리눅스 커널을 위한 함수 단위 업데이트 성능 개선 기법

임병홍*, 김인혁**, 엄영익**
*성균관대학교 임베디드소프트웨어학과
**성균관대학교 전자전기컴퓨터공학과
e-mail : { dd8562, kkojiband, yieom } @ece.skku.ac.kr

Improving Function-level Update Performance For Linux Kernel

Byoung-Hong Lim*, In-Hyuk Kim**, Young-Ik Eom**
*Dept. of Embedded software, Sungkyunkwan University
**Dept. of Electrical and Computer Engineering, Sungkyunkwan University

요 약

기존의 동적 커널 업데이트 시스템에서 주로 사용되는 함수 단위의 재구성 기법으로는 트랩과 점프가 있다. 이러한 기법들을 사용하면 커널 서비스의 중단 없이 함수 단위로 커널을 업데이트할 수 있는 이점이 있다. 하지만 커널 업데이트 후, 프로세서가 분기 명령어를 처리하는 과정에 두 가지 문제점이 존재한다. 업데이트 함수에 업데이트가 필요한 함수 내의 분기 명령어 오퍼랜드 값을 그대로 복사하면 의미 없는 메모리 주소로 분기하게 된다. 또한 분기 명령어로 short jump 를 사용하면, 현재 위치에서 8 비트 범위를 벗어난 주소공간에 존재하는 분기 함수에는 접근을 할 수 없는 문제를 안고 있다. 본 논문에서는 이러한 문제점들을 해결하기 위해 short jump 대신 long jump 를 사용하는 방식을 제안하였다. 이를 위해 업데이트가 필요한 함수의 분기 명령어가 갖고 있는 오퍼랜드 값을 추출하여, 업데이트 함수의 분기 명령어가 정상적으로 동작할 수 있도록 오퍼랜드 값을 수정해주는 동적 커널 업데이트 시스템을 설계하고 구현하였다.

1. 서론

최근 리눅스 시스템에는 기능 향상과 취약한 보안의 개선이 절실히 요구되고 있는 실정이다. 이러한 상황적 요구로 기능과 보안의 개선을 위한 패치 파일들이 자주 배포되고 있다. 하지만 업데이트를 하기 위해서는 커널 서비스의 중단, 컴파일, 재시작 등의 번거로운 작업이 필요하다. 실제로 약간의 서비스 개선을 위해 요구되는 커널 업데이트는 번거로운 작업들로 인하여 기피되고 있다.

이러한 문제를 해결하기 위해 커널 서비스를 중단하지 않고 기능을 확장할 수 있는 여러 연구들이 진행되고 있다[1,2]. 그 가운데 하나인 함수 단위의 동적 커널 업데이트 시스템은 고수준 언어를 사용하여 함수 코드를 수정하며, 커널의 서비스를 중단 하지 않고 수정된 함수를 업데이트할 수 있으므로 실제 업데이트 수행의 편의성을 증대한다[4].

그러나 이러한 함수 단위의 동적 커널 업데이트 시스템은 두 가지 문제점을 갖고 있다. 첫 번째 문제점은 short jump 명령어를 사용할 때, 분기 할 수 있는 주소 공간 범위의 한계에서 나타난다. 일반적으로 커널은 빌드된 이미지를 가지고 있다. 이러한 이미지 안에서 사용되는 몇몇 분기 명령어는 short jump 명령어를 사용한다. 그리고 short jump 명령어 뒤에는 부호를 가지는 8 비트의 상대 주소가 오퍼랜드로 사용된다. 부호를 가지는 8 비트 값의 범위는 -128(80h) ~ +127(7Fh)이다. 그러므로 short jump 명령어를 사용하면 현재 분기 명령어의 주소 값을 기준으로 -128 번지부터 +127 번지 사이 내에 위치한 분기 함수에만 접근을 할 수 있다. 따라서 업데이트 함수에서 요구되는 분기 함수의 위치가 현재 분기 명령어를 기준으로 8 비트 값의 범위를 벗어난다면 분기함수로 접근을 할 수 없게 된다.

두 번째 문제점은 분기 명령어의 오퍼랜드 값은 분기 함수를 호출하기 위한 상대주소 값이라는 점에서 나타난다. 정확한 분기 함수를 호출하기 위해서는 현재 분기 명령어가 위치한 주소 값과 주어진 상대주소 값이 더해져야 한다. 따라서 기존의 분기 명령어 오퍼랜드 값을 업데이트 함수의 분기 명령어 오퍼랜드 값으로 그대로 복사하더라도, 기존의 분기 명령어가 위치한 주소 값과 업데이트 함수의 분기 명령어가 위치한 주소 값이 달라서 요청된 분기 함수가 아닌 의미 없는 메모리 주소로 분기하게 되는 오류가 발생한다.

