리눅스 환경에서의 함수 단위 동적 커널 업데이트 시스템의 설계와 구현

(A Dynamic Kernel Update System with a Function Granularity for Linux)

박 현 찬 † 김 세 원 † 유 혁 ††

(Hyunchan Park) (Sewon Kim) (Chuck Yoo)

요 약 동적인 커널의 업데이트는 복잡한 운영체제 커널의 빈번한 기능 개선 및 버그 수정을 동작 중인 커널의 중단없이 수행하는 것이다. 동적인 업데이트를 위해서는 주로 명령어 단위의 업데이트 기법이 사용되지만 어셈블리 언어 수준에서 개발 및 유지, 보수가 이루어지기 때문에 실제 커널에 적용하기 어렵다. 이런 문제점을 극복하기 위해 우리는 C 언어 수준에서 함수 단위로 동적인 커널 업데이트를 수행하는 시스템을 설계하고 리눅스에 구현하였다. 이 시스템은 업데이트 개발 환경을 커널의 개발 환경과 일치시킴으로써 업데이트의 개발과 수행을 편리하게 하여 실제 커널에의 활용 가능성을 증대시킨다. 우리는 이렇게 증대된 활용 가능성을 실제로 알아보기 위해 이 업데이트 시스템을 이용하여 EXT3 파일 시스템을 간단하게 업데이트하는 사례를 보였다.

키워드 : 동적 업데이트, 함수 단위 업데이트, 커널 업데이트

Abstract Dynamic update of kernel can change kernel functionality and fix bugs in runtime. Dynamic update is important because it leverages availability, reliability and flexibility of kernel. An instruction-granularity update technique has been used for dynamic update. However, it is difficult to apply update technique for a commodity operating system kernel because development and maintenance of update code must be performed with assembly language. To overcome this difficulty, we design the function-granularity dynamic update system which uses high-level language such as C language. The proposed update system makes the development and execution of update convenient by providing the development environment for update code which is same for kernel development. We implement this system for Linux and demonstrate an example of update for EXT3 file system. The update was successfully executed.

Key words : Dynamic update, Function-granularity update, Kernel update

1. 서 론

이 논문은 함수 단위의 동적인 커널 업데이트 시스템을 설계하고 구현한 내용을 기술한다. 동적인 커널 업데이트는 현재 동작 중인 커널의 수행을 중단하지 않고 기능을 확장하는 방법이다. 이러한 기법은 커널 프로파일링, 커널 디버깅, 코드 패치의 설치, 그리고 코드 최적화 등을 동작 중인 커널에 적용하기 위해 사용될 수 있다[1].

동적인 업데이트 기법은 특히 실제로 사용되고 있는 운영체제의 커널에 더욱 유용하게 활용할 수 있다. 유닉스(UNIX), 리눅스(Linux), 솔라리스(Solaris) 등 널리 쓰이는 운영체제의 경우 지속적으로 기능이 개선되며 버그가 수정된다. 하지만 이러한 변화들은 사용자의 커널에 즉각적으로 적용되지 않는다. 실제 커널들은 그 코드의 크기가 크고 새로운 업데이트의 적용을 위해서는 시스템의 재기동이 필요하기 때문에 많은 횟수의 미세

† 학생회원 : 고려대학교 컴퓨터학과
hcpark@os.korea.ac.kr
swkim@os.korea.ac.kr
†† 종신회원 : 고려대학교 컴퓨터학과 교수
hxy@os.korea.ac.kr
논문접수 : 2007년 10월 2일
심사완료 : 2008년 1월 23일

Copyright@2008 한국정보과학회 : 개인 목적이나 교육 목적인 경우, 이 저작물의 전체 또는 일부에 대한 복사본 혹은 디지털 사본의 제작을 허가합니다. 이때, 사본은 상업적 수단으로 사용할 수 없으며 첫 페이지에 본 문구와 출처를 반드시 명시해야 합니다. 이 외의 목적으로 복제, 배포, 출판, 전송 등 모든 유형의 사용행위를 하는 경우에 대하여는 사전에 허가를 얻고 비용을 지불해야 합니다.
정보과학회논문지: 시스템 및 이론 제35권 제5호(2008.6)

한 기능 개선과 버그 수정을 매번 적용하기는 어렵다. 따라서 업데이트할 기능을 포함한 코드나 바이너리 이미지를 따로 분리하여 배포하고 동적으로 커널에 설치하면 커널의 기능을 보다 쉽게 업데이트할 수 있다.

이렇게 커널의 기능에 동적인 확장성을 부여하고자 하는 연구는 크게 두 가지 방향으로 진행되고 있다. 첫 번째는 설계 단계에서부터 확장성을 고려한 새로운 커널을 만드는 방법이다. 대표적인 연구로는 SPIN[2], K42[3] 등의 연구가 있다. 이러한 연구는 완전히 새로운 커널 구조를 제안함으로써 문제를 단순하게 만들 수 있다. 그러나 널리 사용되는 기존 커널들과의 호환성 유지 문제로 인해 실제로 적용 가능한 해결 방법이 되기는 어렵다.

두 번째 연구 방향은 이러한 점을 해결하기 위하여 기존 커널에 대해 확장성을 부여하는 연구이다. 대표적으로 KernInst[1], GILK[4], SLIC[5] 등이 있다. 이들은 기존 커널에 대해 동적으로 확장성을 부여하는 연구로 대상 커널은 Solaris, Linux 등이다. KernInst는 사용자에게 업데이트에 필요한 커널의 호출 흐름 그래프(CFG: Call Flow Graph)를 제공하여 안전하게 커널의 기능을 명령어 단위로 확장할 수 있는 기능을 제공한다. GILK 또한 비슷한 기능을 제공하며 사용자는 어셈블리 언어로 작성된 패치 파일을 커널의 안전한 영역에 삽입할 수 있다. SLIC은 어플리케이션이 커널 인터페이스를 호출할 때 해당 호출을 가로채어 확장 모듈이 처리하도록 하는 방식으로 커널의 확장성을 제공한다.

