Hanbit eBook

Realtime 63

Thinking About C/C++

프로그램 생성편

프로그래머가 몰랐던 프로그램의 동작 원리

박수현 지음

한빛미디어 Hanbit Media, Inc.

Thinking About

C/C++

프로그래머가 몰랐던 프로그램의 동작 원리

프로그램
생성편

Thinking about C/C++: 프로그래머가 몰랐던 프로그램의 동작 원리 프로그램 생성편

초판발행 2014년 4월 24일

지은이 박수현 / **펴낸이** 김태헌
펴낸곳 한빛미디어(주) / **주소** 서울시 마포구 양화로 7길 83 한빛미디어(주) IT출판부
전화 02-325-5544 / **팩스** 02-336-7124
등록 1999년 6월 24일 제10-1779호
ISBN 978-89-6848-695-1 15000 / **정가** 9,900원

책임편집 배용석 / **기획·편집** 정지연
디자인 표지 여동일, 내지 스튜디오 [밈], 조판 최송실
영업 김형진, 김진불, 조유미 / **마케팅** 박상용, 서은옥, 김옥현

이 책에 대한 의견이나 오탈자 및 잘못된 내용에 대한 수정 정보는 한빛미디어(주)의 홈페이지나 아래 이메일로 알려주십시오.
한빛미디어 홈페이지 www.hanbit.co.kr / **이메일** ask@hanbit.co.kr

Published by HANBIT Media, Inc. Printed in Korea

지금 하지 않으면 할 수 없는 일이 있습니다.
책으로 펴내고 싶은 아이디어나 원고를 메일(ebookwriter@hanbit.co.kr)로 보내주세요.
한빛미디어(주)는 여러분의 소중한 경험과 지식을 기다리고 있습니다.

저자 소개

지은이_박수현

홍익대학교에서 컴퓨터공학 학사부터 박사까지 마쳤으며, 현재 현대오토에버에 재직 중이다. 약 12년간에 걸친 홍대 생활로 잘 놀 것 같다는 오해를 자주 받고 있다. 사실 홍대 앞 번화가에 대해서는 잘 모르지만, 홍대 근처 어느 집에서 자장면을 시켜야 맛있는지는 조언해 줄 수 있다. 운영체제, 시스템 프로그래밍에 관심이 많다.

저자 서문

중고등학교 시절 나를 끊임없이 괴롭히던 한 가지 생각은 “이 많은 것들을 배워서 도대체 어디에 써먹는 것일까”였다. 물리학자도, 수학자도 될 생각이 없었기에 미적분은 그저 성적 유지를 위해서 공부하는 것에 불과했다. 대학교에 진학하고 컴퓨터 공학을 전공하면서도 이 생각은 그대로였다. 자료구조와 알고리즘 수업을 듣는다고 해서 멋있는 게임을 만들 수도 없었고, 컴퓨터 구조나 운영체제 수업을 듣는다고 임베디드 장치를 만들 수 없었기에 그 허탈감은 이루 말할 수 없었다.

대학원 진학을 고민하던 시절, 술자리에서 한 선배가 필자에게 질문을 던졌다. “넌 명색이 컴퓨터 공학과 졸업자인데 컴퓨터가 어떻게 부팅되는지 정확하게 설명할 수 있느냐”라고 말이다. 망치로 머리를 얻어맞은 느낌이었다. 비록 모든 내용을 다 배우지는 못했지만, 4년 동안 컴퓨터 공학 전공 수업을 들으면서 한 부분씩은 배웠을 텐데 이들이 어떻게 연결되고 어떻게 동작하는지 이해하려는 시도조차 하지 않았다. 수동적으로 배우기만 하고 내가 만든 프로그램이 어떻게 동작하는지, 무엇이 문제인지 고민조차 해 보지 않은 것이다.

대학원에 진학하고 6년이 넘도록 컴퓨터 시스템에 관련된 공부를 하면서 가장 많이 받은 질문은 “왜 인기도 없는 시스템을 공부하느냐”였다. 답은 간단했다. 어떤 언어를 쓰고 어떤 프로그램을 만들고 어떤 운영체제를 사용하든지 그 근간은 시스템이라고 생각하기 때문이다. 아무리 능숙한 프로그래머라 할지라도 언어의 기본적인 개념을 이해하지 못하거나 프로그램이 동작하는 근본 원리나 플랫폼의 세부사항을 알지 못한다면 이는 능숙한 ‘코드 작성자’밖에 안 된다. 실무를 통해 겪었든 수업이나 책을 통해서 공부했든 자신이 몸담은 분야에 대한 기반 지식이 없다면 살아남기 힘들다고 생각한다. 기반 지식보다 더 문제인 것은 이미 배운 내용을 등한

시하고 사용하지 않는다는 점이다. 아마도 이는 필자가 느낀 것과 마찬가지로 이들을 어디에 써먹어야 할지, 왜 필요한지를 이해하지 못하기 때문이라고 생각한다.

