Journal of The Korea Society of Computer and Information
Vol. 18, No. 4, April 2013
www.ksci.re.kr
http://dx.doi.org/10.9708/jksci.2013.18.4.001

C 응용 프로그램의 동적 소프트웨어 업데이트 시스템 개발

신 동 하*, 김 지 현**

An Implementation of Dynamic Software Update System for C Application Programs

Dongha Shin*, Ji-Hyeon Kim**

요 약

DSU(Dynamic Software Update)는 실행 중인 프로세스를 종료하지 않고 새 버전으로 업데이트하는 기술이다. 이 기술을 이용하여 C 응용 프로그램을 업데이트 하는 DSU 시스템들이 소개 되었으며, 각 시스템의 업데이트 방식 및 주요 기능에는 큰 차이가 있다. 본 논문에서는 기존 DSU 시스템의 단점을 해결할 수 있는 새로운 DSU 시스템을 제안한다. 이 시스템은 C 응용 프로그램을 코드, 전역 데이터 및 지역 데이터로 나누어 이들 각 부분의 특성을 고려하여 업데이트 한다. 이 논문에서 제안한 방법은 리눅스 운영체제 상에서 구현 및 시험하였으며 기존 DSU 시스템과 비교하여 다음과 같은 장점을 가진다. 첫째, 구 버전의 코드는 메모리에서 해제되므로 코드 메모리의 낭비가 적다. 둘째, 새 버전에서 수정되지 않은 전역 데이터는 메모리에 새로 할당할 필요가 없으므로 전역 데이터 메모리의 낭비가 적다. 셋째, 업데이트 시 구 버전의 지역 데이터는 스택 재구성 방식을 사용하여 새 버전의 지역 데이터로 복구한다. 본 논문은 새로운 DSU 방식을 제안하였다는 점과 이 방식을 활용하여 완전한 DSU 시스템을 구현하였다는 점에서 의의가 있다.

▸ Keywords : 동적 소프트웨어 업데이트, C 응용 프로그램, 가상 메모리, 소프트웨어 업데이트

Abstract

Dynamic Software Update(DSU) is a technique, which updates a new version of the software to a running process without stopping. Many DSU systems that update C application programs are introduced. However, these DSU systems differ in implementation method or in main features. In this paper, we propose a new DSU system that can solve some disadvantages of existing DSU systems. DSU system presented in this paper splits existing program to code, global data and local

•제1저자 : 신동하 •교신저자 : 신동하
•투고일 : 2013. 01. 31, 심사일 : 2013. 03. 11, 게재확정일 : 2013. 03. 27.
* 상명대학교 컴퓨터과학부 교수(Division of Computer Science, Sangmyung University)
** 상명대학교 일반대학원 컴퓨터과학과(Department of Computer Science, the Graduate School)
※ 본 논문은 2012년 상명대학교 교내연구비에 의하여 지원되었습니다.

data and then updates each part of the program considering the characteristics of the respective parts. The proposed system in this paper is implemented and tested on Linux. Also, we compared our DSU system with other DSU systems and we could find some strength of our DSU system. First, the code memory usage of our DSU system can be efficient since our system does not need to maintain code of an old version. Second, the global data memory waste is small because our system does not need to allocate the global data again which is not modified in the new version. Finally, we restore local data of old version in stack area of the new version using stack reconstruction technique. This paper is meaningful since we proposed a new DSU method and we implemented a full DSU system using the method.

▸ Keywords : Dynamic Software Update, C Application Program, Virtual Memory, Software Update

I. 서 론

일반적으로 소프트웨어를 업데이트하기 위해서는 실행 중인 프로세스를 종료하고 새 버전을 설치한 후 프로세스를 다시 수행한다. 동적 소프트웨어 업데이트(DSU: Dynamic Software Update)는 실행 중인 프로세스를 종료하지 않고 새 버전으로 업데이트하는 기술이다[1]. 이 기술은 중단 없이 수행하여야 하는 응용(application)에 매우 유용하다[2]. 이 기술을 이용하여 서비스의 상시성이 요구되는 서버 환경에 시스템의 종료(shutdown) 없이 서비스를 제공할 수 있으며, 현재 실행중인 프로세스의 상태(state)를 유지할 수 있으므로 프로그램 종료로 인한 손실 또는 유지비용을 절감할 수 있다.

기존에 연구된 DSU 시스템은 업데이트 이후에도 구 버전의 코드를 메모리 상에 유지하거나[3][4], 모든 전역 데이터를 힙 영역에 유지하므로[5][6] 메모리의 낭비가 발생한다. 또한 업데이트 할 필요가 없는 수정되지 않은 전역 데이터도 업데이트 시 마다 다시 업데이트 하여야 하는 문제점이 있다[7][8]. 지역 데이터의 경우 지역 데이터를 업데이트 하지 않거나 특정 데이터 타입의 업데이트가 불가하다는 문제점도 있다.

본 논문은 위와 같은 문제를 해결하기 위하여 새로운 업데이트 방법을 사용하는 DSU 시스템을 소개한다. 이 연구에서 구현한 DSU 시스템은 업데이트 시 각 버전의 프로그램을 코드(code)와 전역 데이터(global data), 지역 데이터(local data)로 나누어 각 부분의 특성을 고려하여 업데이트하며, 기본 아이디어는 다음과 같다. 첫째, 코드는 업데이트 후 가상 메모리 공간(virtual address space)에서 그 위치가 변경되어도 수행 시 문제가 일어나지 않는다. 따라서 본 연구는 프로세스의 공유 메모리 영역(shared memory)에 모든 버전의 코드를 저장한다. 또한 전체 프로그램 업데이트(whole-program update) 방식을 사용하므로, 새 버전(new version)의 코드를 로드 후 구 버전의 코드 영역을 메모리에서 해제(free)하기 용이하다. 둘째, 대부분의 소프트웨어 업데이트에서 전역 데이터는 업데이트 되는 양이 많지 않으므로[9], 부분 업데이트(partial update)를 수행한다. 이를 위하여 최초 버전의 전역 데이터는 가상 메모리 공간의 데이터(data) 영역에 저장하고 이후 버전에서 추가 또는 수정된 데이터는 힙(heap) 영역에 저장하여 관리한다. 셋째, 지역 데이터는 그 데이터가 선언된 함수가 호출되고 반환됨에 따라 생성 및 삭제되므로 스택(stack) 영역에 저장한다. 이 데이터는 새 버전이 수행 된 후 새 버전에 맞게 복구(restore)되어야 하는 데이터이므로, 업데이트 시 구 버전의 스택을 힙 영역에 임시 저장하였다가 저장한 내용을 이용하여 새 버전의 스택을 구성하는 스택 재구성(stack reconstruction) 기술을 이용하여 업데이트한다[5].

본 연구에서 제안한 방식을 사용하는 전체 DSU 시스템은 현재 x86 구조에서 리눅스를 사용하여 개발되어 여러 C 프로그램의 업데이트를 시험하였다.

