Linux Assembly HOWTO : 리눅스 어셈블리 하우투

BoldyshevKonstantin
Linux Assembly (http://linuxassembly.org/)

konst (at) linuxassembly.org

RideauFrancois-Rene
Tunes project (http://tunes.org)

fare (at) tunes.org

도운이
김영휘
http://lachesis.pe.kr (http://lachesis.pe.kr)

lachesis (at) kldp.org

Linux Assembly HOWTO : 리눅스 어셈블리 하우투
지은이 BoldyshevKonstantin, 그리고 RideauFrancois-Rene
도운이 김영휘

펴냄 $Date: 2001/10/27 02:18:17 $

이 문서는 리눅스 어셈블리 하우투이다. 이 문서는 GNU 의 자유 프로그래밍 도구들을 사용하여 (주로) 인텔의 32 비트 플랫폼(IA32 ; i386)의 리눅스에서 어셈블리 언어로 프로그래밍 하는 법을 설명한다. 이 글에서 설명하는 내용들은 다른 하드웨어나 소프트웨어 플랫폼에 적용될 수도 있고, 그렇지 않을 수도 있다. (역주: 이 번역문서는 부족한 저의 문장실력으로 필요성에 의하여 번역한 결과로 잘못된 오역과 의미에 이상없는 부분이 빠질 수도 있음을 미리 알립니다. 따라서 잘못된 번역으로 오는 책임은 저에게 없으며, 만일 수정해야 할 곳이 있다면 저에게 연락을 해주시기 바라며, 원저의 오류라면 원저자에게 연락해주시기 바랍니다. 부족한 제 영어실력으로 인해 번역이 애매한 부분은 원문을 같이 표기해 두었습니다.)

고친 과정
고침 번역 버전 0.6-trans-0.12000. 11. 21 - 2001. 10. 20고친이 라키시스
문서 초벌 번역 완료 : Resource 이후의 것은 미번역

차례

1장. 소개

> **참고:** 하우투 문서에 친숙하다거나 어셈블리와 무관한 잔소리 :-) 들을 보기 싫으시면 이 장을 건너뛰어도 좋다.

1.1. Legal Blurb (법적문제에 관한 잔소리)

이 문서의 복사, 배포 혹은 수정은 GNU Free Documentation License (http://www.gnu.org/copyleft/fdl.html) Version 1.1 에 따른다. with no Invariant Sections, with no Front-Cover Texts, and no Back-Cover texts. A copy of the license is included in the GNU Free Documentation License appendix.

> 이 문서의 가장 최근의 공식판은 Linux Assembly (http://linuxassembly.org/howto.html) 나 LDP (http://linuxdoc.org/docs.html) 에서 구할 수 있다. 문서의 새로운 버젼을 확인하는 것을 잊지 밀라.

> **참고:** 역자 주 : 현재 최신 버젼은 0.6d 이다. 번역문은 0.6 을 번역한 것이다.

1.2. 서문

이 문서는 free 소프트웨어를 사용한 32비트 x86 어셈블리 프로그래밍이나 프로그램들에 대해 질문하는 사람들에게 답변하기 위해 제작되었으며, 특히 리눅스 운영체제를 타겟으로 삼고 설명을 진행하고 있다. 문서의 여러 부분에서 특정 소프트웨어나 문서를 구하기 위한 url 들을 잠조하고, 언급하고 있다. 이 문서는 또한 주된 목적이 아닌 자유 소프트웨어가 아니거나, x86 기반이 아니거나, 32비트 어셈블러가 아니기도 한 것들에 대한 언급을 하기도 한다. 여러분이 좋아하는 플랫폼에서의 프로그래밍에 대한 FAQ 와 문서들이 있음을 항상 기억하라. 여러분이 그에 대한 내용을 발견하려면 그(특정 플랫폼에서의)에 관한 문서 들을 참조하도록 하라. (어셈블리 프로그래밍을 바로 찾지 말고, 특정 플랫폼에 관한 문서를 찾으면 된다.)

어셈블리 프로그래밍의 주된 목적은 운영체제, 인터프리터, 컴파일러, 게임 등과 같은 분야에서 C 로 구현하기 힘든 부분을 (퍼포먼스가 주된 issue 가 되는 부분) 건드리기 위한 것이기 때문에 이 문서에서는 그러한 종류의 소프트웨어 개발과 관련된 부분에 촛점을 맞출 계획이다.

만약 여러분이 **free** 소프트웨어 (http://www.gnu.org/philosophy/)가 무엇인지 모른다면, GNU General Public License (GPL 혹은 copyleft) (http://www.gnu.org/copyleft/gpl.html) 을 주의깊이 읽어보기 바란다. 아주 많은 free 소프트웨어(역자 주 : 자유 소프트웨어 라고 번역하겠다.)들이 GNU GPL 을 따른다. 배포되는 자유 소프트웨어의 소스 중에 `COPYING`(혹은 `COPYING.LIB`) 이라는 이름의 파일에서 읽어볼 수 있다. 또한, 자유 소프트웨어 재단 (http://www.fsf.org) (Free software foundation : FSF) 에서 제공하는 문서들도 도움이 될 것이다. 이러한 자유 소프트웨어가 아주 흥미있는(!) 이유는 그 소프트웨어들이 소스코드와 함께 제공되기 대문이다. 게다가 여러분은 소스코드를 수정하거나, 버그를 잡아서 다시 배포할 수도

있으며, 심지어 그 코드에서 차용한 코드를 가진 프로그램을 만들 수도 있다. (출처를 밝히고, 자신의 코드를 GNU GPL 하에 공개한다면)

1.3. 이 문서에 기여하는 법

이 문서는 진화(!)해 가는 문서이다. 여러분은 자유롭게 이 문서에서 다루는 주제에 대해 질문할 수 있으며, 또한, 답도 해 줄수 있다. 또한 주어진 답의 오류를 정정해서 다시 답해줄 수도 있다. 새로운 소프트웨어를 알려줄 수도 있으며, 이 문서의 버그나 비효율적인 점을 저자에게 알려줄 수도 있다.

제발!! 기여해 달라!! 문서의 저자에게 메일을 : maintainer.

> 참고**:** 이 글을 쓰는 시점에, 이 문서의 관리자는 Francois-Rene Rideau 가 아니라 Konstantin Boldyshev 이다.(버젼 0.5부터 그러했다.) 나(저자)는 이 문서를 나를 대신해서 유지/보수할 진지한 해커를 한동안 찾아 왔으며, Konstantin이 나의 후임이 되었음을 기쁘게 발표하는 바이다.

2장. 어셈블리가 정말로(!) 필요한가?

음... 여러분이 무슨일을 하는지 내가 알 수 있는 방법이 없기 때문에, 여러분의 일에 어셈블리가 정말 필요한지 내가 판단할 수는 없다. 대신, 거듭된 삽질(-_-)의 결과로 얻은 귀중한 경험의 조언을 아래에 적어 보겠다.

2.1. Pros and Cons - 득실(어셈블리를 사용함으로써 생기는 장점, 단점)

2.1.1. 어셈블리의 장점

어셈블리는 매우 저급의(? - very low things, 하드웨어 레벨의) 일들을 처리할 수 있다. :

- 여러분이 어셈블리를 사용하면, 특정 프로세서에만 있는 레지스터나 I/O 를 바로 접근할 수 있다.(you can access machine-dependent registers and I/O)
- 크리티컬 섹션 등에서 데드락과 같은 일을 야기할 수 있는 여러개의 프로세스 쓰레드들의 코드 동작을 매우 정확하게(!) 컨트롤 할 수 있다. 즉, 코드 하나하나의 동작을 모두 제어하고, 여러분이 생각한 대로 움직이게 할 수 있다.(역자 주: 물론, 여기에는 그만큼의 노력과 댓가가 따른다.)
- 일반적인 컴파일러가 제공하는 환경과, 생성하는 코드의 규칙들을 무시한 채 작업할 수 있다. 다시 말해서, 속도의 최적화 등을 위해 메모리 할당에 관한 규칙이나, 쓰레딩, 함수 호출 규칙과 같은 것들을 일시적으로 무시하고 프로그래밍 할 수 있다.
- 서로 다른 규약을 사용하는 여러개의 코드들 간의 인터페이스를 구축할 수 있다. (예를 들면, 다른 컴파일러들로 생성된 코드들이 상호 작용할 수 있도록 만든다든지, low-레벨 인터페이스가 상이한 코드들의 인터페이스를 통일한다든지 하는 일 말이다.)
- 일반적인 고수준 언어(c 나 pascal 같은)용 컴파일러로는 생성하기 힘든 특정 프로세서의 특수한(!) 프로그래밍 모드에 접근할 수 있다. 이를테면, 인터페이스 스타트업을 위해 프로세서의 16비트 모드를 사용한다든지, 펌웨어용 프로그램을 만든다든지, 인텔 컴퓨터에서 레거시 코드(legacy code)를 만든다든지 하는 일을 어셈블리를 사용해서 할 수 있다.
- 또한, 코드 최적화를 제대로 시키지 못하는 나쁜(!) 컴파일러의 코드를 직접(!) 최적화 시켜서 그런 컴파일러가 생성한 느려터지고 불필요한 루프나 루틴들을 매우 빠르고, 효율적인 루프로 바꾸어 줄 수 있다. (그러나, 매우 좋은 free 컴파일러들이 있다. gcc 처럼)
- 여러분의 하드웨어 설정에만 완벽하게 최적화된 코드를 직접 제작할 수 있다. 그럼으로써, 범용적인 (일반적인) 호환(?)을 위한 불필요한 군살을 여러분의 코드에서 뺄 수 있다.
- 만약 여러분이 새로운 언어를 만들고, 그 언어를 위해 컴파일러를 만든다면, (그런 일을 하는 사람은 몇명 안되고, 또, 자주 그러지도 않지만) 여러분의 컴파일러가 최적의 코드를 생산할 수 있도록 조정할 수도 있을 것이다.

2.1.2. 어셈블리의 단점

어셈블리는 매우 저수준의 언어이다. (이진 기계어 인스트럭션을 그대로 심볼화 한것에 지나지 않는다.) 이것은 다음과 같은 의미를 가진다 :

- 어떤 프로그램을 처음부터 어셈블리 코드를 이용하여 작성하는 일은 매우 오래 걸리고도 짜증나는 작업이다.
- 버그가 발생할 확률이 매우(!) 매우 높다.
- 게다가 버그를 추적해서 잡아내기란 끔찍할 정도로 어렵다.
- 여러분이 작성한 코드를 이해하기도 어려울 뿐만 아니라 수정하기도 어렵기 그지없다. 즉, 유지 보수가 어렵다.
- 어셈블리로 코딩을 한 결과물은 현존하거나 앞으로 개발될지도 모르는 다른 아키텍쳐의 머신으로 이식할 수 없다.
- 여러분이 어셈블리로 최적화시켜 작성한 코드는 여러분이 개발한 타겟이 되는 아키텍쳐와 동일한 아키텍쳐를 가지고, 또, 동일한 설정을 하고, 동일한 환경을 가진 기계에서만 수행될 것이다. 예글 들면, 인텔 계열의 프로세서들(Intel 386, 486, Pentium, PPro, PII, PIII, PIV; Cyrix 5x86, 6x86, M2; AMD K5, K6 (K6-2, K6-III), K7 (Athlon, Duron).등) 만 보더라도 다들 프로세서의 디자인이나 그 외 여러 요소들(상대적 클럭 지연시간(?), 수행속도, 용량, 캐쉬, 램, 버스, 디스크, FPU 의 존재여부, MMX, 3DNOW, SIMD확장 등등... 나열하기도 힘들다) 의 차이로 인해서 프로세서마다 또, 그 주변 환경에 따라서 완벽하게 다른 최적화 테크닉이 필요하다. 게다가 프로세서의 설계/디자인은 계속적으로 변하고, 바뀌고, 발전하고 있기 때문에, 방금 나열한 목록과 또한, 여러분이 최적화시킨 코드가 최신의(!) 코드가 되기를 바라는것은 말도 안된다.
- 게다가, 자잘한 세밀한 부분에 신경을 쓰느라 프로그램 전체의 알고리즘이나 설계에 신경을 쓰지 못할 가능성이 다분히 존재한다.

  you spend more time on a few details and can’t focus on small and large algorithmic design, that are known to bring the largest part of the speed up (e.g. you might spend some time building very fast list/array manipulation primitives in assembly; only a hash table would have sped up your program much more; or, in another context, a binary tree; or some high-level structure distributed over a cluster of CPUs)

- 알고리즘 설계를 조금만 바꾸어도 이미 만들어둔 어셈블리 코드의 전체를 다시 만들어야 할지도 모른다.
- 뭐, 표준 벤치마크 등, 엄청난 수행속도가 필요하지 않은 프로그램이라면, 상용 컴파일러들이 만들어내는 어셈블리 코드도 손으로 만든것 처럼 좋은 코드를 생산한다. gcc 도 상당히 좋은 컴파일러이다.
- 그리고, 컴파일러 뉴스그룹 (comp.compilers (news:comp.compilers)) 의 moderator 인 John Levine 이 말한 내용을 인용하겠다.

  "컴파일러는 복잡한 자료구조를 다루기 쉽게 도와둔다. 그리고, 컴파일러를

사용한 프로그램은 그리 지루하지도 않으며, 게다가 매우 좋은 품질의 코드를 생산해 준다."

> "compilers make it a lot easier to use complex data structures,
> and compilers don't get bored halfway through
> and generate reliably pretty good code."

컴파일러들은 거대한 프로그램 전체에 걸쳐 함수들과, 모듈들 간의 인터페이스 코드 최적화와 같은 것들을 정확하게 수행해 준다.

2.1.3. 결론

요약하자면, 여러분은 어셈블리가 필요한 경우를 가끔 발견할 수 있을 것이다. 그리고, 어떤 경우에는 꼭 필요하지는 않아도 어셈블리로 코딩하는 것이 유용한 경우를 발견할 수도 있을 것이며, 여러분이 그렇게 느낀다면, 그렇게 하라. 단지, 다음 사항들을 명심하고, 그렇게 한다면 좋을 것이다. :

- 어셈블리 코드의 사용을 최소화하라
- 매우 잘 정의된 인터페이스 내부로 어셈블리 코드를 캡슐화(!)하라.
- 여러분의 어셈블리 코드가 보다 고수준의 언어패턴으로부터 자동적으로 생성된 코드인가? (예를 들면 gcc 의 인라인 어셈블리 매크로 등과 같은)
- 그 프로그램들을 어셈블리 코드로 번역한 후 비교해 보라

  have automatic tools translate these programs into assembly code

- 가능한 만큼 코드들이 최적화 되었는가(have this code be optimized if possible)
- 위의 사항들을 잘 체크해 보고, 최적화된 좋은 코드를 생산하는 컴파일러의 back-end 를 잘 이용하라.

  All of the above, i.e. write (an extension to) an optimizing compiler back-end.

비록 운영체제 개발과 같이 어셈블리가 꼭 필요한 것처럼 보이는 경우라 할지라도 위에서 언급한 내용들을 주의깊게 생각해 본 후에 어셈블리의 사용 여부를 결정하라

리눅스의 커널 소스를 한번 보도록 하라 : 리눅스의 커널은 필요한 정말 최소(!)의 어셈블리 코드만 사용되었다. 그 결과로 빠르고, 신뢰성있고, 포터블하며, 유지보수하기 용이한 운영체제의 코드가 생성되게 되었다. 게다가 둠과 같은 게임조차도 속도 향상을 위한 일부 매우 작은 루틴을 제외하고는 대부분 c 를 이용해서 만들어졌다는 사실을 항상 상기하라.

2.2. 어셈블리를 사용하지 않는 방법

2.2.1. 효율이 높은 코드를 생성하기 위한 일반적인 절차

뉴스그룹 comp.compilers (news:comp.compilers) 의 Charles Fiterman 이 사람이 수작업한 어셈블리 코드와 컴파일러가 생성한 어셈블리 코드를 비교해서 한 말을 인용하겠다.