이 책은 그래서 쓰기 시작했다. 컴퓨터공학을 전공하지 않았거나 잘 모르는 사람이 보기에는 조금 어려울 수도 있고, 실무 경험이 많거나 컴퓨터에 대한 지식이 많은 사람이 보기에는 너무 쉬울지도 모른다. 책을 쓰는 데 많은 도움을 주신 분의 평가처럼 대상 독자층이 참으로 '애매모호'하다.

하지만 나는 이 책을 통해서 여러분이 배웠을지도 모르는 그 지식이 어떻게 사용될 수 있으며 왜 중요한지 꼭 이야기하고 싶었다. 한 줄의 코드 라인에 캐시의 철학을 담을 수 있는 사람이 진정한 프로그래머라고 생각한다. 그리고 이 책을 읽는 많은 분은 캐시가 무엇인지 그리고 어떻게 프로그램을 만드는 것인지 이미 다 알고 있다고 생각한다. 다만 왜 캐시가 중요하고, 캐시가 어떻게 프로그램에 영향을 미치는지 그리고 작성하는 코드가 캐시의 동작에 어떤 의미를 가지는지를 모를 뿐이라고 생각한다. 비단 캐시뿐만 아니라 이 책에서 설명하는 모든 부분은 익히 아는 내용일 것이다. 다만 이들이 여러분의 프로그램과 어떤 연관이 있는지 모를 뿐이다.

사실 이 책에서 이야기하는 내용은 몰라도 회사에서 일하고 프로그램을 만들고 돈을 버는 데에는 아무 지장이 없다. 다만 이 책을 다 읽고 덮을 때 즈음 여러분이 프로그램의 소스 코드뿐 아니라 그 외에 많은 부분을 둘러볼 수 있는 좀 더 넓은 시야를 갖게 되길 바란다.

마지막으로 이 책이 나오기까지 많은 도움을 주신 모든 분께 감사의 말을 전한다.

대상 독자 및 참고사항

초급 | 초중급 | **중급** | 중고급 | 고급

이 책은 C/C++ 등 프로그래밍 언어에 어느 정도 익숙하며 프로그램을 만들고 실행해 본 경험이 있는 중급 이상의 프로그래머를 대상으로 한다.

이 책에서는 주로 POSIX를 따르는 Unix 계열의 운영체제와 C/C++ 언어를 주로 다루고 있다. 이 책에 수록된 예제는 대부분 이해를 도우려고 작성된 슈도 코드 또는 스켈레톤 코드 형태이며 상당수가 불완전하다. 각 코드의 실행 환경은 대부분 설명 부분에 기재하였으나, 없는 경우 리눅스와 GCC를 사용하였다고 보아도 무방하다.

한빛 eBook 리얼타임

한빛 eBook 리얼타임은 IT 개발자를 위한 eBook입니다.

요즘 IT 업계에는 하루가 멀다 하고 수많은 기술이 나타나고 사라져 갑니다. 인터넷을 아무리 뒤져도 조금이나마 정리된 정보를 찾는 것도 쉽지 않습니다. 또한 잘 정리되어 책으로 나오기까지는 오랜 시간이 걸립니다. 어떻게 하면 조금이라도 더 유용한 정보를 빠르게 얻을 수 있을까요? 어떻게 하면 남보다 조금 더 빨리 경험하고 습득한 지식을 공유하고 발전시켜 나갈 수 있을까요? 세상에는 수많은 종이책이 있습니다. 그리고 그 종이책을 그대로 옮긴 전자책도 많습니다. 전자책에는 전자책에 적합한 콘텐츠와 전자책의 특성을 살린 형식이 있다고 생각합니다.

한빛이 지금 생각하고 추구하는, 개발자를 위한 리얼타임 전자책은 이렇습니다.

1. eBook Only - 빠르게 변화하는 IT 기술에 대해 핵심적인 정보를 신속하게 제공합니다.

   500페이지 가까운 분량의 잘 정리된 도서(종이책)가 아니라, 핵심적인 내용을 빠르게 전달하기 위해 조금은 거칠지만 100페이지 내외의 전자책 전용으로 개발한 서비스입니다. 독자에게는 새로운 정보를 빨리 얻을 수 있는 기회가 되고, 자신이 먼저 경험한 지식과 정보를 책으로 펴내고 싶지만 너무 바빠서 엄두를 못 내는 선배, 전문가, 고수 분에게는 보다 쉽게 집필할 수 있는 기회가 될 수 있으리라 생각합니다. 또한 새로운 정보와 지식을 빠르게 전달하기 위해 O'Reilly의 전자책 번역 서비스도 하고 있습니다.