본 논문의 2장에서는 동적 소프트웨어 업데이트 기술에 대하여 설명하고 기존에 연구된 주요 DSU 시스템을 소개한다. 3장에서는 이 연구에서 구현한 DSU 시스템의 설계 방법을 기술한다. 4장에서는 개발한 각각의 DSU 시스템 도구에 대하여 설명하고, 5장에서는 개발한 DSU 시스템을 이용하여 업데이트를 수행하고 시험한 결과를, 6장에서는 이 DSU 시스템을 다른 DSU 시스템과 비교하여 평가한 내용을 기술한

다. 마지막으로 7장에서는 본 논문의 결론 및 기여 효과에 대하여 기술한다.

II. 관련 연구

1. 주요 용어

1.1. 동적 소프트웨어 업데이트 과정

동적 소프트웨어 업데이트는 현재 프로그램을 실행하는 프로세스를 종료하지 않고 새 버전의 프로그램으로 업데이트하는 기술로, 업데이트 시 다음과 같은 과정을 거친다[6].

1) 구 버전의 프로그램인 Π를 수행중인 프로세스 P가 시간 t에 상태 s인 상태로 잠시 중단(pause)된다.
2) P의 프로그램이 새 버전의 프로그램 Π'로 바뀌고, 상태 맵핑 함수(state mapping function)로 얻은 새 버전의 상태 s′로 프로그램의 수행이 재개(resume)된다.

2)에서 언급한 상태 맵핑 함수란, 구 버전의 프로그램이 중단 되었을 때의 상태 s를 입력받아 새 버전의 프로그램이 재개 하는 상태인 s′로 연결시키는 함수, 즉 S(s)=s′ 인 함수 S를 말한다[6].

1.2. 업데이트 가능한 프로그램

DSU 시스템을 사용하기 위해서는 기존 프로그램을 업데이트 가능한 프로그램(updatable program)으로 변환하여야 하는데, 업데이트 가능한 프로그램은 기존 프로그램에 업데이트를 위하여 필요한 코드와 데이터를 추가한 프로그램을 말한다. 업데이트 가능한 프로그램에는 업데이트 지점(update point)도 추가되는데, 업데이트 지점이란 구 버전의 프로그램 수행 중 정상적인 업데이트를 위한 조건을 만족시키면서 안전하게 업데이트가 가능한 코드 상의 특정 지점으로 일반적으로 프로그래머가 지정한 위치에 설정된다.

1.3. 스택 재구성 기술

스택 재구성 기술은 업데이트 시 구 버전의 스택을 임시 저장 후 이를 바탕으로 새 버전의 스택으로 재구성하는 기술로 다음과 같은 순서로 이루어져 있다[5]. 구 버전의 프로그램이 업데이트 지점을 수행 시 그 동안 생성된 스택의 내용을 저장하여 두었다가 새 버전의 프로그램이 로드된 후 저장해 둔 구 버전의 스택 내용을 사용하여 새 버전의 스택을 인공적으로 구축한다. 그 다음 새 버전의 프로그램이 구 버전에서 업데이트가 일어난 위치에서 프로그램의 수행을 계속한다.

2. 주요 DSU 시스템

2.1 Ginseng (STUMP)

미국의 University of Maryland에서 연구한 Ginseng[3]과 다음 버전인 STUMP[4]는 안전한 업데이트 지점(safe update point)을 자동으로 찾기 위하여 데이터 타입의 변화를 추적하는 정적 분석(static analysis)을 사용한다.

Ginseng 및 STUMP에서 코드 업데이트는 함수 단위로 이루어지므로, 먼저 모든 함수를 포인터를 사용하여 간접 접근 하도록 변환한다. 코드 업데이트 시에는 새 버전에서 변경된 함수를 메모리에 동적 할당하고, 구 버전을 가리키던 함수 포인터를 새로 추가된 함수의 메모리를 가리키도록 한다. 전역 데이터 업데이트는 매개 함수(mediator function)를 이용하여 전역 데이터에 간접 접근하도록 한다. 또한 지역 데이터는 업데이트 하지 않는다.

Ginseng은 프로그램 수행 시 정적 분석에 따른 오버헤드가 발생하며, 코드 상의 모든 함수와 전역 데이터를 간접 접근하므로, 이에 따른 오버헤드가 발생한다. 또한 사용하지 않는 구 버전의 코드와 전역 데이터를 유지해야 하므로, 이에 따른 메모리 공간의 낭비가 발생하며, 구 버전의 지역 데이터가 유지되지 않는다.

2.2 UpStare

미국의 Arizona State University에서 개발한 UpStare[5][6]는 스택 재구성 기술을 이용 하며, 이를 위하여 구 버전 및 새 버전의 소스 코드를 스택 재구성 기능이 추가된 업데이트 가능한 프로그램으로 변환하여 업데이트 한다.

이 연구에서 코드 업데이트는 전체 프로그램 업데이트를 수행하며, 모든 전역 데이터를 힙 영역에 유지한다. 또한 지역 데이터는 스택 재구성 기술을 이용하여 업데이트 한다.

이 연구에서는 스택 재구성 기술을 이용하므로 스택 내의 모든 지역 데이터를 쉽게 업데이트 할 수 있다. 그러나 전역 데이터 사용 시 동적 메모리 접근에 따른 오버헤드가 있다.

2.3 Ekiden (Kitsune)

미국의 University of Maryland에서 연구한 다른 DSU 시스템인 Ekiden[7]과 Ekiden의 다음 버전인 Kitsune[8]는 DSU 시스템을 라이브러리 형식으로 제공한다.

코드 업데이트 시 수정된 함수 각각이 아닌 전체 프로그램 단위로 업데이트 하는 전체 프로그램 업데이트 방식을 사용하며, 업데이트 시 모든 전역 데이터를 업데이트 한다. 또한 지

역 데이터 업데이트 시 시스템 호출 함수 setjmp와 longjmp를 사용한다[10].

이 연구에서는 모든 전역 데이터를 업데이트 하므로 변화되지 않은 데이터를 업데이트하는 불필요한 업데이트가 발생하며, 업데이트 후 새 프로세스로 변경되므로 포인터와 같은 특정 데이터 타입의 데이터를 업데이트 시 문제가 발생한다. 또한 지역 데이터 업데이트 시 시스템 호출 함수 setjmp와 longjmp를 이용하므로 기존의 지역 데이터가 손상되거나 시스템 호출 함수 longjmp 호출 시 유효하지 않은 스택 프레임을 복구하여 문제가 발생할 수 있다. 뿐만 아니라 Ekiden은 새 버전의 프로그램을 수행하기 위해 새 프로세스를 생성하고 구 버전의 상태를 복사한다. 이 때 새 프로세스를 생성하는 오버헤드가 발생한다.

III. DSU 시스템 설계

동적 소프트웨어 업데이트에서 일반적으로 구 버전과 새 버전은 거의 같은 프로그램이지만 일부 함수 및 전역 변수 등에 차이가 있다.

본 논문에서 제안하는 업데이트 방법은 구 버전의 프로그램을 코드, 전역 데이터, 지역 데이터로 나누어 한 프로세스 상에서 새 버전의 프로그램으로 업데이트 하는 것이다. 여기서 코드는 한 프로세스 상에서 업데이트를 수행하기 위하여 공유 메모리 영역에 로드한다. 전역 데이터의 경우, 구 버전의 전역 데이터는 데이터 영역에, 새 버전에서 추가 또는 수정된 전역 데이터는 힙 영역에 로드한다. 지역 데이터는 스택 영역에 저장한다.