본 논문은 다음과 같이 구성된다. 2 장에서는 관련 연구를 소개하고, 3 장에서는 동적 커널 업데이트 시스템의 컴포넌트를 설계한다. 4 장에서는 위에서 제시

된 문제점들을 해결하기 위한 기법의 동작을 설명하고, 5 장에서는 구현된 시스템의 성능을 평가한다. 6 장에서는 결론을 제시한다.

2. 관련연구

동작 중인 커널의 중단 없이 동적인 업데이트를 하기 위해 주로 사용되는 함수 단위의 동적 커널 재구성 기법으로는 트랩(Trap) 기반[3]과 점프(Jump) 기반[4,5]이 있다. 두 동적 커널 재구성 기법은 업데이트가 필요한 함수의 기계 코드 명령어를 업데이트 함수 영역으로 분기할 수 있는 분기 명령어로 덮어쓰고, 업데이트 함수 영역 끝에서 본래의 수행 흐름으로 복귀하는 방식을 사용한다. 이러한 방식은 분기 명령어로의 간단한 수정으로 업데이트가 가능한 장점이 있다. 또한 업데이트 함수 코드의 크기에 대해 고려할 필요 없다. 반면에 분기에 의한 성능 저하와 분기 명령어의 종류에 따른 메모리 영역 접근에 제한이 생긴다는 단점이 있다.

이번 장에서는 트랩과 점프를 기반으로 하는 동적 커널 재구성 기법의 수행 내용을 알아보고 두 기법을 비교한다.

2.1 트랩 기반 동적 커널 재구성 기법의 수행 내용

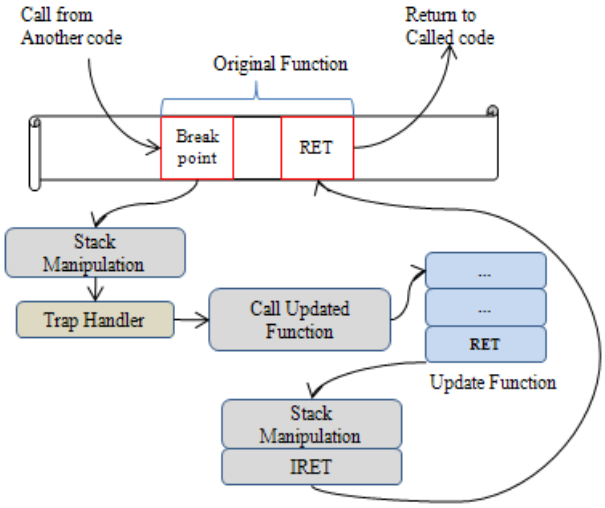
위에서 간단하게 소개된 동적 커널 재구성 기법에서, 트랩 기반 동적 커널 재구성 기법의 수행 내용은 다음과 같다. 먼저 업데이트가 필요한 함수가 호출된다. 업데이트가 필요한 함수의 처음 코드 부분에는 업데이트 함수로 분기를 위한 트랩 명령어가 있다. 트랩이 발생되면 트랩 핸들러가 호출된다. 트랩 핸들러가 호출 되기 전에 스택에는 트랩 핸들러를 수행한 후 돌아올 현재 위치의 값, 에러코드와 레지스터 등의 데이터가 저장된다. 수행된 트랩 핸들러는 업데이트 함수를 호출한다. 이 때도 핸들러 내 수행되던 현재 위치의 값은 스택에 저장된다. 업데이트 함수를 수행한 후, 이전에 스택에 저장된 모든 복귀 값과 데이터들을 읽으면서 트랩 핸들러부터 본래의 수행되던 코드까지 차례대로 복귀된다.

그림 1 은 트랩 기반 동적 커널 재구성 기법의 동작 순서를 보여준다.