2. 무료로 업데이트되는 전자책 전용 서비스입니다.

   종이책으로는 기술의 변화 속도를 따라잡기가 쉽지 않습니다. 책이 일정 분량 이상으로 집필되고 정리되어 나오는 동안 기술은 이미 변해 있습니다. 전자책으로 출간된 이후에도 버전 업을 통해 중요한 기술적 변화가 있거나 저자(역자)와 독자가 소통하면서 보완하여 발전된 노하우가 정리되면 구매하신 분께 무료로 업데이트해 드립니다.

3. 독자의 편의를 위해 DRM-Free로 제공합니다.

구매한 전자책을 다양한 IT 기기에서 자유롭게 활용할 수 있도록 DRM-Free PDF 포맷으로 제공합니다. 이는 독자 여러분과 한빛이 생각하고 추구하는 전자책을 만들어 나가기 위해 독자 여러분이 언제 어디서 어떤 기기를 사용하더라도 편리하게 전자책을 볼 수 있도록 하기 위함입니다.

4. 전자책 환경을 고려한 최적의 형태와 디자인에 담고자 노력했습니다.

종이책을 그대로 옮겨 놓아 가독성이 떨어지고 읽기 힘든 전자책이 아니라, 전자책의 환경에 가능한 한 최적화하여 쾌적한 경험을 드리고자 합니다. 링크 등의 기능을 적극적으로 이용할 수 있음은 물론이고 글자 크기나 행간, 여백 등을 전자책에 가장 최적화된 형태로 새롭게 디자인하였습니다.

앞으로도 독자 여러분의 충고에 귀 기울이며 지속해서 발전시켜 나가도록 하겠습니다.

차례

1 | 들어가기

『Thinking about: C/C++ 〈프로그램 개발편〉(한빛미디어, 2014년 월)』[01]에서는 프로그램(또는 소프트웨어)을 만들기 위해서 어떤 준비를 해야 하는지 살펴보았다. 어떤 개발 방법을 사용하고 어떤 프로그래밍 언어가 적합한지, 프로그램 설계 과정에서 무엇을 고민해야 하는지에 대해서 다각적으로 검토하고 생각해 보아야 한다. 물론 애자일 개발 방법과 같이 즉흥적으로 떠오르는 대로 코드를 작성하고 개발을 진행할 수도 있겠지만, 대부분은 약간의 계획과 조율이 필요할 것이다.

이제 맞닥트리는 것은 코드가 제대로 실행되는 프로그램이 되느냐의 문제다. 소스 코드가 프로그램이 되는 과정에서 가장 큰 비중을 차지하는 것은 뭐니뭐니해도 (그리고 익히 알다시피) 툴 체인이라 흔히 불리는 컴파일러와 링커, 로더 등의 프로그램이다. 이들은 개발자가 만든 소스 코드를 입력으로 받아서 문법을 분석하고, 오류가 있는지 찾으며 적당한 언어(예컨대 어셈블리어) 등으로 변환한 다음, 마지막으로 프로그램이 실행하는 데 필요한 라이브러리나 기타 객체 파일을 연결해준다. 비주얼 스튜디오나 GCC 등 코드를 만들고 무의식 중에 실행하는 프로그램들이 바로 이렇게 소스 코드를 프로그램으로 만들어준다.

개발하면서 지금까지 이런 툴 체인들이 어떻게 돌아가는지 사실 크게 궁금한 적은 없었을 것이다. 프로그램에 문제가 있으면 대부분은 소스 코드가 잘못된 것이며, 어디서 무슨 문제가 발생하길래 에러 메시지가 출력되는지는 모르지만, 아무튼 항상 소스 코드를 들여다보기에 바빴을 것이다. 많은 경우 소스 코드를 수정하는 것만으로도 프로그램이 생성되지 않는 문제는 해결되었을 것이며, 당연히 툴 체인의 동작 과정에는 관심이 없었을 것이다. 컴파일러 등은 그 존재만으로도 완전무결하

01 http://www.hanbit.co.kr/ebook/look.html?isbn=9788968486937

며, 감히 이런 프로그램들에 오류가 있으리라고는 꿈에도 상상하지 않았을 것이다.

반은 맞고, 반은 틀렸다. 컴파일러에도 버그는 종종 있었으며, 프로그래머가 모르는 사이에 문제가 해결되었을 뿐이다. 버그가 자주 발생하지 않을 뿐이지 컴파일러나 링커, 로더 역시 사람이 만든 프로그램이며 실수가 있을 수 있다. 그리고 프로그래머가 정말 알아야 할 것은 무슨 실수가 있느냐가 아니라, 툴 체인이 정상적으로 동작해서 만들어낸 결과 프로그램이 기대하는 것과 다른 모습일 수 있다는 것이다. 프로그램이 생성되었지만, 이상하게 동작할 가능성이 큰 것이다.