> 사람이 수작업한 코드가 언제나 더 우수한 성능을 보인다. 이유는
> 사람은 먼저, 고수준의 언어로 전체 프로그램을 만든다.
> 그리고, 그렇게 해서 생성한 코드의 시간을 잡아먹는 등 비효율적인 부분을 찾아낸다.
> 그리고나서, 컴파일러가 생성한 어셈블리 코드를 자신이 수정한 효율적이고 적은 코드로 대체한다.
> 마지막으로, 기계가 생성한 코드에 비교해 조금 더 나은 성능의 코드를 수작업해서
> 만들어낸다.
> 인간이 만든 코드가 언제나 더 뛰어나다. 인간은 기계를 사용할 수 있기 때문이다.

2.2.2. 최적화된 좋은 코드를 생성하는 컴파일러를 가진 언어들

ObjectiveCAML, SML, CommonLISP, Scheme, ADA, Pascal, C, C++, 등과 같은 언어들은 모두 좋은 free 컴파일러를 가지고 있으며, 그러한 컴파일러를 사용해서 생성한 코드들은 매우 최적화된 코드를 만들어준다. 게다가 종종, 수작업으로 만든 어셈블리 코드보다 더 나은 최적화를 보여주기도 한다. 그에 더하여, 여러분이 보다 높은 수준의 세부적인 것에, 즉, 알고리즘이나, 프로그램의 전체 설계와 같은 부분에 보다 신경을 쓰도록 도와준다. 그리고, 그처럼 설계와 같은 부분에 보다 신경을 씀으로써, 자잘한, 몇몇 부분에서의 약간의 수행속도의 손실은 보다 나은 디자인에 의한 효과가 충분히 상쇄시켜 주고도 남는다. 물론, 상용의 앞서 언급한 언어를 지원하는 훌륭한 컴파일러들도 있다.

LISP, Scheme, Perl 등의 언어의 컴파일러는 c 코드를 생성하기도 한다. 그렇게 해서 생성된 c 코드는 c 컴파일러에 의해서 최적화되며, 속도도 꽤 빠른 편이다.

2.2.3. 여러분의 코드의 수행속도를 높이는 일반적인 절차

코드의 수행속도를 높이기 위해 여러분은 프로파일 툴이 계속적으로 수행성능의 병목 현상이 일어난다고 지적하는 부분의 코드에 대해서만 최적화의 작업을 해 주면 된다.

여러분이 코드의 일부분이 너무 느리다는 것을 알아채면, 다음과 같은 조치를 취해 보라

- 먼저, 더 나은 알고리즘을 사용하도록 하라.
- 그리고 나서, 그 코드를 인터프리트 하지 말고, 컴파일(!) 하도록 하라.
- 여러분의 컴파일러의 tweak optimization 을 활성화시켜서 컴파일하라.

- 그리고 나서, 컴파일러에게 최적화 시키는 법에 관한 힌트를 주어라 (최적화 정보를 리스프로 만들고, gcc 등으로 등록시켜라. 대부분의 컴파일러에는 많은 옵션이 있다.) (역자 주:번역이 너무 매끄럽지 못하군요 -_-;)

  then give the compiler hints about how to optimize (typing information in LISP; register usage with GCC; lots of options in most compilers, etc).

- 그와같이 한다면, 어셈블리 프로그래밍을 충분히 대신할 수 있을 것이다.

마지막으로, 여러분이 어셈블리 코드의 작성을 마무리하기 전에 생성된 코드들을 검사해서 최적화되지 않은 상태로 컴파일러가 생성한 코드가 정말 문제가 되는지를 확인해 보라. 대부분, 그렇지 않을 것이다. 오히려 컴파일러가 생성해 준 코드가 여러분이 직접 작성한 코드보다 더 나을 수도 있다. 특히, 오늘날의 복잡한, multi-pipelined 아키텍쳐 기반의 하드웨어를 사용하는 환경에서의 프로그램일 경우에는 더욱 그러하다.

프로그램에서 시간을 많이 잡아먹는 부분은 정말 그러한데(?-번역이 이상합니다...-_-;) 왜냐하면, 오늘날의 초고속 :-)의 프로세서들에서는 실행 시간을 늦추는 주 원인으로, 메모리 접근(메모리에 접근하는데 걸리는 시간은 레지스터에 접근하는 데 걸리는 시간의 몇배에서 몇십배는 더 길다), 캐쉬의 부재, TLB (역자주 : Transtate lookasaide buffer:인텔 계열의 프로세서에서 메모리 주소의 상호변환을 보다 빨리 하기 위해 descriptor 를 저장하는 캐쉬와 비슷한 것) 의 부재, 페이지 폴트 등의 원인이 있다. 따라서, 레지스터를 최적화 하는것은 점차 별 쓸모가 없어지게 되었다. 대신, 자료 구조와 쓰레딩의 조절을 통해서 메모리에 접근하는 횟수를 줄이는 것이 보다 나은 속도를 얻는데 더 효과적이 되었다. 아마도, 완전히 다른 방향에서의 접근이 도움이 될지도 모른다.

2.2.4. 컴파일러가 생성한 코드에 대한 고찰

컴파일러가 생성해 낸 어셈블리 코드를 고찰해 보아야 할 많은 이유가 있다. 여러분이 해야 할 일은

- 컴파일러가 생성한 코드가 수작업으로 만든 어셈블리 코드보다 더 나은지 정말 검사해 볼 것 (또는, 컴파일러의 옵션을 매우 세밀하게 조정해 볼 것)
- 그럴 경우 (역자 주:컴파일러가 생성한 코드가 마음에 들지 않는다는 생각이 들때) 처음부터 완전히 손으로 만드는 것 보다 컴파일러가 생성한 바로 그 코드를 수정하는 것으로 최적화하라
- 보다 일반적으로 이야기해서, 컴파일러가 만들어낸 코드를 단지, 여러분의 어셈블리 루틴과 외부 루틴의 인터페이스만 제공하는 뼈대만 생성된 스텁(stub) 코드로 간주하고 작업하라.
- (그럴 일은 별로 없겠지만)여러분의 컴파일러가 생성한 코드를 따라 읽어 내려가면서 버그를 찾도록 하라.(track down bugs in your compiler (hopefully the rarer))

결론적으로, 어셈블리로 코딩하는 표준(?)적인 방법은 여러분의 컴파일러에(gcc 일 것이라고 가정하고..) `-S` 옵션을 주고 c 등으로 작성된 프로그램을 컴파일해서 생성된 어셈블리 코드

를 사용해서 프로그램을 만드는 것이다. 대부분의 유닉스 계열에서 사용되는 컴파일러로 가능하다. gcc 를 사용할 경우, `-fverbose-asm` 옵션도 준 상태에서 컴파일한다면, 보다 알기쉬운 어셈블리 코드를 생산할 것이다. 물론, 훌륭한(!) 어셈블리 코드를 얻기를 원한다면, 여러분이 평소에 사용하던 최적화 옵션과 힌트를 컴파일러에 주는 것을 잊지 말라.

2.3. 리눅스와 어셈블리

여러분도 눈치챗을 테지만, 일반적으로, 리눅스에서 프로그래밍을 하면서 어셈블리를 사용할 필요는 없다. 리눅스는 도스와는 달라서, 드라이버와 같은 것들을 어셈블리로 작성할 필요는 없다 (음.. 만약 여러분이 어셈블리로 만들겠다면 할 수 없지만 말이다) 그리고, 오늘날의 좋은 컴파일러들을 이용한다면, 서로 다른 프로세서들을 타겟으로 각각 수행속도의 최적화를 하기를 원하더라도, c 언어를 사용해서 프로그래밍하는 것이 어셈블리로 하는 것보다 더 쉽다. 그러나... 그럼에도 불구하고, c 나 c++ 보다 어셈블리가 더 적절한 경우가 있을 것이다.

여러분이 어셈블리 프로그래밍을 필요로 하거나, 혹은 어셈블리 언어를 사용하기를 원할지도 모른다. 간단히 얘기해서, 어셈블리의 미궁으로 뛰어들어야 하는 경우는 매우 작은 코드가 필요하거나 libc 에 독립적인, 즉, 라이브러리를 필요로 하지 않는 프로그램을 만들어야 하는 좀.. 비정상적인(?) 경우 밖에는 없다. 그리고, 가장 자주 보게 되는 그러한 경우는, 신화시대의(!) 전설적인 광기의 해커처럼 되고 싶어하는 사람의 경우이다. 그런 사람은 주로 스무살 가량의, 모든것을 자신이 다 알아야 하고, 모든것을 스스로 하고 싶어하고, 모든것을 어셈블리로 하고 싶어하는 사람이 많다. :-) (역자 주:역자의 의역이 너무(!) 많았습니다. 아래에 - 그래왔듯이 - 원문을 보입니다.

You may **need** to use assembly, or you may **want** to use assembly. In short, main practical (**need**) reasons of diving into the assembly realm are **small code** and **libc independence**. Impractical (**want**), and the most often reason is being just an old crazy hacker, who has twenty years old habit of doing everything in assembly language.

그러나, 여러분이 리눅스를 임베디드 시스템에 포팅하려는 경우, 전체 프로그램의 크기에 상당히 예민해 질텐데, 즉, 커널, libc, 유틸리티들(file|find|text|sh|etc.) 전부를 단지 몇백 kb 의 작은 크기에 맞추어 넣을려고 하고, 1킬로바이트가 늘어날 때마다 시스템 생산 비용이 너무 많이 들 때가 있을 것이다. 그럴 경우에는 시스템의 일부 혹은 전부를 어셈블리로 재작성하는 것도 한 방법이 될 수 있다. 그리고, 그렇게 한다면, 사이즈는 감동적으로(! 정말로) 작아질 것이다. 예를 하나 들어보자. 어셈블리로 간단한 **httpd** 데몬을 만들면 600바이트 이하로 만들어질 수 있다. 그러면, 여러분은 커널과, httpd 를 탑재한 웹서버를 400 kb 혹은 보다 더 작은 사이즈로 만들 수 있게 된다.

3장. 가용한 어셈블러들

3.1. GCC 인라인 어셈블러

아주 유명한 GNU C/C++ 컴파일러인 GCC 이다. 이 컴파일러는 GNU 프로젝트의 핵심이 되는 32비트 컴파일러이다. 매우, 매우(!) 좋은 코드를 생산하며, x86 아키텍쳐의 프로세서 또한 매우 잘 지원한다. 이 컴파일러에는 C 프로그램 코드 중간에 인라인 어셈블리를 삽입할 수 있는 기능을 가지고 있는데, 이를테면, 레지스터 할당과 같은 것을 gcc 가 스스로 하게 하거나, 또는 프로그래머가 원하는 대로 조정해서, 원하는 레지스터를 할당하게 할 수도 있다. 게다가, gcc 는 거의 대부분의 플랫폼에서 사용가능하다. 리눅스, *BSD, VSTa, OS/2, *DOS, Win* 등과 같이 광범위한 운영체제에서 사용 가능하다.

3.1.1. GCC 를 어디서 구할 것인가?

가장 주된 사이트는 GNU 의 FTP 사이트 (ftp://prep.ai.mit.edu/pub/gnu/gcc/ (ftp://prep.ai.mit.edu/pub/gnu/gcc/)) 이다. 이 사이트는 GNU 프로젝트의 산물인 많은 자유 소프트웨어들이 컴파일된 형태 혹은 소스의 형태로 발표되는 곳이다. 리눅스에 맞춰서 설정 및 컴파일된 버젼들은 ftp://metalab.unc.edu/pub/Linux/GCC/ (ftp://metalab.unc.edu/pub/Linux/GCC/) 에서 찾을 수 있다. 또한, 세계 각처에 FTP 미러 사이트들이 있다. 원한다면, CD-ROM 으로도 구할 수 있다.

GCC 의 개발은 일전에 GCC2.8 과 EGCS 패키지의 두개의 가지로 나누어졌었다. 그러나 최근에 와서 이 두 패키지는 통합되어서 개발 및 배포되고 있다. 현재 GCC 의 웹페이지는 http://gcc.gnu.org (http://gcc.gnu.org) 이다.

여러분이 즐겨 사용하는 운영체제를 타겟으로 만들어진 소스나 바이너리 또한 여러분이 자주 가는 여러분 운영체제의 FTP 사이트들에서 찾을 수 있을 것이다.

GCC 의 DOS 버젼은 DJGPP 라고 불리우는데, http://www.delorie.com/djgpp/ (http://www.delorie.com/djgpp/) 에서 구할 수 있다.

또한 Win32 용의 GCC 포트도 있다 : cygwin (http://sourceware.cygnus.com/cygwin/ (http://sourceware.cygnus.com/cygwin/)) 과 mingw (http://www.mingw.org) 가 그것이다.

심지어 OS/2 용의 GCC 포트도 있다. 이것은 EMX 라고 불리운다. 구할 수 있는 URL 은 다음과 같다. ftp://ftp-os2.cdrom.com/pub/os2/emx09c/ (ftp://ftp-os2.cdrom.com/pub/os2/emx09c/). 여기서 구할 수 있는 EMX 는 도스에서도 작동하며, 패키지에는 유닉스 에뮬레이션을 위한 목적은 라이브러리 루틴도 포함되어 있다. 앞의 사이트에서 여기저기를 둘러보고 찾아 보라.

There is also an OS/2 port of GCC called EMX; it works under DOS too, and includes lots of unix-emulation library routines.

3.1.2. GCC 인라인 어셈블러에 관한 문서를 어디서 찾을 수 있는가

GCC 문서는 TeXinfo 포멧으로 문서파일들을 포함한다. 여러분은 그 문서들은 TeX 을 이용하여 컴파일해 출력해 볼 수도 있고, `.info` 파일로 변환해서 emacs 에서 브라우징 하면서 읽을수도 있다. 또는, `.html` 파일로 변환하여 웹 브라우저에서 읽을 수도 있고... 기타 여러분들이 좋아하는 포멧으로 변환할 수 있다. .info 파일은 GCC를 제대로 설치했다면 함께 설치되었을 것이다.

여러분이 정보를 얻기 위해서 찾아야 할 섹션은 `C Extensions::Extended Asm::` 섹션이다.

또한, `Invoking GCC::Submodel Options::i386 Options::` 섹션에 있는 내용도 상당히 유용할 것이다. 특히나 이 섹션을 자세히 살펴보면, i386 계열의 레지스터를 위해 규정된 이름에 대한 정보를 얻을 수 있는데 : `abcdSDB` 가 각각 `%eax`, `%ebx`, `%ecx`, `%edx`, `%esi`, `%edi`, `%ebp` 과 하나씩 일치하게 된다. (단 `%esp` 에 해당하는 것은 없음),

웹의 DJGPP 게임즈 리소스 페이지에 어셈블리에 관한 내용이 매우 많이 있었는데... 지금은 서버가 죽어 있는것 같다. 그러나, 자료는 DJGPP 사이트 에 복구되어 있다. 이 사이트에는 다른 유용한 정보들도 많이 존재한다 : http://www.delorie.com/djgpp/doc/brennan/ (http://www.delorie.com/djgpp/doc/brennan/), 또한, 정말 유용한 자료인 DJGPP Quick ASM Programming Guide (http://www.castle.net/~avly/djasm.html) 도 찾아볼 수 있다.

GCC 는 어셈블리 소스를 어셈블 하기 위해서 GAS 에 의존하며, GAS 의 문법을 따른다(아래를 보라:) 인라인 어셈블리를 쓸 때, 퍼센트 문자는 따옴표로 둘러싸야 한다는 것을 명심하라. 퍼센트 문자는 GAS 로 넘겨지게 된다. 자세한 내용은 아래의 GAS 에 관련된 섹션을 보라.

