를 만들어준다.

ldconfig는 링커 이름을 만들지는 않는다; 보통 이것은 라이브러리 설치할때 만들어지고, 링커 이름은 "가장최근의" 불리는 이름이나 실제 이름으로 만들어진다. 나는 링커 이름을 불리는 이름의 심볼릭 링크로 사용할 것을 추천한다. 왜냐하면, 대부분의 경우 당신이 라이브러리를 업데이트 한다면, 당신은 링크 시킬때에 자동적으로 사용하고 싶어할 것이기 때문이다. 나는 H. J. Lu에게 왜 ldconfig가 자동적으로 링커이름을 만들어주지 않냐고 물어보았다. 그의 설명은 이렇다. 아마 당신을 최신버전의 라이브러리 코드를 돌리고 싶어할지 모른다. 하지만, 그 대신에 예전의(아마도 호환되지 않는) 버전의 라이브러리로 개발하고 싶어할 수 도 있다. 따라서, ldconfig는 프로그램이 어떤 라이브러리를 사용할 것인지 어떤 가정도 하고 있지 않다. 따라서, 설치자는 링커가 어떤 라이브러리를 사용할것이지에 대한 심볼릭 링크를 특별히 수정해 주어야 한다.

따라서, `/usr/lib/libreadline.so.3`는 `/usr/lib/libreadline.so.3.0`과 같은 실제 이름에 ldconfig에 의해 심볼릭 링크가 된 충분히 검증받은 불리는 이름이다. `/usr/lib/libreadline.so.3`을 심볼릭 링크하는 `/usr/lib/libreadline.so` 같은 링커 이름이 있을 수 있다.

3.1.2. 파일 시스템 배치

공유 라이브러리는 파일시스템의 어딘가에 위치해야만 한다. 대부분의 오픈소스 소프트웨어는 GNU표준을 따른다; 더 많은 정보를 위해서는 info:standards#Directory_Variables (info:standards#Directory_Variables)를 찾아보아라. GNU표준은 소스코드를 배포할때 표준으로 모든 라이브러리를 /usr/local/lib에 올리기를 추천한다(그리고 모든 명령어는 /usr/local/bin에 위치하기를 추천한다). 그들은 또한 이 표준을 오버라이드하고, 인스톨 순서를 정해주기위한 관례를 정의한다.

파일시스템 계층 표준(FHS = Filesystem Hierarchy Standard)는 배포판의 어디에서 무엇을 해야하는지 논의한다(http://www.pathname.com/fhs을 참조하라). FHS에 따르면, 대부분의 라이브러리는 /usr/lib에 인스톨 되어있어야만 한다. 하지만, 시작시에 요구되는 라이브러리는 /lib에 있어야 하고, 시스템의 일부가 아닌 라이브러리는 /usr/local/lib에 있어야 한다.

위의 두 문서사이에 정말 충돌이 있지는 않다; GNU표준은 소스코드의 개발자에게 기본적 것들을 추천한다. 반면에, FHS는 배포자(시스템 패키지 관리 시스템을 통해 소스코드의 기본적인 것을 오버라이드하는 사람)에게 기본적인것을 추천한다. 일반적으로 이것은 잘 돌아간다: "최근의"(아마도 버그가 있을지도 모르는!) 소스코드는 당신이 다운로드해서 설치를 하면 "local"디렉토리(`/usr/local`)에 설치될 것이고, 그 코드가 패키지를 발전시키면, 관리자는 배포판의 표준 위치로 그 코드를 표준으로 위치시킬 수 있다. 만약 당신의 라이브러리가 라이브러리를 통해 호출되는 프로그램을 호출한다면, 당신을 그런 프로그램을 /usr/local/libexec에 놓아야 한다(배포판에서는 /usr/libexec가 된다). 하나의 복잡한 점은, Red Hat종류의 시스템은 라이브러리의 탐색시에 /usr/local/lib을 기본으로 포함하지 않는다는 것이다; 아래의 /etc/ld.so.conf에 대한 논의를 보라. 다른 표준 라이브러리 위치는 X-windows를 위한 /usr/X11R6/lib을 포함한다. /lib/security는 PAM 모듈을 위한것이지만, 이것들은 DL라이브러리로 적재된다는 것을 명심하라(이것도 아래에서 논의된다).

3.2. 라이브러리 사용 방법들

모든 리눅스 시스템을 포함한 GNU glibc기반 시스템에서는, ELF 바이너리 실행파일의 시작은 프로그램로더가 적재되고 실행된 후에 한다. 리눅스 시스템에서는, 이 로더는 /lib/ld-linux.so.X(여기서 X는 버전 숫자)라는 이름이 붙는다. 이 로더는 프로그램에서 쓰이는 다른 모든 공유 라이브러리를 찾아주고 적재시켜준다.

탐색될때 찾아지는 디렉토리의 리스트는 /etc/ld.so.conf에 저장된다. 많은 Red Hat기반 배포판은 기본적으로 /usr/local/lib을 /etc/ld.so.conf에 저장하지 않는다. 나는 이것이 버그라고 생각한다. 따라서, Red Hat기반 시스템에서 많은 프로그램을 돌리기 위해 /etc/ld.so.conf에 /usr/local/lib을 추가하는것은 버그 "픽스"이다.

라이브러리에 몇개의 함수를 추가하고 싶은데, 라이브러리의 나머지 부분을 유지하고 싶다면, 오버라이드하는 라이브러리의 이름(.o파일)을 /etc/ld.so.preload에 넣어라; 이 "미리 적재되는" 라이브러리는 기본 셋에 대해 우선순위를 가질것이다. 이 미리 적재되는 파일은 일반적으로 긴급패치에 사용된다; 배포판은 일반적으로 출시될때 그런파일들을 포함하지 않는다.

프로그램 시작시에 이런 디렉토리를 다 찾는것은 매우 비효율적인 일이다. 따라서, 보통 캐싱 정렬이 사용된다. ldconfig(8)은 기본으로 /etc/ld.so.conf를 읽고 동적 링크 디렉토리들(표준 관례를따르는)에서 적절한 심볼릭 링크를 만들고, /etc/ld.so.cache에 캐시를 써 넣으면, 다른 프로그램에서 사용된다. 이것은 라이브러리 접근의 속도를 높여준다. 관련된것은 DLL이 추가

되거나 삭제되거나 DLL디렉토리가 변할때도 ldconfig이 작동해야 한다는 것이다; ldconfig를 동작시키는 것은 라이브러리를 설치할때 패키지 관리자가 수행해야할 작업 중 하나이다. 그리고나서, 시작시에 동적 로더가 /etc/ld.so.cache를 사용하고 필요한 라이브러리를 로드한다.