그래도 프로그래머는 열심히 소스 코드만 들여다볼 것이다. 툴 체인이 무슨 일을 하는지 모르니까 말이다. 이 책에서 컴파일러나 링커, 로더가 정확히 어떤 일을 하는지 모든 것을 알아보지는 못하더라도, 최소한 프로그램에 영향을 미칠 수 있는 몇 가지 중요한 점들은 짚어 보도록 하자.

2 | 컴파일러

컴파일러는 컴퓨터 분야에서 하나의 축복이다. 컴파일러가 만들어지지 않았다면 아직도 '01010010'로 프로그램을 작성하고 있었을지도 모른다. CPU의 모든 명령어를 완벽히 이해해야 하고, 프로그램이 실행되는 컴퓨터 시스템이 어떤 하드웨어로 구성되었는지 어떤 운영체제를 사용하는지 등 모든 내용을 숙지해야 한다. 그리고 그 모든 경우에 따라 각기 다른 프로그램 코드를 작성해야만 했을 것이다.

프로그래머가 이 모든 것을 잊고 소스 코드에만 집중하도록 하는 그 중심에는 컴파일러가 존재한다. 물론 이 모든 일을 컴파일러 혼자서 처리하지는 않는다. 전처리기와 컴파일러, 어셈블러, 링커와 로더, 운영체제 등 컴퓨터 시스템의 다양한 요소와 어우러져서 소스 코드는 결국 실행할 수 있는 프로그램으로 변신하게 된다.

프로그램을 빌드하는 과정에서 발생할 수 있는 문제나 프로그램이 실행되면서 발생하는 문제들은 모두 이 요소들과 밀접한 연관을 가지고 있다. 대부분 프로그램의 소스 코드가 잘못 작성된 이유로 문제가 발생하지만, 모든 경우가 소스 코드 잘못은 아니다. 컴퓨터는 결국 사람이 시키는 일만 하고 똑똑하게 처리하지도 못한다. 그리고 자신이 어떤 일을 하는지도 잘 모른다. 그러므로 프로그래머는 프로그램을 만드는 과정에 개입되는 요소들을 정확히 이해해야 하며 이들을 어떻게 조절할지 잘 알아야만 한다.

2.1 컴파일러는 도대체 어떤 일을 하는가

컴파일러에 관해 이야기하기 전에 우선 컴파일러가 하는 일을 확실히 알아야 한다. 컴파일러는 정확한 의미로는 하나의 프로그램 언어로 작성된 소스 코드를 다른 프로그램 언어로 변환하는 일을 담당한다. 대부분은 C나 C++와 같은 언어를 어셈블리어로 변환한다.

많은 사람이 가지는 오해 중 하나는 컴파일러가 소스 코드의 전처리나 어셈블리 언어로 변환 그리고 링커를 통해서 실행 파일을 만들어내는 모든 과정을 처리한다는 것이다. 다음 장에서 링커에 관해 살펴보겠지만, 중요한 점은 컴파일러는 소스 코드를 변환하는 일만 담당한다는 점이다. 물론 최근에는 이러한 개념이 모호해져서 실행 파일을 만드는 전체 툴 체인을 CC(Compiler Collection)라고 부르기도 하지만, 여기서는 컴파일러가 소스 코드를 다른 언어로 변환하는 일만 담당한다고 생각하도록 하자.

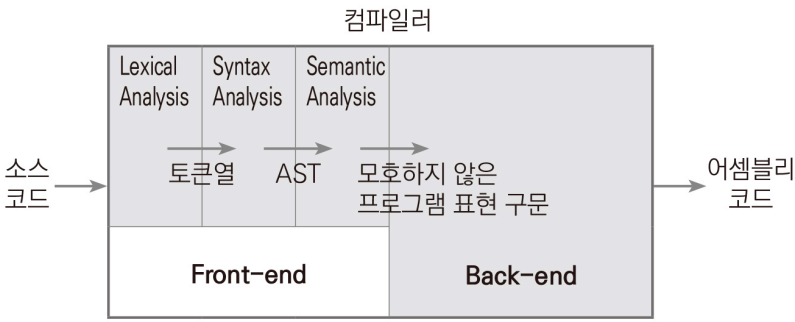
컴파일러는 주어진 프로그래밍 언어로 작성된 소스 파일을 읽어 들여서 먼저 토큰으로 구분한다. 토큰은 쉽게 말해서 하나의 단어라고 생각할 수 있다. 'if(a == 0)'이라는 문장을 받아들이면 컴파일러의 스캐너는 이들을 공백 문자, 괄호, 세미콜론, 개행문자(\n) 단위로 구분하여 단어들을 추출하고 이들을 목록으로 구성한다. 따라서 앞의 문장은 if, a, ==, 0이라는 네 개의 단어로 분해된다. 컴파일러의 파서(Parser)는 만들어진 토큰 목록을 사용해서 프로그래밍 언어의 문법 규칙을 찾는다. 따라서 파서는 'if(a == 0)'라는 문장에 C 언어의 if 문에 해당하는 문법 규칙을 찾아서 적용한다.