1. 코드 업데이트

본 연구에서는 코드 업데이트 시 전체 프로그램 업데이트 방식을 사용한다. 전체 프로그램 업데이트 방식이란, 새 버전의 프로그램으로 업데이트 시 추가, 수정 또는 삭제된 함수만을 업데이트 하는 것이 아니라, 수행중인 프로그램 전체를 업데이트 하는 방식이다. 전체 프로그램 업데이트 방식을 이용하면, 현재 실행중인 함수를 포함하여 즉시 업데이트를 할 수 있으며, 새 버전의 코드를 로드하고 해제하기가 용이하다.

코드는 업데이트 후 프로세스의 가상 메모리 공간에서 그 위치가 변경되어도 수행 시 문제가 발생하지 않는다. 따라서 최초 버전을 포함한 모든 버전의 코드는 기존의 방식과는 다르게 동적 링킹 라이브러리(DLL: dynamic liking library)로 컴파일 후 프로세스의 가상 메모리 공간 중 공유 메모리 영역에 로드하여 수행한다. 이와 같은 방식으로 하나의 프로세스 메모리 공간 상에서 동적 소프트웨어 업데이트를 구현할 수 있다. 아래 그림 1은 코드 업데이트 수행 과정을 나타낸 그림이다. 구 버전의 프로그램을 해제 하기 위하여 시스템 호출 함수 dlclose를 사용하며, 새 버전의 프로그램을 로드하기 위해서 시스템 호출 함수 dlopen을 사용한다[10].

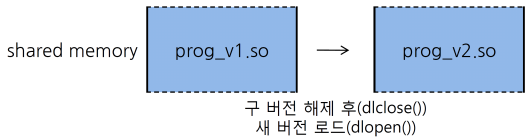


그림 1. 코드 업데이트
Fig. 1. Code Update

본 연구에서는 업데이트 시 새 버전의 DLL을 로드 한 후 구 버전의 메모리를 해제하므로, 구 버전의 코드를 메모리 상에 유지하는 메모리 낭비를 줄일 수 있다.

2. 전역 데이터 업데이트

대부분의 소프트웨어 업데이트는 전역 데이터의 변화 부분이 크지 않다[9]. 따라서 본 연구에서는 전역 데이터 업데이트 시 최초 버전의 전역 데이터를 프로세스의 가상 메모리 공간 중 rodata, data 및 bss 영역에 로드 하고, 새 버전에서 추가 또는 수정된 부분만을 힙 영역에 로드하는 부분 업데이트 방식을 사용한다. 아래 그림 2는 전역 데이터 업데이트 수행 과정을 나타낸 그림이다.

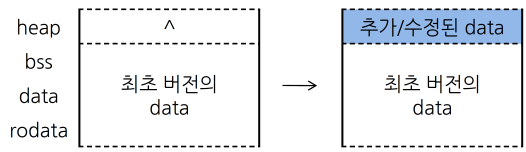


그림 2. 전역 데이터 업데이트
Fig. 2. Global Data Update

전역 데이터 업데이트 시 부분 업데이트 방식을 사용 할 경우, 추가 또는 수정된 전역 데이터만 힙 영역에 저장하므로 동적 할당에 따른 오버헤드가 크지 않다. 또한 수정되지 않은 대부분의 전역 데이터는 실행중인 프로세스의 데이터 영역에 저장되어 있기 때문에 이러한 대부분의 전역 데이터에는 업데이트 후에도 별도의 업데이트 없이 직접 접근이 가능하다.

3. 지역 데이터 업데이트

지역 데이터는 스택 영역에 저장되어 있다. 이 데이터는 새 버전이 수행 된 후 정상적인 수행을 위하여 복구 되어야 하는 상태이기 때문에, 동적 소프트웨어 업데이트에서 구 버전의 함수 수행 도중 생성된 지역 데이터를 새 버전으로 전달하기 위하여 스택 재구성 기술을 사용한다.

본 시스템에서는 스택 재구성 기술을 구현하는 방법 중 2009년 발표된 Kristis Makris의 논문[6]에 기술된 방법을 사용한다. 이 방법은 구 버전 및 새 버전의 소스 코드를 업데이트 가능한 프로그램으로 변환하여 구 버전 수행 시 업데이트 지점에 도착하면 구 버전의 스택을 프레임 당 하나씩 힙 영역에 임시 저장하고 새 버전의 프로그램을 로드한 후, 저장한 스택 프레임을 사용하여 새 버전의 스택을 구성하는 방법이다. 이 방법을 사용하면 구 버전의 프로그램을 실행 중 생성된 프로세스의 상태를 보존 가능하며, 스택 내에 저장된 지역 데이터와 수행 지점 등을 새 버전에 맞게 업데이트 할 수 있다. 아래 그림 3은 지역 데이터 업데이트 수행 과정을 나타낸 그림이며, 구 버전의 스택이 업데이트 후 새 버전의 스택으로 변화됨을 나타낸다.

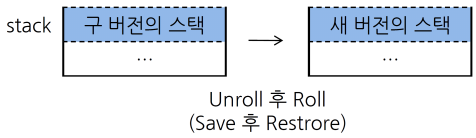


그림 3. 지역 데이터 업데이트
Fig. 3. Local Data Update

본 연구에서는 업데이트 지점을 수동으로 입력받아, 코드 변환 도구를 이용하여 구 버전 및 새 버전의 프로그램을 스택 재구성 기능을 포함하는 업데이트 가능한 프로그램으로 변환한다.

IV. DSU 시스템 구현

본 연구에서는 기존 프로그램을 업데이트 가능한 프로그램으로 변환하고, 업데이트를 수행하는 전체 DSU 시스템을 구현하였다. 이번 장에서는 기존 프로그램을 업데이트 가능한 프로그램으로 변환하는 과정을 설명한 후, 개발한 DSU 시스템을 구성하는 주요 모듈인 DSU 라이브러리, DSU 관리 명령어 및 DSU 변환 도구의 구현 내용에 관하여 기술한다.

1. 업데이트 가능한 프로그램 생성 과정

본 연구에서 구현한 DSU 시스템을 사용하여 업데이트를 수행하기 위해서는 기존의 프로그램을 업데이트 가능한 프로그램으로 변환하여야 한다. 이 때 앞서 3장에서 설계한 바와 같이 프로그램을 코드, 전역 데이터 및 지역 데이터로 나누어 업데이트 가능한 프로그램으로 변환한다. 업데이트 가능한 프로그램은 구 버전과 새 버전으로 나누어 생성한다.