그리고, 정말 유용한 예제들의 집합체인 `linux/include/asm-i386/` 디렉토리 아래를 살펴보라. 또한, 리눅스 커널의 서브디렉토리들도 살펴보도록 하라. 유용한 예제들을 정말로 많이 얻을 수 있을 것이다.

3.1.3. GCC 를 이용하여 인라인 어셈블리 코드를 생성하기

커널 헤더들에서 찾아볼 수 있는 어셈블리 루틴들은 `extern inline` 함수들 내부에 들어가 있기 때문에, (만약 여러분이 여러분이 만든 헤더파일을 가지고 있으며, 여러분의 어셈블리 코드를 리눅스 커널의 그것처럼 명료하게 작성하고 싶다면 여러분도 그처럼 하면 된다) 이들 루틴들이 사용가능하게 하기 위해서 GCC 는 반드시 `-O` 플래그와 함께 실행되어야 한다. (또는 `-O2`, `-O3` 등과 같은 옵션과 함께 실행되어야 한다.) 그렇게 하지 않으면, 여러분의 코드는 컴파일될런지는 몰라도, 제대로 링크되지는 않을 것이다. 왜냐하면, 프로그램이 링크될 때, 링커는 인라인이 아닌 `extern` 함수들을 여러분의 프로그램이 링크될 라이브러리에서 찾기 때문이다. 또 다른 방법은 루틴들의 fallback 버젼이 존재하는 라이브러리와 링크시키는 것이다.

Because assembly routines from the kernel headers (and most likely your own headers, if you try making your assembly programming as clean as it is in the linux kernel) are embedded in extern inline functions, GCC must be invoked with the -O flag (or -O2, -O3, etc), for these routines to be available. If not, your code may compile, but not link properly, since it will be looking for non-inlined extern functions in the libraries against which your program is being linked! Another way is to link against libraries that include fallback versions of the routines.

인라인 어셈블리가 사용하기 싫다면, `-fno-asm` 플래그를 주면 된다. 그렇게 하면, 컴파일러는 확장 문법인 인라인 어셈블리가 발견되었을 경우, 종료하게 된다. 그렇지 않으면, 링커가

인식하지 못하는 외부함수인 `asm()` 함수에 대한 호출을 생성하게 되어버린다. (번역이 좀 이상하네요... -_-a) 그 반대의 플래그는 `-fasm` 옵션으로써, asm 키워드에 대한 취급방법을 복구한다(?)

which will have the compiler die when using extended inline asm syntax, or else generate calls to an external function named `asm()` that the linker can't resolve. To counter such flag, `-fasm` restores treatment of the `asm` keyword.

보다 일반적인 것으로써, x86 플랫폼에서 실행되는 gcc 에 매우 유용한 플래그가 있는데, 바로

gcc -O2 -fomit-frame-pointer -W -Wall

이다. `-O2` 는 일반적으로, 쓸만한 최적화 레벨이다. 그 외의 최적화 레벨은 컴파일하는데 시간이 더 오래 글리고, 더 큰 코드를 생산한다. 그럼에도 불구하고, 생성된 코드의 수행성능은 약간 더 빨라질 뿐이다. 과도한 최적화 옵션을 주어서 효과적인 것은 매우 타이트한 루프가 있을때 뿐이다. 여러분이 정말, 강력한(!) 최적화 컴파일을 필요로 하는 경우에는 `-O6` 의 레벨까지 사용하는 것은 고려해 봄직하다.

`-fomit-frame-pointer` 옵션은 쓸데없는 frame포인터(역자주 : stack 의 내용을 복구하는 것과 관련되어서 컴파일러 내부적으로 생성하는 포인터입니다.) 를 유지하는 코드를 생성하지 않도록 한다. 그렇게 하면 생성된 어셈블리 코드는 보다 작고, 빠른 것이 되며, 더 나은 최적화를 위한 레지스터들의 확보를 가능하게 해 준다. 단, 디버깅 툴 (**gdb** 와 같은..) 을 사용한 디버깅이 불가능해진다.

`-W -Wall` 옵션은 유용한 경고메세지들을 발생시키도록 한다. 경고메세지를 통해 여러분은 혹시 발생할지도 모를 어리석은 에러들을 미리 체크할 수 있게 한다.

여러분은 일련의 프로세서 종속적인 최적화 옵션을 줄 수 있는데, 예를 들면, `-m486` 등과 같은 옵션이 그것이다. 그러한 CPU 특정의 옵션을 줌으로써, GCC 는 여러분이 타겟으로 삼는 프로세서에 보다 더 잘 맞고, 최적화된 코드를 만들어 줄 것이다. 또한, 최근의 GCC 는 `-mpentium` 과 같은 플래그도 가지게 되었다. (GCC 2.7.x 또는 그 이하의 버젼에서는 지원하지 않는다.) 또한 PGCC (http://goof.com/pcg/), 소위 펜티엄 GCC 라는 것은 더 많은 옵션들을 가지고 있다. 구할 수 있는 장소는 다음과 같다 : http://goof.com/pcg/ 이처럼 특정 프로세서에 최적화 시키는 플래그들은 리눅스의 커널을 컴파일 할 때 볼 수 있을 것이다. 여러분의 GCC 에 딸려온 TeXInfo 문서를 확인하면 보다 많은 정보를 얻을 수 있다.

`-m386` 옵션은 코드의 크기를 최적화시킬 수 있도록 해준다. 그것이 중요한 이유는 메모리 사용율이 매우 높은 컴퓨터들에서 속도에 최적화되었지만, 그 최적화로 인해 크기가 매우 큰 프로그램을 실행시킨다면, 십중팔구 스와핑이 발생하게 되며, 그로 인해 원래의 목적인 빠른 속도의 효과에 반하는 일이 발생한다. 이럴 경우에는, C 를 쓰는 것 보다, 코드의 분할(?)을 구현한 언어 (functional language 혹은 FORTH 라고 불리우는 언어를 사용해서, 바이트코드나 워드코드에 기초한 구현을 하도록 하라.)

In such settings, it might be useful to stop using C, and use instead a language that favors code factorization, such as a functional language and/or FORTH, and use a bytecode- or wordcode- based implementation.

여러분은 최적화 옵션을 파일별로 다 다르게 지정할 수 있다. 그래서, 매우 좋은 성능이 요구되는 루틴을 담고 있는 파일에는 최고 레벨의 속도를 낼 수 있도록 최적화를 수행하고, 다른

파일에는 생성되는 코드의 크기를 작게 하는 방향으로 최적화할 수도 있다는 것을 유념하라.

좀 더 나은 최적화를 시키려면 `-mregparm=2` 옵션을 사용하거나, 해당하는 함수 특성을 사용하는것이 도움이 될 것이다. 그러나, 다른 코드(**libc**등과 같은)의 링크시에 몇가지 문제가 발생할 수 있다는 사실을 명심하라(역자주 : 번역이 정말-_- 매끄럽지 못합니다.) To optimize even more, option `-mregparm=2` and/or corresponding function attribute might help, but might pose lots of problems when linking to foreign code, **including libc**.

외부의 함수들을 정확하게 선언해 줌으로써, 올바른 순서에 따른 호출이 발생하도록 조정해 주거나, 외부 라이브러리들을 여러분이 만든 프로그램과 같은 레지스터를 사용하는 호출 컨벤션을 사용하도록 재컴파일 해 줌으로써, 문제를 극복할 수 있기도 하다.

There are ways to correctly declare foreign functions so the right call sequences be generated, or you might want to recompile the foreign libraries to use the same register-based calling convention...

`/usr/lib/gcc-lib/i486-linux/2.7.2.3/specs` 을 편집해서 여러분의 make 에 위에서 설명한 플래그들이 디폴트가 되도록 할 수도 있다.(그러나 그 경우, `-W -Wall` 옵션은 덧붙이지 않는 것이 나은 경우도 있다.) 여러분의 시스템에서 GCC specs 파일의 정확한 위치는 **gcc -v** 를 함으로써 알 수 있다.

3.1.4. 매크로의 지원

GCC 는 여러분이 여러분의 인라인 어셈블리 코드 내에 레지스터 규약(? - constraints)을 명시하는 것을 가능하게 합니다. 그럼으로써, 최적화 할 때 컴파일러가 그 사실을 참고하여 인라인 어셈블리 코드가 패턴들의 집합으로 생성되도록 합니다. (-_-; 너무 번역을 못하는 거 같군여)

GCC allows (and requires) you to specify register constraints in your inline assembly code, so the optimizer always know about it; thus, inline assembly code is really made of patterns, not forcibly exact code.

따라서 여러분은 여러분의 어셈블리를 CPP 매크로 라든지, 인라인의 C 함수들 내부에 포함시킬 수 있습니다. 그렇게 하면, 누구든지, 그 어셈블리 코드를 평범한 C 함수나 매크로처럼 사용할 수 있게 됩니다. 인라인 함수들은 매크로와 매우 비슷하게 보이지만, 때로는 더욱 명확하여 쓰기가 더 쉽다. 그러나, 매크로를 사용하든, 인라인 함수를 쓰던, 계속 중복된 코드를 쓰게 된다는 것을 명심하라. 따라서, `1:` 과 같은 스타일의 로컬 레이블만이 그러한 어셈블리 코드에서 정의되고, 사용되어야 한다. 그러나, 매크로는 로컬 레이블로 정의되지 않은 레이블이 파라미터로 넘겨지는 것을 허용한다. (혹은, 여러분은 부가적인 메타 프로그래밍 기법을 사용할 수 있을 것이다.) 또한, 인라인 어셈블리 코드를 확장하게 되면, 그 코드 안의 잠재적인 버그 또한 나타날 수 있다. 그러므로, 그러한 인라인 어셈블리 코드에서 레지스터 사용방법을 이중으로 고찰하도록 하라.

Thus, you can make put your assembly into CPP macros, and inline C functions, so anyone can use it in as any C function/macro. Inline functions resemble macros very much, but are sometimes cleaner to use. Beware that in all those cases, code will be duplicated, so only local labels (of `1:` style) should be defined in that asm code. However, a macro would allow the name for a non local defined label to be passed as a parameter (or else, you should use additional meta-programming methods). Also, note that propagating inline asm code will spread potential bugs in them; so watch out doubly for register constraints in such inline asm code.

마지막으로, C 언어 그 자체가 어셈블리 프로그래밍의 매우 좋은 추상화라는 것을 명심하고, C 를 사용함으로써, 어셈블리 프로그래밍시 발생할 수 있는 많은 난점들 극복하고, 보다 신뢰도 높은 프로그램을 여러분이 만들 수 있게 될 것이다.

Lastly, the C language itself may be considered as a good abstraction to assembly programming, which relieves you from most of the trouble of assembling.

3.2. GAS

GAS 는 GCC 가 사용하는 GNU 어셈블러이다.

3.2.1. 어디에서 GAS 를 찾을 수 있는가

GCC 를 찾은 곳과 같은 장소에 있다. 패키지 binutils 안에 패키징되어 있다.

최신의 GAS 버젼은 HJLu 에서 구할 수 있다. url 은 다음과 같다 : ftp://ftp.varesearch.com/pub/support/hjl/binutils/ (ftp://ftp.varesearch.com/pub/support/hjl/binutils/).

3.2.2. AT&T 문법

GAS 가 원래 32-bit 의 유닉스 컴파일러를 지원하도록 처음에 고안되었기 때문에, GAS 는 표준 AT&T 문법을 사용한다. 표준 AT&T 문법은 m68k 어셈블러의 문법과 상당히 유사하다. 현재 UNIX 계열 어셈블리의 표준 문법이기도 하다. 이 문법은 Intel 문법에 비해서 좋지도 않고, 나쁘지도 않다. 단지 다를 뿐이다. 여러분들이 그것을 사용해보게 되면, 인텔 문법에 비해 좀 더 많은 규칙이 있음 을 알게 될 것이다.

아래에 GAS 문법의 주요 사항들을 나열하겠다.

- 레지스터의 이름은 `%`문자로 시작한다. 그래서, 레지스터를 표기할 때 다음처럼 표시한다 : `%eax`, `%dl` 등등.. 이렇게 하는 것은 외부의 C 등에서 정의된 심볼을 아무런 혼란 없이 곧바로 어셈블리 소스에 포함시키는 것을 가능하게 해 준다.
- 오퍼랜드의 순서가, 소스가 먼저 오고, 데스티네이션이 뒤에 온다. 인텔의 문법과는 반대이다. 그래서, 인텔의 문법으로 쓴 문장이 `mov eax,edx` 과 같다면, GAS 의 문법으로는 `mov %edx,%eax` 과 같이 쓴다. (edx 레지스터의 내용을 eax 레지스터로 옮겨라...는 뜻)
- 오퍼랜드의 크기가 인스트럭션의 접미어로 붙는다. 가능한 접미어는 `b` 는 바이트를(8-bit), `w` 는 워드를(16-bit) `l` 는 long 형을(32-bit) 나타내는 접미어이다. 예를 들자면, 바로 앞 단락에서 언급한 인스트럭션의 올바른 문법은 `movl %edx,%eax` 가 되어야 한다. 그러나, GAS 는 엄격하게 AT&T 문법의 규칙을 준수할 것을 요구하지 않는다. 그래서, 오퍼랜드의 크기를 나타내는 접미어는 오퍼랜드로 쓰인 레지스터의 이름으로부터 오퍼랜드의 크기를 유추할 수 있을 경우에는 생략해도 된다. 또한 그렇지 않을 경우라고 하더라도, 32-bit 의 크기를 가졌다고 가정한 상태에서 동작한다(경고 메세지를 뿌린다)

- 오퍼랜드에 레지스터가 아니라 상수 등을 사용할 때에는 그 오퍼랜드 앞에 `$` 의 접두어를 붙여준다. 즉, `addl $5,%eax` 은, `%eax` 레지스터에 long 형 숫자 5 를 넣으라는 뜻이다.
- 오퍼랜드에 접두어가 없을 경우에는 그것이 메모리 주소라는 것을 의미한다. 그래서, `movl $foo,%eax` 은 변수 `foo` 의 주소 를 레지스터 `%eax` 에 넣으라는 뜻이 된다. 그러나, `movl foo,%eax` 은 변수 `foo` 의 값을 `%eax` 레지스터에 넣으라는 뜻이다.
- 인덱싱이나 간접 주소지정 방식은 인덱스 레지스터를 이용하거나 간접 메모리 주소의 번지를 괄호 안에 넣음으로써 표현되어진다. 예를 들어, `testb $0x80,17(%ebp)` 과 같은 표현은 `%ebp` 가 가리키는 메모리 번지에서 17번째 오프셋에 있는 바이트 값의 최상위 비트를 테스트하라는 말이 된다.

주의 : AT&T 문법을 이용한 어셈블리 소스코드와 인텔의 문법을 이용한 소스 코드 간에 서로 변환을 시켜 주는 프로그램은 거의 없다.(few programs)

Note: There are few programs which may help you to convert source code between AT&T and Intel assembler syntaxes; some of the are capable of performing conversion in both directions.

GAS 는 패키지에 TeXinfo 형식으로 된 매우 이해하기 쉬운 문서를 가지고 있다. `.info` 페이지 형식으로 Emacs 등에서 브라우징 할 수도 있다. 파일명은 gas.doc 이거나 as.doc 일 것이다. 의심의 여지 없이 최고의 문서는 소스코드 자체이다.

다시한번, 리눅스의 소스가 최상의 예제임을 상기하라. `linux/arch/i386/` 디렉토리 아래에서 `kernel/*.S`, `boot/compressed/*.S`, `mathemu/*.S`. 과 같은 파일을 열어서 읽어 보아라. 최고의 예제이다.

만약 여러분이 언어를 하나 고안하고 있거나, 쓰레드 패키지를 프로그래밍하고 있다면, 다른 언어(OCaml (http://para.inria.fr/), Gforth (http://www.jwdt.com/~paysan/gforth.html), etc.) 나 스레드 패키지들 (QuickThreads, MIT pthreads, LinuxThreads, etc) 을 살펴 보기를 원할 것이다. 그리고, 살펴 보면 된다!!!

