프로그램 라이브러리 하우투

David A. Wheeler지음

버전 0.60, 2000년 4월 27일

번역 : 황 진 희(*jeenee@soback.kornet.net* <mailto:jeenee@soback.kornet.net>)

2000년 5월

이 하우투 문서는 프로그래머가 어떻게 리눅스환경에서 프로그램 라이브러리 를 생성하고 사용할수 있는지를 다루고 있으며 라이브러리에는 정적 라이브러 리, 공유 라이브러리 및 동적(DL) 라이브러리가 있다.

목 차

제 1 절 개요

이 하우투 문서는 프로그래머가 어떻게 GNU의 툴셋을 이용하여 리눅스에 서 프로그램 라이브러리를 생성하고 이용할수 있는지를 다룬다. 프로그램 라이 브러리란 단순하게 보면 나중에 어떤 프로그램 코드에 끼워넣어질수 있는 컴 파일된 코드(그리고 데이터)가 들어있는 파일이며, 이를 통해서 프로그램들은 더욱 모듈화될수 있으며, 재컴파일이 빠르며, 갱신이 용이해진다. 프로그램 라 이브러리는 정적, 공유, 동적(DL) 라이브러리의 세가지 형태로 대별된다. 먼저 프로그램이 실행되기전에 미리 실행프로그램에 설치되는 정적 라이브 러리가 다루어 질것이며, 그 다음은 프로그램 시작시에 적재되어 여러 프로그 램에 의해 공유되는 공유 라이브러리, 그리고 마지막으로 프로그램 실행중의 어떤 순간에도 적재 및 사용이 가능한 동적 라이브러리를 다룰 것이다. 동적 라이브러리를 다른 어떤 한 라이브러리의 포맷이라고 보기는 어려울 것이다.(정적 라이브러리이건 공유 라이브러리이건 모두 동적 라이브러리로 사용 가능함) 대신 동적 라이브러리가 다른점은 그것이 프로그래머에 의해서 이용 되는 방식에 있는 것이다. 이 책의 다른 장에는 그것의 예들과 참고할 만한 문 헌들을 실을 것이다.

이 책은 이제는 대부분의 리눅스 배포본에서 사용되는 ELF 포맷의 실행 프로그램과 라이브러리만을 다룰 것이다. GNU의 gcc 툴셋은 ELF외에 다른 형태의 라이브러리도 다룰수 있는데 아직도 대부분의 리눅스 배포판은 이미 퇴물이된 a.out형의 포맷을 다루고는 있다, 하지만 이것은 이책의 범위를 벗 어난 얘기이다.

어떤 사람들은 동적 링크 라이브러리(DLL)란 용어를 사용하여 공유라이브 러리를 칭하기도 하고, 때론 정적이건 공유건 동적 라이브러리로 사용될 때 DLL이라 칭하기도 하고 또, 때로는 위의 두 조건을 모두 만족할 때 DLL이란 용어를 쓰기도 하나 이것은 별로 중요한 얘기가 아니다. 이문서는 당신이 어 떤 의미를 선택하건간에 리눅스에서의 DLL을 다뤄줄 것이다.

만약 여러 시스템에 포팅될수 있는 프로그램을 개발하려 한다면 LINUX Tool을 바로 사용하는 것 보다는 GNU Libtool의 사용을 고려해 보길 바란다. GNU Libtool은 공유 라이브러리의 사용(생성과 설치등)의 복잡성을 일관되고 이식적인 인터페이스를 통해서 제거해주는 전형적인 라이브러리 지원 스크립 트이다. 리눅스에서 GNU Libtool은 Tool들중에서 최상위에 구현되어 있으며 그 사용관례도 이 하우투 문서에 기술되어있다. DL 라이브러리의 이식가능 한 인터페이스 구현을 위해서는 여러가지 이식성 구현장치가 있으며 그 중 GNU Libtool 은 "Libltdl" 이 포함되어 있다. 다른 선택으로는 glib 라이브러리 (glibc와 혼동하지말것)의 동적 모듈 적재의 이식성지원기능을 사용할수 있다. 그것을 사용하려 한다면 이 문서말고 [10]*developer.gnome.org/doc/API/glib/glib-dynamic-loading-of-modules.html* `<http://developer.gnome.org/doc/API/glib/glib-dynamic-loading-of-modules.html>` 에서 보길 바란다. 다시 한번 말 하건데 리눅스에서 라이브러리의 사용에 관한 제반 기능들의 구현이 이 문서 에서 기술될 것이며 리눅스용 코드를 개발중이거나 디버깅 중이라면 이문서의 내용이 유용할것이다.

이 하우투 문서의 메인주소는 [11]*www.dwheeler.com/program-library* `<http://www.dwheeler.com/program-library>` 이며 LDP([12]*http://www.linuxdoc.org* `<http://www.linuxdoc.org>`)에 제공된다. 저작권자는 David A. Wheeler이며 GPL의 라이센스를 따른다.

제 2 절 정적 라이브러리

정적 라이브러리는 단지 통상의 오브젝트 화일을 모은것이다. 통상적으로 정 적 라이브러리는 .a라는 접미어로 끝난다. 이 오브젝트 집합은 ar(archiver) 프로그램에 의해서 생성된다. 정적 라이브러리는 공유 라이브러리가 가지는 장 점들 때문에 이제는 옛날처럼 그렇게 많이 사용되어 지지는 않는다. 그렇지만 아직도 사용되어 지고 있고 역사적으로 가장먼저 존재했고 설명이 간단하다. 정적 라이브러리는 재컴파일없이 프로그램에 링크될수 있기 때문에 재컴파일 에 필요한 시간이 절약될수 있다. 요즘은 컴퓨터의 빠른 실행시간때문에 이런 재컴파일에 드는 시간 절약은 옛날처럼 그렇게 중요한 이유가 될수는 없을 것이다. 정적 라이브러리는 개발자들이 소스코드의 재공없이 프로그래머들로 하여금 링크하여 사용하도록 할때는 가끔씩 유용한 것이된다. (이때 라이브러 리의 판매자는 이익이지만 라이브러리를 이용하고자 하는 프로그래머들에게 는 유용한것이 아니다.) 이론상으로 실행파일에 링크되는 정적 ELF 라이브러 리 코드는 1-5%정도 빨리 수행되어져야하나 실제에 있어서는 다른 여러가지 복합적인 이유로 인하여 이런 예는 잘 맞지 않는것 같다.

정적 라이브러리를 생성하거나 기존의 정적 라이브러리에 부가적 오브젝트 파일을 추가할 경우 다음의 명령을 사용한다.

ar rcs my_library.a file1.o file2.o 위의 예의 명령은 file1.o file2.o라는 오브젝트 파일을 mylibrary.a에 추가하거나 존재하지 않을경우 이 파일을 생성한다. 정적 라이브러리 생성과 관련한 더 이상의 정보는 ar(1)을 보기 바란다.