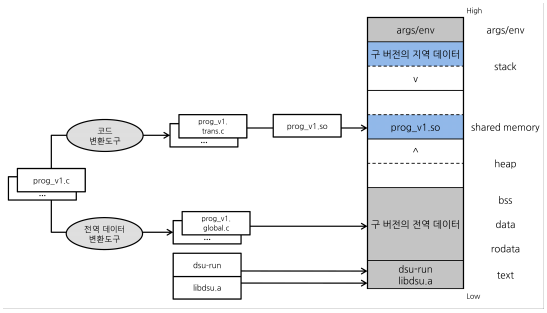


그림 4. 구 버전의 업데이트 가능한 프로그램 생성 과정
Fig. 4. Generating Process of Updatable Program of Old Version

먼저 구 버전의 업데이트 가능한 프로그램 생성 과정은 위 그림 4와 같다. 첫째로 코드 업데이트를 위하여 코드 변환 도구를 사용한다. 이 도구는 구 버전의 프로그램에서 데이터 영역을 제외한 코드 부분만을 분리하며, 분리된 코드에 스택 재구성을 위한 스택 unroll/roll 기능이 추가되도록 변환한다. 변환된 파일은 DLL 파일로 컴파일 하여 프로세스의 가상 메모리 공간 중 공유 메모리 영역에 로드한다. 둘째로 전역 데이터 변환 도구를 사용하여 구 버전의 프로그램에서 전역 데이터 부분만을 분리한다. 이렇게 분리한 데이터는 프로세스 가상 메모리 공간 중 데이터 영역, 즉 bss, data 및 rodata 영역에 로드된다. 이 전역 데이터의 메모리는 해제되지 않으며 새 버전에서도 업데이트 없이 그대로 사용된다. 지역 데이터는 구 버전의 프로그램을 수행함에 따라 생성된다.

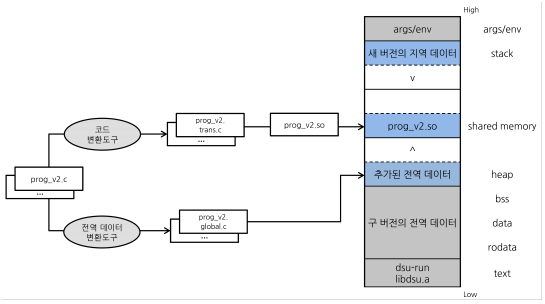


그림 5. 새 버전의 업데이트 가능한 프로그램 생성 과정
Fig. 5. Generating Process of Updatable Program of New Version

새 버전의 업데이트 생성 과정은 위 그림 5와 같다. 먼저 새 버전의 프로그램도 코드 변환 도구를 사용하여 데이터 영역을 제외한 코드 부분만을 분리한 후, 스택 unroll/roll 기능이 추가되도록 변환한다. 변환된 파일은 DLL 파일로 컴파일하여, 공유 메모리에 로드되어 있던 구 버전을 해제 후, 공유 메모리 영역에 로드한다. 다음으로 전역 데이터 변환 도구를 사용하여 새 버전에서 추가 또는 수정된 전역 데이터 부분만을 분리한다. 이렇게 분리된 데이터는 구 버전과 다르게 힙 영역에 로드한다. 이 때 새 버전에서 추가 또는 수정되는 전역 변수는 많지 않으므로, 대부분의 전역 데이터는 데이터 영역에 저장되어 있으며, 코드 변경 또는 업데이트 없이 사용 가능하다.

이렇게 변환된 구 버전과 새 버전의 업데이트 가능한 프로그램은 DSU 관리 명령어 및 DSU 라이브러리와 같은 프로세스 내에서 수행되고 업데이트 된다. DSU 시스템의 자세한 구현 내용은 아래 각 절에서 설명한다.

2. DSU 라이브러리

본 연구에서는 구 버전과 새 버전의 업데이트 가능한 프로그램을 DSU 시스템 상에서 수행하기 위하여 DSU 라이브러리를 구현하였다. DSU 라이브러리의 구현 코드는 컴파일 후 정적 라이브러리(static library) 파일에 저장된다.

DSU 라이브러리는 DLL 파일 관리, 업데이트 관리, 스택 unroll/roll 관리 및 심볼 테이블 관리 함수들로 구성되어 있다.

첫째, 하나의 프로세스 메모리 공간 상에서 동적 소프트웨어 업데이트를 구현하기 위하여 최초 버전을 포함한 모든 버전의 코드를 기존의 방식과는 다르게 DLL로 컴파일하고, 수행 중인 프로세스가 프로그램 수행 중 이를 로드하여 수행 하여야 한다. 이를 위하여 DLL 파일 관리 함수는 DLL로 컴파일된 각 버전의 프로그램을 프로세스의 가상 메모리 공간 중 공유 메모리에 로드한 다음, 로드한 DLL에 포함된 특정 함수의 주소를 알아낸 후 이를 호출하여 수행한다. 또한 업데이트 시 새 버전의 DLL을 로드 후 구 버전의 메모리를 해제하는 기능을 수행한다.

둘째, 업데이트 관리 함수는 구 버전 수행 중 업데이트 신호를 전달하기 위하여 리눅스 운영체제의 신호(signal) 중 신호 SIGUSR1을 사용한다. 이러한 업데이트 신호를 정의하고, 받은 업데이트 신호를 처리하기 위하여 신호 핸들러 함수와 시그널 및 주요 DSU 전역 데이터를 초기화 하는 함수를 구현하였다.

셋째, 동적 소프트웨어 업데이트에서는 구 버전의 함수 수행 도중 생성된 지역 데이터를 새 버전으로 전달하기 위하여 스택 unroll/roll 기술을 사용하여 스택 내의 지역 데이터와 함수의 연속 지점을 업데이트 한다. 이러한 unroll/roll 기능을 사용하기 위해서는 소스 코드의 변환이 필요하며, 기존 소스 코드의 주어진 함수 시작 부분 및 함수 호출 부분 뒤에 스택 unroll/roll 코드를 삽입하여 변환된 소스 코드를 생성한다.

마지막으로 동적 소프트웨어 업데이트를 구현하기 위해서는 앞서 로드된 DLL의 전역 데이터 값을 뒤에 로드될 DLL의 전역 데이터 값으로 전달할 필요가 있다. 이 때 많은 변수 이름을 보다 효율적으로 관리하기 위하여 심볼 이름을 저장하고 관리하는 효율적인 프로그램이 필요하다. 이를 위하여 본 시스템에서는 해쉬 테이블(hash table)을 사용한 심볼 테이블 관리 프로그램을 구현하여 사용한다.

3. DSU 관리 명령어

DSU 관리 명령어는 DSU 시스템을 동작시키고 업데이트를 수행하는 명령어로 리눅스 시스템 상에서 명령어처럼 사용 가능하며 명령어 dsu-run, dsu-stat 및 dsu-update로 구성된다.

먼저 명령어 dsu-run은 DSU 시스템에서 DLL로 컴파일된 최초 버전의 프로그램을 수행하는 명령어로, 인수로 주어진 DLL 파일 내의 함수 main()을 실행한다. 이 명령어가 수행되는 프로세스의 데이터 영역에는 최초 버전의 전역 데이터가 저장되어 있다.

다음으로 명령어 dsu-stat은 DSU 시스템에서 현재 수행중인 프로그램의 정보인 현재 DSU 시스템 프로세스의 pid 및 DLL 파일 경로명을 출력하는 명령어이다. 이 명령어는 위의 명령어 dsu-run을 수행한 후 사용 가능하며, 만약 어떤 프로그램이 DSU 시스템 상에서 수행 중이지 않으면 이 명령어는 아무 정보도 출력하지 않는다.