그런데, FreeBSD는 이 캐시를 위해 다른 파일이름을 사용한다. FreeBSD에서는, ELF 캐시는 /var/run/ld-elf.so.hints이고 a.out 캐시는 /var/run/ld.so.hints이다. 이것들은 ldconfig(8)에 의해서 업데이트된다. 따라서, 몇몇 다른 상황들에서만 이 장소의 차이가 문제가 된다.

3.3. 환경 변수들

여러가지 환경변수는 이 과정을 제어할 수 있다. 그리고 이 과정을 오버라이드하는 환경변수들이 존재한다.

3.3.1. LD_LIBRARY_PATH

이 특별한 실행을 위해 당신은 일시적으로 다른 라이브러리를 대체할 수 있다. 리눅스에서, 환경변수 LD_LIBRARY_PATH는 표준의 디렉토리들을 찾기전에 찾아보게되는 라이브러리의 디렉토리들의 콜론으로 구분되는 셋이다; 이것은 새 라이브러리나 특별히 제작한 표준이 아닌 라이브러리를 디버깅할때 유용하다. 환경변수 LD_PRELOAD는 /etc/ld.so.preload가 하는 것처럼 표준 셋을 오버라이드하는 공유 라이브러리를 함수와 함께 나열한다. 이것들은 /lib/ld-linux.so라는 로더에 의해 구현된다. LD_LIBRARY_PATH가 많은 유닉스 시스템에서 작동하는 반면 모든 시스템에서 작동하지 않는다는 것을 말하고 싶다; 예를들어, HU-UX에서는 이 기능이 환경변수 SHLIB_PATH에 의해서 가능하고, AIX에서는 LIBPATH에 의해 가능하다(같은 문법과, 콜론으로 구분되는 리스트로 가능하다).

LD_LIBRARY_PATH는 개발과 검사를 위해 편리하다. 그러나 보통의 유저의 보통의 사용을 위해서 설치 과정에서 변경되면 안된다; 왜 그런지는 http://www.visi.com/~barr/ldpath.html의 "Why LD_LIBRARY_PATH is Bad"에서 찾아보라. 하지만, 이 기능은 여전히 개발과 검사를 위해 유용하고, 다른방식으로 해결하지 못하는 것을 해결하는데 유용하다. 만약 당신이 환경변수 LD_LIBRARY_PATH를 설정하고 싶지 않다면, 리눅스에서 당신은 프로그램 로더를 직접 불러서 인자를 넘겨줄수도 있다. 예를들어, 다음은 환경변수 LD_LIBRARY_PATH의 경로 이외의 주어진 PATH를 사용할 것이고, 실행가능 프로그램을 돌릴 것이다.

```
/lib/ld-linux.so.2 --library-path PATH EXECUTABLE
```

인자없이 ld-linux.so를 돌리는 것은 당신이 이렇게 사용하는데에 도움을 줄 것이다. 하지만, 이것을 보통의 용도로 사용하지 마라. 이것들은 디버깅용이다.

3.3.2. LD_DEBUG

GNU C에서 또다른 유용한 환경변수는 LD_DEBUG이다. 이것은 dl* 함수를 위해 만들어졌다. 따라서 그들이 하고 있는 것들에 대한 매우 장황한 정보를 준다. 예를 보자:

```
export LD_DEBUG=files
command_to_run
```

라이브러리를 다룰때 파일과 라이브러리의 동작을 보여주고, 어떤 의존성이 발견되었고, 어떤 SOs(sonames)가 어떤 순서로 로드되었는지 말해준다. LD_DEBUG를 “bindings”로 설정하는 것은 심볼제한에 대한 정보를 보여주고, “libs”에 설정하는은 것은 라이브러리 탐색경로에 대해서 보여주고, “version”으로 설정하는 것은 버전 의존성을 보여준다.

LD_DEBUG를 “help”로 설정하고 프로그램을 돌리면 여러가지 옵션을 표시할 것이다. 다시, LD_DEBUG는 보통의 사용을 위해 있는 것이 아니라, 디버깅과 검사를 위해 편리한 것이다.

3.3.3. 다른 환경 변수들

로딩과정을 제어할 수 있는 많은 환경변수들이 있다; 그것들의 이름은 LD_나 RTLD_로 시작한다. 대부분의 다른 환경변수들은 로더 프로세스의 저 수준의 디버깅이나 특별한 용도의 구현을 위해 존재한다. 그것들 대부분은 문서화가 잘 되어있지 않다; 당신이 그것들에 대해 알고 싶어한다면 최상의 방법은 로더의 소스코드를 읽는 것이다(gcc의 일부).

특별한 조치가 취해지지 않는다면, 동적 연결 라이브러리에 사용자의 권한을 허락하는 것은 setuid/setgid가 걸린 프로그램에게 매우 위험하다. 따라서, GNU 로더(프로그램이 시작시에 프로그램의 나머지를 로드하는 로더)에서 setuid/setgid프로그램이라면 이 변수들(다른 비슷한 변수들)은 무시되거나 그들이 할 수 있는 역할이 매우 제한된다. 로더는 프로그램의 퍼미션을 체크해서 setuid/setgid인지 확인한다; uid/euid가 틀리거나, gid/egid가 틀리면 로더는 프로그램이 setuid/setgid라고 가정(또는 그럼 프로그램에서 파생된것)하고 따라서, 링크를 연결하는 동작에 매우 제한을 가하게 된다. 당신이 만약 GNU glibc라이브러리 소스코드를 읽었다면, 당신은 다음과 같은 것을 보았을 것이다; elf/rtld.c와 sysdeps/generic/dl-sysdep.c를 보아라. 이것은 당신이 uid/gid가 euid/egid가 같으면 프로그램을 불러서 환경변수들이 최대의 효과를 나타낼 수 있다는 것을 의미한다. 다른 유닉스같은 시스템에서는 다른 방식으로 처리하지만 같은 이유로 처리한다: setuid/setgid프로그램이 환경변수들에 의해 나쁘게 처리되면 안된는 이유이다.