이러한 연구들은 공통적으로 기존 커널을 전혀 수정하지 않거나 약간의 수정을 통해 확장성을 제공한다. 하지만 사용자가 확장하고자 하는 코드를 어셈블리 언어로 작성해야 하거나[1,4], 시스템 콜이나 가상 파일 시스템(Virtual File System)처럼 인터페이스가 잘 정의(well-defined)되어 있어야만 확장이 가능한[5] 한계가 존재한다. 이러한 문제점들은 현재 시스템에 확장성을 제공하는데 현실적인 어려움으로 작용한다. 따라서 커널에 동적인 확장성을 부여하기 위한 두 가지 연구 방향은 새로운 커널을 디자인하는 커다란 부담이 있거나, 기존 커널에 적용하기에 현실적인 어려움이 있다.

우리는 이 논문에서 위에서 언급한 문제점들을 해결하기 위해 함수 단위로 커널을 업데이트할 수 있는 시스템을 설계하고 구현하였다. 이 시스템의 특징은 다음과 같다. 1) C 언어 레벨에서 기존의 함수 코드를 수정하고 그 내용을 동작 중인 커널에 반영할 수 있도록 하여 실제 업데이트 수행의 편의성을 높였다. 2) 트랩 기반 동적 재구성 기법[6](Trap-based dynamic instrumentation)을 통해 함수 단위의 업데이트가 이루어진다. 이 기법은 하드웨어 환경에 독립적인 특성을 가지기 때문에 여러 환경에서 사용되는 리눅스에 적용하기 적합하다. 3) 이러한 주요 특징과 더불어 우리는 동적으로 수행된 업데이트를 시스템의 재기동 시에도 지속적으로 유지시키는 영속성(persistency)과 업데이트의 수행과 그 내용이 커널의 동작에 오류를 일으키지 않음을 검증하는 무결성(integrity) 등을 고려하여 이 시스템을 설계하였다.

우리는 제안된 시스템을 2.6.15 버전의 리눅스 커널에 LKM(Loadable Kernel Module) 시스템을 확장하여 구현하였고, 사용자는 이 시스템을 기존 리눅스 소스 코드에 약간의 패치를 가하는 형태로 설치하여 사용할 수 있다. LKM 시스템은 동적인 커널 확장성을 지원하기 위해 리눅스에 본래 포함되어 있는 시스템이지만 기존 커널의 내용을 업데이트하기보다 새로운 기능을 추가하는 방법으로 동작하기 때문에 활용이 제한적이다. 그러나 리눅스 커널에서 제공하는 기본적인 시스템이기 때문에 이 시스템을 확장하여 구현하는 방향을 선택하였다.

2. 함수 단위 업데이트 기법

커널 업데이트를 함수 단위로 수행하는 것은 기존의 명령어 단위 업데이트 기법의 단점인 편의성을 보완하기 위해 본 논문에서 제안하는 기법이다. 사용자는 함수 단위의 업데이트를 통해 커널 기능을 확장할 수 있다. 또 함수의 추가도 가능하며 추가된 함수를 커널의 다른 부분에서 호출하는 것도 가능하다.

2.1 명령어 단위 업데이트 기법과의 비교

동적 업데이트의 단위로 활용할 수 있는 가장 작은 단위는 명령어 단위이다. 명령어 단위의 업데이트 기법은 우선 가장 작은 단위라는 점에서 업데이트에 소요되는 오버헤드를 최소화할 수 있는 장점을 지닌다. 그러나 개발 과정과 유지 보수 면에서 사용자에게 최대한의 편의성을 제공하지는 못한다.

명령어 단위의 커널 업데이트를 수행하고자 하는 사용자는 어셈블리 언어 수준의 명령어와 바이트 코드로 표현된 명령어를 해석하고 다룰 수 있어야 한다. 그리고 그것은 사용하는 하드웨어 구조(hardware architecture)에 따라 다르기 때문에 매우 어려운 문제이다. 또 만약 이러한 지식을 갖추고 있는 사용자라고 해도 커널의 어떤 부분을 수정할 것인지를 알아내야 하는 문제를 쉽게 해결하기는 어렵다. 수정하고자 하는 부분의 바이트 코드를 해석할 수 있어야 하는데, 일반적인 커널 이미지는 컴파일러에 의해 최적화(optimizing)되어 생성되기 때문이다. 따라서 명령어 단위의 업데이트 기법을 실제로 활용하는 데는 큰 어려움이 있다. 그러므로 우리는 본 논문에서 리눅스에서 사용하는 C 언어 수준에서 작업할 수 있는 함수 단위의 업데이트 기법을 제안하고자 한다.

2.2 트랩 기반 동적 재구성 기법

동적 재구성을 구현하기 위해서는 일반적으로 코드 스플라이싱(code splicing)과 코드 리플레이스먼트(code replacement)의 두 가지 기법을 사용한다. 코드 스플라이싱 기법은 기존의 기계 코드 명령어를 새로운 코드 영역으로의 분기 명령어(JMP, CALL, RET 등)로 덮어쓰고, 새로운 코드 영역 끝에서 다시 본래의 수행 흐름으로 복귀하는 방식이다.

코드 리플레이스먼트는 기존 코드 위에 새로운 코드를 덮어쓰는 동작으로만 이루어진다. 따라서 코드 스플라이싱은 추가하고자 하는 명령어 코드 크기에 대한 제한이 없고 분기명령어의 작은 수정만을 필요로 한다. 대신 분기명령어로 인한 제한이 생겨난다. 분기명령어의 종류에 따라 이동할 수 있는 메모리 영역에 제한이 생길 수 있고 코드의 크기가 다를 수 있기 때문에 덮어쓰고자 하는 기존 코드 명령어와 크기가 일치하지 않을 수 있다. 또 분기로 인해 성능도 하락할 수 있다. 코드 리플레이스먼트는 새로운 코드의 크기가 기존의 영역보다 클 경우 사용이 불가능하기 때문에 너무나 제한적이다.