마지막으로 명령어 dsu-update는 현재 DSU 시스템에서

수행 중인 프로그램을 새 버전의 프로그램으로 업데이트하는 명령어이다. 이 명령어를 수행한 후 구 버전이 업데이트 지점을 수행하면 업데이트 신호 SIGUSR1을 전달 후, 스택 unroll/roll이 수행되면서 새 버전으로 업데이트가 일어난다. 업데이트 시 스택 재구성을 위한 스택의 내용은 심볼 테이블 관리 함수를 사용하여 저장하고 관리한다.

4. DSU 변환 도구

본 연구에서 구현한 DSU 시스템을 사용하기 위해서는 DSU 변환 도구를 사용하여 주어진 버전의 프로그램을 업데이트 가능한 프로그램으로 변환하여야 한다. DSU 변환 도구는 컴파일러 개발 도구인 flex[11]와 bison[12]을 이용하여 구현하였으며, 코드 변환 도구와 전역 데이터 변환 도구로 구성된다.

이 변환 도구는 ANSI C 문법으로 작성된 C 프로그램의 문법을 가정하고 개발하였으므로, 리눅스 운영체제에서 사용되는 C 언어 컴파일러인 GCC가 인식하는 C 언어로 작성된 프로그램의 변환에는 완벽하지 않을 수 있다.

4.1. 코드 변환 도구

코드 변환 도구는 인수로 주어진 C 프로그램의 함수 템플릿을 분석하여 각 함수를 스택 unroll/roll 기능이 추가된 업데이트 가능한 코드로 변환하는 프로그램이다. 이 도구의 입력 파일인 함수 템플릿에는 함수의 프로토타입, 함수의 지역 데이터 선언 및 이 함수가 부르는 함수 호출들이 포함된다.

이 변환 도구는 ANSI C 문법을 사용하여 입력 프로그램을 분석하여, 이 함수의 지역 데이터 이름 및 업데이트 함수 이름 및 업데이트 지점 등을 분석하고 이 정보를 바탕으로 스택 unroll/roll 기능이 추가된 업데이트 가능한 코드를 생성한다. 이 때 코드 변환 도구에 의하여 변환되어야 하는 함수는 업데이트 지점 수행 시점에 스택에 저장되어 있는 지역 데이터의 대상 함수이다. 예를 들어 어떤 응용 프로그램에서 함수 main()이 함수 read_line() 및 print_line()을 부르고, 함수 read_line() 및 print_line()의 시작 지점이 업데이트 지점이라면, 3개의 함수인 main(), read_line(), print_line()이 스택 unroll/roll을 위하여 변환되어야 하는 함수이다. 함수 main()을 코드 변환 도구를 통하여 변환하고자 할 때, 입력 파일에 이 함수의 모든 부분을 넣지 않고, 이 함수에 대한 프로토타입, 지역 변수 선언, 및 이 함수 내에서 다른 함수를 부르는 함수 호출 문장만을 포함시킨다. 아래 그림 6은 코드 변환 도구의 보기로 볼드 및 이텔릭체로 표시된 부분이 코드 변환 도구 사용 후 추가되는 내용이다.

입력 파일

```
int main(int argc, char **argv)
{
  mainloop();
}
```

출력 파일

```
int main(int argc, char **argv)
{
  {
    if (dsu_signal_unroll == 1) {
      dsu_signal_unroll = 0;
      dsu_stack_unroll = 1;
      return 0;
    } else if (dsu_stack_roll == 1) {
     struct table *tablep = dsu_pop_table();
     int label = dsu_restore_label(tablep);
     dsu_restore_symbol(tablep, "argc", &argc, sizeof(argc));
     dsu_restore_symbol(tablep, "argv", &argv, sizeof(argv));
     dsu_free_table(tablep);
     if (label == 0) {
        printf("main(...): rolled... \n");
        goto label_mainloop
      }
    }
  }
label_mainloop:
  mainloop();
  {
    if (dsu_stack_unroll == 1) {
      struct table *tablep = dsu_push_table("main");
      dsu_save_label(tablep, 0);
      dsu_save_symbol(tablep, "argc", &argc, sizeof(argc));
      dsu_save_symbol(tablep, "argv", &argv, sizeof(argv));
      printf("main(...): unrolled... \n");
      return 0;
    }
  }
}
```

그림 6. 코드 변환 도구 보기
Fig. 6. Example of Code Transformation Tool

4.2 전역 데이터 변환 도구

전역 데이터 변환 도구는 인수로 주어진 C 프로그램 파일을 분석하여 전역 데이터 선언 부분만을 찾아내어 출력하는 프로그램이다. 이 변환 도구는 초기 버전의 전역 데이터를 모두 모아 data 및 bss 영역에 넣기 위하여 사용된다.

전역 데이터 변환 도구는 입력으로 주어지는 ANSI C 프로그램을 읽어 구문 분석한 다음 이 프로그램 내에 존재하는 모든 함수 정의 부분에 대하여 시작 줄, 열 번호, 끝 줄 및 끝 열 번호를 알아낸다. 그 후 입력 프로그램에서 함수 정의 부분을 제거한 프로그램을 생성한다. 이를 구현하기 위하여 구

문 분석 도중에 하나의 함수 정의 및 끝 부분에 도달 시 줄 및 열 번호를 기록한다. 이 과정에서 전처리기인 cpp가 먼저 수행된다. 어휘 분석기(lexical analyzer) 및 구문 분석기(syntax analyzer) 프로그래밍에 사용된 C 문법은 공개된 ANSI C YACC 문법[13]을 사용하였다. 아래 그림 7은 전역 데이터 변환 도구의 보기로 볼드 및 이탤릭체로 표시된 부분이 전역 데이터 변환 도구 사용 후 삭제되는 내용이다.

입력 파일

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

FILE *fp
int line_no = 0;

int read_line(char *ptr)
{
  ...
}

int print_line(char *ptr)
{
  ...
}

int main(int argc, char *argv[])
{
  ...
}
```

출력 파일

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

FILE *fp
int line_no = 0;
```

그림 7. 전역 데이터 변환 도구 보기
Fig. 7. Example of Global Data Transformation Tool

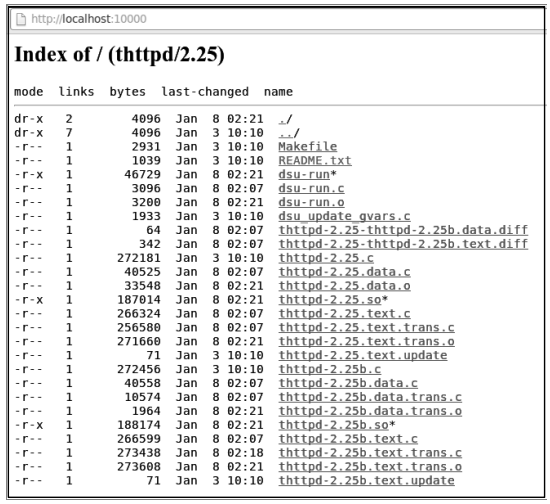
V. 시험

이 장에서는 x86 구조 상에서 리눅스 kernel 버전 2.6과 버전 3.2를 사용한 DSU 시스템을 구현하여 C 응용 프로그램을 업데이트 가능한 프로그램으로 변환하고 시험하였다.