3.4. 공유 라이브러리 만들기

공유 라이브러리를 만드는 것은 쉽다. 처음으로, gcc-fPIC나 fpic플래그를 사용해서 공유 라이브러리로 사용될 오브젝트 파일을 만들어라. -fPIC나 -fpic옵션은 “위치에 독립적인 코드”를 만들어주고, 공유 라이브러리의 조건을 만족시킨다; 아래의 차이점을 보라. 그리고 이 형식을 따라서 공유라이브러리를 만들어라:

```
gcc -shared -Wl,-soname,your_soname \
    -o library_name file_list library_list
```

두개의 오브젝트 파일(a.o, b.o)를 만들고 이것들 모두를 포함하는 공유 라이브러리를 만드는 예제이다. 컴파일이 디버그 정보(-g)와 경고정보(-Wall)를 포함하는데, 이것들은 공유라이브러리를 위해 필요한것은 아니지만, 추천되는 정보라는 것을 주의하라. 컴파일은 오브젝트 파일을 만들고(-c), -fPIC옵션을 요구한다.

```
gcc -fPIC -g -c -Wall a.c
```

```
gcc -fPIC -g -c -Wall b.c
gcc -shared -Wl,-soname,libmystuff.so.1 \
    -o libmystuff.so.1.0.1 a.o b.o -lc
```

주의할 만한 가치가 있는 것들이 있다:

- 꼭 필요한 경우가 아니라면, 결과로 생긴 라이브러리를 분해하거나, 컴파일러 옵션으로 -fomit-frame-pointer 옵션을 주지마라. 결과로 나온 라이브러리는 잘 동작할 것이고, 위의 행동은 디버거를 무용지물로 만든다
- 코드를 생성하기 위해 -fPIC이나 -fpic을 사용하라. 코드를 생성하기 위해 -fPIC이나 -fpic을 사용하는 것은 타겟에 따라서 다르다. -fPIC을 사용하는것은 언제나 동작한다. 하지만, -fpic을 사용하는 것보다 큰 코드를 생성할 것이다(PIC은 더 큰코드를 위한것이라서 더 많은 양의코드를 만든다는 것을 기억하라). -fpic옵션은 작고 빠른 코드를 만든다. 하지만, 전역심볼이나 코드의 크기 같은 것에서 플랫폼에 독립적이다. 링커는 공유 라이브러리를 만들때 이 옵션이 맞는지 말해줄 것이다. 어느것을 써야 할지를 모를때, 나는 언제나 동작하는 -fPIC을 선택한다.
- 몇몇의 경우에서, 오브젝트 파일을 만들기위해 gcc를 호출하는 것은 “-Wl,-export-dynamic” 옵션을 포함할 것이다. 보통 동적 심볼테이블은 동적 오브젝트에 의해 사용되는 심볼만 포함한다. 이 옵션은(ELF파일을 만들때) 동적 심볼테이블에 모든 심볼을 추가한다(더 많은 정보를 위해 ld(1)를 참고하라). ’역 의존성’이 있을때 이 옵션을 필요로 할 것이다. 즉, DL라이브러리가 라이브러리를 로드하는데 프로그램에서 필요한 심볼이지만, 관례에 의해 정의되지 않은 심볼을 필요할 경우 사용된다. “역 의존성”이 작동하기 위해서, 주 프로그램은 심볼이 동적으로 동작하게 해야 한다. 리눅스 시스템에서만 사용한다면, “-Wl,export-dynamic”대신에 “-rdynamic”을 사용할수도 있을 것이다. 하지만, ELF문서에 따르면 “-rdynamic”플래그는 리눅스가 아닌 시스템의 gcc에서 항상 작동하는 것은 아니다.

개발과정동안, 다른 많은 프로그램에서 사용되는 라이브러리를 수정하고 싶을때가 있을 것이다 -- 그리고 당신은 프로그램들이 “개발상의”라이브러리를 사용하는것을 원치 않을 것이고, 어떤 특정 응용프로그램만이 그것을 사용하기를 원할것이다. ld의 “rpath”옵션은 어떤 특정한 프로그램이 컴파일 될 때 실시간으로 라이브러리의 패스를 정해주는 역할을 한다. gcc에서 당신은 다음과 같은 방식으로 rpath를 지정해 줄 수 있다:

```
-Wl,-rpath,$(DEFAULT_LIB_INSTALL_PATH)
```

당신이 라이브러리 클라이언트 프로그램을 설치할때 이 옵션을 사용한다면, 그것이 충돌을 일으키지 않거나, 라이브러리를 숨기는 다른 기술을 사용하도록 하기위해 LD_LIBRARY_PATH를 사용하는 것을 걱정할 필요가 없다.

3.5. 공유 라이브러리를 설치하고 사용하기

당신이 공유 라이브러리를 만들었다면 당신은 그것을 설치하고 싶어 할 것이다. 간단한 방법은 표준 디렉토리(예들를어, /usr/lib)중 하나에 카피하고 ldconfig(8)을 실행시키는 것이다.

첫째로, 당신은 공유라이브러리를 어딘가에 설치하고 싶어할 것이다. 그리고나서, 당신은 실제이름을 불리는이름으로 심볼릭링크를 걸어야만 할것이다(버전 숫자가 없는 불리는 이름이다. 즉, “.so”로 끝나서 사용자들이 버전에 상관없이 사용하게 하는 것이다). 간단한 접근법은 다음을 실행시키는 것이다:

```
ldconfig -n directory_with_shared_libraries
```

마지막으로, 너의 프로그램을 컴파일할때 당신이 쓰려하는 정적, 공유 라이브러리에 대해 링커에게 말해줘야 한다. -l이나 -L옵션을 쓰면 된다.

당신이 라이브러리를 표준 공간(예를들어, 당신은 /usr/lib을 수정해야하는 것은 아니다)에 설치하고 싶지 않을 경우, 다른 접근법이 있다. 이 경우에, 당신은 다른 어딘가에 설치하고 프로그램이 라이브러리를 찾도록 충분한 정보를 주면된다. 이 방법에는 여러가지가 있다. 간단한경우로 gcc의 -L 플래그를 줄 수 있다. “표준이 아닌”공간에 있는 특정한 프로그램을 가지고 있다면 “rpath”를 사용할 수 있다. 물론 환경변수를 사용해서 해결하는 방법도 있다. 특별히, 당신은 콜론으로 구분되어지는 표준공간에서 검색 전에 찾아지는 공유라이브러리들의 디렉토리들의 모임인 LD_LIBRARY_PATH를 사용할 수 있다. 만약 당신이 bash를 사용한다면, my_program은 다음과 같은 방법으로 될 수 있다:

```
LD_LIBRARY_PATH=.:$LD_LIBRARY_PATH  my_program
```

몇 개의 함수를 오버라이드 하구 싶다면, 오버라이드 할 오브젝트 파일을 만들고 LD_PRELOAD를 설정하라;이 오브젝트 파일의 함수는 그 함수들을 오버라이드 할 것이다(다른 함수들은 원래 있던대로 있을 것이다).