정적 라이브러리를 당신이 만들었다면 사용하기를 원할 것이다. 이를 사용하 려면 실행 프로그램을 생성하기 위한 컴파일과 링크과정의 한 부분에 이 파 일을 포함시키면 된다. 만약에 gcc(1)을 사용하여 실행파일을 만든다면 -l 옵 션을 사용하여 당신이 만든 라이브러리를 명시하면 된다. (info:gcc을 참조하 면 더 많은 정보를 얻을수 있음). 링커인 ld(1)를 바로 사용하여도(-l,똔느 -L 옵션) 되나 ld(1)의 인터페이스가 자주 변하므로 gcc(1)을 쓰는 것이 더 나 을 것이다.

제 3 절 공유 라이브러리

공유 라이브러리란 프로그램이 시작될 때 이 프로그램에 의해서 적재되는 라 이브러리를 말한다. 어떤 공

유 라이브러리가 바르게 적재었다면 이후에 실행되 는 모든 프로그램들은 자동적으로 이 새로운 공유 라이브러리를 사용할 것이 다. 실재에 있어서, 리눅스에서 사용되는 접근방식은 아래와 같은 기능들을 허 용하므로 더욱 탄력적이며 또한, 세련되게 기능한다.

* 라이브러리를 갱신하면서 이 라이브러리의 구버전 지원용 버전이 없이도 여전히 구 버전의 라이브러리의 사용을 원하는 프로그램을 지원할수 있다.

* 특정 프로그램의 수행시 라이브러리 또는 심지어 라이브러리의 특정함수의 오버라이딩(대치)이 가능하다.

* 이 모든 기능을 기존의 라이브러리를 이용하여 프로그램 실행중에 가능하 다.

관 례

공유 라이브러리가 이러한 기능들을 지원하기 위해서는 사용자가 반드시 따 라야할 이용상의 몇가지 관례와 가이드라인이 있다. 당신은 한 라이브러리를 일컫는 이름들의 차이 특히 "soname"과 "real name"을 구별할수 있어야하 며, (또한 이들이 어떻게 상호작용 하는가도 알아야 한다) 또한, 라이브러리가 파일 시스템상의 어디에 위치해야 하는가도 알아야 한다.

공유 라이브러리 이름들

모든 공유 라이브러리는 "soname"이라 불리는 특별한 이름을 가지고 있다. "soname"은 접두어 "lib", 라이브러리 이름, ".so"라는 문구, 그 다음 마침표와 인터페이스 변경시 증가되는 버전 넘버로 구성된다. (예외로 가장 저수준의 C라이브러리들은 "lib"로 시작되지 않는다) 완전한 공식적 "soname"에는 라이 브러리가 속한 디렉토리명이 앞부분에 들어간다. 운용중인 시스템에서는 완전 한 공식적 "soname" 이란 공유라이브러리의 실제 이름("real name")에 대한 심볼릭 링크(실제 코드를 담고있는 파일이 아닌)가 될 것이다.

모든 공유 라이브러리는 실제 라이브러리 코드가 저장되어 있는 파일의 파일 명인 "real name"을 또한 가진다. 실제 파일명은 "soname"에 마침표와 마이너 버전 넘버, 마침표 그리고 릴리즈 넘버를 더한 것이다. 릴리즈 넘버는 선택사 항이다. 마이너 넘버와 릴리즈 넘버로 정확히 어떤 버전의 라이브러리가 설치 되어 있는지 알게 함으로써 시스템 구성을 조절할수 있다. 이 번호들은 라이브 러리를 기술하기위한 문서화시에 사용된 번호와 맞지 않을수도 있음에 유의하 라. 이렇게 함으로써 라이브러리의 문서화가 더 용이해진다.

부가적으로 컴파일러가 라이브러리를 요구할 때 사용하는 이름이 하나 더 존 재한다.(나는 이를 "linker name"이라 일컬을 것이다.) 여기에는 어떠한 버전넘 버도 포함되지 않는 "soname"이 해당된다. 공유라이브러리를 관리하는 핵심은 이러한 이름들을 잘 분별하는 것이다. 프 로그램에서 필요한 공유 라이브러리를 내부적으로 표현할때는 그 라이브러리 의 "soname"을 써야한다. 반대로 만약에 당신이 공유 라이브러리를 만들었다 면 자세한 버전 정보를 포함한 실제 파일명을 주어야 한다. 당신이 새로운 버 전의 라이브러리를 설치할때는 몇 개의 정해진 디렉토리들중의 한곳에 설치한 뒤에 ldconfig라는 프로그램을 실행하여야 한다. ldconfig는 디렉토리에 존재하 는 파일들을 점검한 뒤 실제 파일명의 심볼릭 링크로써 soname을 생성하며 동시에 캐쉬 파일인 /etc/ld.so.cache를 셋팅한다.

ldconfig는 linker name을 셋팅하지 않는다. 이것은 보통 라이브러리의 설치 시에 셋팅을 하는데, 간단하게 가장 최근의 soname이나 real name에 심볼릭 링크를 만들면 된다. 대부분의 경우 라이브러리를 갱신하고 자동적으로 링크시 에 사용하기를 원할것이므로 soname에 심볼릭 링크하기를 권장한다. H. J. Lu 에게 ldconfig가 왜 자동으로 linker name을 셋팅하지 않느냐고 물었더니 그 의 설명은 가장 최근의 라이브러리로 링크시키고 싶을때가 있을것이며 또한 이전 버전의 라이브러리를 개발하거나할때 그것을 링크하려 할 때가 있으므 로 ldconfig는 여러분이 어떤 라이브러리와 링크하고 싶은지 가정하지 않는 것이며 그래서 설치자는 어떤 라이브러리가 링크될것인지를 명시해야 하는것 이라 했다.

이상의 얘기를 종합한 예를 들어보면 ldconfig가 /usr/lib/libreadline.so.3.0과 같은 real name의 심볼릭 링크로 셋팅한 /usr/lib/libreadline.so.3 형태의 완 전한 공식적 soname이 있을것이며 /usr/lib/libreadline.so.3에 심볼릭 링크된 link name으로써 /usr/lib/libreadline.so가 또한 있을 것이다.

파일 시스템상의 위치

공유 라이브러리는 파일 시스템 상의 특정위치에 존재해야한다. 오픈 소스 소 프트웨어들은 GNU의 표준을 따르는 경향이 있으며 더 자세한 내용은 info문 서의 [13]info:standards#Directory_Variables 부분을 살펴보길 바란다. GNU 표 준은 소스코드의 배포시에 모든 라이브러리를 /usr/local/lib에 설치하길 권 장한다(/usr/local/bin에는 명령들이 모임). GNU표준은 또한 이러한 디폴트들 을 무시하거나 설치루틴을 호출하는 절차 등에 관한 관례들을 정의하고 있다. 파일시스템 계층적 구조표준(FHS:File system Hierarchy Standard) 또한 소 스배포시에 어떤 위치에 무엇을 두어야 하는지에 대해 논하고 있다.(*www.pathname.com/fhs* `<http://www.pathname.com/fhs>` 를 참조하기 바람). FHS에서는 대부분의 라이브러리를 /usr/lib에 설치할 것을 권장한다, 그러나 시스템 시작시 필요한 라이브러리는 /lib에 또한, 비시스템 라이브러리는 /usr/local/lib에 설치되어야 한다.