괄호를 빼먹었거나 세미콜론 대신 쉼표를 사용했을 때 에러 메시지를 출력하는 것은 컴파일러의 몫이다. 컴파일러는 기본적으로 소스 파일 하나 단위로 동작하며, 일반적으로 .s 확장자를 가진 어셈블리 코드 파일을 생성한다. 어셈블리 코드는 함수나 변수의 이름, if나 if-else 구문, for/while 반복 구문, switch 구문 등을 위한 레이블 정보들이 포함되어 있고, 어셈블러에 의해서 객체 파일(Object File)로 변환되는 것이 일반적이다. 물론 자바는 어셈블리 코드 대신 바이트 코드를 사용하지만, 여기서는 C와 C++를 다루고 있으니 어셈블리 코드라고 이해하자.

일반적으로 컴파일러가 소스 코드를 토큰으로 분해하고 분석해서 인식할 수 있는 문법으로 구성하는 단계까지를 컴파일러의 Front-end라 하고, 문법 트리를 사

용하여 최적화하고 다른 프로그래밍 언어로 변환하는 부분을 Back-end라고 지칭한다. 물론 이 과정을 한 단계로 처리하는 컴파일러도 있지만, 요즘 컴파일러는 Front-end와 Back-end의 두 단계로 구성되는 것이 일반적이다.

[그림 2-1] 컴파일러 구조



Front-end는 언어에 정의된 문법에 따라서 AST나 중간 언어 코드를 만들어내고, Back-end는 이를 목적 CPU에 맞는 어셈블리 코드로 바꾼다.

컴파일러는 소스 코드를 변환하는 과정에서 여러 가지 일을 수행한다. 소스 코드에 잘못된 점은 없는지 검사하며 변수나 구조체, 함수의 인자, 반환값 등의 형을 검사하기도 한다. 검사 과정에서 형이 서로 다르면 경고 표시를 하는데 자동으로 형 변환을 수행하면 약타입 언어Weak-type Language, 명시되지 않은 형 변환을 금지하고 컴파일을 중단하면 강타입 언어Strong-type Language라고 한다. 여기서 다루는 C와 C++는 모두 약타입 언어다. C++ 컴파일러는 컴파일 과정에서 템플릿을 사용한 어셈블리 코드를 만들기도 한다.

컴파일러가 하는 또 다른 중요한 일은 바로 최적화다. 사실 최적화는 프로그래밍 언어의 명세서에 포함되지 않는 내용이므로 컴파일러를 만드는 회사나 그룹에 따라 최적화 기법이나 정책 등이 많이 다르다. 하지만 컴파일러의 역사가 오래되었고 이론적인 배경 또한 구성이 탄탄한 만큼 정책이나 구현에서 약간의 차이를 보일 뿐

기술적인 부분에서 아주 뚜렷한 차이가 있는 것은 아니다. 하나의 컴파일러에서 사용되는 최적화 기법은 다른 컴파일러에서도 거의 같게 사용될 가능성이 매우 높다.

최적화는 대개 프로그래머가 작성한 프로그램의 실행에 영향을 미치지 않는 범위에서 일어난다. 영향을 미치는 범위는 최적화 수준이나 정의에 따라 조금씩 다르긴 하지만, 기본적으로 최적화가 적용된 프로그램은 최적화를 하지 않았을 때의 프로그램 실행과 같아야 한다.

그런데 정말 모든 최적화가 프로그래머가 의도한 대로 프로그램이 동작하는 것을 보장할 수 있을까?

2.2 프로그램 실행이 안 되는 것은 전부 소스 코드 탓이다?

프로그램이 동작하지 않는 것을 전부 소스 코드 탓으로 돌리기에 프로그램은 너무나도 복잡하고 관여하는 요소도 매우 많다. 이 모든 것을 다 알기는 어려우므로 여기서는 특히 컴파일러가 어떤 일을 저지를 수 있는지 알아보자. 아는 것이 힘이며 힘이 셀수록 야근을 덜 하거나 빨리 끝낼지도 모르는 일이다.

다음 코드를 살펴보자.