보통은 당신이 라이브러리를 걱정없이 업데이트 할 수 있다; 만약 API가 바뀌면 라이브러리 제작자는 불리는 이름을 바꾸어야 한다. 이런방식으로, 한 시스템에 많은 라이브러리가 있을 것이고, 정확한것이 각각의 프로그램에서 선택되어서 사용되어진다. 그러나, 어떤 프로그램이 같은 불리는 이름을 가지는 라이브러리에 대해서 업데이트 한것이 잘 동작하지 않는다면, 너는 예전 버전의 라이브러리를 다른곳에 옮겨두고 그 프로그램의 이름을 바꾸고(예전이름에다가 “.orig”를 붙인다), 라이브러리 사용을 리셋하고 실제의 새로 이름이 붙은 프로그램을 부르는 작은 “감싸는(wrapper)” 스크립트를 만들수 있다. 당신이 원한다면, 숫자 관례가 허용하는 한 같은 디렉토리에다가 예전의 라이브러리를 다른곳에 옮겨 둘 수 있다. 감싸는 스크립트는 다음과 같다:

```
#!/bin/sh
export LD_LIBRARY_PATH=/usr/local/my_lib:$LD_LIBRARY_PATH
exec /usr/bin/my_program.orig $*
```

당신이 프로그램을 쓸때 꼭 이것에 의존하지는 말아라; 프로그램의 라이브러리가 예전것과 호환이 되는지 확인하거나 호환되지 않는 개정을 했을때 버전넘버를 증가시켰는지 체크하라. 이것은 최악의 경우의 “긴급” 접근법이다.

당신은 공유 라이브러리의 리스트를 ldd(1)을 사용해서 볼수 있다. 따라서, 예를들어 당신은 ls에 사용되는 공유 라이브러리를 볼 수 있다.

```
ldd /bin/ls
```

일반적으로 당신은 그 이름이 속해있는 디렉토리에 따라 관계되는 불리는 이름의 목록을 볼 것이다. 특별히 모든 경우에서 다음의 두가지의 의존성을 가질것이다:

- /lib/ld-linux.so.N (N은 1이상인데 보통 2이다). 이것은 다른 라이브러리를 로드하는 라이브러리이다.
- libc.so.N (N은 6이상이다). 이것은 C라이브러리이다. 다른 언어에서 C라이브러리를 사용하고자 하면(적어도 그들의 라이브러리에서 구현하려할때), 다른 대부분의 프로그램들이 이것을 포함할 것이다.

주의 : 당신이 신뢰하지 않는 프로그램에 ldd를 실행시키지 말아라. ldd(1) 매뉴얼을 보면 확실하겠지만, ldd는 (어떤 경우들에 있어서) 환경변수를 설정하고(ELF 오브젝트나, LD_TRACE_LOADED_OBJECTS) 프로그램을 실행시킴으로서 동작한다. 이것은 신뢰하지 못하는 프로그램에 있어서 (ldd의 정보를 보여주는것 대신에) 임의의 코드를 실행시킬 수 있다. 따라서, 안전을 위해서 당신이 신뢰하지 못하는 프로그램은 ldd를 사용하지 말아라.

3.6. 호환되지 않는 라이브러리들

새버전의 라이브러리가 예전버전과 이진-호환이 안된다면, 불려지는 이름이 바뀌어야 한다. C에서는 이진 호환이 안되게 되는 4가지 기본 경우가 있다.

1. 함수의 내용이 바뀌어서 본래의 스펙과 맞게 동작하지 않는 경우.
2. 수출된(exported) 데이타 아이템이 변한경우(예외 : 데이터 구조가 계속 라이브러리 내에 존재한다면, 데이터 구조의 내부에 아이템을 추가하는 것은 괜찮다)
3. exported 함수가 제거될 경우
4. exported 함수의 인터페이스가 변할 경우

이런경우를 피하려한다면, 이진-호환이 되게 라이브러리를 유지할 수 있다. 다시 말해서 그런경우를 피하고 싶다면 응용 이진 인터페이스(ABI=Application Binary Interface)를 할 수 있다. 예들들어, 예전것들을 지우지 않고 새로운 함수를 추가하고 싶을 수 있다. 당신은 구조에 새로운 아이템을 추가할 수 있다. 하지만, 예전 프로그램이 새로운 아이템을 추가한 구조를 인식하지 못한다거나 (프로그램이 아닌)라이브러리가 새로운 공간을 할당하지 못하거나, 추가의 아이템을 옵션으로 만드는것(또는 라이브러리가 그것들을 채우게 하는것)등등을 확실히 해야한다는 것을 주의하라. 주의하라 - 유저가 배열에 구조를 사용하고 있다면 당신은 배열의 구조를 늘릴수 없다.

C++(그리고 컴파일된 템플릿이나 다른 디스패치된 메소드를 지원하는 언어)에서 상황은 교묘해진다. 위의 모든 상황이 이루어져야 하고, 더 해주어야 할 이슈들이 있다. 상황은 컴파일된 코드에서 "보이지 않게" 구현되기 때문에 C++이 어떻게 구현되었는지 알지 못하면 애매모호해지는 의존성을 낳는 정보들 때문에 일어난다. 엄밀하게 말하면, 이것은 "새로운" 이슈는 아니다. 단지, 컴파일된 C++코드가 너에게 놀라게 할 수 있을것이다. 다음에 나오는 것들은 (아마도 부족하겠지만) Troll Tech's Technical FAQ (http://www.trolltech.com/developer/faq/tech.html#bincomp)에 나오는 이진 호환을 유지하기 위해 C++에서 사용하면 안되는 것들의 목록이다.

1. 가상함수를 재 구현하는것(그렇지 않으면 원래의 구현에서는 안전하다). 왜냐하면 컴파일러는 컴파일시에(링크시가 아니라) SuperClass::virtualFunction()를 구하기 때문이다.
2. 가상 멤버함수를 더하거나 제거하는것. 왜냐하면 이것은 모든 서브클래스의 vtlb의 크기와 구조를 바꾸기 때문이다.
3. 데이터 멤버의 타입을 바꾸거나 인라인 멤버함수를 통해 접근할 수 있는 데이터 멤버를 옮기는것.
4. 새 노드를 추가하는 것 이외에 클래스 구조를 바꾸는것.
5. private데이터 멤버를 추가하거나 삭제하는것. 왜냐하면 이것은 모든 서브클래스의 크기와 구조를 바꾸기 때문이다.
6. 인라인이 아닌 public, protected 멤버 함수를 제거하는것.
7. public, protected 멤버함수를 inline으로 만드는것.
8. inline함수를 바꾸어서, 예전버전이 제대로 작동하지 않는것.
9. 포팅할 프로그램의 멤버 함수의 접근권한(public, protected, private)을 바꾸는것. 왜냐하면, 어떤 컴파일러는 함수의 이름에 접근권한을 붙이기 때문이다.