DSU 시스템의 동작을 확인하기 위하여 작성한 시험 프로그램은 구 버전에서 새 버전으로 프로세스의 상태 전달을 확인하기 위한 파일 입출력 프로그램, 스택 재구성 기능을 시험하기 위한 프로그램, 심볼 테이블 기능을 시험하기 위한 프로그램 등이 있다. 이러한 시험 프로그램 중 HTTP서버 프로그램인 thttpd[14]의 버전 2.25를 2.25b로 업데이트하는 시험을 설명한다. 두 버전 사이의 차이점은 다음과 같다. thttpd의 새 버전인 버전 2.25b에는 하나의 전역 데이터가 추가 되었고, 버전 2.25에서 3개의 전역 데이터가 수정되었으며, 하나의 전역 데이터가 삭제되었다. 또한 추가 되거나 삭제된 함수는 없으며, 몇몇 함수의 내용이 변경되었다.

본 연구에서는 위와 같은 thttpd 프로그램 버전 2.25와 2.25b를 DSU 변환 도구를 이용하여 업데이트 가능한 프로그램으로 변환 후, DSU 관리 명령어 dsu-run과 DSU 라이브러리를 컴파일하여 버전 2.25를 수행하였다. 그 후 DSU 관리 명령어 dsu-stat으로 현재 수행중인 DSU 시스템의 pid와 현재 수행중인 구 버전의 파일명을 확인 후 DSU 관리 명령어 dsu-update를 이용하여 새 버전의 프로그램으로 업데이트 하는 시험을 수행하였다. 이 때 시험을 위하여 root 권한이 필요하며, thttpd의 신호 문제로 인하여 DSU 시스템 관리 명령어를 gdb 내에서 수행하였다.

위와 같은 과정을 거쳐 thttpd 프로그램의 업데이트 시험을 수행 한 결과는 다음과 같다. 먼저 아래 그림 8은 DSU 관리 명령어 dsu-run을 사용하여 DSU 시스템 상에서 thttpd 프로그램의 버전 2.25를 실행한 보기이다.


http://localhost:10000

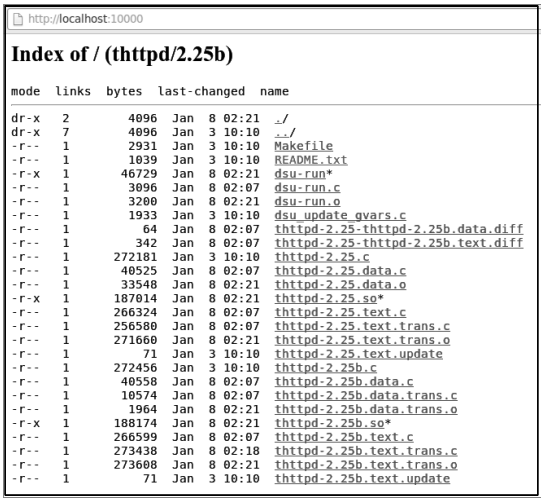
Index of / (thttpd/2.25)

mode	links	bytes	last-changed	name
dr-x	2	4096	Jan 8 02:21	../
dr-x	7	4096	Jan 3 10:10	.../
-r--	1	2931	Jan 3 10:10	Makefile
-r--	1	1039	Jan 3 10:10	README.txt
-r-x	1	46729	Jan 8 02:21	dsu-run*
-r--	1	3096	Jan 8 02:07	dsu-run.c
-r--	1	3200	Jan 8 02:21	dsu-run.o
-r--	1	1933	Jan 3 10:10	dsu_update_gvars.c
-r--	1	64	Jan 8 02:07	thttpd-2.25-thttpd-2.25b.data.diff
-r--	1	342	Jan 8 02:07	thttpd-2.25-thttpd-2.25b.text.diff
-r--	1	272181	Jan 3 10:10	thttpd-2.25.c
-r--	1	40525	Jan 8 02:07	thttpd-2.25.data.c
-r--	1	33548	Jan 8 02:21	thttpd-2.25.data.o
-r-x	1	187014	Jan 8 02:21	thttpd-2.25.so*
-r--	1	266324	Jan 8 02:07	thttpd-2.25.text.c
-r--	1	256580	Jan 8 02:07	thttpd-2.25.text.trans.c
-r--	1	271660	Jan 8 02:21	thttpd-2.25.text.trans.o
-r--	1	71	Jan 3 10:10	thttpd-2.25.text.update
-r--	1	272456	Jan 3 10:10	thttpd-2.25b.c
-r--	1	40558	Jan 8 02:07	thttpd-2.25b.data.c
-r--	1	10574	Jan 8 02:07	thttpd-2.25b.data.trans.c
-r--	1	1964	Jan 8 02:21	thttpd-2.25b.data.trans.o
-r-x	1	188174	Jan 8 02:21	thttpd-2.25b.so*
-r--	1	266599	Jan 8 02:07	thttpd-2.25b.text.c
-r--	1	273438	Jan 8 02:18	thttpd-2.25b.text.trans.c
-r--	1	273608	Jan 8 02:21	thttpd-2.25b.text.trans.o
-r--	1	71	Jan 3 10:10	thttpd-2.25b.text.update

그림 8. thttpd 버전 2.25 수행 보기
Fig. 8. Example of running thttpd version 2.25

이 때, DSU 관리 명령어 dsu-stat을 입력하면 DLL로 컴파일된 구 버전의 프로그램인 thttpd-2.25.so가 수행 중임을 알 수 있다.

구 버전을 수행하는 도중 DSU 관리 명령어 dsu-update 입력 시 스택 재구성을 위한 unroll/roll 기능이 수행되고, 그 이후 새 버전이 수행된다. 아래 그림 9는 DSU 관리 명령어 dsu-update 입력 후 thttpd 프로그램의 버전 2.25b로 업데이트 된 보기이다.