이러한 두 기법의 단점들을 보완하기 위해 우리는 분기 명령어로서 트랩을 사용하는 코드 스플라이싱 기법[6]을 사용하였다. 이러한 명령어로서 대표적인 것은 인텔 아키텍처(Intel architecture)의 "INT" 명령어가 있다. [6]에 의하면, 트랩 명령어를 분기 명령어로 사용함으로써 얻을 수 있는 장점은 다음과 같다. 1) 특정한 하드웨어 환경에 의존적이지 않고 독립성을 보장할 수 있으며, 2) 코드 스플라이싱에 직접적으로 요구되는 분기 명령어의 요구량이 작다(2 바이트). 우리는 이러한 장점들이 우리의 시스템에 필요하다고 판단하였다. 특히 첫 번째 장점은 리눅스가 여러 하드웨어 환경에서 널리 사용되는 점을 감안하면 대단히 중요하다는 것을 알 수 있다. 두 번째 장점은 2.1절에서 언급했던 명령어 단위 업데이트 기법에서 발생하는 분기 명령어의 크기에 따른 문제점을 해결해준다. 그림 1은 트랩 기반 동적 재구성 기법의 동작을 나타낸 것이다. INT 127은 하나의 예제로서 127번 트랩을 발생시킨다는 뜻이다.

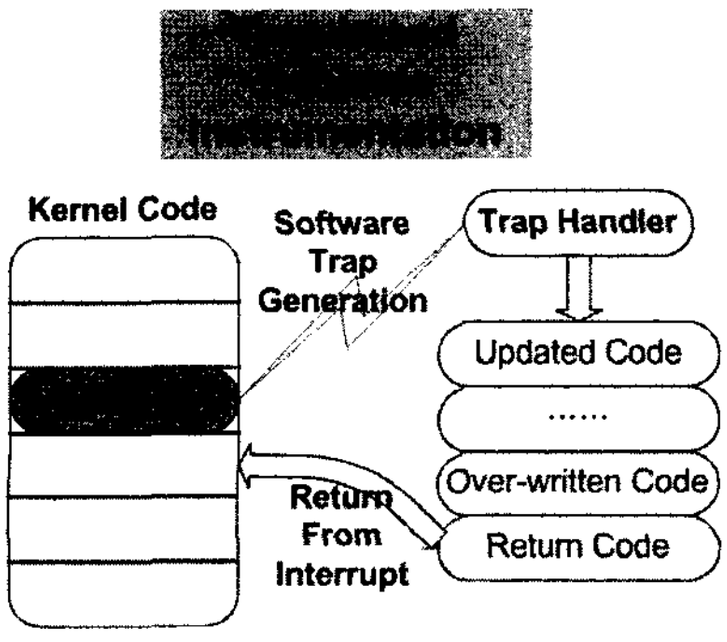

그림 1 트랩 기반 동적 재구성 기법

3. 업데이트 시스템 설계

함수 단위 커널 업데이트 기법을 이용한 업데이트 시스템의 전체 설계 내용에는 함수 단위 업데이트 기법을 수행하기 위한 주요 컴포넌트들과 그 외 다른 부가적인 컴포넌트들이 포함된다. 각각의 컴포넌트들은 유저 영역 또는 커널 영역에 위치하고 이를 통해 두 부분으로 나누어 볼 수 있다. 유저 영역의 컴포넌트는 사용자가 커널의 업데이트 시스템에 접근할 수 있는 인터페이스를 제공한다. 커널 영역에 있는 업데이트 시스템이 업데이트를 수행하는 핵심적인 역할을 한다. 업데이트 시스템의 전체 구조와 상호간의 동작은 그림 2와 같고, 아래에서 각 컴포넌트들의 역할을 기술한다.

3.1 업데이트 시스템 컴포넌트

• 업데이트 어시스턴트(Update Assistant)

유저 레벨 어플리케이션의 형태로 함수 단위의 업데이트를 수행하기 위한 개발 환경을 제공한다. 예를 들어, 커널 내 업데이트 대상 함수 A와 변경된 함수 A'가 있다면 이러한 정보를 정형화된 리눅스 LKM 모듈-업데이트용 스켈레톤(skeleton) 모듈 소스 파일-에 적재하고 커널에 업데이트를 요청한다. 비슷한 역할을 수행하는 유틸리티로 리눅스에서 사용되는 Insmod가 있다. Insmod는 커널 모듈 오브젝트(kernel module object)로 컴파일된 파일을 리눅스의 모듈 시스템 서비스들을 이용하여 커널 영역에 배치시키고 모듈을 가동시키는 역할을 한다. 이러한 동작을 이용하여 우리는 업데이트할 코드를 커널 영역에 배치시키고 커널 내 업데이트 시스템에 업데이트를 요청하는 동작을 수행할 것이다.