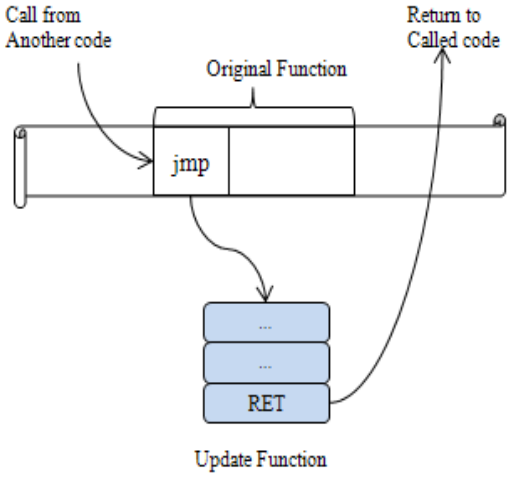
2.2 점프 기반 동적 커널 재구성 기법의 수행 내용

점프 기반 동적 커널 재구성 기법의 수행 내용은 다음과 같다. 먼저 업데이트가 필요한 함수가 호출된다. 업데이트가 필요한 함수의 처음 코드 부분에는 업데이트 함수로 분기를 위한 점프 명령어가 있다. 점프가 발생되면 트랩과는 다르게 바로 업데이트 함수가 호출된다. 업데이트 함수를 수행한 후, 본래의 수행되던 코드로 복귀한다.

그림 2 은 점프 기반 동적 커널 재구성 기법의 동작을 나타낸 것이다.



(그림 1) 트랩 기반 커널 동적 재구성 과정



(그림 2) 점프 기반 커널 동적 재구성 과정

2.3 트랩과 점프 기반의 동적 커널 재구성 기법 비교

위에서 설명한 두 동적 커널 재구성 기법의 수행 과정은 다음과 같은 차이점을 보인다. 트랩 기반 동적 커널 재구성 기법은 업데이트 함수를 실행하기 위해서 트랩에서 업데이트 함수까지 여러 번의 분기 호출과 복귀를 한다. 게다가 레지스터와 복귀 값 등 많은 데이터를 스택에 쓰고 읽기를 반복한다. 하지만 점프 기반 동적 커널 재구성 기법은 점프 명령어를 통한 한번의 호출과 스택을 사용한다. 위에서 볼 수 있듯이 트랩 기반은 점프 기반의 동적 재구성 기법보다 오버헤드가 크다.

5 장에서는 구현된 동적 커널 업데이트 시스템에서 트랩과 점프 기반의 동적 커널 재구성 기법의 오버헤드 차이를 알아본다.

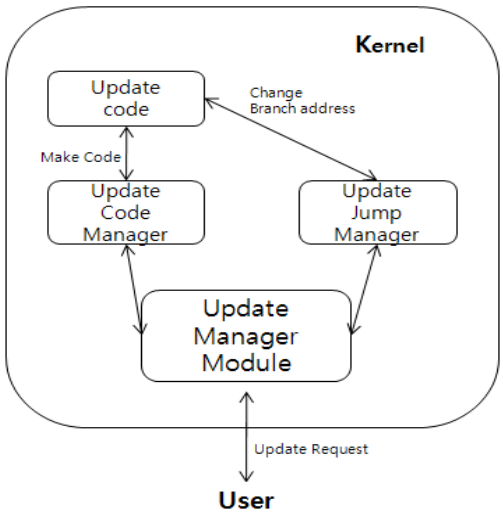
3. 동적 커널 업데이트 시스템 설계

3.1 시스템 컴포넌트

함수 단위의 동적 커널 업데이트 시스템에는 동적 커널 재구성 기법을 수행하기 위한 컴포넌트들이 포함된다. 아래에는 각각의 컴포넌트들 역할을 기술한다.

그림 3 은 동적 커널 업데이트 시스템의 전체 구조와 컴포넌트 상호간의 동작을 나타낸다.

· 업데이트 매니저 모듈(Update Manager Module)
  업데이트 매니저 모듈은 커널의 메모리 영역에서 모듈로 수행되며, 유저 영역에서의 요청을 받아들인다. 그리고 업데이트 코드 매니저와 업데이트 점프 매니저를 순차적으로 수행하여 업데이트를 전체적으로 관리한다.
· 업데이트 코드 매니저(Update Code Manager)
  업데이트 함수를 커널의 메모리 영역에 적재한다. 그리고 업데이트가 필요한 함수에서 업데이트 함수로 점프할 수 있도록 업데이트가 필요한 함수의 처음 코드 부분을 트랩 명령어 또는 점프 명령어와 업데이트 함수의 주소 값으로 수정한다.
· 업데이트 점프 매니저(Update Jump Manager)
  업데이트 코드에서 short jump 를 사용하는 분기 명령어 오퍼랜드를 분기 함수로의 정확한 상대 주소로 수정한다.



(그림 3) 커널의 함수를 업데이트하기 위한 컴포넌트들의 구조

3.2 업데이트 수행 과정

위에서 소개된 컴포넌트들은 다음과 같은 일련의 작업을 수행한다. 유저 영역에서 발생된 업데이트 요청이 업데이트 매니저 모듈로 전달되면, 업데이트 코드 매니저는 업데이트 요청에 해당하는 업데이트 함수를 커널 메모리 영역에 적재한다. 그리고 업데이트 점프 매니저는 적재된 업데이트 코드에서 short jump 를 사용하는 분기 명령어와 오퍼랜드를 분기 함수로의 정확한 상대 주소로 수정한다.