마지막으로, 단지, C 프로그램을 어셈블리로 컴파일 하기만을 원한다면, 단지, 여러분이 원하는 인스트럭션의 종류와 문법만 알면 될 것이다. 위에서 여러분이 정말 어셈블리 프로그래밍이 필요한가를 판단하도록 돕는 섹션을 보도록 하라 : See section Do you need assembly? above.

3.2.3. GAS 의 인텔 문법 지원

binutils 2.10 부터 GAS 는 인텔 계열의 문법 또한 지원한다. `.intel_syntax` 지시어를 이용하여 활성화 시킨다.

3.2.4. 16-비트 모드

Binutils (2.9.1.0.25+) 이후의 버전부터는 i386 계열의 컴퓨터들에서 16비트 모드(레지스터와 주소지정방식)를 완벽하게 지원한다. 어셈블리 코드에서 16비트와 32비트 모드를 전환할 때는 `.code16` 과 `.code32` 지시자를 사용하면 된다.

또한, 몇몇 사람들(oskit 의 저자들도 포함)이 사용했던 깔끔한 트릭이 있는데, 그것은 GCC 로 하여금 16 비트 리얼모드용의 코드를 생성하도록 강제하는 것인데, 인라인 어셈블리 문장인 `asm(".code16\n")` 을 씀으로써 그렇게 했다. 그 문장을 사용하면, GCC 는 여전히 32비트의 주소지정 방식을 사용한 코드를 생산하지만, GAS 가 적당한 32비트의 접두어를 붙여준다.

3.2.5. 매크로 지원

GAS 에는 얼마의 매크로와 관련된 기능도 포함되어 있다. 자세한 내용은 texinfo 문서를 보라. 매크로에 더하여, GCC 는 `.s` 확장자의 파일을 GAS 에 바로 보내여야 할 순수한 어셈블리로 인식하는데, 또한, `.S` 의 확장자를 가지는 파일은 GAS 로 넘기지 전에 먼저 CPP 를 거친 후에 넘겨야 할 파일로 인식한다. 계속 말하는 것이지만, 리눅스의 소스를 예제삼아 보라.

GAS 는 또한, GASP (GAS Preprocessor) 를 가지고 있는데, GASP 가 하는 일은 일반적인 매크로어셈블리의 트릭을 GAS 에 더해주는 것이다. GASP 는 GNU binutils 패키지 안에 GAS 와 더불어서 배포된다. GASP 는 CPP 나 m4 유틸리티처럼 필터로써 동작한다. 자세한 것은 저자가 잘 모르겠으나, texinfo 형식의 문서가 따라오므로, 문서를 참조하기 바란다. 저자에겐 GAS 와 GASP 의 조합이 일반적인 macro-assembler 처럼 느껴졌다.

3.3. NASM

Netwide Assembler project 는 매우 괜찮은 i386 어셈블러를 제공한다. 그 어셈블러는 C 로 쓰여졌으며, 충분히 모듈화 되어 있다. 그리고, 지금은 거의 모든 알려진 문법과, 목적 파일의 형식을 지원하게 되었다.

3.3.1. NASM을 어디서 구할 수 있는가

http://www.cryogen.com/nasm/, http://nasm.rm-f.net

여러분이 자주 가는 metalab 미러 사이트의 `devel/lang/asm/` 디렉토리에 있다. 아마도 rpm 이나 deb 형식의 패키지로도 구할 수 있을 것이다.

3.3.2. NASM 으로 무엇을 할 수 있는가

문법은 인텔 스타일이다. 또한, 이해하기 쉬운 매크로 프로세싱에 대한 지원도 한다.

지원되는 목적 코드의 형식은 다음과 같다 : `bin`, `aout`, `coff`, `elf`, `as86`, `obj` (DOS), `win32`, `rdf` (NASM 고유의 포맷).

NASM 은 또한 LCC 컴파일러의 백엔드로도 사용될 수 있다.

여러분이 BCC 를 16비트 컴파일러로 사용하지 않는다면 여러분은 아마도, AS86 이나 MASM 보다 NASM 을 사용해야 할 것이다. NASM 은 온라인으로 계속 지원이 이루어지고 있

고, 또한, 거의 모든 플랫폼에서 돌아가기 때문이다.

Note: NASM 에는 디스어셈블러인 NDISASM 도 딸려온다.

NASM 은 사람의 손으로 작업해낸 파서를 탑재하고 있는데, 그로 인해 GAS 보다 훨씬 빠르다. (역자 주 : 그 이후의 말은... 해석이 좀 힘들군요 -_- 원문을 그대로 두겠습니다.) Its hand-written parser makes it much faster than GAS, though of course, it doesn't support three bazillion different architectures. If you like Intel-style syntax, as opposed to GAS syntax, then it should be the assembler of choice..

3.4. AS86

AS86 은 16비트와 32비트 모두를 지원하는 80x86 용의 어셈블러이다. 이 프로그램은 Bruce Evans 의 C 컴파일러(BCC) 패키지의 일부이다. 거의 대부분 인텔의 문법을 따르지만, 주소 지정방식 등에서 약간의 차이를 보인다.

3.4.1. AS86 을 어디서 구할 수 있는가

완전히 시대에 뒤떨어진 AS86 의 버젼은 HJLu 에 의해 배포된다. 용도는 단지 리눅스 커널을 컴파일하는것 밖에 없다. bin86 패키지 (현재 버젼 0.4) 안에 포함되어 있다. 리눅스 GCC 를 구할 수 있는 곳이면 어디에나 있다. 그러나, 저자는 리눅스를 컴파일하는 용도 외에는 아무도 AS86 을 쓰지 않는다고 말해주고 싶다. 게다가 리눅스를 컴파일하는데에도, 미닉스의 오브젝트 파일 포맷을 지원하는 부분 외에는 쓰이지 않는다. 이 분야는 GNU binutils 등과 같은 다른 툴에서는 전혀 지원하지 않는 분야이다. 그리고, AS86 은 32비트 모드에서 몇가지 버그를 내포하고 있다. 그냥 AS86 을 리눅스를 컴파일하는 데에만 사용하라.

Bruce Evans 의 컴파일러의 최신 버젼은 항상 FreeBSD 와 함께 배포된다. 2.1 버젼의 배포판에서는 소스를 찾을 수 없었다 :(저자의 홈페이지에 소스를 두겠다. http://www.tunes.org/~fare/files/asm/bcc-95.3.12.src.tgz (http://www.tunes.org/~fare/files/asm/bcc-95.3.12.src.tgz)

어쨌든 Linux/8086 (aka ELKS) 프로젝트는 bcc 를 유지/보수 하고 있기는 하다. (저자는 32 비트 패치가 포함되었다고는 생각하지 않는다) http://www.linux.org.uk/ELKS-Home/ (http://www.linux.org.uk/ELKS-Home/) (혹은 http://www.elks.ecs.soton.ac.uk (http://www.elks.ecs.soton.ac.uk)) 와 ftp://linux.mit.edu/pub/linux/ELKS/ (ftp://linux.mit.edu/pub/linux/ELKS/) 를 둘러보도록 하라. 저자가 여기 올라오는 개발 버젼들을 모두 확인해 볼 수 없었다. 이 글을 읽으시는 분이 관심이 있다면 조사해 보고 저자에게 이야기해 준다면 좋겠다.

그리고, 무엇보다 이 툴의 보다 최신의 버젼은 리눅스의 a.out 포맷의 실행파일 형식을 지원한다. 그래서, 여러분은 여러분의 코드를 리눅스 프로그램에 링크시켜서 사용하거나, GNU binutils 패키지의 툴들을 여러분의 프로그램의 자료를 유지하는데 사용할 수 있을 것이다. AS86 프로그램은 이전 버젼 혹은 다른 프로그램과 아무런 충돌 없이 여러분의 시스템에 공존할 수 있다.

1995 년 3월 12일 이전의 버젼의 BCC 는 모든 세그먼트의 푸시/팝 을 16비트로 하는 오류가 있는데, 그것은 32비트 모드로 프로그래밍 하는데 있어서 상당히 귀찮고, 골치아픈 문제이다. 저자는 TUNES 프로젝트가 AS86 을 사용할 당시 그에대한 패치를 했었다.
http://www.tunes.org/~fare/files/asm/as86.bcc.patch.gz
(http://www.tunes.org/~fare/files/asm/as86.bcc.patch.gz) 브루스 에반스도 이 패치를 승인하였다. 그렇지만, 그는 bcc 의 새로운 릴리즈는 발표하지 않았다.

3.4.2. 어셈블러를 작동시키는 방법 : How to invoke the assembler?

아래에 BCC 를 사용해서 `.s` 파일로부터 a.out 형식의 `.o` 오브젝트 코드와 `.l` 의 리스팅 파일을 생성시키는 생성시키는 GNU makefile 의 엔트리가 있다.

```
%.o %.l: %.s
    bcc -3 -G -c -A-d -A-l -A$*.l -o $*.o $<
```

여러분이 리스팅 파일을 생성하기를 원하지 않는다면, `%.l` 과 `-A-l`, 그리고 `-A$*.l` 을 삭제하면 된다. 그리고, a.out 말고 다른 파일이 출력되게 하려면, bcc 의 문서를 참조하여 다른 지원되는 형식을 찾아보거나, GNU binutils 패키지의 objcopy 유틸리티를 사용하라.

3.4.3. AS86의 문서를 찾을 수 있는 곳

문서들은 bcc 패키지 안에 포함되어 있다. FreeBSD 사이트인
http://www.tunes.org/~fare/files/asm/bcc-95.3.12.src.tgz
(http://www.tunes.org/~fare/files/asm/bcc-95.3.12.src.tgz) 에서 맨페이지를 구할 수 있었다.
ELKS 의 개발자들이 더 많이 알고 있을 것이다. 흠... 그리고, 좀 더 알고 싶다면, AS86 의 소스 그 자체가 좋은 문서가 될 것이다. 주석이 그리 잘 되어 있는 것은 아니지만, 프로그래밍 스타일은 매우 직선적(?)이다. 여러분은 ELKS 프로젝트나 TUNES 프로젝트 0.0.0.25 에서 AS86 이 어떻게 쓰였는지 살펴볼 수도 있을 것이다.

3.4.4. 매크로 지원

AS86 은 간단한 매크로도 지원한다. 그렇지만, 저자는 그에대한 문서는 찾지 못하였다. AS86의 소스는 매우 straightforward 하게 쓰였으므로, 여러분이 원한다면, 소스를 살펴보고, 원하는 부분을 쉽게 이해할 수 있을 것이다. 여러분이 단순한 매크로 이상의 더 많은 것을 원한다면 외부의 필터 (뒤에서 설명한다)를 이용할 수도 있다.

3.4.5. 만약 이 새로운 버젼을 이용해서 리눅스를 더이상 컴파일 할 수 없다면 어떻게 해야 하는가 : What if I can't compile Linux anymore

with this new version?

Linus 는 쏟아지는 메일더미에 파묻혀 :) 있다. 그리고, HJLu (bin86 의 공식적 유지/보수자) 가 as86 의 as86 의 최신 버전을 사용한 코드의 재 구축 보다 구버전의 해킹을 선택했기 때문에 저자가 as86 에 리눅스를 컴파일 할 수 있도록 한 패치가 제대로 받아들여지지 않았다고 생각한다. (역자주 : 번역이 이상해서 원문을 싣습니다) Linus is buried alive in mail, and since HJLu (official bin86 maintainer) chose to write hacks around an obsolete version of as86 instead of building clean code around the latest version, I don't think my patch for compiling Linux with a modern as86 has any chance to be accepted if resubmitted.

그러나, 별 문제 없다. bin86 패키지에서 as86 을 `/usr/bin/` 에 그대로 놔두고, bcc 가 최신의 as86 을 `/usr/local/libexec/i386/bcc/as` 로 인스톨하도록 해 주면 된다. 여러분은 명시적으로 /좋은/ 버젼의 as86 을 호출할 필요는 없다. 제대로 된 옵션과 함께 호출되면, bcc 가 모든것을 제대로 조정해 주기 때문이다. 리눅스의 a.out 포맷으로의 변환까지 포함해서 말이다. 그래서, 파일을 as86 을 바로 사용하지 말고, bcc 를 프론트엔드로 이용해서 어셈블하라. Now, this shouldn't matter: just keep your as86 from the bin86 package in `/usr/bin/`, and let bcc install the good as86 as `/usr/local/libexec/i386/bcc/as` where it should be. You never need explicitly call this "good" as86, because bcc does everything right, including conversion to Linux a.out, when invoked with the right options; so assemble files exclusively with bcc as a frontend, not directly with as86.

지금은 gas 도 16비트 코드도 지원하고, NASM 도 그렇게 할 모양이므로, 리눅스는 AS86 의 그늘에서 벗어날 수 있을 것이다. 누가 알리... Since GAS now supports 16-bit code, and NASM looks promising, maybe Linux will get rid of AS86, anyway? Who knows!

3.5. 다른 어셈블러들

지금부터 소개하고자 하는 어셈블러들은 일반적으로 사용하는 것이 아닌 어셈블러들이다. 앞서 언급한 어셈블러들이 불만족스럽다면 지금부터 소개하는 어셈블러들을 사용하라. 그러나 앞서 언급한 어셈블러들이 불만족스러울 이유가 없다. :) 그래서 저자는 앞서의 어셈블러들을 사용하기를 추천하는 바이다. 하지만, 여러분이 개발하는 소프트웨어에 사용하는 어셈블러를 일률적으로 같은 것으로 할 필요는 없다.

3.5.1. Win32Forth 어셈블러

Win32Forth 는 **free** 32-bit ANS FORTH system 이다. Win32 환경, 즉, 윈95, 윈NT 등에서 성공적으로 사용할 수 있는 어셈블러이다. 이 어셈블러는 reflective FORTH language 에 포함되어 있는 공개 32비트 어셈블러이다. (문법은 prefix 나 postfix 모두를 지원한다) 매크로 프로세싱도 reflective language FORTH 의 능력 하에서 성공적으로 잘 지원된다. 그러나, 단점이 있다면, 지원되는 입.출력 내용이 Win32Forth 에서만 사용할 수 있다는 것이다. (.obj 파일로의 덤프는 없다. 그러나 여러분이 필요하다면 그러한 기능을 추가할 수 있다) 이 어셈블러를 찾을 수 있는 장소는 다음과 같다 : ftp://ftp.forth.org/pub/Forth/Compilers/native/windows/Win32For/ (ftp://ftp.forth.org/pub/Forth/Compilers/native/windows/Win32For/).

Win32Forth is a **free** 32-bit ANS FORTH system that successfully runs under Win32s, Win95, Win/NT. It includes a free 32-bit assembler (either prefix or postfix syntax) integrated into the reflective FORTH language. Macro processing is done with the full power of the reflective language FORTH; however, the only supported input and output contexts is Win32For itself (no dumping of `.obj` file, but you could add that feature yourself, of course). Find it at ftp://ftp.forth.org/pub/Forth/Compilers/native/windows/Win32For/.

3.5.2. TDASM

TDASM (the Table Driven Assembler) 은 어떤 어셈블리 언어로도 포팅이 가능한 공개 어셈블러이다. 이 어셈블러는 타겟이 되는 마이크로프로세서가 무엇이든 간에 그에 해당하는 컴파일 작업을 정의하는 테이블을 이용하여서 해당 마이크로프로세서의 컴파일러로 사용이 가능한 어셈블러이다.

The Table Driven Assembler (TDASM) is a **free** portable cross assembler for any kind of assembly language. It should be possible to use it as a compiler to any target microprocessor using a table that defines the compilation process.

구할 수 있는 장소는 http://www.penguin.cz/~niki/tdasm/ (http://www.penguin.cz/~niki/tdasm/) 이다.