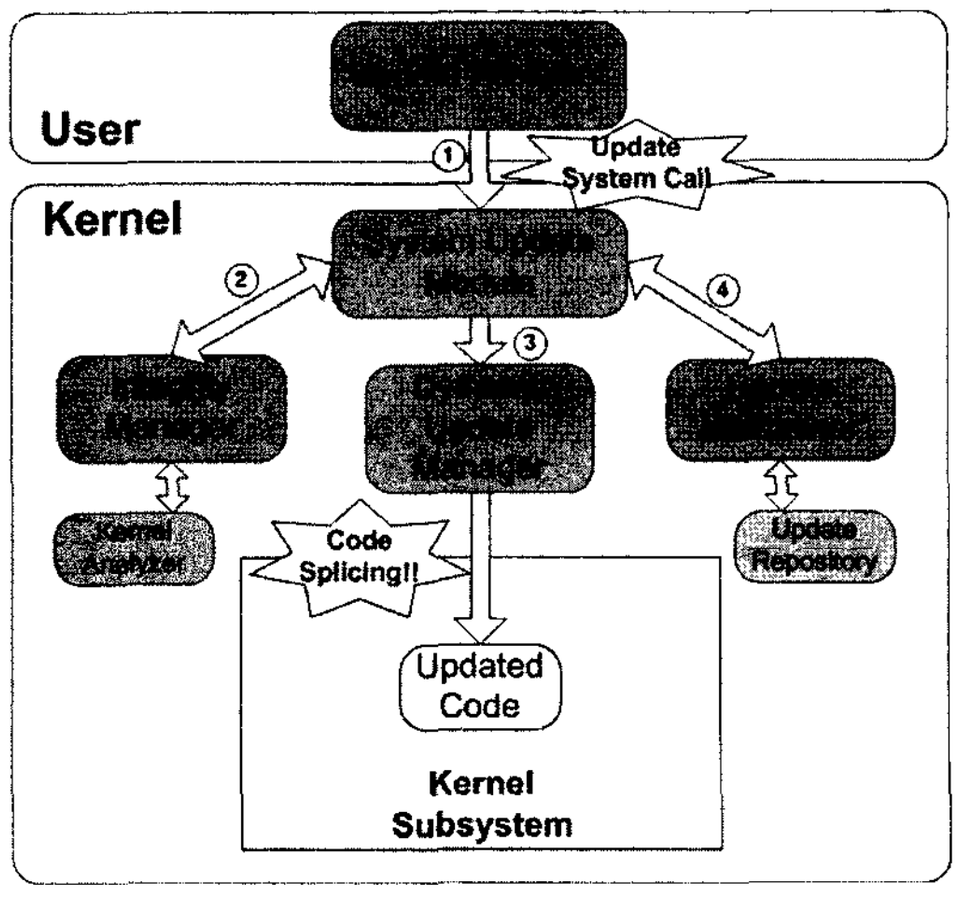

그림 2 전체 업데이트 시스템 구조

• 시스템 업데이트 모듈(System Update Module)

커널에 위치한 업데이트 시스템의 일부로 유저-커널 간의 인터페이스를 제공하고 업데이트를 수행하는 주체적 역할을 담당한다. 이 컴포넌트는 사용자가 요청한 업데이트 정보를 처리하고 다른 컴포넌트들을 이용하여 실제 동적 업데이트를 수행한다. 그리고 업데이트 어시스턴트가 필요로 하는 서비스들을 제공한다. 예를 들어 업데이트 어시스턴트는 커널 심볼에 대한 보다 많은 정보를 필요로 하는데, 이는 LKM 보다 많은 범위의 커널 심볼을 다루어야 하기 때문이다. 따라서 이러한 정보를 제공해줄 시스템 콜이 필요하고, 이를 시스템 업데이트 모듈에서 제공한다.

• 동적 업데이트 관리자(Dynamic Update Manager)

함수 단위의 동적 업데이트를 담당하는 핵심 컴포넌트이다. 시스템 업데이트 모듈이 요청한 함수 업데이트에 따라 해당 함수를 트랩 기반 동적 재구성 기법을 이용하여 새로운 함수를 호출할 수 있도록 수정한다.

• 업데이트 메인테이너(Update Maintainer)

동적으로 이루어진 업데이트가 시스템의 재기동시에도 적용될 수 있도록 유지하는 역할을 담당한다. 동적인 업데이트는 일반적으로 메모리의 코드 영역을 수정함으로써 이루어지는데, 이러한 변경 사항은 재기동 시에 다시 적용되지 않는다. 따라서 이차적인 저장매체에 기록되어 있는 커널 이미지를 수정하는 방식 등으로 동적인 업데이트를 정적으로 유지해줄 기법이 필요하다. 우리는 업데이트 저장소(update repository)를 구성하여 동적으로 이루어진 업데이트 정보를 저장한다. 그리고 해당 정보를 커널이 새로이 부팅될 시에 동적인 업데이트를 다시 수행하도록 하는 간단한 구현으로 우선적인 업데이트 유지를 시도할 계획이다. 그러나 이러한 방법에는 한계가 있다. 함수 단위 업데이트를 구현하기 위해 우리는 기존 함수에서 트랩을 이용하여 새로운 함수를 호출하는 방식을 사용하고 있다. 이러한 경우 함수 호출의 경로가 길어지게 되고 트랩을 거쳐야 하므로 단순한 함수 호출보다 처리 시간이 길어지게 된다. 따라서 이러한 성능의 저하를 막기 위해 커널 이미지에 직접 업데이트 내용을 반영하는 구현이 필요하다.

• 무결성 관리자(Integrity Manager)

앞서 언급한 바와 같이 동적인 업데이트는 업데이트의 신뢰성과 업데이트로 인한 시스템의 무결성을 보장하기 위한 방법이 필요하다. 무결성 관리자는 이러한 역할을 담당하게 된다. 업데이트된 코드가 커널에 무해한지, 업데이트는 언제 이루어져야 하는지를 관리하고 결정한다.

3.2 업데이트 수행 과정

위와 같이 구성된 컴포넌트들은 업데이트가 수행될 때 서로 통신을 수행하며 순차적으로 동작한다. ① 업데이트 어시스턴트가 사용자의 업데이트 요청을 커널로 전달하면, ② 시스템 업데이트 모듈이 이를 무결성 관리자로 보내 코드를 검증한다. ③ 코드의 검증 후, 동적 업데이트 매니저를 통해 실제 커널 업데이트가 수행되고, ④ 이후 업데이트 메인테이너가 이미 수행된 업데이트의 유지를 담당하게 된다.