4. 구현

4.1 업데이트 함수를 커널 영역에 할당

설계된 업데이트 매니저 모듈을 커널에 등록하기 위해서 리눅스의 insmod 유틸리티를 사용한다. insmod 유틸리티로 등록된 업데이트 매니저 모듈은 유저 영역에서 업데이트 요청이 오면, 해당 업데이트 함수를 커널 메모리 영역에 적재한다.

4.2 업데이트 함수 내의 short jump 를 사용하는 분기 명령어 오퍼랜드 수정

업데이트가 필요한 함수 내의 short jump 를 사용하는 분기 명령어들은 8 비트 주소를 사용한다. 하지만 업데이트 함수 내의 분기 명령어들은 32 비트 주소의 long jump 를 사용하여, 0000h 부터 0FFFFh 번지까지 제한 없이 점프 할 수 있도록 해야 한다. 이를 위해 gcc 유틸리티의 defsym 옵션으로 업데이트 함수 내의 short jump 를 사용하는 분기 명령어들의 오퍼랜드를 32 비트 주소의 임의의 값으로 만들어준다.

그리고 업데이트가 필요한 함수 내 short jump 를 사용하는 분기 명령어들의 오퍼랜드 값을 추출한다. 추출된 오퍼랜드 값과 업데이트 함수 내에 있는 분기 명령어의 메모리 영역에 위치한 값을 계산하여, 업데이트 함수의 분기 명령어 오퍼랜드를 수정한다

그림 4 은 업데이트가 필요한 함수의 8 비트 주소에 해당하는 short jump 를 사용하는 분기 명령어 코드를 보여준다. 그림 5 은 업데이트 함수에 defsym 옵션을 적용하여 수정한 임의의 32 비트 주소에 해당하는 코드를 보여준다.

```
c014d690 <sys_newcall>:
c014d690:       55                      push   %ebp
c014d691:       89 e5                   mov    %esp,%ebp
c014d693:       83 ec 04                sub    $0x4,%esp
c014d696:       e8 2d 75 fb ff          call   c0104bc8 <mcount>
c014d69b:       eb 3d                   jmp    c014d6da
c014d69d:       8b 45 08                mov    0x8(%ebp),%eax
c014d6a0:       03 45 0c                add    0xc(%ebp),%eax
c014d6a3:       03 45 10                add    0x10(%ebp),%eax
c014d6a6:       c9                      leave
c014d6a7:       c3                      ret
```

(그림 4) 업데이트가 필요한 함수의 short jump 를 사용하는 분기 명령어 코드 부분

```
c014d690 <sys_newcall>:
c014d690:       55                      push   %ebp
c014d691:       89 e5                   mov    %esp,%ebp
c014d693:       83 ec 04                sub    $0x4,%esp
c014d696:       e8 2c 75 fb ff          call   c0104bc8 <mcount>
c014d69b:       e9 00 00 00 00          jmp    c014d6a0
c014d6a0:       8b 5d 0c                mov    0xc(%ebp),%ebx
c014d6a3:       0f af 5d 08             imul   0x8(%ebp),%ebx
c014d6a7:       0f af 5d 10             imul   0x10(%ebp),%ebx
c014d6ab:       83 c4 04                add    $0x4,%esp
c014d6ae:       89 d8                   mov    %ebx,%eax
c014d6b0:       5b                      pop    %ebx
c014d6b1:       5d                      pop    %ebp
c014d6b2:       c3                      ret
```

(그림 5) 업데이트 함수에서 defsym 옵션을 적용하여 long jump 를 사용하는 분기 명령어 코드부분

4.3 업데이트 필요한 함수 코드 수정

업데이트가 필요한 함수대신 업데이트 함수가 수행되도록, 업데이트가 필요한 함수의 처음 코드 부분을 트랩 명령어 또는 점프 명령어와 업데이트 함수의 주소 값으로 덮어쓴다.

5. 실제 사용 및 평가

5.1 리눅스 시스템에서의 실제 사용

함수 단위의 동적 커널 업데이트 시스템은 인텔 플랫폼 기반의 리눅스 커널 2.6.27.10 버전에서 구현되었다. 구현된 시스템을 실험하고 평가하기 위해서 새로운 시스템 콜을 추가하였다. 그리고 새롭게 추가된 시스템 콜 핸들러의 처리 함수를 트랩 또는 점프 기반의 동적 커널 재구성 기법을 사용하여 함수를 업데이트하는 실험을 수행 하였다.