http://localhost:10000

Index of / (thttpd/2.25b)

mode	links	bytes	last-changed	name
dr-x	2	4096	Jan 8 02:21	./
dr-x	7	4096	Jan 3 10:10	../
-r--	1	2931	Jan 3 10:10	Makefile
-r--	1	1039	Jan 3 10:10	README.txt
-r-x	1	46729	Jan 8 02:21	dsu-run*
-r--	1	3096	Jan 8 02:07	dsu-run.c
-r--	1	3200	Jan 8 02:21	dsu-run.o
-r--	1	1933	Jan 3 10:10	dsu_update_gvars.c
-r--	1	64	Jan 8 02:07	thttpd-2.25-thttpd-2.25b.data.diff
-r--	1	342	Jan 8 02:07	thttpd-2.25-thttpd-2.25b.text.diff
-r--	1	272181	Jan 3 10:10	thttpd-2.25.c
-r--	1	40525	Jan 8 02:07	thttpd-2.25.data.c
-r--	1	33548	Jan 8 02:21	thttpd-2.25.data.o
-r-x	1	187014	Jan 8 02:21	thttpd-2.25.so*
-r--	1	266324	Jan 8 02:07	thttpd-2.25.text.c
-r--	1	256580	Jan 8 02:07	thttpd-2.25.text.trans.c
-r--	1	271660	Jan 8 02:21	thttpd-2.25.text.trans.o
-r--	1	71	Jan 3 10:10	thttpd-2.25.text.update
-r--	1	272456	Jan 3 10:10	thttpd-2.25b.c
-r--	1	40558	Jan 8 02:07	thttpd-2.25b.data.c
-r--	1	10574	Jan 8 02:07	thttpd-2.25b.data.trans.c
-r--	1	1964	Jan 8 02:21	thttpd-2.25b.data.trans.o
-r-x	1	188174	Jan 8 02:21	thttpd-2.25b.so*
-r--	1	266599	Jan 8 02:07	thttpd-2.25b.text.c
-r--	1	273438	Jan 8 02:18	thttpd-2.25b.text.trans.c
-r--	1	273608	Jan 8 02:21	thttpd-2.25b.text.trans.o
-r--	1	71	Jan 3 10:10	thttpd-2.25b.text.update

그림 9. thttpd 버전 2.25b 수행 보기
Fig. 9. Example of running thttpd version 2.25b

이 때 DSU 관리 명령어 dsu-stat 입력 시, DLL로 컴파일 된 새 버전의 프로그램인 thttpd-2.25b.so가 구 버전의 프로세스가 수행되던 동일한 프로세스에서 수행 중임을 알 수 있다.

이를 통하여 동일한 프로세스 내에서 구 버전의 프로세스 상태가 새 버전으로 전달됨을 확인하였다. 또한 최초 버전을 수행하는 프로세스와 새 버전을 수행하는 프로세스가 동일하며, 새 버전의 프로그램이 구 버전과 동일한 프로세스에서 종료 없이 동적으로 업데이트 후 정상 동작함을 알 수 있다.

VI. 평가

이번 장에서는 2장에서 설명한 주요 DSU 시스템과 본 연구에서 구현한 DSU 시스템을 비교하고 평가한다.

이 연구의 평가를 위하여 기존의 DSU 시스템 연구 사례를 살펴보았으나, 구현 방법에 따른 성능 측정 결과에 대하여 발표된 사례가 없다. 또한 각 DSU 시스템에서 업데이트를 수행하는 프로그램이 각기 다르므로 시스템의 성능을 수치로 비교하는 것은 무리가 있다. 따라서 아래와 같이 각 DSU 시스템의 구현 방법과 특징을 토대로 하여 DSU 시스템 수행 시 불필요한 메모리 사용과 오버헤드가 발생 할 수 있는 원인을 분석하여 비교 및 평가 하였다.

본 연구에서 제안하는 DSU 시스템의 구현 방법은 기존 방법과 비교하여 다음과 같은 특징이 있다. 첫째, 코드 업데이트는 전체 프로그램 업데이트 방식을 사용하며, 코드 상의 모든 함수를 호출 시 직접 접근한다. 또한 업데이트 시 새 버전의 코드를 공유 메모리에 로드 후 구 버전의 코드 영역을 해제하므로 코드 메모리의 낭비가 적다. Ginseng[3]은 코드 업데이트 시 함수 단위의 업데이트를 수행하며, 업데이트 시 구 버전의 함수를 가리키던 모든 함수 포인터를 새로운 버전의 함수를 가리키도록 변경한다. 이와 같이 코드 내의 모든 함수를 간접 참조하도록 하므로 함수 간접 접근에 따른 오버헤드가 있으며, 구 버전의 함수를 유지하는 메모리 낭비가 있다. Ginseng의 다음 버전인 STUMP[4]에서는 새 버전에서 삭제된 함수를 메모리에서 해제하나, 구현 방법은 동일하므로 함수 간접 참조에 의한 오버헤드는 동일하게 발생한다. UpStare[5][6], Ekiden[7] 및 Ekiden의 다음 버전인 Kitsune[8]에서는 전체 프로그램 업데이트를 수행하고 함수를 직접 호출하므로, 본 연구에서 제안하는 코드 업데이트 방식과 유사하다. 하지만 Ekiden은 새로운 프로세스를 생성하여 업데이트를 수행하므로, 새 프로세스 생성에 따른 오버헤드가 있다.

둘째, 전역 데이터 업데이트는 부분 업데이트 방식을 사용하므로, 최초 버전의 전역 데이터는 프로세스의 메모리 공간 중 데이터 영역에 저장하고 새 버전에서 수정된 전역 데이터만 업데이트하고 힙 영역에 저장한다. 따라서 새 버전에서 수정되지 않은 전역 데이터는 메모리에 새로 할당 할 필요가 없으므로 전역 데이터의 낭비가 적다. 새 버전에서 수정된 전역 데이터만 업데이트하므로 불필요한 업데이트를 줄일 수 있으며, 데이터 영역에 저장되어 있는 수정되지 않은 대부분의 전역 데이터에 동적 할당과 간접 접근 없이 직접 접근할 수 있으므로 오버헤드가 적다. 하지만 Ginseng과 STUMP는 모든 전역 데이터를 직접 접근하지 않고 매개 함수를 이용하여 간접 접근하므로 간접 접근에 따른 오버헤드가 있다. 또한 새 버전의 전역 데이터를 모두 메모리에 새로 할당해야 하므로 전역 데이터의 메모리 낭비가 있다. UpStare는 부분 업데이트를 사용하므로 새 버전에서 수정된 데이터만 힙 영역에 추가한다. 하지만 모든 전역 데이터를 힙 영역에 저장하므로 동적 메모리 할당 및 접근에 따른 오버헤드가 있다. Ekiden과 Kitsune는 새 버전에서 수정되지 않은 전역 데이터도 메모리에 새로 할당해야 하므로 전역 데이터의 메모리 낭비가 있다.

또한 Ekiden은 업데이트 후 새 프로세스로 변경되므로 포인터와 같은 특정 데이터 타입의 데이터를 업데이트 시 문제가 발생한다.

셋째, 본 연구에서 지역 데이터 업데이트는 스택 재구성 방식을 이용하며, 업데이트 시점에 스택 내의 모든 지역 데이터를 복구할 수 있다. Ginseng과 STUMP는 지역 데이터를 업데이트 하지 않으므로, 업데이트 이후 지역 데이터가 유지되지 않는다. UpStare는 스택 재구성 방식을 이용하므로 본 연구의 방식과 유사하다. Ekiden과 Kitsune는 업데이트가 필요한 지역 데이터를 따로 표시하고 이를 위한 코드를 생성함에 따른 오버헤드가 있다. 또한 업데이트 시 시스템 호출 함수 setjmp와 longjmp를 이용하여 지역 데이터를 저장 및 복원하는데, 이는 기존의 지역 데이터가 손상되거나 시스템 호출 함수 longjmp 호출 시 유효하지 않은 스택 프레임을 복구하여 문제가 발생할 수 있다. 아래 표 1은 위 비교 내용을 정리한 표이다.