위의 주어진 목록대로, C++라이브러리 개발자들은 때때로 이진 호환성을 깨는 업데이트를 계획해야만한다. 다행스럽게도, 유닉스 기반 시스템은 동시에 로드되는 라이브러리의 많은 버전을 가지고 있어서, 디스크 공간을 조금 할당한다면, 유저들은 예전의 라이브러리를 요구하는 "예전"프로그램을 돌릴수 있다.

4. 동적 적재(DL) 라이브러리

동적 적재 라이브러리는 프로그램이 시작할때가 아닌 다른 시기에 적재되는 라이브러리이다. 이것들은 플러그인이나 모듈을 구현할때 적합하다. 왜냐하면 그것들이 필요해질때까지 적재를 기다릴 수 있기 때문이다. 예를들어, PAM(Pluggable Authentication Modules)시스템은 관리자가 인증을 관리하는 것을 허용하기 위해 DL라이브러리를 사용한다. 이것들은 또한 때때로 코드를 머신 코드로 바꾸고 효율을 위해 멈추지 않고 컴파일된 코드로 만드는 인터프리터를 구현하는데 유용하다. 이것은 실시간의(just-in-time) 컴파일러나 MUD(multi-user dungeon)을 구현하는데에 유용하다.

리눅스에서, DL 라이브러리는 그들의 포멧의 관점에 있어서 특별하지 않다; 그들은 표준의 오브젝트 파일로 만들어지거나, 위에서 언급된 표준의 공유 라이브러리 파일로 만들어진다. 주요 차이점은 프로그램의 링크시나 시작시에 적재되지 않는다는 것이다; 대신, 라이브러리를 열거나, 심볼을 찾거나, 에러를 조정하거나, 라이브러리를 닫는 특별한 API가 있다. C 사용자들은 이런 API를 사용하기위해 dlfcn.h라는 헤더파일을 포함해야 한다.

리눅스에서 사용되는 인터페이스는 솔라리스에서 사용되는 인터페이스와 같다. 이것을 나는 "dlopen()" API라고 불를 것이다. 그러나, 이 같은 인터페이스가 모든 플랫폼에서 지원되는 것은 아니다; HP-UX는 shl_load()방법을 사용하고, 윈도우즈는 완전히 다른 인터페이스의 DLLs을 사용한다. 만약 당신이 폭넓은 포팅성이 목표라면, 당신은 이런 다른 플랫폼을 숨기는 라이브러리를 생각해야 할것이다. 하나의 방법은 모듈의 동적 적재를 지원하는 glib 라이브러리이다; 그것은 이 함수들의 포팅가능한 인터페이스를 구현하기위해 플랫폼의 동적 적재 루틴을 사용한다.
http://developer.gnome.org/doc/API/glib/glib-dynamic-loading-of-modules.html의 링크에서 glib에 대해 많은 것을 찾아볼 수 있다. glib 인터페이스가 이 문서에 잘 설명되있기 때문에 더 언급하지 않겠다. 다른 방법은 GNU libtool (http://www.gnu.org/software/libtool/libtool.html)의 일부인 libltdl을 사용하는 것이다. 당신이 이것보다 더 기능적인것을 원한다면, CORBA ORB(Object Request Broker)를 찾아보아라. 당신이 리눅스와 솔라리스에서 직접 지원되는 것을 아직도 원하고 있다면, 계속 읽어나가라.

4.1. dlopen()

dlopen(3)함수는 라이브러리를 열고 그것이 사용되도록 준비시켜준다. C에서 프로토타입은 다음과 같다:

```
void * dlopen(const char *filename, int flag);
```

파일이름이 "/"(즉, 절대경로)로 시작한다면 dlopen()은 즉시 사용하려 한다(라이브러리를 찾으려 하지 않는다). 그렇지 않으면, dlopen()은 다음과 같은 순서로 라이브러리를 찾는다:

1. 사용자의 LD_LIBRARY_PATH의 환경변수의 콜론으로 구분지어진 디렉토리
2. (/etc/ld.so.conf에서 파생된) /etc/ld.so.cache에 명시되어있는 라이브러리의 목록
3. /lib, /usr/lib. 순서를 주의하라; 이것은 예전의 a.out로더가 사용한 순서의 역이다. 예전의 a.out로더는 프로그램을 로드할때 /usr/lib을 찾고 /lib을 찾았다(ld.so(8)의 man페이지를 참고하라). 보통은 어떤 디렉토리나 다른디렉토리 하나에만 라이브러리가 있기 때문에(둘다 없을수도 있다) 문제가 되지 않는다. 그리고 같은이름을 가진 다른 라이브러리는 잠재적인 위험을 가지고 있다.

dlopen()에서 `flag`의 값은 "동적 라이브러리가 실행되면서 코드의 정의되지 않은 심볼을 처리하라"의 의미를 가진 RTLD_LAZY이거나, "dlopen()이 리턴하기 전에 모든 정의되지 않은 심볼을 처리하고, 그것이 되지 않을 경우 실패하라"의 의미를 지닌 RTLD_NOW가 있다. RTLD_GLOBAL은 `flag`의 값속에 옵션으로 쓸수 있다. 의미는 라이브러리에 적재되는 외부 심볼들은 그 후에 적재되는 라이브러리에 의해서 가능하다는 것이다. 디버깅시에, RTLD_NOW를 쓰기를 원할것이다; RTLD_LAZY는 정의되지 않은 심볼이 있으면 알아볼수 없는 에러를 내기 때문이다. RTLD_NOW를 쓰면서 라이브러리를 여는 것은 약간의 시간이 더 걸린다(하지만 나중에 찾는 속도는 빨라진다); 만약 이것이 유저 인터페이스 문제를 일으킨다면, 당신은 나중에 RTLD_LAZY로 바꿀 수 있다.

라이브러리들이 서로 의존한다면(예를들어 X가 Y에 의존한다면), 당신은 의존당하는 것을 먼저 로드해야한다(이 예제에서 Y를 로드하고 X를 로드해야 한다).

dlopen()의 리턴 값은 다른 DL라이브러리 루틴이 사용하기에 애매모호하게 느껴지는 "핸들"이다. dlopen()은 로드가 성공하지 못하거나 당신이 체크해야 할 필요가 있다고 생각하면 NULL을 리턴한다. dlopen()에 의해 같은 라이브러리가 한번이상 로드되면 같은 파일 핸들이 리턴된다.

라이브러리가 _init이라 이름지어진 루틴을 사용한다면(export), 그 코드는 dlopen()이 반환하기 전에 실행된다. 당신은 초기화 루틴을 구현하기위해 당신의 라이브러리에서 이 사실을 사용했을것이다. 더 자세한 내용을 위해 5.2절을 보아라.