[코드 2-1]

```
      #include <stdio.h>
      #include <unistd.h>
      #include <pthread.h>

      int i;

01    void* foo(void *data)
02    {
03       extern int i;
```

```
04      i = 0;
05      while(1)
06      {
07         if(i)
08         {
09            printf("Hello, World!\n");
10         }
11      }
12      return 0;
13   }

14   void* bar(void *data)
15   {
16      extern int i;
17      i = 0;
18      while(1)
19      {
20         sleep(1);
21         i = !i;
22      }
23   }

     int main( )
     {
        int tid;
        int status;
        pthread_t p_thread[2];

24      tid = pthread_create(&p_thread[0], NULL, foo, NULL);
25      tid = pthread_create(&p_thread[1], NULL, bar, NULL);

26      pthread_join(p_thread[0], (void **)&status);
```

```
27      pthread_join(p_thread[1], (void **)&status);

        return 0;
    }
```

01 ~ 23 foo와 bar는 각각 스레드로 실행된다.
24 ~ 25 foo와 bar 스레드를 만들고 실행한다.
26 ~ 27 스레드가 종료될 때까지 기다린다.

스레드 프로그램을 작성해 본 적이 있다면 [코드 2-1]에 대해서 딱히 설명할 필요가 없겠지만, 모르는 분들을 위해서 간단히 설명하겠다. [코드 2-1]은 POSIX의 PTHREAD라는 라이브러리를 사용해서 두 개의 스레드를 생성하고 프로그램이 실행되면 동시에 두 개의 스레드가 시작하도록 구성되었다. main 함수가 시작되면 foo와 bar 함수를 사용해서 두 개의 스레드가 시작되고, main 함수는 두 개의 스레드가 종료할 때까지 기다렸다가 프로그램을 종료한다. foo 함수는 변수 i 값을 계속 감시하다가 i 값이 0이 아니면 화면에 특정 메시지를 출력한다. bar 함수는 1초 동안 아무런 동작도 하지 않다가 i 값이 0이면 1로, 1이면 0으로 바꾼 뒤 다시 1초 동안 잠든다. main, foo, bar는 모두 동시에 동작하므로 이 프로그램을 실행하면 1초 동안 화면에 계속 메시지가 출력되다가 1초 동안 가만히 있는 것을 반복하리라 기대하고 이 코드를 작성하였다.

하지만 이 프로그램은 기대한 대로 동작하지 않는다. 물론 사용하는 컴파일러나 운영체제 설정에 따라서 기대한 대로 동작할 수도 있다. 여기서 중요한 것은 이 코드에서 잘못된 부분은 없다.[01] 스레드 생성이 잘못되거나 인자를 잘못 전달하거나 그 흔한 스택 오버플로도 없다. 배열값의 범위를 넘어선 변수 접근도 없다. 하지만 프로그램이 제대로 동작하지 않으면 프로그래머 대부분은 소스 코드에 어떤 문제가 있으리라 짐작하고 찾을 때까지 프로그램을 실행하고, 고치고, 실행하고, 디버깅을

01 물론 소스 코드에서 잘못된 부분이 없음을 의미한다.

반복할 것이다.

소스 코드에 문제가 없는데도 이렇게 고치려고 시도하는 것은 프로그램의 실행이나 생성에 관여하는 요소가 무엇인지 잘 모르거나 혹은 그들이 잘못 동작하리라고는 꿈에도 생각하지 못하기 때문이다.

결론부터 말하면 [코드 2-1]이 동작하지 않는 것은 바로 컴파일러의 최적화 기법 때문이다. 컴파일러는 다양한 방법으로 프로그램의 소스 코드에 최적화를 시도하며 이 과정에서 소스 코드의 일부가 사라지게 된다. 컴파일러의 최적화 기법은 그 종류만 해도 수십 가지에 이르지만 우선 [코드 2-1]이 제대로 동작하지 못하게 만드는 원흉부터 살펴보자.

2.3 컴파일러 최적화 기법에서 발생하는 문제

컴파일러 최적화는 사람이 만든 기법이므로 상식적인 수준에서 이해할 수 있는 기법들이 대부분이다. 물론 이런 기법들을 실제로 컴파일러에 구현하는 것은 별개의 문제다. 최적화 기법을 설명하기 전에 알아야 할 것은 CPU에서 실행되는 명령어가 메모리에 적재되는 것과 명령어가 실행되는 동안 어떤 데이터를 처리하기 위해서 메모리에 접근하는 것이 별개로 진행된다는 점이다. 다음 어셈블리 코드를 살펴보자.

[코드 2-2]

```
01    mov $16(%ebp), %eax
02    add %eax, $12(%ebp)
```

01 ~ 02 이 코드는 이를테면 'A+B;'와 같다. $12(%ebp)가 A이고 $16(%ebp)가 B다.