4. 구현

4.1 함수 단위 동적 업데이트의 수행

우리는 3장에서 설계한 업데이트 시스템을 인텔 플랫폼 기반의 리눅스 커널 2.6.15 버전에 구현하였다. 우선 기존 커널에 20번 트랩에 대한 신호를 처리하는 트랩 핸들러를 추가하였다. 20번 트랩은 함수 업데이트로 인해 변경된 함수의 초반부에서 업데이트된 함수 코드로의 분기를 위해 발생된다. 커널의 수행 중에 20번 트랩이 발생하면 위에서 새로 추가된 트랩 핸들러가 이를 처리한다. 핸들러의 수행 내용은 다음과 같다. 1) EAX 레지스터 값을 통해 어떤 함수로부터 발생한 트랩인지 구분한다. 이는 커널 내의 여러 함수들이 제각기 업데이트된 상황에서 각각의 함수에 해당하는 업데이트 코드로 정확하게 분기하기 위해 필요하다. 2) 함수 인자의 전달과 인터럽트 서비스 루틴(interrupt service routine) 수행의 보호를 위한 스택 처리를 수행한다. 이것은 다음 절에서 자세히 설명하도록 한다. 3) 업데이트 코드를 수행한다. 4) 복귀값(return value) 전달을 위한 스택 처리를 수행한다. 5) 핸들러의 동작을 종료하고 IRET 명령어를 통해 인터럽트 서비스 루틴에서 복귀한다. 이 동작으로 복귀하는 주소는 업데이트된 본래 함수이지만, 즉시 RET 명령이 호출되어 본래 수행 흐름으로 복귀하게 된다. 이러한 과정을 통해 업데이트된 함수가 수행되고, 전체 과정을 그림으로 표현하면 그림 3이며 위에서 설명된 부분은 그림의 ②부터 ⑦까지의 과정이다. ①은 함수의 호출을 나타낸다.

위에서 구현된 트랩 핸들러는 업데이트된 함수의 수행을 위한 것이다. 함수의 동적인 업데이트는 동적 업데이트 매니저에 의해 이루어진다. 업데이트는 그림 4와 같은 일련의 코드를 대상 함수의 초반부에 덮어쓰는 것으로 구현된다. 이 코드는 앞서 설명한 핸들러의 동작에 상응하여 작성된 것이다. 처음의 MOV 명령의 인자에는 $0x0 대신 각 함수에 대해 부여되는 업데이트 번호가 들어가게 되고, 이 업데이트 번호를 이용해 업데이트 매니저가 업데이트된 여러 함수를 구분하여 관리한다. 이러한 관리는 업데이트 번호를 인덱스로 하고 함수의 주소와 스플라이싱에 의해 덮어씌워진 8 바이트의 코드를 저장하는 테이블을 커널 내에 두고 이루어진다. 그리고

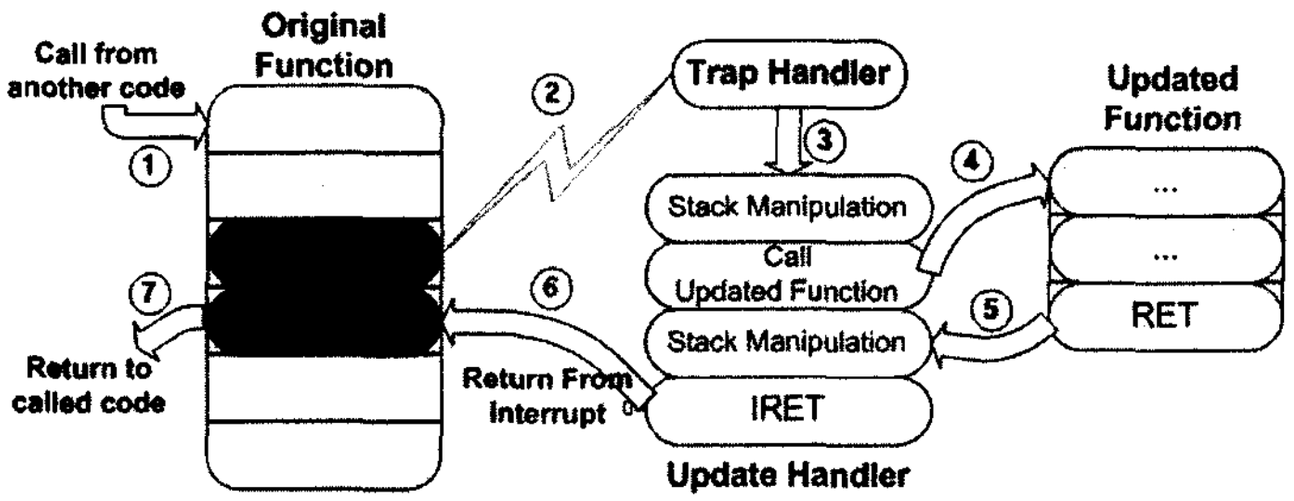

그림 3 업데이트된 함수의 호출 과정

```
0xb8, 0x00, 0x00, 0x00, 0x00    - MOV  $0x0,%EAX
0xcd, 0x14                      - INT  $20
0xc3                            - RET
```

그림 4 코드 스플라이싱 기법에 이용되는 분기 코드

이러한 정보를 이용하여 동적 업데이트 매니저는 업데이트된 함수를 원상태로 복구(update rollback)시킬 수도 있다.

4.2 구현 상의 문제점과 해결 방안

4.1절에서와 같은 트랩 핸들러, 동적 업데이트 매니저 구현을 통해 함수 단위의 동적인 업데이트를 수행하고 관리할 수 있지만, 세부적인 구현상의 문제점들이 존재한다. 우리는 두 가지의 커다란 문제점을 발견하고 이를 해결하였다.

4.2.1 함수 인자 전달을 위한 스택 처리 기법

첫째는 트랩 기반 동적 재구성 기법을 함수 단위 재구성 기법의 구현을 위해 사용할 때 발생하는 함수 인자 전달 문제이다. 우리의 실험 환경인 인텔 구조의 CPU는 트랩이 발생했을 때 수행 흐름을 트랩 인터럽트 서비스 루틴(트랩 핸들러)으로 옮기고 본래 수행 흐름으로 복귀하기 위한 정보- EFLAGS, CS, EIP 레지스터의 데이타와 에러 코드(error code)-를 스택에 저장한다. 이러한 동작이 함수 호출 과정에서 이루어짐으로써 스택에 저장되어 함수로 전달되는 인자들이 해당 정보들이 그 위에 저장됨으로 인해 정확하게 전달되지 않는다. 또 이로 인해 커널은 부정확한 동작을 하게 된다. 이 같은 문제는 새로운 코드를 통해 수행되는 함수의 결과값이 스택에 저장될 때에도 마찬가지로 발생한다.