위의 두 표준이 서로 모순되는 듯 보이지만 실제로 그렇지는 않다. GNU의 표준은 소스코드의 개발자를 위한 디폴트값을 제안하는 것이고, FHS는 소스 의 배포자(시스템의 디폴트 소스코드를 선택적으로 그 시스템의 패키지 관리 도구를 통해서 대치시키는)를 위한 디폴트값을 제안하는 것이다. 실제로 이것 은 상당히 효과적인 것이다, 즉, 당신이 다운로드한 가장 최신의(버그의 가능 성이 있는) 소스코드는 자동적으로 로컬 디렉토리인 /usr/local에 설치될것이며 그 코드가 이제 완벽해지면 패키지 매니저를 이용하여 시스템에 설치된 디 폴트(라이브러리)를 대치하고 배포시의 표준 디렉토리로 설치할수 있는 것이 다. 라이브러리를 통해서만 호출가능한 프로그램들을 라이브러리가 호출하는 경우에는 반드시 그 프로그램들을 usr/local/libexec(배포본에서는 /usr/libexec 임)에 두어야함을 명심하라. 한가지 혼동은 레드햇의 배포본들은 라이브러리를 찾는 과정에서 /usr/local 디렉토리를 제외 시킨다는 것인데, 아래에서 논의 될 /etc/so.conf부분을 참고하길 바란다. 그외에 또다른 하나의 표준 라이브러리 디렉토리가 있는데 X-윈도우 시스템이 사용하는 /usr/X11R6/lib이 그 것이다. /lib/security 디렉토리는 PAM모듈에서 사용됨에 유의하라. 이 디렉 토리는 통상 동적 라이브러리로써 적재된다.(역시 뒷부분에서 논의될것임)

라이브러리는 어떻게 사용되는가

모든 리눅스 시스템이 포함되는 GNU의 glibc-기반의 시스템은 ELF 이진 실 행 프로그램의 시작시에 자동적으로 프로그램 로더를 적재하고 실행시킨다. 리눅스 시스템에서 이 로더는 /lib/ld-linux.so.X(X는 버전 넘버임)이라 불리워 지며 프로그램에서 사용되는 모든 공유 라이브러리를 차례대로 찾아서 적재시 킨다.

탐색되어질 디렉토리의 목차는 /etc/ld.so.conf에 들어 있다. 레드햇에 기반한 배포본에는 일반적으로 /usr/local/lib가 /etc/ld.so.conf파일에 들어 있지않은데 저자가 생각하기로 이것은 일종의 버그가 아닌가 생각한다. 레드햇의 경우 /usr/local/lib을 /etc/ld.so.conf에 첨가하는 것은 많은 프로그램들을 수행시키 기위해 흔히 있는 수정작업이다.

만약 라이브러리내의 특정한 함수만을 대치시키고 나머지는 그대로 유지하려 할때는 대치시킬 라이브러리(.o파일들)들을 /etc/ld.so.preload 파일에 명시하면 되는데 이러한 프리로드 라이브러리들은 표준적인 함수보다 우선권을 가지게 된다. 이러한 프리로딩 파일들은 보통 응급 패치본에서 사용되는데 정상적 배포본에는 잘 포함되지 않는다.

매 프로그램의 시작시마다 이 모든 디렉토리들을 탐색하는 것은 전반적으로 매우 비효율적이기 때문에 실제적으로는 캐쉬가 사용된다. ldconfig 프로 그램은 통상 /etc/ld.so.conf를 읽어서 모든 필요한 동적 링크 디렉토리들로부 터 적당한 심볼릭 링크를 생성한다.(그래서 이름 사용의 규칙을 잘 따라야 한다) 그 다음으로 /etc/ld.so.cache 에 이 내용들을 기록하며 이 다음 부터 는 프로그램들이 이 파일을 이용하는 것이다. 이렇게 해서 라이브러리의 액세 스 속도는 상당히 향상되는 것이다. 위의 내용이 함축하는 바는 DLL이 추가 또는 제거될 때, 또는 DLL의 디렉토리가 변경될때는 반드시 ldconfig가 실행되어야 한다는 것이며, 그래서 패키지 매니저가 라이브러리를 설치할 때, 이 과정은 통상으로 수행되는 절차인 것이다. 그래서 프로그램 시 작시 동적 로더는 실제로는 /etc/ld.so.cache를 사용하여 필요한 라이브러리를 적재하는 것이다.

환경 변수

여러 종류의 환경변수들이 위의 이러한 프로그램 시작시의 적재과정들을 조 절할수 있으며 또한 이 과정들을 대치하도록 하는 환경변수들도 있다. 예를 들 면 이 수행과정중, 어떤 라이브러리를 잠정적으로 교체하여 수행할수도 있다. 리눅스에서 환경변수 LD_LIBRARY_PATH는 콜론으로 분리되어 있

는 표준 적인 라이브러리셋에 앞서서 가장 먼저 탐색되어질 라이브러리 디렉토리집합 이다. 이렇게 하면 새로운 라이브러리를 디버깅하거나 비표준적인 라이브러리 를 어떤 특수한 용도로 사용할 때에 유용할것이다. 환경변수 LD_PRELOAD 는 /etc/ld.so. preload처럼 표준적인 함수들에 오버라이드될 함수들이 있는 오 브젝트 파일의 리스트를 나타낸다. 위의 이러한 기능들은 로더 /lib/ld-linux.so 가 구현한다.

적재과정을 제어하는 다른 종류의 환경변수들이 있다; 이름이 LD_ 또는 RTLD_로 시작한다. 이러한 변수들의 대부분은 적재과정의 저수준 디버깅이나 특수기능의 구현을 위해서 사용되어 진다. 이들 대부분이 잘 문서화되지 않았 기 때문에 자세히 알려면 소스코드를 읽어나가는 것이 최선의 방도일 것이다. 이렇게 동적링크 라이브러리에 대한 제어권을 사용자에게 허락하는 것은 특 별한 조치가 취해지지 않는다면 setuid/setgid 등의 프로그램에 있어서는 대단 히 위험한 것이다. 그래서 GNU로더의 경우 만약에 프로그램이 setuid 또는 setgid일 경우에는 이러한 환경변수들(또는 이와 유사한 다른 변수들)은 무시 되거나 아니면 할수있는 기능이 대단히 제한된다. 로더는 프로그램이 setuid 또는 setgid인지 점검하기위해 프로그램의 credential를 체크한다; uid와 euid(effective user ID)가 다르거나 gid와 egid(effective group ID)가 다른경 우에 로더는 이 프로그램을 setuid/setgid(또는 상속받은것)이라 가정하고 링 크과정의 제어기능을 대단히 제한하게 된다. GNU의 glibc라이브러리의 소스코 드를 읽어보면 이러한 내용을 잘 볼수 있을것이며 특히 elf/rtld.c, sysdeps/ generic/dl-sysdep.c를 보길 바란다. 이것이 뜻하는 바는 만약에 당신이 uid, gid를 euid, egid와 같게하고 프로그램을 호출하면 이러한 환경변수들은 완전 한 영향력을 미칠수 있다는 뜻이다. 다른 유닉스 유사 시스템들에서는 이러한 상황을 다르게 다루기도 하지만 같은 이유로 인해서 setuid/setgid 프로그램 은 환경변수에 의해서 잘못된 영향을 받아서는 않되는 것이다.