3.5.3. Terse

Terse (http://www.terse.com) 는 x86 계열을 위한 가장 컴팩트한 어셈블러 문법을 가지고 있는 프로그래밍 툴이다. 하지만, 이 어셈블러는 evil proprietary software 이다 (역자주:마땅한 말을 못찾았습니다) 이 어셈블러의 공개 클론을 만드는 프로젝트가 있었다고는 하는데, 지금은 더 이상 개발되지 않는 것으로 필자는 알고 있다. 만약 여러분이 terse 와 비슷한 문법을 가진 어셈블러를 찾는다면, NASM 의 프론트엔드로 그 문법을 개발하면 될 것이다. 관심있다면, comp.compilers (news:comp.compilers) 에서 역사적인 토의들(?)을 훑어볼 수 있을 것이다.

> 1999/07/11 19:36:51, the moderator wrote:
>
> "There's no reason that assemblers have to have awful syntax. About
> 30 years ago I used Niklaus Wirth's PL360, which was basically a S/360
> assembler with Algol syntax and a a little syntactic sugar like while
> loops that turned into the obvious branches. It really was an
> assembler, e.g., you had to write out your expressions with explicit
> assignments of values to registers, but it was nice. Wirth used it to
> write Algol W, a small fast Algol subset, which was a predecessor to
> Pascal. As is so often the case, Algol W was a significant
> improvement over many of its successors. -John"

3.5.4. HLA

HLA (http://webster.cs.ucr.edu) 는 **H**igh **L**evel **A**ssembly language 의 약자이다. 이 어셈블러는 변수 선언이나 함수 선언, 함수 호출과 같은 부분에서 파스칼이나 C/C++ 과 같은 고급언어와 매우 비슷한 문법을 사용한다. 그리고 이 어셈블러는 표준 머신 인스트럭션을 위한 수정된 어셈블리 언어 문법을 사용한다. 또한 if, while, repeat, until 등과 같은 고급언어에서 제공하는 제어 구조를 가진다.

HLA 는 free 이지만, Win32 환경에서만 동작한다. HLA 는 MASM 용의 코드를 생산하기 때문에 목적파일을 생성하기 위한 최종 어셈블링과 링킹을 위해 여러분은 MASM 과 MS-link 의 32비트 버젼이 필요하다. 그러나, HLA 의 MASM 버젼의 출력을 TASM 에서 컴파일 될 수 있도록 변환해주는 **m2t** 포스트 프로세서 프로그램도 따라온다. 불행히도, NASM 은 지원되지 않는다.

3.5.5. TALC

TALC (http://www.cs.cornell.edu/talc/) 는 또다른 MASM/Win32 기반의 공개 컴파일러이다. (그럼에도 불구하고, ELF 형식의 출력도 지원하는 것 같다)

TAL 은 **T**yped **A**ssembly **L**anguage 의 약자이다. (역자주:정말.. 아래의 문장들은 전문적인 용어라.. 번역이 힘들군요.. 누군가 번역해주실 분은... 짧으니까 저에게 메일 주십시오. 반영하겠습니다) It extends traditional untyped assembly languages with typing annotations, memory management primitives, and a sound set of typing rules, to guarantee the memory safety, control flow safety,and type safety of TAL programs. Moreover, the typing constructs are expressive enough to encode most source language programming features including records and structures, arrays, higher-order and polymorphic functions, exceptions, abstract data types, subtyping, and modules. Just as importantly, TAL is flexible enough to admit many low-level compiler optimizations. Consequently, TAL is an ideal target platform for type-directed compilers that want to produce verifiably safe code for use in secure mobile code applications or extensible operating system kernels.

3.5.6. x86 용의 프리가 아니거나 32비트가 아닌 어셈블러들

여러분은 Raymond Moon’s x86 assembly FAQ 에서 x86 어셈블리 프로그래밍의 기초적인 지식과 함께 그러한 어셈블리에 대해 보다 많은 자료들을 찾을 수 있다.

Note : 모든 도스 기반의 어셈블러들은 리눅스의 DOS 에뮬레이터를 통해서 사용되어야 한다. 최근의 도스 기반의 어셈블러들은 GNU BFD 라이브러리에서 지원하는 COFF 나 다른 목적 파일 포맷들을 지원하기 때문에 여러분은 그러한 어셈블러들을 다른 공개의 32비트 툴들과 함께 사용할 수도 있을 것이다. 아마도 GNU objcopy(binutils 패키지에 있다) 를 변환용 필터로 이용할 수도 있을 것이다.

4장. 메타프로그래밍

어셈블리 프로그래밍은 지루한 작업이다. 그러나, 그렇다고 결정적인 부분에서 어셈블리를 사용하지 않을수는 없다.

여러분은 제대로 작업을 수행하기 위해서 적절한 툴을 사용해야 할 것이다.... 다시 말하자면, 어셈블리 언어가 반드시 필요하지 않다면 어셈블리를 사용해서는 안된다. 대신, C, OCaml, perl, Scheme, 등의 고급언어를 사용하는 것이 대개의 경우 더 나은 선택이 될 것이다.

그러나, 앞서 언급한 고급 프로그래밍 언어가 기계 자체를 만족할 만한 수준으로 컨트롤 할 수 있는 기능을 제공하지 못하는 경우가 있다. 다시 말해서, 그러한 언어가 제공하는 머신에 대한 제어기능 보다 더 세밀하게 하드웨어를 컨트롤 해야 하는 경우에 어셈블리가 유용하거나, 필요할 것이다. 그러한 경우, 여러분은 매크로 프로세싱이나 메타 프로그래밍 툴을 유용하게 쓸 수 있다. 그러한 툴을 사용하면, 여러분은 패턴을 이용한 프로그래밍을 할 수 있게 된다. 다시 말해서, 한 패턴에 대한 정의를 내려 두면, 그것을 이용하여 안전하게, 또 재사용 가능하게 매크로 처럼 사용하여 프로그래밍을 할 수 있게 되는 것이다. 그러한 방식으로 프로그래밍을 한다면, 여러분이 만든 코드는 보다 안전하며, 수정할 때에도 패턴의 정의만 바꿔주면 자동으로 수정된 버젼의 소스가 다시 생성되게 하는 등의 여러가지 잇점이 있다. 일반적으로 사용하는 어셈블러만 가지고는 심지어 작은 루틴을 C 와 연결하려고 할 때에도 이와같이 복잡한 기능을 수행할 수 없다.

4.1. 외부 필터들

여러분이 즐겨 사용하는 어셈블러에서 지원하는 매크로가 무엇이든, 여러분이 주로 사용하는 언어가 무엇이든지간에 (심지의 C 일 경우에도) 그 언어가 여러분이 생각하기에 충분히 expressive(모든 것을 표현할 수 있는) 하지 않다면, 여러분은 여러분의 소스 파일을 외부의 필터를 통과해서 재생성(?)되게 할 수 있으며, Makefile 에서 다음에 보여지는 것과 같은 간단한 규칙만 추가 함으로서 그렇게 할 수 있다.

```
%.s:    %.S other_dependencies
        $(FILTER) $(FILTER_OPTIONS) < $< > $@
```

4.1.1. CPP

CPP 는 실은 그리 표현적이지 못하다(is truly not very expressive), 하지만, 간단한 일을 하기 위해서는 CPP 로도 충분하다. CPP 는 표준이며, GCC 에 의해서 투명하게 호출되어진다.

CPP 의 한계에 대한 예를 들자면, 여러분은 여러분의 객체가 선언되어서, 사용할 때, 그 객체가 유효한 부분의 마지막에 자동으로 소멸자(destructor) 가 호출되게 하는 기능을 구현할 수는 없다.

As an example of its limitations, you can’t declare objects so that destructors are automatically called at the end of the declaring block; you don’t have diversions or scoping, etc.

CPP 는 대부분의 C 컴파일러 패키지들에 포함되어 있다. 그러나, 특별히 뛰어나거나 하지도 않다.

4.1.2. M4

M4 는 여러분에게 매우 훌륭하고, 강력한 매크로 프로세싱 도구가 되어 줄 것이다. 게다가 Turing 과 같은 언어를 지원하고, 재귀, 정규 표현식 등도 지원한다. 여러분은 CPP 로 하지 못했던 모든 것을 M4 를 가지고 할 수 있게 될 것이다.

M4 gives you the full power of macroprocessing, with a Turing equivalent language, recursion, regular expressions, etc. You can do with it everything that CPP cannot.

m4 를 사용한 매크로 프로그래밍의 예제를 구하기를 원한다면, macro4th (this4th - ftp://ftp.forth.org/pub/Forth/Compilers/native/unix/this4th.tar.gz) (ftp://ftp.forth.org/pub/Forth/Compilers/native/unix/this4th.tar.gz) 나 the Tunes 0.0.0.25 sources(ftp://ftp.tunes.org/pub/tunes/obsolete/dist/tunes.0.0.0/tunes.0.0.0.25.src.zip) (ftp://ftp.tunes.org/pub/tunes/obsolete/dist/tunes.0.0.0/tunes.0.0.0.25.src.zip) 을 참조하기 바란다.

그러나, However, its disfunctional quoting and unquoting semantics force you to use explicit continuation-passing tail-recursive macro style if you want to do **advanced** macro programming (which is remindful of TeX -- BTW, has anyone tried to use TeX as a macroprocessor for anything else than typesetting ?). This is NOT worse than CPP that does not allow quoting and recursion anyway.

구해야 할 M4 의 버전은 GNU m4 1.4 혹은 그 이상의 버젼이다. 가장 많은 기능과 가장 적은 버그 혹은 제한을 가지고 있는 버전이기 때문이다. m4 는 설계될 때, 간편한 사용을 목적으로 설계되고, 빠른 속도를 목표로 해서 설계되지 않았기 때문에, 무엇을 하든지 좀.. 느리다. 그러나, 어셈블리 프로그램을 만들때에는 별 문제가 되지 않는다. 어셈블리로 몇백만 라인이나 되는 양을 코딩할 것인가? 그렇지 않다.

4.1.3. 여러분이 만든 필터를 이용한 매크로 프로세싱

여러분은 perl 이나 awk, sed 등을 가지고도 여러분 자신의 간단한 매크로 확장 필터를 만들 수 있다. 오히려 간단한 것은 이러한 툴을 가지고 만드는 것이 더 나을지도 모른다. 그러나. 매크로 프로세싱이란 말은 "the hard way(좀 더 힘든 방법?)" 라는 뉘앙스를 풍긴다. You can write your own simple macro-expansion filter with the usual tools: perl, awk, sed, etc. It can be made rather quickly, and you control everything. But, of course, power in macroprocessing implies "the hard way".

4.2. 메타프로그래밍

매크로를 확장시켜 주는 외부필터를 사용하는 대신 사용할 수 있는 또다른 방법이 있는데, 그것은 여러분의 프로그램의 일부 혹은 전체를 만들어주는 프로그램을 짜는 것이다.

무슨 말인고 하니, 예를 들어 설명해 보면, 여러분은 소스 코드를 출력해주는 프로그램을 사용할 수도 있다. 그러한 프로그램의 예를 들자면,

- 사인(sine)이나 코사인(cosine) 등의 함수값의 테이블을 찾아서 적절한 값을 생성시켜 주는 프로그램
- 이진파일에서 소스-폼(source-form) 을 추출하는 프로그램
- 여러분의 비트맵을 컴파일해서 고속 출력 루틴을 만들어주는 프로그램
- 초기화와 종료 코드를 만들어주거나, 문서를 추출하거나, 하는 프로그램

  to extract documentation, initialization/finalization code, description tables, as well as normal code from the same source files,

- 어떤 일을 하는 쉘이나 펄 코드를 어셈블리 코드로 바꾸어 주는 프로그램
- 한곳에서 정의된 데이터를 여러가지의 상호 참조하는 테이블과 코드 청크들로 생성시켜 주는 프로그램
- 등이 있을 수 있다. (역자의 생각으로는 yacc 나 lex 등과 같은 프로그램을 이야기하는 것 같다 : 역자주 : yacc 나 lex 를 이용해서 간단한 컴파일러나 어셈블러를 만들 수 있다. 사용할 랭귀지의 문법을 입력하면, 해당하는 컴파일러가 생성된다.)

그러한 프로그램들을 한번 고려해 보라!

4.2.1. 컴파일러의 백엔드로 제공되는 어셈블러의 이용

GCC, SML/NJ, Objective CALM, MIT-Scheme, CMUCL 등과 같은 컴파일러들은 고유의 어셈블러 백엔드를 갖추고 있다. 여러분은 이러한 백엔드를 이용해서 어셈블리 코드를 얻을 수 있다. 여러분이 사용하는 언어를 이용해서 만든 프로그램을 앞서 언급한 컴파일러로 컴파일해서 훌륭한 어셈블리 코드를 얻을 수 있고, 그 코드를 여러분이 더 낫게 수정해서 마음에 드는 어셈블리 코드를 얻을 수 있다.

4.2.2. 뉴저지 머신-코드 툴킷

프로그래밍 언어라는 아이콘을 사용해서 기계어를 다루는 툴킷의 개발과 관련된 프로젝트가 있다(실험적인 ML 버젼도 제공한다) http://www.eecs.harvard.edu/~nr/toolkit/ (http://www.eecs.harvard.edu/~nr/toolkit/)을 한번 둘러보라

4.2.3. TUNES 프로젝트

Free Reflective Computing System 을 위한 TUNES Project(http://www.tunes.org) (http://www.tunes.org) 에서도 자체적인 어셈블러를 Scheme 언어의 확장으로 개발하고 있다. 아직 제대로 실행되는 것은 아니지만 말이다.

tunes 프로젝트에서 개발중인 어셈블러는 abstract syntax 트리를 다루는데, 그렇게 하여, 디스어셈블러로도, 어셈블러 문법 번역기로도, 컴파일러나 어셈블러들의 공통적인 백엔드로도 사용할 수 있도록 하고 있다. 또한, Scheme 이라는 언어의 강력함은 매크로 프로세싱이나 메타 프로그래밍처럼 도전할 만한 분야이다.(?)

The assembler manipulates abstract syntax trees, so it could equally serve as the basis for a assembly syntax translator, a disassembler, a common assembler/compiler back-end, etc. Also, the full power of a real language, Scheme, make it unchallenged as for macroprocessing/metaprogramming.

5장. 함수 호출 규칙

5.1. Linux

5.1.1. GCC 로의 링킹 : Linking to GCC

만약 여러분이 C 와 어셈블리가 혼합된 프로젝트를 수행하고 있다면, 지금 소개하고자 하는 방법을 추천하고자 한다. 리눅스 커널에서 `.S` 의 확장자를 가진 파일들 중에서, gas 가 처리하는 (as86 이 처리하는 파일이 아님!) 파일들을 통해서 유용한 예제를 발견할 수 있다.

32비트의 argument 들은 스택에 푸시될 때, 32비트의 근거리 반환 주소 (32-bit near return address) 의 윗부분에 역순으로 푸시되어진다. 따라서, 그것들이 pop 될 때에는 올바른 순서로 pop 되어진다. `%ebp`, `%esi`, `%edi`, `%ebx` 레지스터들은 피호출자가 저장하게 되고, 다른 레지스터들은 호출자가 저장하게 된다. `%eax` 는 결과가 32비트 일때 결과를 저장하는 용도로 사용되어지고, 만약, 결과가 64비트라면, 그때는 `%edx:%eax` 레지스터 쌍이 결과를 저장하는데 사용된다.

FP stack 에 대해서는 확실해 잘 모르겠지만, 저자는 결과가 `st(0)` 에 저장되고, 스택 전체가 호출자가 저장하는 용도로 쓰인다고 생각한다.

GCC 가 레지스터들의 용도를 미리 지정해서 위에서 설명한 호출 규칙을 수정하는 옵션을 가지고 있다는 것을 잊지 않도록 하라. 자세한 내용은 i386 `.info` 페이지를 참조하라.