우리는 인자를 함수로 전달하고 인터럽트 서비스 루틴을 호출하는 동작이 정확히 이루어지도록 하기 위해서 트랩 핸들러 내에서 업데이트된 함수를 호출하기 전에 스택에서 네 개의 4 바이트 데이타를 다른 메모리 공간에 임시로 저장한 후, 함수의 수행이 끝난 이후 다시 스택으로 옮긴다. 단, 스택에 대한 수행이므로 두 동작은 역순으로 이루어져야 한다. 이런 간단한 처리를 통해 인터럽트 서비스 루틴 내에서 정확한 함수 호출을

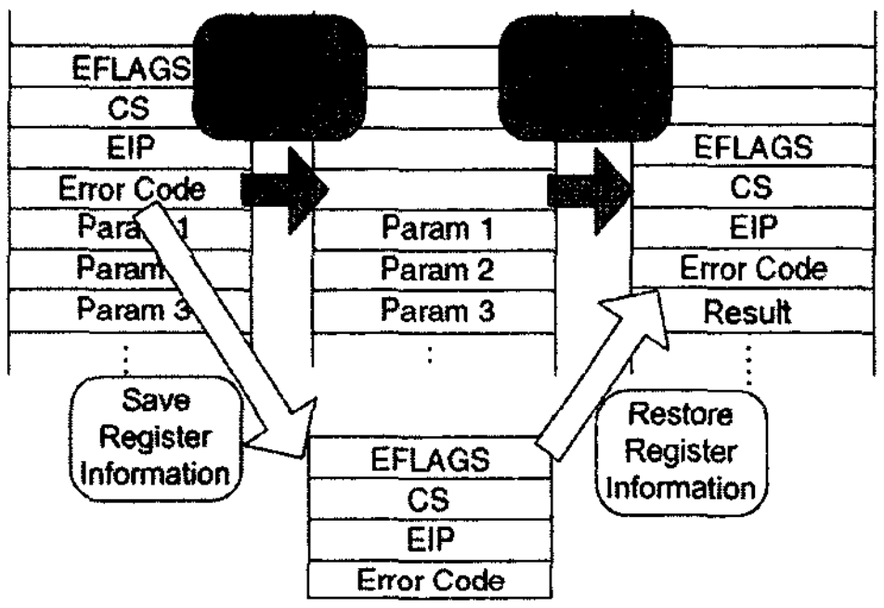

그림 5 커널 스택 내의 함수 인자 전달 과정

수행할 수 있다. 그림 5에서 위와 같은 스택 처리의 전체 과정과 스택의 상태를 그림으로 나타내었다.

4.2.2 커널 내부 심볼에의 접근을 위한 기법

두 번째는 커널 외부에서 커널 함수들을 이용하여 업데이트 패치를 개발하고자 할 때 발생하는 커널 내부 심볼에의 접근 문제이다. 리눅스 커널은 LKM 인터페이스의 활용을 위해 커널 내부 함수 중 일부를 동적으로 접근 가능하게 노출(export)시킨다. 이를 통해 커널 모듈 내에서 해당 함수의 호출이 가능하다. 그러나 노출된 함수의 개수와 종류가 극히 제한적이기 때문에 함수 단위 동적 업데이트 시스템을 이용한다 해도 이러한 제약 때문에 커널의 확장성을 실질적으로 넓히는 데 한계가 생긴다.

이 문제를 해결하기 위해 우리는 LKM 시스템에 커널 함수에 대한 무제한적인 접근을 허용하는 다른 인터페이스를 추가하였다. sys_update_module() 시스템 콜 형태로 제공되는 이 인터페이스는 기존에 LKM 시스템을 위해 존재하는 sys_init_module() 시스템 콜(system call)을 확장한 것이다. 이 시스템 콜은 커널 모듈을 커널에 설치하는 동작을 수행하는데, 커널 모듈이 커널 내의 함수를 동적으로 호출할 수 있도록 함수의 주소를 반환해주는 함수를 내부에서 호출한다. 이때 사용되는 함수는 본래 외부로 노출된 함수만을 접근할 수 있게 제한되어있지만 우리는 여기서 kallsyms_lookup_name()

함수를 사용함으로써 모든 함수 심볼에 접근할 수 있도록 수정하였다.

이러한 자유로운 접근 허용은 커널의 안정적인 동작을 침해할 가능성이 존재한다. LKM 시스템은 본래 이러한 보안 문제 해결을 위해서 이용 권한을 관리자 계정에만 부여하는 등 여러 기법을 사용하고 있지만 보다 강력하게 커널의 안전을 지킬 수 있는 기법이 부가적으로 필요할 것으로 생각된다. 예를 들면 [5]에서 SFI[7](software fault isolation) 기법 등이 동적인 커널 업데이트 기법의 보안을 위해 사용될 수 있음을 언급하고 있다. 이러한 기법들을 적절히 사용함으로써 우리는 동적인 커널 확장성과 동시에 커널의 무결성(integrity)을 얻을 수 있을 것이다.

5. 실제 사용 및 평가

완성된 시스템을 실제로 사용하고 평가하기 위해 우리는 EXT3 파일 시스템을 업데이트하는 간단한 실험을 수행하였다. 그 과정을 단계별로 기술하도록 하겠다.