EBP는 인텔 아키텍처에서 스택의 바닥을 가리키는 용도로 사용되는 레지스터다. [코드 2-2]는 스택의 어떤 영역에서 값을 읽어와서 EAX 레지스터에 저장한 뒤 이

값을 다시 스택의 다른 영역에 더하는 역할을 한다. CPU가 [코드2-2]를 실행하려면 우선 코드를 메모리에 적재한 뒤 해당 명령어를 CPU에 적재한다. 명령어를 해석하고 필요에 따라 메모리나 캐시에 접근하여 원하는 값을 가져온다. [코드2-2]에서 가장 많은 실행 시간을 차지하는 것은 아마도 $16(%ebp)와 $12(%ebp)라고 지정된 스택의 특정 영역에서 값을 읽어오거나 값을 쓰는 행위일 것이다.

캐시에서 값을 읽어오고 값이 없으면 메모리에서 읽어와야 하며, 값을 읽어올 때까지 그다음 명령어는 실행되지 않는다. 만약 $16(%ebp)라는 스택 영역에 저장된 값이 바뀌지 않는 상수값이라면 코드를 어떻게 작성할 수 있을까? 예를 들어 저장된 값이 10이라면 다음과 같이 작성할 수 있다.

[코드 2-3]

```
01    mov 10, %eax
02    add %eax, $12(%ebp)
```

01 ~ 02 이 코드는 'A+=10;'과 같다.

명령어에 필요한 값이 이미 기록되어 있으므로 메모리에 접근해서 값을 가져올 필요가 없다. 사실 더 간단히 줄이면 다음과 같다.

[코드 2-4]

```
      add 10, $12(%ebp)
```

물론 CPU가 이런 형태의 명령어를 지원하는 경우에 한해서다. 명령어가 실행되려면 어차피 해당 명령어가 메모리를 거쳐 CPU로 전달되어야 하는데 필요한 상수값을 메모리가 아닌 명령어에 직접 삽입하는 것은 추가적인 메모리 접근을 줄일 수 있으므로 성능상 이점이 매우 많다. 이러한 이유로 컴파일러는 불필요한 변수

등을 상수로 대체하고, 상수로 이루어진 계산식을 미리 처리하는 방식으로 최적화를 수행한다. 대부분 변수는 DU$^{Define-Use}$ chain을 통해서 언제까지 그 값이 유지되고 언제 값이 변하며 언제 값을 사용하는지 나타낼 수 있다. 요즘 컴파일러들은 DU chain과 같이 분석된 프로그램 정보를 이용하여 변수에 대한 최적화를 시도하는데 이는 소스 코드 수준에서 분석된다. 대표적인 최적화 기법으로 Constant Propagation과 Constant Folding, 그리고 Code Elimination을 들 수 있다.

■ Constant Propagation / Constant Folding / Code Elimination

Constant Propagation은 한 번만 정의되고 계속 같은 값이 사용되는 변수일 때 이를 상수로 대체하는 것이고, Constant Folding은 상수로 이루어진 계산식을 미리 계산해서 다른 상수값으로 대체하는 기법이다. 다음 코드를 보자.

[코드 2-5]

```
01   int a = 10;
     int b = a * 20;
     int c;

     c = b - 2;
     if(c > a)
        c = c * 2;
     return c;
```

01 변수 a는 10이 저장된 뒤 값이 변경되지 않는다.

사실 큰 의미 없는 코드다. 변수 a는 초기에 10이라는 값이 할당된 뒤 그 값이 변하지 않는다. 컴파일러는 Constant Propagation을 적용해서 변수 a가 사용되는 곳에 모두 10을 적용한다. 그렇다면 코드는 다음과 같이 변할 것이다.

[코드 2-6]

```
      int a = 10;
01    int b = 10 * 20;
      int c;

      c = b - 2;
02    if(c > 10)
        c = c * 2;
      return c;
```

01 ~ 02 a 대신 10이 직접 사용된다.

이제 변수 b에는 10 * 20이라는 결과값이 저장된다. Constant Folding을 통해서 10 * 20은 200이라는 상수로 대체된다.

[코드 2-7]

```
      int a = 10;
01    int b = 200;
      int c;

      c = b - 2;
      if(c > 10)
        c = c * 2;
      return c;
```

01 10 * 20을 200으로 대체한다. 이제 b는 200이라는 값이 저장되고 변경되지 않는다.

다시 Constant Propagation을 통해 b 자리에 200이라는 상수값을 직접 사용한다.

[코드 2-8]

```
int a = 10;
int b = 200;
int c;

c = 198;
// Constant Folding에 의해 c = 198로 대체된다. 원래 코드는 c = 200 - 2다.
if(c > 10)
   c = c * 2;
return c;
```

다시 Constant Propagation을 통해서 c 값이 상수로 대체된다.