4.2. dlerror()

에러는 dlerror()를 호출함으로써 보고될수 있다. dlerror()는 dlerror(), dlsym(), dlclose()중 마지막 부른것의 에러의 스트링을 반환한다. 하나의 이상한점은 dlerror()를 부르고 나서 dlerror()를 부르는 것은 그 사이에 다른 에러가 없으면 NULL을 반환한다는 것이다.

4.3. dlsym()

당신이 사용할 수 없다면 DL라이브러리를 로딩하는 의미가 없다. DL라이브러리를 사용하는 중요한 루틴은 dlsym(3)이다. 이것은 열려진 라이브러리의 심볼의 값을 찾아준다. 이 함수는 다음과 같이 정의된다:

```
void * dlsym(void *handle, char *symbol);
```

핸들은 dlopen에 의해 반환된 값이고, 심볼은 NIL로 끝나는 스트링이다. 당신이 할 수 있다면 있다면, dlsym()의 결과를 void* 포인터로 저장하지 마라. 왜냐하면, 당신은 사용할때마다 캐스트해서 사용해야 하기때문이다(그리고 다른사람들이 프로그램을 유지하는데에 더 적은 정보를 줄것이다).

dlsym()은 심볼이 없으면 NULL을 반환할 것이다. 만약 당신의 심볼에 NULL이나 0이 없다는 것을 알면 좋겠지만, 다음과 같은 잠재적인 문제가 있다: 당신이 받은 NULL이 에러인가, 아니면 심볼의 값이 NULL인가? 표준적인 해결법은 dlerror()를 부르고(전에 있던 에러들을 없앤다), dlsym()을 불러서 심볼의 값을 부르고, dlerror()를 불러서 에러가 났는지 체크한다. 단순한 코드는 다음과 같다:

```
dlerror(); /* clear error code */
s = (actual_type) dlsym(handle, symbol_being_searched_for);
if ((err = dlerror()) != NULL) {
 /* handle error, the symbol wasn't found */
} else {
 /* symbol found, its value is in s */
}
```

4.4. dlclose()

dlopen()의 반대는 dlclose()로서, DL 라이브러리를 닫아준다. DL 라이브러리가 동적 파일 핸들의 링크 수를 관리하기 때문에, 동적 라이브러리는 dlopen이 성공한 만큼 모두 dlclose를 불러주기 전에 다 할당이 없어지지는 않는다. 따라서, 같은 프로그램이 같은 라이브러리를 여러번 불러주는것은 문제가 되지 않는다. 라이브러리가 할당이 없어진다면, 함수의 _fini가 불린다(존재한다면); 더 자세한 정보를 위해 5.2절를 보아라. 주의하라 : dlclose()는 성공하면 0을 리턴하고, 아니면 0이 아닌 값을 리턴한다; 어떤 리눅스 메뉴얼은 이것을 언급하고 있지 않다.

4.5. DL 라이브러리 예제

여기에 있는 예제는 dlopen(3)의 맨페이지에 나오는 예제이다. 이 예제는 math라이브러리를 로드해서, 코사인 2.0을 출력한다. 또한, 각각의 단계에서 에러 체크를 한다(이렇게 하기를 추천한다).

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

int main(int argc, char **argv) {
    void *handle;
    double (*cosine)(double);
    char *error;

    handle = dlopen ("/lib/libm.so.6", RTLD_LAZY);
    if (!handle) {
        fputs (dlerror(), stderr);
        exit(1);
    }

    cosine = dlsym(handle, "cos");
    if ((error = dlerror()) != NULL)  {
        fputs(error, stderr);
        exit(1);
    }

    printf ("%f\n", (*cosine)(2.0));
    dlclose(handle);
}
```

만약 이 프로그램의 파일이름이 "foo.c"라면, 다음과 같은 명령으로 프로그램을 만들 수 있을 것이다:

```
gcc -o foo foo.c -ldl
```

5. Miscellaneous

5.1. nm 명령

nm(1)명령은 주어진 라이브러리의 심볼의 리스트를 보고한다. 이것은 정적, 공유 라이브러리에서 모두 동작한다. 주어진 라이브러리에서, nm(1)은 정의된 심볼의 이름, 각각의 심볼의 값, 심볼의 타입을 나열한다. 이것은 또한 라이브러리에서 가능하다면(-l옵션을 보라), 심볼이 소스코드의 어디에서 정의되었는지 알려준다(파일이름과 줄 번호를 표시함으로서).

심볼타입은 더 많은 설명을 요구한다. 타입은 한글자로써 표시된다; 소문자는 타입이 지역적이라는 것이고, 대문자는 타입이 전역적(외부적)이라는 것을 의미한다. 보통의 심볼타입들은 다음과 같다. T (코드섹션에서 보통의 정의), D (초기화 데이터 섹션), B (초기화되지 않은 데이터 섹션), U (정의되지 않음; 심볼이 라이브러리에의해 사용되지만 라이브러리에 의해 정의되지 않았다), W (약함; 다른 라이브러리가 이 심볼을 정의한다면, 그 정의가 이 정의를 오버라이드 할 수 있다).

당신이 함수의 이름을 알지만, 어떤 라이브러리에서 정의되있는지 기억하지 못한다면, grep으로 라이브러리를 찾으면서 nm의 “-o”옵션(각각의 줄의 파일 이름을 앞에 붙인다)을 사용할 수 있다. bash에서 당신은 /lib, /usr/lib와 /usr/lib, /usr/local/lib의 하부 디렉토리에서 “cos”를 다음과 같이 찾을 수 있다.

```
nm -o /lib/* /usr/lib/* /usr/lib/*/* \
      /usr/local/lib/* 2> /dev/null | grep 'cos$'
```

nm에 대해 더 많은 것을 알고 싶으면 info:binutils#nm에 “정보” 문서가 있따.

5.2. 특별함수 _init, _fini

두개의 특별한 함수가 초기화와 종결화의 모둘을 돕는다:_init와 _fini. “_init”함수가 라이브러리에 의해 내보내진다면, 라이브러리가 처음에 열어질때 불릴것이다(dlopen()이나 공유라이브러리의 형태로). C 프로그램에서, 이것은 당신이 _init이란 이름이 붙은 어떤 함수를 정의했다는 것을 의미한다. 이것에 대한 _fini함수가 있는데, 이것은 클라이언트가 라이브러리 사용을 끝낼때 불리는 함수이다(0나 프로그램의 정상종료를 가져오는 dlclose()의 형태로). 이 함수의 C 프로토타입은 다음과 같다:

```
void _init(void);
void _fini(void);
```

“.o”파일을 gcc에서 컴파일 할 때, “-nostartfiles”옵션을 넣어라. 이것은 C컴파일러가 .so파일에 대해 시스템 시작 라이브러리를 링크하도록 해준다. 그렇지 않으면, 당신은 “multiple=definition”에러를 만날 것이다. Jim Mischel과 Tim Gentry이 _init, _fini함수에 대한 논의에서 글을 쓰는데 도움을 줬을 뿐만 아니라 이런 제안을 해준것에 의해 감사한다.

5.3. 공유 라이브러리는 스크립트가 될 수 있다

GNU 로더가 특별한 스크립트 언어를 통해 보통의 라이브러리 형식대신 텍스트 파일의 형태로 공유 라이브러리를 유지하게 하는것은 의미가 없다. 이것은 간접적으로 다른 라이브러리를 합치게 됨으로서 유용하다. 예를들어, 나의 시스템중 하나에서 돌아가는 `/usr/lib/libc.so`의 리스트가 있다:

```
/* GNU ld script
   Use the shared library, but some functions are only in
   the static library, so try that secondarily.  */
GROUP ( /lib/libc.so.6 /usr/lib/libc_nonshared.a )
```

이것에 대해 더 많은 정보를 얻기위해 ld링커 스크립트를 보아라(ld명령어). 일반적인 정보는 info:ld#Options와 info:ld#Commands에서 얻을수 있고, info:ld#Option Commands에서 명령어의 정보를 얻을수 있다.

5.4. GNU libtool

만약 당신이 다른 많은 시스템에 포팅할 응용프로그램을 제작한다면, 당신은 라이브러리를 만들고 설치하기 위해 GNU libtool (http://www.gnu.org/software/libtool/libtool.html)를 사용하는 것을 고려해야 할 것이다. GNU libtool은 스크립트를 지원하는 일반적인 라이브러리이다. Libtool은 일관적이고 포팅가능한 인터페이스속에 공유라이브러리의 복잡도를 줄여준다. Libtool은 오브젝트 파일을 만들고, (정적, 공유)라이브러리를 링크하고, 실행파일을 링크하고, 실행파일을 디버그하고, 라이브러리를 설치하고, 실행파일을 설치하는데 포팅가능한 인터페이스를 제공한다. 이것은 또한 libltdl을 포함하는데, 동적 적재 프로그램의 포팅가능한 감싸는 역할을 한다. 더 많은 정보를 위해서 http://www.gnu.org/software/libtool/manual.html의 문서를 참고하라.

5.5. 공간을 위해 심볼을 제거하기

생성된 파일에 포함되는 모든 심볼은 디버깅에 유용하지만, 너무 많은 공간을 차지한다. 만약 공간이 필요하다면, 그것들의 일부를 지워라.

최선의 접근법은 오브젝트 파일을 정상적으로 생성하고, 디버깅과 검사를 먼저한다(이것들을 가지고 디버깅, 검사를 하는것은 매우 편하다). 그리고나서, 프로그램을 완전히 검사했으면, strip(1)을 사용해서 심볼들을 지워라. strip(1) 명령은 심볼을 지우는데 많이 편리하게 해준다; 자세한것은 그것의 문서를 참조하라.

다른, GNU ld옵션의 다른 것으로는 “-S”와 “-s”가 있다; “-S”는 출력파일에서 (모든 심볼이아닌)디버거 심볼 정보를 빠뜨린다. “-s”옵션은 출력파일에 모든 심볼 정보를 써 넣는다. “-Wl, -S”와 ”-Wl,-s”로서 gcc를 통해 이 옵션을 쓸 수 있다. 당신이 항상 심볼을 제거하고, 이 옵션이 불필요한것이라면, 그냥 편하게 느껴라. 하지만, 이것은 덜 유연한 방법일 것이다.

5.6. 매우 작은 실행화일들

당신은 Whirlwind Tutorial on Creating Really Teensy ELF Executables for Linux (http://www.muppetlabs.com/~breadbox/software/tiny/teensy.html)문서가 유용하다는 것을 알것이다. 이것은 매우작은 프로그램들이 어떻게 만드는지 설명한다. 사실, 일반적인 상황에서 이런 트릭을 사용할 필요는 없다. 하지만 이것은 ELF가 어떻게 동작하는지 보여주는 좋은예이다.

5.7. C++초기화 속도 높이기

KDE 개발자는 많이 재 배치를 해야하게때문에 큰 GUI C++응용프로그램이 시작하는데 오래걸린다는 것을 안다. 이것에 여러가지 해결책이 있다. 더 많은 정보를 위해 Making C++ ready for the desktop (by Waldo Bastian) (http://www.suse.de/~bastian/Export/linking.txt)를 참고하라.

6. 그외의 예제들

다음의 예제들은 모든 3가지 방법들(정적, 공유, 동적 적재 라이브러리)의 예제들이다. libhello.c는 보통의 라이브러리이고, libhello.h는 헤더파일이다. demo_use.c는 보통의 라이브러리 호출 파일이다. 다음으로, 정적, 공유 라이브러리를 어떻게 사용하는지 보여주는 script_static, script_dynamic 스크립트가 나온다. 그리고, 공유 라이브러리를 동적 적재 라이브러리로 어떻게 사용하는지 보여주는 demo_dynamic.c, script_dynamic이 나온다.

6.1. File libhello.c

```
/* libhello.c - demonstrate library use. */

#include <stdio.h>

void hello(void) {
  printf("Hello, library world.\n");
}
```

6.2. File libhello.h

```
/* libhello.h - demonstrate library use. */
```

```
void hello(void);
```

6.3. File demo_use.c

```
/* demo_use.c -- demonstrate direct use of the "hello" routine */

#include "libhello.h"

int main(void) {
 hello();
 return 0;
}
```

6.4. File script_static

```
#!/bin/sh
# Static library demo

# Create static library's object file, libhello-static.o.
# I'm using the name libhello-static to clearly
# differentiate the static library from the
# dynamic library examples, but you don't need to use
# "-static" in the names of your
# object files or static libraries.

gcc -Wall -g -c -o libhello-static.o libhello.c

# Create static library.

ar rcs libhello-static.a libhello-static.o

# At this point we could just copy libhello-static.a
# somewhere else to use it.
# For demo purposes, we'll just keep the library
# in the current directory.

# Compile demo_use program file.

gcc -Wall -g -c demo_use.c -o demo_use.o

# Create demo_use program; -L. causes "." to be searched during
# creation of the program.  Note that this command causes
# the relevant object file in libhello-static.a to be
```

```
# incorporated into file demo_use_static.

gcc -g -o demo_use_static demo_use.o -L. -lhello-static

# Execute the program.

./demo_use_static
```

6.5. File script_shared

```
#!/bin/sh
# Shared library demo

# Create shared library's object file, libhello.o.

gcc -fPIC -Wall -g -c libhello.c

# Create shared library.
# Use -lc to link it against C library, since libhello
# depends on the C library.

gcc -g -shared -Wl,-soname,libhello.so.0 \
    -o libhello.so.0.0 libhello.o -lc

# At this point we could just copy libhello.so.0.0 into
# some directory, say /usr/local/lib.

# Now we need to call ldconfig to fix up the symbolic links.

# Set up the soname.  We could just execute:
# ln -sf libhello.so.0.0 libhello.so.0
# but let's let ldconfig figure it out.