표 1. 기존 DSU 시스템과의 비교
Table 1. Comparison of Existing DSU Systems and Ours

	시스템	비교 내용
코드 업데이트	Ginseng /STUMP	-메모리 낭비 -함수 간접 접근 오버헤드
	UpStare	-메모리 낭비 최소화
	Ekiden /Kitsune	-프로세스 생성 오버헤드
	본 DSU 시스템	-메모리 낭비 최소화 -모든 함수 직접 접근
전역 데이터 업데이트	Ginseng /STUMP	-데이터 간접 접근 오버헤드
	UpStare	-동적 메모리 접근 오버헤드
	Ekiden /Kitsune	-불필요 업데이트 발생 -특정 데이터 타입 업데이트 불가 (포인터 포함)
	본 DSU 시스템	-불필요 업데이트 및 간접 접근 오버헤드 없음 -대부분의 데이터를 동적 메모리 할당 없이 사용
지역 데이터 업데이트	Ginseng /STUMP	-업데이트 불가
	UpStare	-모두 업데이트 가능
	Ekiden /Kitsune	-데이터 손상 위험
	본 DSU 시스템	-모두 업데이트 가능

VII. 결론

본 연구에서는 DSU의 일반적 특성, 프로세스를 구성하는 코드와 전역 및 지역 데이터가 가지는 특성 및 많은 소프트웨어 업데이트 시스템의 특성 등을 이용하여 새로운 업데이트 방식을 사용하는 DSU 시스템을 제안하였다. 또한 리눅스 상에서 C 응용 프로그램을 업데이트 하는 전체 DSU 시스템을 구현하였다.

기존 DSU 시스템은 구현 방법 상 구 버전의 코드를 유지하기 위한 불필요한 메모리를 유지하여야 하거나, 변화되지 않은 전역 데이터를 업데이트하기 위한 불필요한 업데이트가 추가적으로 필요하다.

그러나 본 연구에서 제안하는 방식은 코드를 전체 프로그램 업데이트 하므로 기존 연구에 비해 메모리 낭비가 적다. 전역 데이터는 부분 업데이트 하므로 새 버전으로 업데이트 시 마다 모든 전역 데이터를 업데이트 하는 오버헤드와 새 버전을 위해 새 프로세스를 생성하는 오버헤드를 줄일 수 있다. 또한 데이터 영역에 저장된 대부분의 전역 데이터에 업데이트 없이 직접 접근 가능한 장점이 있다.

본 연구에서 제안한 방식을 사용하여 구현한 DSU 시스템을 x86 구조에서 리눅스를 수행시켜 ANSI C 문법으로 작성된 C 응용 프로그램들과 thttpd 서버 프로그램을 새 버전으로 업데이트 하도록 시험하였다. 그 결과 업데이트 이후에도 프로세스의 상태가 보존되며, 구 버전과 동일한 프로세스에서 새 버전의 프로그램이 정상적으로 동작함을 확인할 수 있었다.

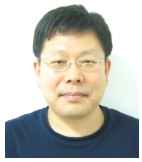
본 연구에서는 C 응용 프로그램을 동적으로 업데이트 하는 새로운 업데이트 방식을 제안하고 전체 DSU 시스템을 구현하였다. 이러한 결과는 새로운 업데이트 방식의 DSU 시스템 설계에 기여할 수 있을 것으로 기대된다. 또한 본 연구를 더욱 발전시켜 다양한 컴파일러가 인식하는 업데이트 프로그램의 분석 및 자동 생성을 위한 DSU 시스템 개발에 기여할 수 있을 것으로 기대된다.

참고문헌

[1] D. Gupta, and P. Jalote, "On-line Software Version Change Using State Transfer Between Processes," Software Practice and Experience, Vol. 23, No. 9, pp. 949-964, September. 1993.
[2] Jae Hong Cheon, Dea-Woo Park, "A Dynamic

Update Engine of IPS for a DoS Attack Prevention of VoIP," Journal of the Korea Society of Computer and Information, Vol. 11, No. 6, pp. 165-174, December. 2006.
[3] I. Neamtiu, M. Hicks, G. Stoyle and M. Oriol, "Practical Dynamic Software Updating for C, " Proceedings of Programming Language Design and Implementation, pp. 72-83, Ottawa, Canada, June 2006.
[4] I. Neamtiu and M. Hicks, "Safe and timely dynamic updates for multithreaded programs," Proceedings of Programming Language Design and Implementation, pp. 13-24, Dublin, Ireland, June 2009.
[5] K. Makris and R. A. Bazzi, "Immediate Multi-Threaded Dynamic Software Updates Using Stack Reconstruction," Proceedings of the 2009 conference on USENIX Annual Technical Conference, pp. 397-410, June 2009.
[6] K. Makris, "Whole-Program Dynamic Software Updating," PhD thesis, Arizona State University, December 2009.
[7] C. M. Hayden, E. K. Smith, M. Hicks and J. S. Foster, "State Transfer for Clear and Efficient Runtime Updates, In Third Workshop on Hot Topics in Software Upgrades, pp. 179-184, April 2011.
[8] C. M. Hayden, E. K. Smith, M. Denchev, M. Hicks and J. S. Foster, "Kitsune: Efficient, General-purpose Dynamic Software Updating for C," Proceedings of the ACM international conference on Object oriented programming systems languages and applications, pp. 249-264, October 2012.
[9] D. Gupta, P. Jalote, Senior Member, IEEE, and G. Baura, "A Formal Framework for On-line Software Version Change," IEEE Transactions on Software Engineering, Vol. 22, No. 2, pp. 120-131, February 1996.
[10] Linux Man Pages, Nov 2003, http://www.linuxmanpages.com/man3/
[11] V. Paxon, W. Estes and J. Millaway, Lexical Analysis With Flex, Edition 2.5.35, The Regents of the University of California, 2008, http://flex.sourceforge.net
[12] C. Donnelly and R. Stallman, Bison, Version 2.3, Free Software Foundation, Inc., 2006, http://www.gnu.org/software/bison
[13] Jutta Degener, ANSI C Yacc Grammar, 2011, http://www.quut.com/c/ANSI-C-grammar-y.html
[14] thttpd- tiny/turbo/throttling HTTP server 2.25b, Jul 2012, http://acme.com/software/thttpd/

저 자 소 개



신 동 하

1980년: 경북대학교 전자공학과 전자계산기전공 학사
1982년: 서울대학교 전자계산기공학과 석사
1994년: University of South Carolina 컴퓨터과학과 박사
1982년~1996년: 한국전자통신연구원 책임 연구원
1997년~현재: 상명대학교 컴퓨터과학부 교수
관심분야: 가상화, 임베디드 컴퓨팅, 논리 프로그래밍, 리눅스 및 윈도우 시스템 프로그래밍
Email: dshin@smu.ac.kr

김 지 현

2010년: 상명대학교 컴퓨터과학과 이학사
2013년: 상명대학교 일반대학원 컴퓨터과학과 석사
관심분야: 임베디드 시스템, 리눅스 시스템 프로그래밍, 운영체제, 모바일 시스템
Email: kim.jihyeon2@gmail.com