[코드 2-9]

```
int a = 10;
int b = 200;
int c;
01  if( 198 > 10)
02     c = 198 * 2;
return c;
```

01 ~ 02 c 대신 198을 직접 사용한다.

이제 Constant Folding과 Constant Propagation을 통해서 마지막으로 c 값이 모두 상수로 대체된다. 그렇게 되면 변수 a, b, c가 사용되는 곳이 없으므로 불필요한 코드가 된다. 이렇게 사용되지 않거나 절대 실행되지 않는 코드를 제거하는 기법을 Code Elimination이라고 한다. [코드 2-9]에서는 선언문 int a = 10, int b = 200, int c와 더불어 항상 그 판단 결과가 참이 되는 if(198 > 10) 구문 역시 제거된다. 따라서 이 코드는 최종적으로 다음과 같이 변한다.

[코드 2-10]

```
    return 396;
```

거짓말 같은가? 실제로 인텔 CPU를 사용하는 리눅스 운영체제에서 GCC에 최적화 수준을 2레벨로 지정하고, 이 코드를 어셈블리 코드로 변환하였을 때 다음 어셈블리 코드를 얻었다.

[코드 2-11]

```
01  movl $396, %eax
    ret
```

01 일반적으로 인텔 CPU는 함수 반환값을 EAX 레지스터로 전달한다.

딱 두 줄이다. EAX 레지스터에 396을 대입하고 반환한다. 'return 396;'이 그대로 어셈블리 코드로 변환된 내용이다. Constant Propagation, Constant Folding, Code Elimination 외에도 수많은 최적화 기법이 있지만, [코드 2-1]이 왜 제대로 동작하지 않는지 이해하기 위해서는 이 세 가지만 알아도 충분하다.

다시 [코드 2-1]로 돌아가서 생각해보자. 이 최적화 기법들은 대개 함수 단위로 실행되므로 우선 함수 foo에 최적화 기법이 적용되면 어떻게 될지 생각해 보자.

[코드 2-12]

```
    void* foo(void *data)
    {
01      extern int i;
        i = 0;
        while(1)
        {
```

```
            if(i)
            {
               printf("Hello, World!\n");
            }
         }
         return 0;
      }
```

01 변수 i는 foo 함수 외부에 정의되어 있다.

변수 i에 값 0이 대입되면 그 이후에는 값이 변경되지 않으므로 Constant Propagation에 의해 변수 i는 0으로 대체된다. 따라서 이 코드는 다음과 같이 변한다.

[코드 2-13]

```
      void* foo(void *data)
      {
         extern int i;
         i = 0;
         while(1)
         {
01          if(0)
            {
               printf("Hello, World!\n");
            }
         }
         return 0;
      }
```

01 변수 i는 0으로 대체된다. 이제 if 구문은 항상 거짓이 되며 실행되지 않는다.

이제 if 구문의 결과는 항상 거짓이 되므로 if 구문 안에 포함된 printf 구문도 절대 실

행되지 않는 코드가 된다. 이제 남은 것은 while(1)로 이 코드에 의해서 'return 0;' 구문 역시 실행되지 않는다. 따라서 Code Elimination에 의해 [코드 2-14]와 같이 변한다. int i 대입 구문은 외부에서 사용되는 변수이므로 해당 값에 대한 초기화는 남아있다.

[코드 2-14]

```
      void* foo(void *data)
      {
         extern int i = 0;
01       while(1);
      }
```

01 while 구문 아래 코드는 실행되지 않으므로 제거되었다.

실제로 GCC에 3레벨 수준의 최적화를 적용하면 foo 함수가 다음과 같이 생성된다.

[코드 2-15]

```
8048580: movl $0x0,0x804a024 // extern int i = 0;
804858a jmp 804858a <foo+0xa> // while(1);
```

i에 값 0을 대입하고 계속 같은 명령어로 뛴다. while(1)에 해당하는 부분이다. bar 함수에서 i 값을 변경해도 화면에는 메시지가 출력되지 않는다. 프로그래머가 소스 코드를 작성했지만, 컴파일러가 원하는 코드를 만들어내지 않은 것이다. 물론 이 수준의 최적화는 상당히 공격적이고 자주 사용하지 않는 편이긴 하다. 하지만 2레벨 수준의 최적화에서도 프로그램은 제대로 동작하지 않는다. 비록 3레벨 수준만큼 코드가 줄어들지는 않지만, 최적화로 인해서 프로그램이 동작하지 않는다는 점은 같다. 대부분 프로그램을 생성하는 과정에서 2레벨 수준의 최적화를 사용하므로 이 문제는 상당히 자주 발생할 가능성이 높다.