공유 라이브러리의 생성

공유 라이브러리의 생성은 쉽다. 먼저 gcc의 fPIC플래그("position inde- pendent code" 공유라이브러리로 사용할 때 필수)를 사용하여 오브젝트 파일 을 생성한다. 그 다음엔 다음과 같은 명령형태로 공유 라이브러리를 생성한다.

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

두 개의 오브젝프 파일(a.o, b.o)을 포함하는 하나의 공유 라이브러리를 생성 하는 예를 들면 아래와 같다. 컴파일시에 공유 라이브러리에 필요한 것은 아니 지만 권고되는 사항인 디버깅 정보(-g옵션)와 경고(- wall옵션)의 생성에 유의 하라, 그리고 -c옵션으로 오브젝트 파일이 생성되며 분명히 -fPIC 옵션이 사 용되고 있다.

```
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
```

이때 주의사항 몇가지를 적어보며 아래와 같다.

* 컴파일 결과 생성될 라이브러리를 명기하는 것을 잊지 말것이며 -fomit, -frame, -pointer 등의 컴파일러 옵션은 진짜 필요할 경우외엔 쓰지 말 것. 결과적으로 생성된 라이브러리는 동작하지만 이렇게 하면 디버거는 무용지물 이 된다.

* -fpic가 아닌 -fPIC를 사용하여 코드롤 생성시켜야 한다.(-fpic를 쓸 경우 분 기점에서 큰 위치변화가 필요할경우 완전한 위치 독립형 코드를 얻을수 없다.)

공유 라이브러리의 설치 및 사용

이제 만약에 당신이 하나의 공유 라이브러리를 만들었다면 그 다음으로 설치 를 원할 것이다. 가장 간단한 방법은 만들어진 공유 라이브러리를 표준 디렉토 리들중의 하나(예를 들면 /usr/lib)에 복사하고 ldconfig(8)를 실행시키는 것이 다.

만약에 (/usr/lib 디렉토리의 변경권을 가지지 못하거나 해서) 위처럼 할수 없 는 경우가 있다면 환경변수를 사용하여 조정할수 있다. 먼저 어딘가에 공유 라 이브러리를 생성해야한다. 그 다음에 필요한 것은 real name에 관한 심볼릭 링크로써 soname을 생성(또는 버전명을 전혀 명시하지 않는 사용자을 위한 버전명이 포함되지 않은 soname을 심볼릭 링크로 생성)하는 것인데 이를 위 한 가장 간단한 방법은 다음과 같이 ldconfig를 실행하는 것이다.

ldconfig -n directory_with_shared_library

그 다음으로 표준 라이브러리 보다 앞서서 탐색될 라이브러리 디렉토리 목록 인 LD_LIBRARY_PATH 변수를 설정한다. 만약에 bash라면 다음과 같이 하 여 my_program을 포함되게 할수 있다.

LD_LIBRARY_PATH=.:$LD_LIBRARY_PATH my_program

만약 특정한 함수만을 오버라이드 하려면 오버라이딩 오브젝트 파일을 생성 한 뒤에 LD_PRELOAD를 설정해서 할수 있다. 즉, 이 파일에 있는 함수들이 기존 함수를 오버라이드한다.

일반적으로 라이브러리를 갱신하는데 있어 특별한 고려사항은 없다. 만약 API가 변화한다면 라이브러리의 제작자는 soname을 변경할 것이다. 같은 soname을 가지면서 내용은 갱신된 신버전의 라이브러리를 무시하고 구버전의 라이브러리를 사용하려면, 구버전의 라이브러리를 어딘가에 복사하고 그것의 이름을 변경(이를테면 옛날이름.orig등)한 뒤 이 이름 변경된 라이브러리를 실 제로 호출하여 사용하도록 재설정하는 작은 wrap스크립트를 작성하여 강제적 으로 구버전을 사용하게 할수있다. 관례상 여러버전의 라이브러리들이 같은 디 렉토리에 있을수 있지만 이 구버전의 라이브러리를 따로 특정한 디렉토리에 위치시킬수도 있다. wrap 스크ㄹ트는 아마 아래와 같을 것이다.

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

한 프로그램이 사용하는 공유 라이브러리의 목록은 ldd(1)를 써서 알 수 있 다. 그래서, 예를 들면 ls명령이 사용하는 공유 라이브러리의 목록은 다음의 예 와 같이 알수있다

ldd /bin/ls

보통 프로그램에서 의존하고 있는 라이브러리의 soname의 목록과 포함된 디 렉토리가 보일것이다. 실질적으로 다음의 두 모듈의존성은 항상 보일것인데 그 것은 아래와 같다.

* /lib/ld-linux.so.N(N은 1이상 적어도 2). 이 라이브러리는 다른 모든 라이브 러리를 적재한다.

* libc.so.N(N은 6또는 그이상). 이것은 C 라이브러리이다. 다른 프로그래밍 언어들에서도(적어도 그 언어의 라이브러리를 구현하기위해서) C 라이브러리 를 쓰는 경향이 있다. 그래서 모든 프로그램에서 적어도 이 라이브러리는 포 함될 것이다.

경고: ldd를 신뢰할수 없는 프로그램에 사용하여서는 않된다. 이는 ldd의 매뉴 얼에도 명백히 언급된 얘기이다. ldd는 직접 그 프로그램을 호출함으로써 수행 되기 때문에 예기치 못한 코드의 실행을 가져올수 있기 때문이다.

비호환 라이브러리

새로운 버전의 라이브러리가 구버전의 라이브러리와 이진코드 차원에서 호환 되지 않는다면 soname은 변경될 필요가 있다. 이진코드에서 비호환되는 경 우에는 다음의 네가지 이유가 가장 기본적이다.

1. 함수의 행동이 변경되어 원래의 함수스펙(규격)을 더 이상 만족하지 못할 때.
2. 외래 데이터 아이템이 변했을때(예외:옵션 아이템을 라이브러리의 구조 마 지막 부분에 첨가하는 것은 이러한 아이템구조가 라이브러리의 내부에 할당되 어 있는한 괜찮음)
3. 외래함수가 제거되었을 때.
4. 외래함수의 인터페이스가 변경되었을 때.

만약 이러한 원인들을 당신이 피해간다면 당신의 라이브러리는 이진모드에 서 호환가능할 것이다. 다른 식으로 얘기하자면 당신 응용프로그램의 이진 인터페이스(ABI) 호환성은 이러한 변경을 하지않는다면 지켜

질 것이다. 예 를들면 당신은 새로운 함수를 추가하되 기존함수들을 삭제하지않길 원할수 있 을 것이다. 당신은 구버전의 프로그램이 라이브러리 구조의 끝부분에 아이템 을 추가하거나, 또는 (응용 프로그램이 아닌)라이브러리에서만 이러한 구조를 할당하거나, 또는 어떤 특정한 아이템을 옵션화하는(또는 라이브러리에서 그 아이템을 채워넣는)등등의 이러한 변경에 영향을 받지않음이 확실할때만 그 라이브러리 구조에 이러한 아이템추가를 할수있는 것이다. 사용자들이 배열을 사용하고 있을시에는 그러한 구조확장을 아마 할 수 없을 것임에 유의하라.