실험은 다음과 같은 일련의 과정에 따라 수행되었다. 우선 업데이트 매니저 모듈 소스를 컴파일하여 모듈 오브젝트를 생성한다. 생성된 모듈 오브젝트를 리눅스의 insmod 유틸리티를 이용하여 커널의 메모리 영역에 적재한다. 이렇게 적재된 업데이트 매니저 모듈은 업데이트 함수를 커널 메모리 영역에 적재한다.

이러한 과정을 거쳐 적재된 업데이트 함수에서 short jump 를 사용하는 분기 명령어들이 사용될 경우, 해당 명령어의 오퍼랜드를 gcc 유틸리티의 defsym 옵션으로 32 비트의 임의의 값으로 만들어준다. 그리고 업데이트 함수의 분기 명령어가 정상적으로 수행될 수 있도록, 업데이트가 필요한 함수 코드에서 short jump 를 사용하는 분기 명령어 오퍼랜드 값을 추출한다. 추출된 오퍼랜드 값과 업데이트 함수 내에 있는 분기 명령어가 위치한 주소 값을 계산하여 업데이트 함수의 분기 명령어 오퍼랜드 값을 수정한다. 마지막으로 업데이트가 필요한 함수에서 트랩이나 점프가 발생되어 업데이트 함수가 실행될 수 있도록, 업데이트가 필요한 함수의 처음 코드 부분을 트랩이나 점프 명령어와 업데이트 함수의 주소 값으로 덮어쓴다. 새로 등록된 시스템 콜을 호출하여 업데이트 함수가 정상적으로 동작하는지 확인한다.

5.2 평가

위의 실험을 통하여 구현한 시스템이 업데이트 함수 내의 분기 명령어를 long jump 로 변경하고 오퍼랜드의 값을 수정하여, 정상적으로 업데이트를 수행함을 볼 수 있었다. 또한 시스템에서 트랩과 점프 기반의 동적 커널 재구성 기법을 각각 실행해보며 동적 업데이트에 걸린 시간을 측정해 보았다.

그 결과로 업데이트에 소요된 시간이 트랩 기반 업데이트에서는 11046 nsec 였으며, 점프 기반 업데이트 수행에서는 3108 nsec 였다. 트랩보다 점프 기반의 업데이트가 3 배 이상의 빠른 처리 속도를 보였다. 따라서 트랩보다 점프를 기반으로 하여 업데이트를 수행하는 것이 더 적은 오버헤드를 가진다고 볼 수 있다.

6. 결론

본 논문에서는 기존의 함수 단위 동적 업데이트 시스템의 문제점을 해결할 수 있는 시스템을 설계하고 구현하였다. 기존의 문제점을 해결하기 위한 short jump 명령어에서 long jump 명령어로의 변경과 오퍼랜드 값의 수정에 대한 설계 및 이에 필요한 기술들을 설명하였다. 이를 통해 업데이트 함수의 분기 명령어까지 정확히 실행할 수 있게 되었다. 또한 완성된 시스템에서 트랩과 점프 기반의 동적 커널 재구성 기법을 모두 수행해 봄으로써 두 기반 모두 업데이트가 성공적으로 실행됨을 확인할 수 있었다. 게다가 점프를 기반으로 한 동적 커널 재구성 방식이 트랩 기반보다 성능 상으로 더 우세함을 알 수 있었다.

7. 참고 문헌

[1] A. Tamches and B. P. Miller, ”Fine-Grained Dynamic Instrumentation of Commodity Operating System Kernel,” Proc. of the third symposium on OSDI, 1999.

[2] D. J. Pearce, P. H. J Kelly, T. Field and U Harder,, “Gilk : A dynamic instrumentation tool for the linux kernel,” Proc. of the 12th International Conf. on Modeling Tools and Techniques for Computer and Communication System Performance Evaluation(TOOLS 2002), 2002

[3] Y. Kim, J. Choi and C. Yoo, “A Trap-based Mechanism for Runtime Kernel Modification,” Proc. of 6th International Conf. on Computer and Information Technology, 2006

[4] M. Hiramatsu and S. Oshima, “Djprobe - Kernel probing with the smallest overhead,” Proc. of the Ottawa Linux Symposium, 2006

[5] H. Chen, R. Chen, F. Zhang, B. Zang and P. C. Yew, “Live Updating Operating Systems Using Virtualization,” Proc. of the 2nd international conference on VEE, 2006