5.1 EXT3 파일 시스템에서의 실제 사용

① 업데이트 함수 작성

아래와 같이 업데이트 함수를 작성한다.

```
원본 함수 - int ext3_mark_inode_dirty()
업데이트 함수 - int new_ext3_mark_inode_dirty()
내용 - Inode 번호, 파일 크기 등의 디버그 메시지(debug
       message) 출력
```

② 모듈에 추가

업데이트용 스켈레톤 모듈 소스 파일에 업데이트할 내용을 기술한다. 이 스켈레톤 파일은 개발자가 업데이트를 편하게 수행할 수 있도록 모듈을 통한 업데이트 요청을 미리 형식에 맞춰 제공되는 것이다. ①의 업데이트 요청을 포함하는 업데이트 모듈은 아래와 같다. register_update_func() 함수가 지정된 함수의 업데이트를 수행하고, cleanup_update_ func()가 업데이트를 취소하고 원상태로 복구시킨다. 아래 내용과 함께 new_ext3_mark_inode_dirty() 함수의 내용이 기술되어야 한다.

```
#define UPDATE_NUMBER 1
int init_module(void) {
register_update_func(UPDATE_NUMBER, (void_func_t*)
new_ext3_mark_inode_dirty, (unsigned int)
ext3_mark_inode_dirty);
return 0;
}
void cleanup_module(void) {
cleanup_update_func(UPDATE_NUMBER);
}
```

③ 동작 중인 커널에 설치

②에서 작성된 모듈 소스를 컴파일하면 리눅스 모듈 오브젝트가 만들어지고, Insmod 유틸리티를 확장해서 만든 module_updater 유틸리티를 수행하면 해당 업데이트가 커널에 설치된다. 이 유틸리티는 Insmod와 같이 리눅스 모듈 오브젝트를 커널에 설치하는 동작을 수행하되, 4.2.2절에서 언급한 sys_update_module() 함수를 이용하여 커널 내 모든 심볼에 접근할 수 있도록 하는 유틸리티이다.

④ 동작의 검증

위와 같은 수행을 통해 동적인 업데이트가 완료되었으므로 업데이트된 함수가 정확히 수행되는지 검증한다. 현재 업데이트된 함수는 파일 시스템의 특정 Inode의 내용이 변경되었을 때마다 호출된다. 따라서 파일 시스템의 내용을 변경함으로써 업데이트의 결과를 알 수 있다. 그림 6에서 그 결과를 확인할 수 있다. 업데이트 모듈이 커널에 설치되고, 출력하도록 한 디버그 메시지가 문제없이 출력되는 것을 알 수 있다.

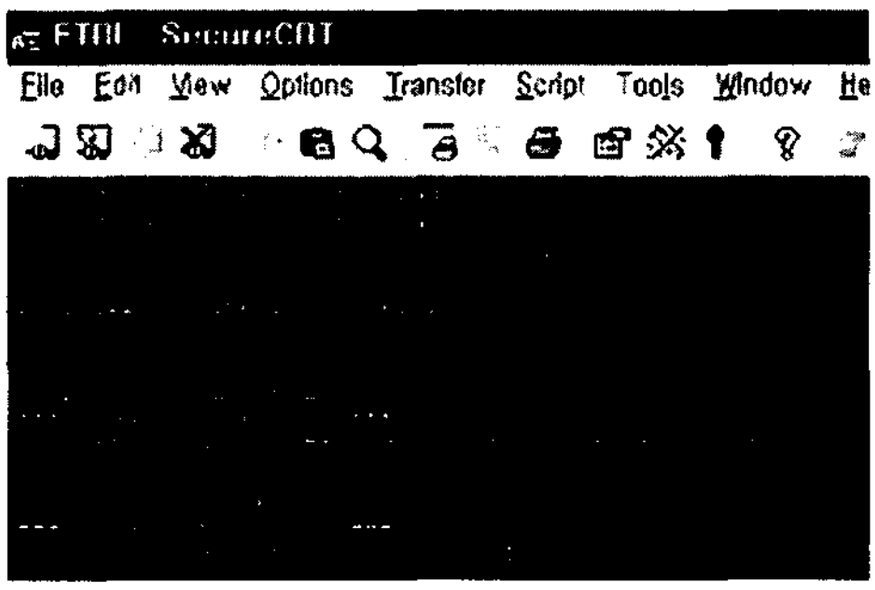

그림 6 업데이트 수행 결과

5.2 평가

함수 단위의 동적 업데이트 시스템을 이용한 업데이트 과정의 편의성은 같은 내용의 업데이트를 기존의 방식을 이용해 수행했을 경우와 비교하여 크게 두 가지 장점을 가진다.

첫째로 수정하고자 하는 함수만을 포함한 바이너리 이미지만으로도 실행중인 EXT3 파일 시스템을 업데이트 할 수 있다. 리눅스에서 커널의 동적인 확장을 위해 제공하는 LKM 시스템은 파일 시스템 전체를 커널 모듈의 형태로 커널에 로드 할 수 있게 한다. LKM 시스템을 이용하여 위와 같이 EXT3 파일 시스템의 일부 기능을 업데이트하기 위해서는 수정된 기능이 포함되어 컴파일된 전체 EXT3 파일 시스템 모듈을 필요로 하게 된다.

둘째로 커널 메모리에 이미 로딩된 함수를 직접 수정하기 때문에 파일 시스템을 언마운트 하는 등의 시스템

서비스 가용성을 저해하는 작업이 필요치 않다. LKM 시스템을 이용해 업데이트하는 경우, 만일 업데이트하고자 하는 EXT3 파일 시스템이 실행중인 커널에 의해 사용되고 있는 경우라면, 우선 EXT3 파일 시스템이 마운트(mount)된 디스크를 언마운트(unmount)한 후에 새로운 EXT3 파일 시스템 모듈을 적재하고 다시 해당 디스크에 마운트해야 한다. 결국 업데이트가 이루어지는 동안은 해당 디스크에 대한 파일 시스템 서비스를 이용할 수 없다. 이는 커널의 동적인 확장 가능성과 직접 연관되는 사항으로, 커널은 동작을 멈추지 않았지만 업데이트를 수행하고자 하는 대상 서브 시스템은 실질적으로 동작을 멈추었기 때문에 동적인 업데이트의 특성이 충족되었다고 보기는 어렵다.

이러한 업데이트 과정의 편의성과 동적인 업데이트의 특성을 만족시키는 장점 외에도 업데이트가 수행되는 속도가 빨라지고 업데이트에 필요한 모듈의 사이즈도 줄어드는 부가적인 장점이 있다. 또 개발자가 업데이트를 위한 작업을 보다 편하게 수행할 수 있는 등의 장점 또한 존재한다.