제 4 절 동적(DL) 라이브러리

동적 라이브러리는 프로그램의 시작시가 아닌 다른때에 적재되는 라이브러리 를 말한다. 플러그인등의 모듈은 그것의 필요시까지 적재를 기다리는 것이 허 용되므로 이들의 구현에 특히 유용하다. 예를들면 PAM(Pluggable Authenti- cation Module:적재가능 인증모듈)등의 시스템에서는 관리자가 인증을 구성또는 재구성하는데 동적 라이브러리를 사용한다. 또한 동적 라이브러리는 수시로 언어코드를 기계어로 컴파일하고, 컴파일된 모듈들을 효율적으로 멈춤없이 사 용코저하는 인터프리터를 구현하는데도 유용하다. 이러한 방법은 저스트 인타 임 컴파일러(JIT)나 멀티 유저던전(MUD:multi user dungeon)의 개발에도 유 용하다.

리눅스에서는 동적 라이브러리는 포맷만의 관점에서 본다면 특별한 것이 아 니다. 즉, 그것은 위에서 논의한 표준적인 오브젝트 파일이나 표준적인 공유 라이브러리로써 만들어진다. 가장 중요한 차이점은 그것이 프로그램의 링크시 나 시작시에 자동으로 적재되는 것이 아니라는 것이다. 그 대신으로 라이브러 리를 열고 심볼을 살피며 에러를 제어하고 닫는등의 API가 제공된다. C의 사 용자는 이러한 API를 사용하기 위해서 <dlfcn.h>를 include하여야 한다. 리눅스에서 사용되는 인터페이스를 나는 "dlopen() API"라고 부르며 솔라리 스의 그것과 핵심에 있어서는 동일하다. 그러나 이러한 동일한 방식의 인터페 이스가 모든 플랫폼에서 같은 것은 아니다. HP-UX는 shl_load()라는 다른 메 커니즘을 사용하며 윈도우즈에서는 DLL을 완전히 다른 인터페이스로 다룬 다. 이식성에 목표를 둔다면 플랫폼간의 이질성을 숨기게해주는 몇개의 wrapping 라이브러리의 사용을 고려해야한다. 한가지 방법으로 glib 라이브러 리의 동적 모듈적재 기능이 있다. 이것은 이러한 함수들의 이식성 구현을 위해 플랫폼의 하부에 있는 동적적재루틴을 사용한다. 이에관한 더 자세한 내용은 [15]*developer.gnome.org/doc/API/glib/glib-dynamic-loading-of-modules.html* `<http://developer.gnome.org/doc/API/glib/glib-dynamic-loading-of-modules.html>` 에서 배울수 있다. 여기 문서에서 잘 설명되어 있으므로 더 이상 의 설명은 하지 않겠다. 다른 방법은 [16]GNU libtool중에 하나인 libltdl을 쓰는 것이다. 이것의 다양한 기능을 쓰기위해서는 아마도 COBRA Object Broker(ORB)를 보길 바란다. 리눅스와 솔라리스에서 제공하는 기능에 여전히 관심이 있다면 계속 읽어주길 바란다.

dlopen()

dlopen(3) 함수는 라이브러리를 열어서 사용준비를 한다. C에서 함수의 원 형은 아래와 같다.

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

만약에 파일의 이름이 /(즉, 절대경로)로 시작한다면 dlopen()은 라이브러 리를 탐색하지 않을것이다. 그렇지 않다면 dlopen()은 다음과 같은 순서로 라 이브러리를 탐색한다.

1. 콜론으로 분리된 사용자의 LD_LIBRARY 환경변수의 각 디렉토리 목록.
2. /etc/ld.so.cache에 명세된 라이브러리들
3. /lib, 그다음은 /usr/lib

dlopen()에서 플래그의 값은 동적 라이브러리로 부터의 코드 수행시에 미정의 된 심볼값을 해결함을 뜻하는 RTLD_LAZY, dlopen()이 리턴되기전 모든 미 정의된 심볼의 값이 해결되는, 그렇지 않으면 실패를 뜻하는 RTLD_NOW 중의 하나가 되어야 한다. 라이브러리내에서 정의된 외래심볼이 순차적으로 호출되는 라이브러리들에 의해서 유효해짐을 뜻하는 RTLD_ GLOBAL또한 사용되거나 옵션이 될 수 있다. 디버깅 중이라면 RTLD_LAZY는 미해결된 심볼참조에 의한 예측 불가능의 에러를 일으킬수 있으므로 RTLD_NOW를 사용하길 원할 것이다. RTLD_NOW는 라이브러리를 여는데 약간 느리다. 이것이 문제가 된다면 RTLD_LAZY로 나중에 전환하면된다.

만약 라이브러리들이 서로 의존할 경우(예를 들면 X가 Y를 의존한다면), 의 존이 되어주는 라이브러리(이 예에서는 Y)를 먼저 로드하여야 한다. dlopen()의 리턴값은 (적재되는 동적 라이브러리의 일종의 파일) 핸들(handle) 이다. 그것은 다른 동적 라이브러리 루틴에서 사용되어질 어떠한 값이라 고려될수 있을 것이다. 적재시도가 실패하면 dlopen()은 NULL을 리턴할 것이며 점검해 보길 바란다. 같은 라이브러리가 두번 이상 dlopen()에 의해서 로드된 다면 같은 파일 핸들이 리턴된다.

만약 라이브러리에서 _init로 명명된 루틴을 외부 호출 한다면 그 코드가 dlopen()의 리턴전에 수행된다. 이러한 사실을 이용해 당신의 라이브러리에서 초기화 루틴을 구현하면 될 것이다.

dlerror()

에러는 dlerror()을 호출함으로써 보고되는데, 이 함수는 가장 최근의 dlopen(), dlsym(), 또는 dlclose()로 인해 발생한 에러를 기술하는 스트링 을 리턴한다. 한가지 특별한 점은 dlerror()를 호출하고 난뒤에는 그 다음부 터는 또다른 에러가 발생하기 전까지 dlerror()는 NULL을 리턴한다는 것이다.

dlsym()

동적 라이브러리를 적재하고 사용하지 못한다면 아무런 의미도 없을 것이다. dlsym(3)은 라이브러리 사용의 가장 주된 루틴이며 이 함수는 오픈된 라이브 러리에서 심볼값을 조사한다. 이함수는 다음과 같이 정의되어있다.