여러분은 GCC 의 표준 함수 호출 규칙을 따르는 함수를 위해서 `cdecl` 이나 `regparm(0)` attribute 를 선언해야만 한다는 것을 기억하라. GCC info 페이지의 `C Extensions::Extended Asm::` 섹션을 참조하라. 또한, Linux 가 `asmlinkage` 매크로를 어떻게 정의하고 있는지도 참조하라.

5.1.2. ELF 와 a.out 문제

몇몇 C 컴파일러들은 밑줄 문자를 다른 심볼들 보다 먼저 확장한다. 다른 컴파일러는 그렇게 하지 않지만 (역자 주 : 원문의 내용은 Some C compilers prepend an underscore before every symbol, while others do not. 입니다)

Linux 의 a.out GCC 는 그러한 확장을 한다. 그렇지만, ELF GCC 는 하지 않는다.

여러분이 그러한 특성을 극복할 필요가 있다면, 현존하는 패키지들은 어떻게 하는지를 참조하라. 예를 들어, 오래된 리눅스의 소스트리나, 혹은 Elk, qthreads, OCaml 등과 같은 패키지를 구해서 살펴볼 수 있을 것이다.

여러분은 또한, 다음과 같은 문장을 삽입함으로써 함축된(implicit) C -> asm 의 renaming 규칙을 override 할 수도 있을 것이다.

```
void foo asm("bar") (void);
```

그렇게 하면, `foo()` C 함수는 실제로 어셈블리에서는 `bar` 이라고 불리우게 될 것이다.

binutils 패키지의 **objcopy** 유틸리티가 여러분의 a.out 오브젝트를 ELF 형식의 오브젝트 파일로 변환해 줄 수 있음을 주목하라. 그리고, 그 반대의 일도 가능할 것이다. 보다 일반적으로 이야기하자면, 그 유틸리티는 더 많은 형태의 파일 형식 변환도 해 줄 것이다.

5.1.3. Direct Linux syscalls

여러분은 종종 C library (libc) 를 사용하는 것이 유일한 방법이며, 직접적인 시스템 콜은 별로 좋지 않다는 말을 들을 것이다. 이것은 사실이다. 그러나 일반적으로 여러분은 libc 가 신성시되고 있지 않다는 것과, 대부분의 경우 libc 는 몇가지 체크만 하고, 커널을 호출한 후, errno 변수를세팅할 뿐이라는 것을 알아야 한다. 여러분은 여러분의 프로그램에서도 이와 같은 것을 쉽게 할 수 있다 (만약 여러분이 그것을 필요로 한다면). 그리고 여러분이 공유 라이브러리를 사용하지 않는다는 이유만으로도 여러분의 프로그램은 몇십배 더 크기가 줄어들 것이며, 수행성능 또한 더 좋아질 것이다. libc 을 어셈블리 프로그래밍에서 사용하거나 하지 않거나 하는 문제는 실용적인 문제라기 보다는 기호나 신념에 관한 문제이다. Linux 는 POSIX 호환을 목표로 하고 있다는 것을 기억하라. 그리고, libc 는 그러한 목표를 만족한다. 이것은 libc system call 의 거의 모든 문법이 커널의 시스템 콜의 실제 문법과 정확히 일치한다는 뜻이다. 그러나 GNU libc(glibc) 는 버전업이 되어 갈수록 더 느려지고 있다. 또한 메모리도 더 많이 잡아먹어 가고 있다. 그래서 직접적인 시스템 콜이 꽤나 실용적인 것이 되어가고 있다. 그렇지만... libc 를 사용하지 않음으로써 생기는 주된 부작용(?, drawback) 은 아마도 여러분이 몇몇의 libc 의 특정 함수들(시스템 콜 wrapper 가 아닌)을 직접 구현할 필요가 생길지도 모른다는 것이다. (이를테면, `printf()` 과 같은...) 그리고, 여러분이 libc 를 사용하지 않기로 결정했다면, 당연히 여러분은 그렇게 할 각오(?)가 되어 있는 것이다. 그렇지 않은가?

여기에 직접적인 시스템 콜을 하는 것의 득과 실이 나열되어 있다 :

이득:

- 가능한한 코드의 크기를 줄일 수 있다
- 가능한 최대의 속도를 낼 수 있다.
- 여러분의 프로그램이나 라이브러리를 여러분이 사용하는 특정 언어나 메모리 요구사항 등 어떤 것에도 적용할 수 있다. 즉, full control 을 얻을 수 있다.
- libc cruft 에 의한 공해가 없다(? 원문 : no pollution by libc cruft)
- C 의 함수 호출 규칙으로 인한 공해가 없다 (원문 : no pollution by C calling conventions) (만약 여러분이 여러분 자신의 언어나 혹은 환경을 개발하고 있다면)
- 정적 바이너리들은 여러분을 libc 업그레이드나 크래쉬와 무관하게 해 줄 것이다. 혹은, 쉘의 `#!` 인터프리터 패스와도 무관하게 해 줄 것이다.

  원문 : static binaries make you independent from libc upgrades or crashes, or from dangling `#!` path to an interpreter (and are faster)
- 재미있다. (원문 : just for the fun out of it (don't you get a kick out of assembly programming?)

손해:

- 만약 여러분의 컴퓨터의 다른 프로그램 중 libc 를 사용하는 프로그램이 있다면, libc 의 코드를 두개씩이나 가지는 일은 여러분의 메모리를 낭비할 뿐이다.
- 여러 정적 바이너리에서 불필요하게 구현된 서비스들 역시 메모리의 낭비일 따름이다. 그러나 여러분인 여러분의 libc 대체품을 만들어서 공유 라이브러리로 사용할 수 있다.
- 모든것을 어셈블리로 만드는 것 대신, 바이트코드나 워드코드 혹은, structure interpreter 와 같은 것을 사용함으로써 코드의 크기가 괄목할 만큼 줄어들 것이다. (인터프리터 자체는 C 로 만들든, 어셈블리로 만들든 상관 없다.) 여러개의 바이너리들을 작게 유지하는 최상의 방법은, 여러개의 바이너리 파일을 유지하는 것이 아니라, `#!` 등과 같은 쉘 인터프리터를 사용해서 인터프리터 프로세스를 하나 마련하는 것이다. 이러한 방법으로 OCaml 은 워드코드 모드 (최적화된 native code mode 와 반대말) 에서 동작한다. 또한, 그렇게 하는 것은 libc 를 사용하는 것과도 호환성이 있다. 이 방법은 Tom Christiansen 이 Perl PowerTools (http://language.perl.com/ppt/) 에서 유닉스 유틸리티들을 새로 구현한 방법이다. 마지막으로, 하드코드된 경로에 의한 외부 파일에 의존하지 않고서 코드의 크기를 줄이는 방법으로, 단지 하나의 바이너리 파일만 유지하고, 그 파일에 대한 여러가지 이름의 hard 혹은 soft 링크를 유지하는 방법이 있다 : 동일한 바이너리 파일은 최적의 공간에서 여러분이 원하는 모든것을 제공할 것이며, 그것에는 불필요한 서브루틴이나 혹은 필요없는 바이너리 헤더도 없을 것이다. 바이너리는 *`argv[0]`* 에 의해서만 행동을 결정할 것이다. 특정 이름으로 호출되지 않은 경우에는 디폴트로 쉘이 될 수도 있으며, 아마도 인터프리터로 동작하지 않을 수도 있다.

  (원문을 함께 개제합니다 : Finally, one last way to keep things small, that doesn't depend on an external file with a hardcoded path, be it library or interpreter, is to have only one binary, and have multiply-named hard or soft links to it: the same binary will provide everything you need in an optimal space, with no redundancy of subroutines or useless binary headers; it will dispatch its specific behavior according to its *`argv[0]`*; in case it isn't called with a recognized name, it might default to a shell, and be possibly thus also usable as an interpreter!

- 여러분은 리눅스 시스템 콜 이외에도 libc 가 제공하는 많은 기능들을 유용하게 사용할 수 없다 : 즉, 맨페이지 섹션 3 에 나오는 여러가지 기능들, 이를테면, malloc, threads, locale, password, high-level network management 등과 같은 것을 사용할 수 없다는 이야기이다.
- 따라서, 여러분은 libc 의 대부분을 새로 구현해야 할지도 모르다. 즉, `printf()` 로부터 `malloc()` 과 `gethostbyname` 까지도 직접 구현해야 할 것이다. 그렇게 하는 것은 불필요한 일이며, 때때로 매우 지루한 작업이 될 수도 있다. 몇몇 사람들이 벌써 "light" 버전의 libc 를 구현해 두었다는 것에 주목하고, 그러한 작업의 결과물을 체크해 보도록 하라. (레드햇의 minilibc : Rick Hohensee's libsys (ftp://linux01.gwdg.de/pub/cLIeNUX/interim/libsys.tgz), Felix von Leitner's dietlibc (http://www.fefe.de/dietlibc/), Christian Fowelin's libASM (http://www.fowelin.de/christian/computer.linux.assembly.libASM.html), asmutils (http://linuxassembly.org/asmutils.html) 이 프로젝트는 순수 어셈블리 libc 를 가지고 작업하고 있다.)
- 정적 라이브러리를 사용하면 여러분은 libc 업그레이드나 zlibc 패키지 (gzip 으로 압축된 파일들에 대해서 투명한 압축풀기를 수행해 주는) 와 같은 libc add-on 들로부터 이득을 얻

을 수 없게 됩니다.

- libc 에 의해서 추가된 몇몇 인스트럭션은 시스템 콜과 비교해서 믿기지 않을만큼 적은 속도의 오버헤드를 가진다. 만약 수행속도가 중요한 문제라면, 여러분의 주된 문제는 여러분이 시스템 콜을 사용하는데 있을 것이며, 여러분이 사용하는 wrapper 가 구현된 방식에는 문제가 없을 것이다.
- L4Linux 와 같은 micro-kernel 리눅스 버전을 사용할 때에는 시스템 콜의 표준 어셈블리 API 를 사용하는 것이 libc 의 API 를 사용하는 것 보다 훨씬 느릴 수 있다. 그러한 마이크로 커널 리눅스 버젼은 자신만의 보다 빠른 호출 규칙을 가지고 있으며, 표준 호출 규칙을 사용하는 경우 매우 높은 convention translation (호출 규칙 변환) 오버헤드가 걸릴 수 있다. (L4Linux 는 그에 맞는 syscall API 를 이용해서 재컴파일된 libc 와 함께 제공된다 : 물론, 여러분은 여러분의 코드를 제공된 API 를 이용해서 다시 컴파일 해야 할지도 모른다)
- 앞서 다루었던 일반적인 속도 최적화에 대한 토의를 참조하라.
- 만약에 시스템 콜이 여러분이 보기에 너무 느려 보인다면, 여러분은 userland 에 머물기보다는 커널 소스를 직접 해킹해서 원하는 결과를 얻을 수 있다.

만약 여러분이 방금 언급한 득실들을 심사숙고 했으며, 그래도 여전히 direct 시스템 콜을 사용하기를 원한다면, 아래에 몇가지 조언을 첨부해 주겠다 :

- 여러분은 여러분의 시스템 콜 함수들을 `asm/unistd.h` 을 include 하고, 제공된 매크로들을 사용함으로써 C 로 포터블하게(어셈블리를 사용한 포터블하지 않은 형태와 반대되는) 구현할 수 있다.
- 여러분이 그것을 대체하려고 하기 때문에, libc 소스 코드를 구해서, 그것을 해킹해 보라 (and grok them) (그리고, 만약 여러분이 더 잘 할 수 있다고 생각한다면, libc 저자들에게 피드백을 할 수 있도록 하라!)
- 여러분이 원하는 모든것을 해 주는 순수 어셈블리 코드의 예제로써, 이 문서의 뒷부분에 나오는 리소스 목록을 참조하라.Linux assembly resources.

기본적으로, 시스템 콜은 `int 0x80` 인스트럭션을 __NR_ 시스템 콜 번호 를 (`asm/unistd.h` 파일로부터.) `eax` 레지스터에 넣고, 파라미터들 (six개 까지) 은 각각 `ebx`, `ecx`, `edx`, `esi`, `edi`, `ebp` 레지스터에 넣어서 호출한다.

리턴값은 `eax` 레지스터에 담겨지게 되며, 음수의 리턴값은 에러가 있음을 나타낸다. libc 가 `errno` 변수에 집어넣는 값과 대응되는 값이다. 사용자 영역의 스택은 영향을 받지 않는다. 그래서 여러분은 시스템 콜을 행할 동안에 유효한 것을 가질 필요는 없다(?)

원문 : Result is returned in eax, negative result being an error, whose opposite is what libc would put into `errno`. The user-stack is not touched, so you needn't have a valid one when doing a syscall.

> **참고:** `ebp` 레지스터에 여섯번째 파라미터를 담아서 넘겨주는 것은 Linux 2.4 에서부터 볼 수 있다. 그 이전의 리눅스 버젼들은 레지스터에 담기는 다섯개의 파라미터만 인식한다.

Linux Kernel Internals (http://www.linuxdoc.org/LDP/lki/), 문서의 How System Calls Are Implemented on i386 Architecture?

(http://www.linuxdoc.org/LDP/lki/Linux-Kernel-Internals-2.html#ss2.11) 장은 여러분에게 보다 확실한 개념을 제공할 것이다.

프로세스를 처음 시작할 때 넘겨지는 인자들에 대해서는, 일반적인 원칙은 스택이 인자의 갯수를 저장하는 *`argc`* 의 값과, *`*argc`* 를 구성하는 포인터의 리스트와 NULL-terminated 되는 `variable=value` 의 환경변수 (*`environ`*) 값들의 리스트를 저장한다는 것이다.

원문 : As for the invocation arguments passed to a process upon startup, the general principle is that the stack originally contains the number of arguments *`argc`*, then the list of pointers that constitute *`*argv`*, then a null-terminated sequence of null-terminated `variable=value` strings for the *`environ`*ment.

보다 자세한 정보를 원한다면, 리소스 리스트를 참조해서 (: Linux assembly resources) libc 의 C 스타트업 코드의 소스를 읽어 보아라. (`crt0.S` or `crt1.S`) 혹은 리눅스 커널의 소스를 참조하라 . (`exec.c` and `binfmt_*.c` in `linux/fs/`).

5.1.4. 리눅스에서의 하드웨어 입출력

여러분이 리눅스에서 직접 I/O 포트에의 입출력을 수행하고자 한다면, OS 의 중재를 필요로 하지 않는 매우 간단한 일이어서 `IO-Port-Programming` 미니 하우투를 참조하면 될 정도이거나 아니면 커널 디바이스 드라이버를 필요로 해서 여러분이 커널 해킹이나 디바이스 드라이버 개발이나, 커널 모듈과 같은 것들에 대해서 더 많은 것을 습득해야만 하는 경우이거나 둘중 하나의 경우일 것이다. (커널 모듈이나 디바이스 드라이버에 대한 것들은 LDP 에 매우 훌륭한 하우투들이 존재한다.)

혹시 여러분이 원하는 것이 그래픽 프로그래밍인 경우, 다음의 프로젝트들을 참조하라 : GGI (http://www.ggi-project.org/) 혹은 XFree86 (http://www.XFree86.org/) 프로젝트

어떤 사람들은 작고, 견고한 XFree86 드라이버들을 interperted domain-specific language 인 GAL (http://www.irisa.fr/compose/gal/), 로 개발함으로써 보다 효과적인 결과를 얻었다.

원문 : Some people have even done better, writing small and robust XFree86 drivers in an interpreted domain-specific language, GAL (http://www.irisa.fr/compose/gal/), and achieving the efficiency of hand C-written drivers through partial evaluation (drivers not only not in asm, but not even in C!). The problem is that the partial evaluator they used to achieve efficiency is not free software. Any taker for a replacement?

어쨌거나, 이 모든 경우에도, 여러분은 `linux/asm/*.h` 에 있는 GCC 인라인 어셈블리와 매크로를 사용함으로써 전체를 어셈블리로 코딩하는 것보다 효율적으로 일을 수행할 수 있다.