5.3 성능에 관한 고려

마지막으로 성능에 대해 언급하자면, 본 기법의 경우 기존 방식들[1,4]과 달리 분기 명령어로서 JMP 명령어를 이용하지 않고 트랩을 사용하였다. 따라서 성능 상으로는 JMP 명령어를 이용한 방식보다 다소 느린 결과를 보일 수 있다. 이에 관해서는 [6]에서 이미 상세히 분석된 바 있는데, 명령어 수행에 소요되는 시간은 JMP 명령에 비해 트랩이 19배 가량 느린 결과를 보이고 있다. 그러나 JMP는 명령어 길이가 가변적인 시스템 환경에서는 최소한도의 크기를 갖는 JMP를 사용할 수 밖에 없고, 이 때문에 원하는 커널 주소에 도달하기 위해 여러 단계를 거쳐야 하는 구현 상의 문제가 발생한다. [6]에서는 이러한 경우를 모두 고려하면 트랩을 이용하는 것이 성능에서도 이득을 얻을 수 있다는 결론을 내리고 있고, 본 논문에서는 이러한 점과 더불어 2.2절에서 언급한 장점들을 높이 평가하여 트랩 기반의 기법을 사용하였다. 현 시점에서 기법에 관한 단순한 성능을 측정하여 평가한다면 JMP 명령어를 사용한 경우보다 낮은 결과를 보이겠지만, 실제 기법이 적용되어 사용되는 경우에는 성능 평가 시에도 상황에 따라 비슷한 결과나 오히려 우위를 보일 수 있다고 기대한다. 만약 더 낮다고 하더라도 [6]에서 측정한 결과로는 약 1500 사이클 가량의 차이를 보이므로 실제 수행시간으로는 0.5 마이크로초(3 GHz CPU인 경우) 정도로 생각할 수 있기에, 실제 성능에 큰 영향을 미칠 정도는 아니다. 이러한 성능에 관한 부분은 앞으로 더 연구를 진행하며 종합적으로 평가해볼 계획이다.

6. 결론 및 이후의 진행 방향

본 논문을 통해 우리는 동적인 커널 업데이트를 함수 단위 업데이트로 수행할 수 있는 시스템을 설계하고 구현하였다. 이 시스템은 C 언어 레벨에서 함수 단위로 작업할 수 있는 편리한 환경을 제공하고, 이를 통해 기존의 연구들이 어셈블리 언어 레벨에서 바이트 코드 단위로 작업하던 한계를 극복하여 동적인 커널의 확장성이 실제로 적용될 수 있도록 하였다. 이를 위해 세부적으로 트랩 기반의 동적 커널 재구성 기법을 사용하였고 실제 구현 단계에서는 인터럽트 서비스 루틴 내로 함수의 인자를 전달하기 위한 기법, 노출되지 않은 커널 함수 모두에 접근할 수 있는 기법을 개발하였다. 또 제안한 시스템을 사용하여 간단한 실험을 통해 업데이트 과정이 실제 동적으로 이루어지는 것을 검증하고 수행 과정의 편의성이 증대되었음을 보였다.

우리는 현재 3장에서 설명한 여러 컴포넌트 중 시스템 업데이트 모듈과 동적 업데이트 매니저가 구현하였으며 이를 통해 핵심적인 기법은 완성된 상태이다. 그러나 업데이트 메인테이너가 구현되어야 업데이트가 지속적으로 유지될 수 있고, 무결성 관리자가 구현되어야 커널의 정상적인 동작을 보장한 상태에서 동적인 업데이트 기능을 활용할 수 있다. 부가적인 기능이지만 실제 시스템에서의 사용을 목표로 한다면 둘 모두 반드시 필요한 기능이다. 이후에는 두 컴포넌트의 기능을 실현하기 위한 연구와 구현을 진행할 계획이다. 그리고 현 단계에서 부분적인 성능을 평가하기보다 실제 기법이 커널에 적용되어 활용될 때의 종합적인 성능에 대한 평가를 수행하여야 한다.

참 고 문 헌

[1] Ariel Tamches and Barton P.Miller, Fine-grained dynamic instrumentation of commodity operating system kernels, PhD thesis, University of Wisconsin, 2001.

[2] B. Bershad et al., Extensibility, Safety and Performance in the SPIN Operating System, In Proceedings of the 15th ACM SOSP, pp. 267-284, 1995.

[3] Andrew Baumann et al., Module Hot-Swapping for Dynamic Update and Reconfiguration in K42, LCA 2005.

[4] D. J. Pearce, P. H. J. Kelly, T. Field, and U. Harder., GILK: A dynamic instrumentation tool for the linux kernel, In Computer Performance Evaluation, pp. 220-226, 2002.

[5] Douglas P. Ghormley et al., SLIC: An extensibility system for commodity operating systems, In USE-

NIX Annual Technical Conference, pp. 39-52, 1998.
[6] Young-Pil Kim, Jin-Hee Choi and Chuck Yoo, A Trap-based Mechanism for Runtime Kernel Modification, 6th International Conference on Computer and Information Technology, 2006.
[7] Robert Wahbe et al., Efficient Software-Based Fault Isolation. In Proceedings of the 14th ACM SOSP, pp. 203-216, 1993.

박 현 찬

2004년 고려대 컴퓨터학과 학사. 현재, 고려대학교 대학원 컴퓨터학과 석박 통합 과정. 관심분야는 운영체제, 임베디드 소프트웨어, 파일 시스템.

김 세 원

2004년 고려대 컴퓨터학과 학사. 2006년 고려대학교 대학원 컴퓨터학과 석사. 현재, 동대학원 박사 과정. 관심분야는 운영체제, 임베디드 소프트웨어, Temperature-aware OS.

유 혁

1982년 고려대학교 전자공학과 학사. 1984년 서울대학교 전자공학과 석사. 1986년 University of Michigan 전산학 석사 1990년 University of Michigan 전산학 박사. 1990년~1995년 Sun Microsystems Lab. 1995년~현재 고려대학교 정보통신대학 컴퓨터학과 교수. 관심분야는 운영체제, 멀티미디어, 네트워크