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

handle은 dlopen()에 의해 리턴되는 값이며 symbol은 NIL-로 끝나는 문자열 이다. dlsym()의 결과값을 void * 타잎의 포인터에 저장하면 매번 사용시마다 캐스트 연산을 수행해야 하므로(그리고, 프로그램을 유지하려는 많은 다른 사 람들에게 적은 정보밖에 주지못할 것이므로) 피할수 있다면 이렇게 하지말길 바란다.

dlsym()은 심볼이 발견되지 않은경우에 NULL값을 리턴하게된다. 만약에 어 떤 심볼이 결코 제로나 NULL값을 가지지 않을 것을 당신이 알고 있다면 아 무런 문제도 없으나 만약 그렇지 않다면 잠재적인 모호성이 존재하게 된다, 즉 만약에 NULL값을 가지게 된 경우 그것이 그런 심볼이 존재하지 않는다는 것 인지 아니면 실제로 그 심볼의 값이 NULL인지 하는 것이다. 표준적인 해결 책은 dlerror()을 호출(존재할수 있는 모든 에러의 가능성을 해소하는것임) 하고 그다음으로 심볼값을 알기위해 dlsym()을 호출한 뒤 dlerror()을 한번 더 호출하여 에러가 발생했는지를 점검하는 것이다. 이 부분의 코드를 발췌한 다면 아래와 같을 것이다.

```
dlerror();
 /* 에러코드의 초기화 */
s = (actual_type) dlsym(handle, symbol_being_searched_for);
 if ((err = dlerror()) != NULL)
  /* 심볼값을 찾지 못할때 */
  else
  /* 심볼값을 찾았을 때 그 값이 s에 있음 */
```

dlclose()

dlclose()는 dlopen()의 역으로써 동적 라이브러리를 닫는데 사용된다. 동적 라 이브러리는 동적 파일 핸들에 관한 링크수효를 계속 유지하고 있으므로 적어 도 dlopen()이 호출된 횟수만큼의 dlclose()의 수행이 없기 전에는 실제적으로 메모리에서 삭제되지 않는다. 그래서 같은 프로그램이 같은 라이브러리를 여 러번 호출하는 것은 아무런 문제가 되지 않는다.

동적 라이브러리의 예 dlopen()의 매뉴얼 페이지에 나오는 동적라이브러리의 예가 아래에 있다. 예 에서는 수학라이브러리가 로드되고 2.0의 cosine값이 출력되며 매순간마다 에 러를 검사(권고됨)한다.

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

```
int main(int argc, char **argv)
    void *handle;
    double (*cosine)(double);
    char *error;
    handle = dlopen ("/lib/libm.so", RTLD_LAZY);
    if (!handle)
        fputs (dlerror(), stderr);
        exit(1);


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


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

만약 위의 프로그램의 이름이 foo.c 라면 다음의 명령을 써서 실행프로그램을 만들 수 있다.

gcc -Wl,export-dynamic -o foo foo.c -ldl -WI, export-dynamic 옵션은 실질적으로 필요한 것은 아니나 때때로 유용함 을 알게될 것이다. 그것은 ld에 정의되어 있는데 ELF파일의 생성시에 이옵션 은 모든 심볼을 다이나믹 심볼테이블에 부가한다. 통상적으로 다이나믹 심볼테 이블은 동적 오브젝트에서 사용되는 심볼만을 가지고 있다. 이 옵션은 dlopen()의 어떤 용도에서 필요하다. -WI, export-dynamic 보다 -rdynamic을 쓰라고 말할수도 있겠지만 ELF문서에 의하면 -rdynamic은 비 리눅스 시스 템의 gcc에서는 항상 제대로 동작하는 것은 아니라고 한다.

제 5 절 부가적 내용들

nm 명령

nm명령은 주어진 라이브러리의 심볼목록을 보고한다. 그것은 공유 및 정적라 이브러리 모두에서 동작한다. 주어진 라이브러리의 정의된 심볼명과 값 그리고 타잎을 알려준다. 이 명령은 또한 라이브러리에서 이 정보가 유효하게 되어있 다면(-ls옵션을 보라) 소스코드상의 어디에서 심볼이 정의되었는가(파일이름과 라인넘버로)도 알려준다.

심볼타잎에 대해서는 좀더 설명이 필요하다. 타잎이 보고될 때 소문자는 지역 변수임을 나타내며 대문자는 전역(외부)변수임을 나타낸다. 전형적인 심볼타잎 에는 T(코드부분에서 보통의 정의), D(초기화된 데이타 부분), B(초기화 않된 데이터 섹션), U(미정의:심볼이 라이브러리에서 사용되나 그 라이브러리에 의 해서 정의되지 않음), W(다른 라이브러리가 이 심볼을 정의하면 그 내용이 이 를 대치(override)함)등이 있다.

함수의 이름은 알고 있지만 정확히 어느 라이브러리에서 정의되어있는지 모 른다면 nm 명령의 -o옵션(각 라인마다 파일명을 앞에 보여줌)을 사용하고 동 시에 grep명령을 써서 라이브러리의 이름을 찾아보길 바란다. Bourne shell에서 /lib, /usr/lib의 모든 라이브러리와 /usr/lib, /usr/local/lib의 직계 서 브디렉토리에서 "cos"를 찾는 것은 아래와 같을 것이다.

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

더 이상의 자세한 내용은 [17]info:binutils#nm을 참조하라 공유 라이브러리는 스크˜Pl트일수 있다.

GNU 로더는 공유 라이브러리가 바이너리 이외에도 특정한 스크립트언어로 작성된 텍스트파일로도 가능하도록 허용하고 있음을 알필요가 있다. 이것은 간 접적으로 다른 라이브러리들을 조합하는데 유용하다. 예를 들어 나의 시스템중 의 하나에 사용된 /usr/lib/libc.so의 목록이 아래에 나와있다.

/* 공유 라이브러리를 먼저 사용하며 단 어떤 함수들은 정적 라이브러리이므 로 그것을 다음차례로 사용하는 GNU ld script임. */ GROUP (/lib/libc.so.6 /usr/lib/libc_nonshared.a) 더 많은 내용은 ld 링커에 관한 texinfo 문서를 보라. 일반적인 정보는 info:ld#Options, info:ld#Commands, 그리고 비슷한 info:ld#Option Commands 등에서 찾아보라

GNU Libtool

다양한 시스템에 이식가능한 응용프로그램을 제작중이라면 GNU Libtool을 사용하여 라이브러리를 만들고 설치할 것을 고려해 보라. GNU Libtool은 전 형적인 라이브러리 지원 스크립트이다. Libtool은 라이브러리 사용의 복잡성 을 일관되고 이식가능한 인터페이스를 통해서 제거해줄 것이다. Libtool은 오 브젝트 파일의 생성, 정적 및 공유라이브러리의 링크, 실행 바이너리의 링크 및 디버그, 라이브러리의 설치, 실행 바이너리의 설치를 위한 이식적 인터페이 스를 제공한다. 그안에는 libltdl이라는 동적적재 프로그램을 위한 이식성 wrapper가 또한 포함된다. 더 많은 정보는 [19]*www.gnu.org/software/libtool/manual.html* `<http://www.gnu.org/software/libtool/manual.html>` 에서 찾아보길 바란다.

제 6 절 사용법의 예시

아래 부터는 (정적, 공유 및 동적 라이브러리의)세가지 형태의 모든 라이브 러리의 예이다. libhello.c는 libhello.h를 헤더로 하는 보통의 라이브러리이며, demo_use.c는 libhello.c를 호출하는 평범한 프로그램이다. 그 다음에는 이것이 어떻게 정적 및 공유라이브러리로 사용되는가의 예를 보여주는 스크립트 (script_static, script_shared)가 주석과 함께 있으며, 그 다음에는 공유 라이 브러리가 어떻게 동적 라이브러리로 사용되는지를 보여주는 demo_ dyanmic.c와 script_dynamic이 있다.

File libhello.c /* libhello.c - 라이브러리로 사용되는 파일 */ #include <stdio.h> void hello(void) printf("Hello, library world.");