/sbin/ldconfig -n .

# Set up the linker name.
# In a more sophisticated setting, we'd need to make
# sure that if there was an existing linker name,
# and if so, check if it should stay or not.

ln -sf libhello.so.0 libhello.so

# Compile demo_use program file.

gcc -Wall -g -c demo_use.c -o demo_use.o
```

```
# Create program demo_use.
# The -L. causes "." to be searched during creation
# of the program; note that this does NOT mean that "."
# will be searched when the program is executed.

gcc -g -o demo_use demo_use.o -L. -lhello

# Execute the program.  Note that we need to tell the program
# where the shared library is, using LD_LIBRARY_PATH.

LD_LIBRARY_PATH="." ./demo_use
```

6.6. File demo_dynamic.c

```
/* demo_dynamic.c -- demonstrate dynamic loading and
   use of the "hello" routine */


/* Need dlfcn.h for the routines to
   dynamically load libraries */
#include <dlfcn.h>

#include <stdlib.h>
#include <stdio.h>

/* Note that we don't have to include "libhello.h".
   However, we do need to specify something related;
   we need to specify a type that will hold the value
   we're going to get from dlsym(). */

/* The type "simple_demo_function" describes a function that
   takes no arguments, and returns no value: */

typedef void (*simple_demo_function)(void);


int main(void) {
 const char *error;
 void *module;
 simple_demo_function demo_function;

 /* Load dynamically loaded library */
 module = dlopen("libhello.so", RTLD_LAZY);
 if (!module) {
   fprintf(stderr, "Couldn't open libhello.so: %s\n",
           dlerror());
```

```
   exit(1);
 }

 /* Get symbol */
 dlerror();
 demo_function = dlsym(module, "hello");
 if ((error = dlerror())) {
   fprintf(stderr, "Couldn't find hello: %s\n", error);
   exit(1);
 }

 /* Now call the function in the DL library */
 (*demo_function)();

 /* All done, close things cleanly */
 dlclose(module);
 return 0;
}
```

6.7. File script_dynamic

```
#!/bin/sh
# Dynamically loaded library demo

# Presume that libhello.so and friends have
# been created (see dynamic example).

# Compile demo_dynamic program file into an object file.

gcc -Wall -g -c demo_dynamic.c

# Create program demo_use.
# Note that we don't have to tell it where to search for DL libraries,
# since the only special library this program uses won't be
# loaded until after the program starts up.
# However, we DO need the option -ldl to include the library
# that loads the DL libraries.

gcc -g -o demo_dynamic demo_dynamic.o -ldl

# Execute the program.  Note that we need to tell the
# program where get the dynamically loaded library,
# using LD_LIBRARY_PATH.

LD_LIBRARY_PATH="." ./demo_dynamic
```

7. 그 외의 정보들

라이브러리에 대한 유용한 정보는 다음과 같은 것들이 있다:

- Daniel Barlow의 “The GCC HOWTO”. 특별히 이 HOWTO는 라이브러리를 만들고 질의하는 컴파일러 옵션을 다룬다. 이것은 여기서 다루지 못한 정보를 다룬다. 이 HOWTO는 http://www.linuxdoc.org에서 볼 수 있다.
- TTS(the Tool Interface Standards)위원회의 “Executable and Linkable Format (ELF)”(이 문서는 사실 이 위원회의 포팅가능한 형식 스펙 v 1.1의 한 챕터이다). 이것은 ELF형식에 관한 정보를 제공하고, ELF형식에 관한 자세한 많은 것을 제공한다. ftp://tsx-11.mit.edu/pub/linux/packages/GCC/ELF.doc.tar.gz를 참고하라. 만약 MIT로부터 파일을 얻는다면, 형식이 이상함에 주의하라; gunzip, untar를 한 후에, 이름을 “ps”파일로 바꾸면, 보통의 ps파일이 될 것이다.
- Hongjui Lu의 “ELF: From the Programmer’s Perspective”. 이것은 ELF에 대해 리눅스, GCC gcc의 스펙 정보를 보여준다. 또한 이 문서는 ftp://tsx-11.mit.edu/pub/linux/packages/GCC/elf.ps.gz에서 구할 수 있다.

8. Copyright and License

이것들은 다른 웹사티트에서 미러링을 허락한다. 하지만, 다음것들을 부탁한다:

- 너의 미러 사이트가 원래 사이트의 업그레이드를 자동적으로 하게하고,
- 원본 사이트http://www.dwheeler.com/program-library가 어디인지 하이퍼 텍스트로 게시하고,
- 작가인 나((David A. Wheeler) 에게 알려주어라.

나를 주로 귀찮게 하는 2가지중의 하나는 지나간 버그들에 대한 이야기를 계속 하는 것이다. 당신이 이 문서의 미러링을 제대로 하고 있다면, 나는 당신에게서 몇년전에 고쳐진 버그에 관한 얘기를 듣지 않아도 된다. 원본 사이트를 미러한다면, 사용자들은 당신의 미러사이트가 새

로운것인지 체크할 수 있다. 나는 매우 보안 문제에 신중하기때문에 일반적인 인터넷접속은 하지 않는다; 만약 이것이 당신에게 문제가 된다면, 다른 방법을 찾아서, 당신의 환경에서 때때로 체크를 하면서 업데이트를 하라.

이 라이센스에 의해, 당신이 문서를 변경할 수 있다. 하지만, 당신이 쓰지 않은것을 당신꺼라고 할 수 없다(따라서, 표절이다). 또한, 개정판이 원본과 같은것이라고 할 수도 없다. 작업을 개정함으로서, 전부가 당신의 판권은 아니다; 이것은 판권의법에 의한 "공공의"일은 아니다. 자세한것은 라이센스를 찾아보라, 특별히 "당신은 파일이 바뀐것에대한 큰 변화를 쓰고, 언제 바뀌었는지 개정된 파일에 명시하라"라는 것을 주의하라. 라이센스가 무엇을 허용하는지 알고 싶으면, 나에게 연락하라. 대부분의 경우, 당신이 바꾼것을 주 집필자(현재는 David A. Wheeler)에게 보내는 것이 좋다. 그러면, 당신이 변화시킨것이 원본에 추가되어서 모든 사람이 보게 될것이다.