5.1.5. Accessing 16-bit drivers from Linux/i386

그러한 일들은 이론적으로 가능하다 (증명 : DOSEMU (http://www.dosemu.org) 가 프로그램들에게 선택적으로 하드웨어 포트 접근 권한을 주는 방법을 보라) 그리고, 저자는 어딘가의 누군가가 실제로 그러한 일을 했다는 소문을 들었다. (PCI 드라이버에서인가? 아니면, VESA 드라이버에서인가... 음... ISA PnP 였던가... 잘 모르겠다) 만약 여러분이 보다 정확한 정보를 가지고 있다면 환영한다. (역자주 : 메일을 달라는 뜻인듯) 어쨌든, 정보를 찾기 매우 좋은 곳

은 리눅스의 커널 소스와 DOSEMU 의 소스이다. (그리고, DOSEMU repository (ftp://tsx-11.mit.edu/pub/linux/ALPHA/dosemu/) 의 다른 많은 프로그램들이다.) 또한, 여러가지의 리눅스의 low-level 프로그램들의 소스들도 도움이 될 것이다. (perhaps GGI if it supports VESA)

기본적으로 여러분은 16비트 보호모드나 혹은 vm86 모드를 이용해야만 할 것이다.

전자(16비트 보호모드) 는 셋업은 비교적 쉬우나 세그먼트 조작이나 절대 세그먼트 주소지정(segment 0 을 주소지정하는 것과같은) 이 없는 잘 동작하는 코드를 만들어야만 동작한다. 우연이라도 모든 사용하는 세그먼트가 LDT(역자주 : Local Descriptor Table, 인텔 아키텍쳐에서 세그먼트와 논리 주소, 물리 주소들을 변환하는 데 사용되는 자료구조) 를 마리 셋업해 버리지 않는 한은 그러하다

원문 : The first is simpler to setup, but only works with well-behaved code that won’t do any kind of segment arithmetics or absolute segment addressing (particularly addressing segment 0), unless by chance it happens that all segments used can be setup in advance in the LDT.

후자(vm86 모드) 는 ’vanilla 16-bit 환경’ 에서 보다 많은 호환성을 제공하기는 하지만, 더 복잡한 핸들링이 필요하다.

두 경우 모두 여러분이 16비트 코드를 사용하기 전에 반드시 해야 할 일들이 있다 :

- 16비트 코드에서 사용될 절대 주소들을 `/dev/mem` 으로부터 여러분의 프로세스의 주소 공간으로 메모리 맵 하라. (이를테면, ROM 이나, 비디오 버퍼, DMA 타겟들, 그리고, memory-mapped I/O 와 같은 것들)
- LDT 와(나) vm86 모드의 모니터를 셋업하라 (setup the LDT and/or vm86 mode monitor)
- 적절한 I/O 접근 권한을 커널로부터 획득하라(위의 섹션을 참조하라)

다시한번 DOSEMU 프로젝트의 소스들에 나오는 내용들을 주의깊이 읽도록 하여라. 특히, 리눅스/i386 에서 ELKS 와 간단한 `.COM` 프로그램들을 돌릴 수 있는 mini-emulator 들의 소스를 자세히 보도록 하라.

5.2. DOS and Windows

대부분의 도스 확장자들은 도스 서비스에 대한 인터페이스를 제공한다. 그들의 문서에서 관련된 내용들을 읽어 보도록 하라. 그러나 종종 그들은 `int 0x21` 을 시뮬레이트 하기만 하며, 여러분이 정말로 real mode 에 있는 것 "처럼" 만들어 준다. (저자는 그들이 정말로 스텁 이상의 무엇인가를 가지고 있는지 의심스럽다. 그리고, 32비트 오퍼랜드들을 가지고서 제대로 동작하는지도 의심스럽다. 대개가 그러한 프로그램들은 리얼 모드 인터럽트를 가상 86 모드(vm86 모드) 로 반영만 할 따름이다.

DPMI 와 관련된 문서들이 다음의 링크들에 존재한다 : ftp://x2ftp.oulu.fi/pub/msdos/programming/ (원래의 x2ftp 사이트가 문들 닫았기 때문에 다음의 링크를 이용하라 : mirror site (ftp://ftp.lip6.fr/pub/pc/x2ftp/README.mirror_sites)).

DJGPP 는 자체의(제한버젼임) glibc 유사품(?) 을 제공한다.

Linux 에서 DOS 용의 바이너리로 크로스 컴파일 하는 것도 가능하다. 여러분 지역의 metalab.unc.edu 의 FTP 미러의 `devel/msdos/` 디렉토리를 보라. 또한, MOSS DOS-확장자를 Utah 대학의 Flux project (http://www.cs.utah.edu/projects/flux/) 에서 구할 수 있다.

다른 문서들이나 faq 들은 보다 도스 중심이다. 우리는 도스환경의 개발을 추천하지 않는다.

Windows and Co. 이 문서는 윈도우즈 프로그래밍에 관한 문서가 아니다. 당신은 다른 곳에서 그에 관련된 많은 것들을 찾을 수 있다. 당신이 알아야 할 것은 Cygnus Solutions (http://www.cygnus.com) 가 cygwin32.dll library (http://sourceware.cygnus.com/cygwin/) 를 Win32 플랫폼에서 GNU 프로그램들을 실행시키기 위해서 개발했다는 것이다. 따라서, 당신은 gcc, gas, 그리고 그 외의 gnu 툴들을 다른 많은 유닉스 응용 프로그램처럼 윈도우에서 실행시킬 수 있다.

5.3. 여러분이 만든 OS

시스템에 대한 완전한 제어 라는 특성은 많은 OS 개발자들을 어셈블리로 유혹하는 특성이다. self-development 를 허용하는 시스템은, 비록, 그것이 다른 시스템의 '위에서' 운영될지라도 (Linux over Mach 나 OpenGenera over Unix 처럼) "OS" 라고 부를 수 있음을 기억하라.

따라서, 보다 쉬운 디버깅을 위해 여러분은 리눅스에서 실행되는 프로세스로서의 여러분의 "OS" 를 먼저 개발하고자 할지도 모른다. (느릴지는 모르지만) 그리고 나서 Flux OS kit (http://www.cs.utah.edu/projects/flux/oskit/) (Linux 와 BSD 드라이버들을 여러분 자신의 OS 에서 사용할 수 있게 해 주는 유틸리티) 를 사용해서 그것을 stand-alone 으로 만들고자 할지도 모르겠다. 여러분의 OS 가 stable 하고, 여러분이 진정으로 여러분의 창조물을 사랑한다면 그때에야 여러분의 하드웨어 드라이버들을 만들 시기가 온 것이다.

이 하우투는 부트로더의 코드나, 32-비트 모드로 진입하는 법이라든지, 인터럽트를 다루는 법이나, 인텔의 보호모드, 혹은 가상/리얼 86 모드라든지 여러분의 오브젝트 파일 포맷이나 호출 규칙에 관한 것들은 다루지 않는다.

여러분이 그러한 것들에 관한 믿을만한 정보를 찾을 수 있는 주된 장소는 현존하는 OS 들과 부트로더들의 소스코드이다. 다음의 웹페이지에 많은 링크들이 있을 것이다 : http://www.tunes.org/Review/OSes.html

6장. 일단 따라해보자(Quick Start)

6.1. 소개

마침내 때가 왔다. 여러분이 아직도 이 문서를 읽고 있으며, 여전히 어셈블리로 무언가를 만들겠다는 광기어린(!) 생각을 가지고 있다면 (여기까지 읽은 사람이라면, 정말 어셈블리 팬임에 틀림없다), 이제부터 시작하는데에 필요한 것을 알려주겠다.

여러분이 앞서 읽은 바와 같이 여러분은 리눅스 운영체제에서 여러가지 방법으로 프로그래밍을 할 수 있다. 이제부터 본 저자는 커널의 시스템 콜을 직접 호출하는 방법을 보여주려 한다. 왜! 직접 커널을 호출하는가? 그렇게 하는 것이 커널이 제공하는 서비스를 호출하는 가장 빠른(!) 방법이기 대문이다. 이 방법을 사용하면, 우리의 코드는 다른 라이브러리에 링크될 필요가 없다. ELF 인터프리터를 사용할 필요도 없다.단지, 커널과 직접 의사소통할 따름이다.

저자는 동일한 일을 하는 샘플 프로그램을 **nasm** 과 **gas**, 이 두개의 어셈블러를 사용해서 각각 프로그래밍 한 소스를 보여주겠다. 자세히 보면, 인텔계열과 AT&T 유닉스 계열의 문법의 차이를 알.. 수도 있을까?

여러분은 또한, 유닉스 어셈블리 프로그래밍 튜토리얼(UNIX assembly programming tutorial - http://linuxassembly.org/intro.html) (http://linuxassembly.org/intro.html)의 소개를 읽어볼 수 있다. 거기에는 리눅스 이외에도, 다른 유닉스 계열의 운영체제를 위한 샘플 코드들이 있다.(역자주 : FreeBSD, BeOS 등)

6.1.1. 필요한 도구들

당연히 어셈블러가 필요하다. -- **nasm** 이나 **gas**.

그리고, 링커가 필요하다. -- **ld** 왜냐하면, 어셈블러는 단지 목적코드(object code)만을 만들어내기 때문이다. 대부분의 배포본에는 gas 와 ld 가 포함되어 있다. binutils 패키지에 패키징되어 있다.

nasm 의 경우, 리눅스용의 문서와 바이너리 패키지를 다운로드해서 설치해야 한다. 다운로드 할 수 있는 사이트는 다음과 같다 : nasm site; 맨드레이크, 수세, Stampede, 데비안 등의 패키지에는 포함되어 있다. 먼저, 확인하라.

정말 깊이 파고들려면, 여러분의 운영체제의 include 파일들을 설치해야 하고, 가능하다면, 커널 소스까지 설치하라 (여러분의 운영체제가 공개 운영체제이면 가능하다.)

6.2. Hello, world!

6.2.1. Program layout

리눅스는 32비트 프로텍티드 모드에서 동작하며, 선형 메모리 모델(flat memory model)을 가진다. 그리고, 이진 실행파일에 ELF 포맷을 가진다.

프로그램은 여러분이 작성한 코드가 있는 `.text` 섹션과 (이 부분은 읽기 전용이다) 여러분의 초기화된(!) data 가 저장되어 있는 `.data` 섹션, 초기화되지 않은(!) 데이타가 저장되는 `.bss` 섹션 및 다른 표준적인 섹션들로 나뉜다. 크게 보면, .text 와 .data 의 두개의 섹션으로 나뉜다. (즉, 코드와 데이타) 그러나, 다른 섹션 없이 .text 섹션만으로도 프로그램이 가능하다. .text 섹션은 프로그램에 반드시 있어야 하는 섹션이다. (역자주 : 상상해보라! 코드 없는 프로그램 -- 말도 안된다.)

이제, 첫번째 프로그램을 작성해 보자 (역자주 : 역시나.. 헬로월드 프로그램이다 _._)

6.2.2. NASM (hello.asm)

```
section     .data                               ;섹션 .data 를 선언한다.(여기부터
                                                ; .data 섹션이다.)

msg     db          "Hello, world!",0xa    ;출력할 사랑스런(!) 스트링이다.
len     equ         $ - msg                     ;친애하는(!) 문자열의 길이

section     .text                               ;여기부터 .text 섹션이다. 코드있음

                    ;ELF 링커나 로더에게 프로그램의 엔트리 포인트를 알려주어야
    global _start   ;한다. 로더 혹은 링커는 일반적으로 엔트리 포인트가 _start
                    ;라고 가정하고 있다. 이 디폴트 설정을 바꿀려면,
                    ;ld -e foo 를 사용하라.

_start:

;stdout 에 우리의 사랑스런 문자열을 출력하자.

        mov     edx,len ;시스템 콜 4번(sys_write)은 세번째 인수로 출력할 메세지의
                        ;길이를 취한다.
        mov     ecx,msg ;시스템 콜 4번은 두번째 인수로 출력할 메세지가 담긴 메모리
                        ;주소를 취한다. 즉, 출력할 메세지로의 포인터를 취한다.
        mov     ebx,1   ;시스템 콜 4번은 첫번째 인수로 파일의 핸들(디스크립터)을
                        ;취한다.
        mov     eax,4   ;eax 레지스터에 호출할 커널 시스템 콜의 번호를 넣어 준다.
                        ;이 경우에는 4번(sys_write) 이다.
        int     0x80    ;커널을 호출한다.

;출력이 끝났으면, exit 를 호출한다.

        mov     ebx,0   ;exit 코드로 0을 준다. (c 코드로는 exit(0);)
        mov     eax,1   ;시스템 콜 1번 (sys_exit)
        int     0x80 ;커널을 호출한다.
```

6.2.3. GAS (hello.S)

```
.data                           # 섹션 .data 가 여기서부터 시작한다.

msg:
    .string "Hello, world!\n"   # 우리의 사랑스런 문자열
    len = . - msg               # 친애하는 문자열의 길이

.text                           # 섹션 .text 가 여기서부터 시작한다.

                    # ELF 링커나 로더에게 프로그램의 엔트리 포인트를 알려주어야
    .global _start  # 한다. 로더 혹은 링커는 일반적으로 엔트리 포인트가 _start
                    # 라고 가정하고 있다. 이 디폴트 설정을 바꿀려면,
                    # ld -e foo 를 사용하라.

_start:

# stdout 에 우리의 사랑스런 문자열을 출력하자.

    movl    $len,%edx   # 시스템 콜 4번(sys_write)은 세번째 인수로 출력할
                        # 메세지의 길이를 취한다.
    movl    $msg,%ecx   # 시스템 콜 4번은 두번째 인수로 출력할 메세지가
                        # 담긴 메모리 주소를 취한다. 즉, 출력할 메세지로의
                        # 포인터를 취한다.
    movl    $1,%ebx     # 시스템 콜 4번은 첫번째 인수로 파일의
                        # 핸들(디스크립터)을 취한다.
    movl    $4,%eax     # eax 레지스터에 호출할 커널 시스템 콜의 번호를
                        # 넣어 준다. 이 경우에는 4번(sys_write) 이다.
    int     $0x80       # 커널을 호출한다.

# 출력이 끝났으면, exit 를 호출한다.

    movl    $0,%ebx     # exit 코드로 0을 준다. (c 코드로는 exit(0);)
    movl    $1,%eax     # 시스템 콜 1번 (sys_exit)
    int     $0x80       # 커널을 호출한다.
```

6.3. 실행파일 만들기

6.3.1. 목적 코드 생성

실행파일을 만들기 위한 첫번째 단계는 소스파일을 컴파일 혹은 어셈블링 해서 목적 코드가 담긴 파일을 만드는 것이다.

nasm 을 사용한다면,

```
$ nasm -f elf hello.asm
```

gas 를 사용한다면,

```
$ as -o hello.o hello.S
```

와 같이 하면 목적 코드가 담겨 있는 `hello.o` 를 생성하게 된다.

6.3.2. 실행파일의 생성

두번째 단계로써, 목적코드가 담긴 파일로부터 링커를 호출하여 실행가능한 파일을 만들어 내게 된다.

```
$ ld -s -o hello hello.o
```

위와 같이 하면 된다. 그렇게 하고 나면, 마침내, `hello` 라는 실행파일을 볼 수 있게 된다.

한번 실행시켜 보아라. 제대로 동작하는가? 끝이다. 이게 전부이다. 매우 간단하지 않은가? 너무나 간단해서 되려 황당하지 않은가?!?!