```
File libhello.h
/* libhello.h - 헤더 파일 */

void hello(void);
```

File demo_use.c /* demo_use.c – 바로 hello루틴을 사용하는 예 */ #include "libhello.h" int main(void) { hello(); return 0; }

```
File script_static
#!/bin/sh
# 정적 라이브러리 데모
# 정적 라이브러리의 오브젝트 파일인 libhello-static.o생성.
# libhello-static이라는 이름을 사용하는 것은 정적 라이브러리
# 와 동적 라이브러리 사용을 엄연히 구별하기 위한것일 뿐이므
# 로 -static이라는 이름을 오브젝트 파일이나 정적 라이브러리에
# 쓸 필요는 없다.
gcc -Wall -g -c -o libhello-static.o libhello.c
# 정적 라이브러리의 생성.
ar rcs libhello-static.a libhello-static.o
# 여기서 libhello-static.a를 나중에 사용하기
```

```
# 위해 어딘가에 복사해 놓을수 있을 것이다.
# 데모 목적으로 사용하기 위해서 우리는
# 현재의 디렉토리에 라이브러리를 두기로 한다.
# demo_use 프로그램의 컴파일
gcc -Wall -g -c demo_use.c -o demo_use.o
# demo_use 프로그램의 생성; -L. 옵션은 "." 가 프로그램
# 의 생성시에 검색되게 한다.  이 명령으로 libhello-static.a에
# 있는 관련된 오브젝트가 파일 demo_use_static에 삽입된다.
gcc -g -o demo_use_static demo_use.o -L. -lhello-static
# 프로그램의 실행
./demo_use_static
```

```
File script_shared
#!/bin/sh
# 공유 라이브러리의 데모
# 공유 라이브러리의 오브젝트 파일 libhello.o의 생성.
gcc -fPIC -Wall -g -c libhello.c
# 공유 라이브러리의 생성.
# -lc를 사용하여 C 라이브러리와 링크한다. 왜냐하면
# libhello가 C 라이브러리에 의존되기 때문이다.
gcc -g -shared -Wl,-soname,libhello.so.0 \
   -o libhello.so.0.0 libhello.o -lc
# 여기서 libhello.so.0.0를 어떤 디렉토리 이를테면
# /usr/local/lib에 복사하여도 된다.
# 이제 ldconfig를 이용 심볼릭 링크를 만든다.
# soname을 셋업함.  아래와 같이 해도 무방하다.
# ln -sf libhello.so.0.0 libhello.so.0
# 여기서는 그냥 ldconfig로 하도록 한다.
/sbin/ldconfig -n .
# linker name을 셋팅한다.
# 좀더 면밀하게 하려면 이미 존재하는 linker name이
# 있는지 확인해보고 만약 그렇다면 그냥 놔둬야 할지
# 아닌지를 잘 체크해야한다.
ln -sf libhello.so.0 libhello.so
# demo_use 프로그램 파일을 컴파일한다.
gcc -Wall -g -c demo_use.c -o demo_use.o
# demo_use 프로그램의 생성.
# -L. 은 "." 이 프로그램의 생성시에 검색되도록
# 한다. "." 가 프로그램 실행시에 검색되는 것이
# 아님에 유의하라.
gcc -g -o demo_use demo_use.o -L. -lhello
# 프로그램 실행. LD_LIBRARY_PATH를 사용
# 하여 공유 라이브러리가 어디에 있는가를
# 프로그램에 알려 주어야한다.
LD_LIBRARY_PATH="." ./demo_use
```

File demo_dynamic.c /* demo_dynamic.c – "hello" routine의 동적적재와 사용의 예 */

/* dlfcn.h 파일이 동적적재 라이브러리 루틴에 필요함 */ #include <dlfcn.h> #include <stdio.h> /* "libhello.h"를 include할 필요가 없음에 유의하라. 그렇지만 그와 관련하여 어떤 것은 반드시 명시해야한다; 즉,

dlsym()으로 얻은 값을 가지고 있을 데이터의 타잎 은 분명히 정해져야 한다. */ /* "simple_demo_function" 타잎은 매개변수와 리턴값이 없는 함수의 타잎을 말한다. */ typedef void (*simple_demo_function)(void);

int main(void) const char *error; void *module; simple_demo_function demo_function; /* 동적 라이브러리를 적재함 */ module = dlopen("libhello.so", RTLD_LAZY); if (!module) fprintf(stderr, "Couldn't open libhello.so: %s", dlerror()); exit(1);

/* 심볼을 가져옴 */ dlerror(); demo_function = dlsym(module, "hello"); if ((error = dlerror())) fprintf(stderr, "Couldn't find hello: %s", error); exit(1);

/* DL 라이브러리의 함수를 호출한다. */ (*demo_function)(); /* 모든 것을 마친 뒤 확실히 클로징한다. */ dlclose(module); return 0;