> 참고**:** (역자 주: 그렇다. 이것이 어셈블리 프로그래밍의 전부이다. 더 깊이 프로그래밍하기 위해서는 커널의 시스템 콜의 번호를 전부 알아야 하고, 인텔 혹은 여러분이 프로그래밍 하기 원하는 플랫폼의 마이크로프로세서의 인스트럭션 세트 및, 특성, 제공하는 기능 등을 잘 알아야 한다. 그러한 것은 해당 마이크로프로세서의 제조업체 홈페이지 등에 가면 자세한 자료를 구할 수 있다. 리눅스 커널의 시스템 콜의 목록과 인수로 레지스터에 무엇을 넣어주어야 하는가에 관한 자료는 http://linuxassembly.org 의 문서를 참조하라.

7장. Resources

7.1. Pointers

Your main resource for Linux/UNIX assembly programming material is:

http://linuxassembly.org/resources.html

Do visit it, and get plenty of pointers to assembly projects, tools, tutorials, documentation, guides, etc, concerning different UNIX operating systems and CPUs. Because it evolves quickly, I will no longer duplicate it here.

If you are new to assembly in general, here are few starting pointers:

- The Art Of Assembly (http://webster.cs.ucr.edu/Page_asm/ArtOfAsm.html)
- x86 assembly FAQ (http://www2.dgsys.com/~raymoon/faq/)
- ftp.luth.se (ftp://ftp.luth.se/pub/msdos/) mirrors the hornet and x2ftp former archives of msdos assembly coding stuff
- CoreWars (http://www.koth.org), a fun way to learn assembly in general
- Usenet: comp.lang.asm.x86 (news://comp.lang.asm.x86); alt.lang.asm (news://alt.lang.asm)

7.2. Mailing list

If you're are interested in Linux/UNIX assembly programming (or have questions, or are just curious) I especially invite you to join Linux assembly programming mailing list.

This is an open discussion of assembly programming under Linux, *BSD, BeOS, or any other UNIX/POSIX like OS; also it is not limited to x86 assembly (Alpha, Sparc, PPC and other hackers are welcome too!).

To subscribe send a blank message to `<linux-assembly-subscribe@egroups.com>`.

List address is `<linux-assembly@egroups.com>`.

List archives are available at http://www.egroups.com/list/linux-assembly/.

8장. Frequently Asked Questions

Here are frequently asked questions (with answers) about Linux assembly programming. Some of the questions (and the answers) were taken from the the linux-assembly mailing list.

1. How do I do graphics programming in Linux?

An answer from Paul Furber (mailto:paulf@icom.co.za):

```
Ok you have a number of options to graphics in Linux. Which one you use
depends on what you want to do. There isn't one Web site with all the
information but here are some tips:

SVGALib: This is a C library for console SVGA access.
Pros: very easy to learn, good coding examples, not all that different
from equivalent gfx libraries for DOS, all the effects you know from DOS
can be converted with little difficulty.
Cons: programs need superuser rights to run since they write directly to
the hardware, doesn't work with all chipsets, can't run under X-Windows.
Search for svgalib-1.4.x on http://ftp.is.co.za

Framebuffer: do it yourself graphics at SVGA res
Pros: fast, linear mapped video access, ASM can be used if you want :)
Cons: has to be compiled into the kernel, chipset-specific issues, must
switch out of X to run, relies on good knowledge of linux system calls
and kernel, tough to debug
Examples: asmutils (http://www.linuxassembly.org) and the leaves example
and my own site for some framebuffer code and tips in asm
(http://ma.verick.co.za/linux4k/)

Xlib: the application and development libraries for XFree86.
Pros: Complete control over your X application
Cons: Difficult to learn, horrible to work with and requires quite a bit
of knowledge as to how X works at the low level.
Not recommended but if you're really masochistic go for it. All the
include and lib files are probably installed already so you have what
you need.

Low-level APIs: include PTC, SDL, GGI and Clanlib
Pros: very flexible, run under X or the console, generally abstract away
the video hardware a little so you can draw to a linear surface, lots of
good coding examples, can link to other APIs like OpenGL and sound libs,
Windows DirectX versions for free
Cons: Not as fast as doing it yourself, often in development so versions
can (and do) change frequently.
Examples: PTC and GGI have excellent demos, SDL is used in sdlQuake,
Myth II, Civ CTP and Clanlib has been used for games as well.
```

```
High-level APIs: OpenGL - any others?
Pros: clean api, tons of functionality and examples, industry standard
so you can learn from SGI demos for example
Cons: hardware acceleration is normally a must, some quirks between
versions and platforms
Examples: loads - check out www.mesa3d.org under the links section.

To get going try looking at the svgalib examples and also install SDL
and get it working. After that, the sky's the limit.
```

2. How do I debug pure assembly code under Linux?

There's an early version of the Assembly Language Debugger (http://www.ellipse.magenet.com/ald.html), which is designed to work with assembly code, and is portable enough to run on Linux and *BSD. It is already functional and should be the right choice, check it out!

You can also try **gdb** ;). Although it is source-level debugger, it can be used to debug pure assembly code, and with some trickery you can make **gdb** to do what you need. Here's an answer from Dmitry Bakhvalov (mailto:dl@gazeta.ru):

```
Personally, I use gdb for debugging asmutils. Try this:

1) Use the following stuff to compile:
   $ nasm -f elf -g smth.asm
   $ ld -o smth smth.o

2) Fire up gdb:
   $ gdb smth

3) In gdb:
   (gdb) disassemble _start
   Place a breakpoint at _start+1 (If placed at _start the breakpoint
   wouldnt work, dunno why)
   (gdb) b *0x8048075

   To step thru the code I use the following macro:
   (gdb)define n
   >ni
   >printf "eax=%x ebx=%x ...etc...",$eax,$ebx,...etc...
   >disassemble $pc $pc+15
   >end

   Then start the program with r command and debug with n.

   Hope this helps.
```

An additional note from ???:

```
I have such a macro in my .gdbinit for quite some time now, and it
for sure makes life easier. A small difference : I use "x /8i $pc",
which guarantee a fixed number of disassembled instructions. Then,
with a well chosen size for my xterm, gdb output looks like it is
refreshed, and not scrolling.
```

If you want to set breakpoints across your code, you can just use `int 3` instruction as breakpoint (instead of entering address manually in **gdb**).

If you're using gas, you should consult gas and gdb related tutorials (http://linuxassembly.org/resources.html#tutorials).

3. Any other useful debugging tools?

Definitely **strace** can help a lot (**ktrace** and **kdump** on FreeBSD), it is used to trace system calls and signals. Read its manual page (**man strace**) and **strace --help** output for details.

4. How do I access BIOS functions from Linux (BSD, BeOS, etc)?

Noway. This is protected mode, use OS services instead. Again, you can't use `int 0x10`, `int 0x13`, etc. Fortunately almost everything can be implemented by means of system calls or library functions. In the worst case you may go through direct port access, or make a kernel patch to implement needed functionality.

5. Is it possible to write kernel modules in assembly?

Yes, indeed it is. While in general it is not a good idea (it hardly will speedup anything), there may be a need of such wizardy. The process of writing a module itself is not that hard -- a module must have some predefined global function, it may also need to call some external functions from the kernel. Examine kernel source code (that can be built as module) for details.

Meanwhile, here's an example of a minimum dumb kernel module (`module.asm`) (source is based on example by mammon_ from APJ #8):

```
section .text

global init_module
global cleanup_module
global kernel_version

extern printk

init_module:
push dword str1
call printk
pop eax
xor eax,eax
```

```
ret

cleanup_module:
push dword str2
call printk
pop eax
ret

str1 db "init_module done",0xa,0
str2 db "cleanup_module done",0xa,0

kernel_version db "2.2.18",0
```

The only thing this example does is reporting its actions. Modify `kernel_version` to match yours, and build module with:

```
$ nasm -f elf -o module.m module.asm
```

```
$ ld -r -o module.o module.m
```

Now you can play with it using **insmod/rmmod/lsmod** (root privilidged are required); a lot of fun, huh?

That’s all for now, folks.

부록 A. History

Each version includes a few fixes and minor corrections, that need not to be repeatedly mentioned every time.

고친 과정

고침 0.6 11 Nov 2000 고친이 konst
HOWTO is completely rewritten using DocBook DTD. Layout is totally rearranged; too much changes to list them here.
고침 0.5n 07 Nov 2000 고친이 konst
Added question regarding kernel modules to FAQ, fixed NASM URLs, GAS has Intel syntax too
고침 0.5m 22 Oct 2000 고친이 konst
Linux 2.4 system calls can have 6 args, Added ALD note to FAQ, fixed mailing list subscribe address
고침 0.5l 23 Aug 2000 고친이 konst
Added TDASM, updates on NASM
고침 0.5k 11 Jul 2000 고친이 konst
Few additions to FAQ
고침 0.5j 14 Jun 2000 고친이 konst
Complete rearrangement of Introduction and Resources sections. FAQ added to Resources, misc cleanups and additions.
고침 0.5i 04 May 2000 고친이 konst
Added HLA, TALC; rearrangements in Resources, Quick Start Assemblers sections. Few new pointers.
고침 0.5h 09 Apr 2000 고친이 konst
finally managed to state LDP license on document, new resources added, misc fixes
고침 0.5g 26 Mar 2000 고친이 konst
new resources on different CPUs
고침 0.5f 02 Mar 2000 고친이 konst
new resources, misc corrections
고침 0.5e 10 Feb 2000 고친이 konst
URL updates, changes in GAS example
고침 0.5d 01 Feb 2000 고친이 konst
Resources (former "Pointers") section completely redone, various URL updates.
고침 0.5c 05 Dec 1999 고친이 konst
New pointers, updates and some rearrangements. Rewrite of sgml source.
고침 0.5b 19 Sep 1999 고친이 konst
Discussion about libc or not libc continues. New web pointers and and overall updates.
고침 0.5a 01 Aug 1999 고친이 konst
Quick Start section rearranged, added GAS example. Several new web pointers.
고침 0.5 01 Aug 1999 고친이 konstfare
GAS has 16-bit mode. New maintainer (at last): Konstantin Boldyshev. Discussion about libc or not libc. Added Quick Start
고침 0.4q 22 Jun 1999 고친이 fare
process argument passing (argc, argv, environ) in assembly. This is yet another "last release by Fare before new maintainer t
고침 0.4p 06 Jun 1999 고친이 fare
clean up and updates
고침 0.4o 01 Dec 1998 고친이 fare

고침 0.4m 23 Mar 1998 고친이 fare
corrections about gcc invocation
고침 0.4l 16 Nov 1997 고친이 fare
release for LSL 6th edition
고침 0.4k 19 Oct 1997 고친이 fare

고침 0.4j 07 Sep 1997 고친이 fare

고침 0.4i 17 Jul 1997 고친이 fare
info on 16-bit mode access from Linux
고침 0.4h 19 Jun 1997 고친이 fare
still more on "how not to use assembly"; updates on NASM, GAS.
고침 0.4g 30 Mar 1997 고친이 fare

고침 0.4f 20 Mar 1997 고친이 fare

고침 0.4e 13 Mar 1997 고친이 fare
Release for DrLinux
고침 0.4d 28 Feb 1997 고친이 fare
Vapor announce of a new Assembly-HOWTO maintainer
고침 0.4c 09 Feb 1997 고친이 fare
Added section Do you need assembly?.
고침 0.4b 03 Feb 1997 고친이 fare
NASM moved: now is before AS86
고침 0.4a 20 Jan 1997 고친이 fare
CREDITS section added
고침 0.4 20 Jan 1997 고친이 fare
first release of the HOWTO as such
고침 0.4pre1 13 Jan 1997 고친이 fare
text mini-HOWTO transformed into a full linuxdoc-sgml HOWTO, to see what the SGML tools are like
고침 0.3l 11 Jan 1997 고친이 fare

고침 0.3k 19 Dec 1996 고친이 fare
What? I had forgotten to point to terse???
고침 0.3j 24 Nov 1996 고친이 fare
point to French translated version
고침 0.3i 16 Nov 1996 고친이 fare
NASM is getting pretty slick
고침 0.3h 06 Nov 1996 고친이 fare
more about cross-compiling -- See on sunsite: devel/msdos/
고침 0.3g 02 Nov 1996 고친이 fare
Created the History. Added pointers in cross-compiling section. Added section about I/O programming under Linux (particu
고침 0.3f 17 Oct 1996 고친이 fare

고침 0.3c 15 Jun 1996 고친이 fare

고침 0.2 04 May 1996고친이 fare

고침 0.1 23 Apr 1996 고친이 fare
Francois-Rene "Fare" Rideau creates and publishes the first mini-HOWTO, because "I'm sick of answering ever the same qu

부록 B. Acknowledgements

I would like to thank all the people who have contributed ideas, answers, remarks, and moral support, and additionally the following persons, by order of appearance:

- Linus Torvalds (mailto:buried.alive@in.mail) for Linux
- Bruce Evans (mailto:bde@zeta.org.au) for bcc from which as86 is extracted
- Simon Tatham (mailto:anakin@pobox.com) and Julian Hall (mailto:jules@earthcorp.com) for NASM
- Greg Hankins (mailto:gregh@metalab.unc.edu) and now Tim Bynum (mailto:linux-howto@metalab.unc.edu) for maintaining HOWTOs
- Raymond Moon (mailto:raymoon@moonware.dgsys.com) for his FAQ
- Eric Dumas (mailto:dumas@linux.eu.org) for his translation of the mini-HOWTO into French (sad thing for the original author to be French and write in English)
- Paul Anderson (mailto:paul@geeky1.ebtech.net) and Rahim Azizarab (mailto:rahim@megsinet.net) for helping me, if not for taking over the HOWTO
- Marc Lehman (mailto:pcg@goof.com) for his insight on GCC invocation
- Abhijit Menon-Sen (mailto:ams@wiw.org) for helping me figure out the argument passing convention

부록 C. Endorsements

This version of the document is endorsed by Konstantin Boldyshev.

Modifications (including translations) must remove this appendix according to the license agreement included below.

`$Id: Assembly-HOWTO.sgml,v 1.3 2001/10/27 02:18:17 kss Exp $`

부록 D. GNU Free Documentation License

GNU Free Documentation License
Version 1.1, March 2000

0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other written document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you".

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (For example, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, whose contents can be viewed and edited directly and straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup has been designed to thwart or discourage subsequent modification by readers is not Transparent. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML designed for human modification. Opaque formats include PostScript, PDF, proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies of the Document numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a publicly-accessible computer-network location containing a complete Transparent copy of the Document, free of added material, which the general network-using public has access to download anonymously at no charge using public-standard network protocols. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has less than five).

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License.

I. Preserve the section entitled "History", and its title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. In any section entitled "Acknowledgements" or "Dedications", preserve the section's title, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section entitled "Endorsements". Such a section may not be included in the Modified Version.

N. Do not retitle any existing section as "Endorsements" or to conflict in title with any Invariant Section.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties--for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections entitled "History" in the various original documents, forming one section entitled "History"; likewise combine any sections entitled "Acknowledgements", and any sections entitled "Dedications". You must delete all sections entitled "Endorsements."

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, does not as a whole count as a Modified Version of the Document, provided no compilation copyright is claimed for the compilation. Such a compilation is called an "aggregate", and this License does not apply to the other self-contained works thus compiled with the Document, on account of their being thus compiled, if they are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one quarter of the entire aggregate, the Document’s Cover Texts may be placed on covers that surround only the Document within the aggregate. Otherwise they must appear on covers around the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License provided that you also include the original English version of this License. In case of a disagreement between the translation and the original English version of this License, the original English version will prevail.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

> Copyright (c) YEAR YOUR NAME.
> Permission is granted to copy, distribute and/or modify this document
> under the terms of the GNU Free Documentation License, Version 1.1
> or any later version published by the Free Software Foundation;
> with the Invariant Sections being LIST THEIR TITLES, with the
> Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST.
> A copy of the license is included in the section entitled "GNU
> Free Documentation License".

If you have no Invariant Sections, write "with no Invariant Sections" instead of saying which ones are invariant. If you have no Front-Cover Texts, write "no Front-Cover Texts" instead of "Front-Cover Texts being LIST"; likewise for Back-Cover Texts.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.