```
File script_dynamic
#!/bin/sh
# 동적 라이브러리 데모
# libhello.so와 friends가 이미
# 생성되었음을 가정한다. ( dynamic의 예를 보라).
# demo_dynamic 프로그램을 오브젝트 파일로 컴파일.
gcc -Wall -g -c demo_dynamic.c
# demo_use 프로그램의 생성.
# 이 프로그램이 사용하는 특정의 라이브러리는 프로그램이
# 시작되고 난뒤가 되기 전까지는 적재되지 않기 때문에 동적
# 라이브러리를 찾기위해 어디를 검색해야 하는지 알릴 필요가
# 없음에 유의하라.
# 그렇지만 동적 라이브러리를 적재하는 라이브러리를
# 포함하기 위해서 -ldl옵션은 반드시 필요하다.
gcc -g -o demo_dynamic demo_dynamic.o -ldl
# 프로그램의 실행. LD_LIBRARY_PATH 를 이용하여
# 프로그램이 어디서 동적 라이브러리를 얻을수
# 있는지 알려 주어야 한다.
.
LD_LIBRARY_PATH="." ./demo_dynamic
```

제 7 절 다른 참고 문헌 및 소스

라이브러리에 관한 특히 유용한 소스와 정보가 다음의 문서에 있다;

- "The GCC HOWTO" -Daniel Barlow저. 특히 이 HOWTO 문서는 라이 브러리의 생성에 필요한 컴파일러 옵션과 라이브러리에 어떻게 질의하는가에 관해서 잘논의 되고 있다. 이 문서는 여기서 다루지 못한 내용을 다루어줄 것 이며 역으로 그 문서가 다루지 못한 내용이 역시 여기에서 소개되고 있는 것 이다. 이 HOWTO 문서는 Linux Documentation Project의 20]*www.linuxdoc.org* <`http://www.linuxdoc.org¿` 에서 찾아볼 수 있다.

Tool Interface Standards (TIS) committee(이는 동일한 위원회에 의한 Portable Formats Specification Version 1.1의 한 Chapter(장)이다.)에 의한 "Executable and Linkable Format (ELF)". 이것은 ELF 포맷에 관한 (GNU 나 리눅스에만 한정되지 않은)자세한 정보들을 제공한다. [21] *tsx-11.mit.edu/pub/linux/packages/GCC/ELF.doc.tar.gz* `<ftp://tsx-11.mit.edu/pub/linux/packages/GCC/ELF.doc.tar.gz>` 을 보면 된다. 파일을 MIT에 서 다운받으면 비정상적인 포맷일 것이다. gunzip, tar로 푼뒤에 "hps"라는 파일을 얻게 될것인데 첫째줄과 맨마지막줄을 없애고 나면 인쇄가능한 포스트 스크립트 파일을 얻을수 있을것이다.

“ELF: From the Programmer’s Perspective” - Hongjui Lu지음. 이 문서 는 리눅스와 GNU gcc-specific의 ELF에 관한 정보를 제공하며 얻을수 있는 곳은 [22]*tsx-11.mit.edu/pub/linux/packages/GCC/elf.ps.gz* `<ftp://tsx-11.mit.edu/pub/linux/packages/GCC/elf.ps.gz>`이다.

제 8 절 저작권과 라이센스

저작권은(C) David A. Wheeler에게 있으며 GNU General Public License (GPL)를 포함한다. 아무런 대가 없이 재배포 할수 있으며 다음과 같은 항에 의 거하여 포함된 텍스트와 프로그램을 번역 및 이용할수 있다.

이 프로그램은 자유 소프트웨어이다; Free Software Foundation에 의해서 출 판되는 GNU General Public License에 의거하여 재배포 또는 변경할수 있다. 버전 2또는 그이상도 마찬가지이다. 이 문서는 그것이 유용하길 바라며 배포되는 것이지만 어떠한 보증은 없다는 것에 유의하라. 특정목적이나 상업적인 사용을 위한 어떤 암시적인 보증도 없 다는 것에 유의하길 바라며 이에 관해서는 GNU General Public License에서 더 자세히 보길 바란다.

GNU General Public License와 이 프로그램을 이미 가지고 있을텐데 만약 그렇지 못하다면 Free Software Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA에 편지하라.

위 내용들은 다른 웹사이트에서 미러링 할수 있으며 다만,

- 미러 사이트는 주 사이트가 갱신되면 자동으로 갱신될수 있어야 하며,
- 주 사이트의 주소(23]*www.dwheeler.com/program − library*`<http://www.dwheeler.com/program-library¿`)를 분명 히 명시해야 하며 이에 하이퍼 텍스트 링크 되어야 하며,

내게(David A. Wheeler) 저자로써 가져야할 접근권을 주어야 한다.

상위의 두 항목은 내가 쓸모없는 버그리포트를 계속해서 받지 않게 해준다. 나는 당신의 잘못된 미러링으로 인해 1년전에 고친 버그의 보고를 받지않길 원한다. 주 사이트로 링크함으로써 사용자는 미러내용이 최신의 것인지 알 수 있다. 나는 매우 강한 보안을 필요로 해서 일반적인 인터넷연결 조차 위험부담 하지 않는 사이트들에서 발생되는 문제들에 매우 민감하다. 만약 당신이 이런 조건이라면 적어도 다른 포인트를 찾아서 자주 스니커넷 업데이트(sneaknet update)하라.

이 라이센스에 의해서 당신은 문서를 변경할수 있지만 당신이 쓰지 않은 내 용을 당신것이라던가 변경된 내용이 원문과 같다고 말할 수는 없다. 문서에 대 한 변경이 전체 문서에 대한 저작권을 주는 것이 아니다. 이일은 저작법에서 말하는, 저작권 소실로 인하여 자유로이 사용할수 있는 그런 성격의 작업이 아 니다. 문서가 변경되었을 때 그 내용과 날짜를 분명히 알려야 한다는 자세한 저작권법의 규칙을 살펴보길 바란다. 보통의 경우 변경된 내용을 문서의 통합 권자(David A. Wheeler)에게 알려 통합시키는 것이 좋을 것이다.

제 9 절 주석

1. file://localhost/export/sunsite/users/gferg/howto/00Program-Library-HOWTO. html#AEN12
2. file://localhost/export/sunsite/users/gferg/howto/00Program-Library-HOWTO. html#AEN25
3. file://localhost/export/sunsite/users/gferg/howto/00Program-Library-HOWTO. html#AEN33
4. file://localhost/export/sunsite/users/gferg/howto/00Program-Library-HOWTO. html#AEN126
5. file://localhost/export/sunsite/users/gferg/howto/00Program-Library-HOWTO. html#AEN168
6. file://localhost/export/sunsite/users/gferg/howto/00Program-Library-HOWTO. html#AEN189
7. file://localhost/export/sunsite/users/gferg/howto/00Program-Library-HOWTO. html#AEN220
8. file://localhost/export/sunsite/users/gferg/howto/00Program-Library-HOWTO. html#AEN233

9. *http://www.gnu.org/software/libtool/libtool.html* <http://www.gnu.org/software/libtool/libtool.html>
10. *developer.gnome.org/doc/API/glib/glib-dynamic-loading-of-modules.ht ml* <http://developer.gnome.org/doc/API/glib/glib-dynamic-loading-of-modules.html>
11. *www.dwheeler.com/program-library* <http://www.dwheeler.com/program-library>
12. *www.linuxdoc.org/* <http://www.linuxdoc.org/>
13. info:standards#Directory_Variables
14. *www.pathname.com/fhs* <http://www.pathname.com/fhs>
15. *http://developer.gnome.org/doc/API/glib/glib-dynamic-loading-of-modules.ht ml* <http://developer.gnome.org/doc/API/glib/glib-dynamic-loading-of-modules.html>
16. *www.gnu.org/software/libtool/libtool.html* <http://www.gnu.org/software/libtool/libtool.html>
17. info:binutils#nm
18. *www.gnu.org/software/libtool/libtool.html* <http://www.gnu.org/software/libtool/libtool.html>
19. *www.gnu.org/software/libtool/manual.html* <http://www.gnu.org/software/libtool/manual.html>
20. *www.linuxdoc.org/* <http://www.linuxdoc.org/>
21. *tsx-11.mit.edu/pub/linux/packages/GCC/ELF.doc.tar.gz* <ftp://tsx-11.mit.edu/pub/linux/packages/GCC/ELF.doc.tar.gz>
22. *tsx-11.mit.edu/pub/linux/packages/GCC/elf.ps.gz* <ftp://tsx-11.mit.edu/pub/linux/packages/GCC/elf.ps.gz>
23. *www.dwheeler.com/program-library* <http://www.dwheeler.com/program-library>