커널 자원 관리 기법 설계 및 구현
(Design and Implementation of Kernel Resource Management Scheme)

김 병 진 † 백 승 재 †† 김 근 은 ††† 최 종 무 ††††
(Byung Jin Kim) (Seung Jae Baek) (Keun Eun Kim) (Jong Moo Choi)

요 약 모듈은 동적으로 커널에 적재 가능한 오브젝트 파일로써 적재된 이후에는 커널의 권한으로 모든 자원에 대해 완벽한 접근 권한을 가진다. 따라서 잘못 작성된 모듈이나 혹은 정상적으로 작성된 모듈이라 할지라도 운영체제의 상황에 따라 비정상 수행되는 경우 시스템 전체의 안정성과 신뢰성에 치명적인 영향을 끼친다. 따라서 본 논문에서는 모듈이 발생시킬 수 있는 다양한 자원관련 문제를 해결하기 위해 커널 자원 보호자를 설계하였다. 커널 자원 보호자는 메모리, 주 번호, 워크 큐 등 운영체제가 관리하는 다양한 자원에 대한 보호를 제공한다. 제안된 기법은 리눅스 2.6.18에 실제 구현되었으며, 실험을 통해 본 논문에서 제안한 커널 자원 관리가 커널 자원을 효율적으로 보호하고 있음을 보였다.

키워드 : 모듈, 폴트 격리, 자원 보호 기법, 운영체제

Abstract Module is an object file that can be loaded into operating system dynamically and has complete privileged access to all resources in kernel. Therefore trivial misuses in a module may cause critical system halts or deadlock situations. In this paper, we propose Kernel Resource Protector(KRP) scheme to reduce the various problems caused by module. KRP provides protections of a variety of kernel resources such as memory, major number and work queue resource. We implement the scheme onto linux kernel 2.6.18, and experimental results show that our scheme can protect kernel resources effectively.

Key words : Module, Fault isolation, Resources protecting scheme, Operating system

1. 서 론

운영체제(Operating System)는 크게 모놀리틱 커널(Monolithic Kernel) 구조와 마이크로 커널(Micro Kernel) 구조로 구분 된다[1]. 모놀리틱 커널 구조의 운영체제에서는 운영체제가 제공하는 모든 기능이 단일 주소공간에서 수행된다. 따라서 시스템 호출이나 인터럽트 처리 시 부가적인 문맥 교환(Context Switching)을 유발하지 않으며, 마이크로 커널 대비 높은 성능을 기대할 수 있다. 그러나 운영체제에 의한 메모리 사용량이 상대적으로 많아질 수 있으며, 운영체제의 기능을 변경/추가 하는 경우 운영체제 전체의 재 컴파일 및 재부팅을 필요로 하는 단점을 가지고 있다.

반면 마이크로 커널은 스케줄링, 메모리 관리, 프로세스 간 통신 등 가장 핵심적인 요소만을 운영체제에 구성하고 그 외의 기능은 서버라 불리는 사용자 모드 프로세스를 통해 처리한다[2]. 따라서 운영체제 자체는 상대적으로 적은 메모리 공간을 차지하며, 제공되는 기능의 변경/추가를 위해 시스템 재부팅이 필요 없는 장점을 가진다. 반면 부가적인 문맥 교환으로 인한 성능저하를 초래할 수 있다.

리눅스(Linux)는 본래 성능 향상을 위해 모놀리틱 커널 구조를 택하고 있으나 모듈(Module)개념을 도입하여

· 이 논문은 2008년도 정부(교육과학기술부)의 재원으로 한국과학재단의 지원을 받아 수행된 연구임(No. R01-2008-000-12028-0)

† 정 회 원 : 벽산정보통신 연구원
suein1209@gmail.com
†† 정 회 원 : 단국대학교 컴퓨터학과
ibanez1383@dankook.ac.kr
††† 정 회 원 : 한국정보사회진흥원 u-응용사업팀 수석연구원
denver333@nfa.kr
†††† 종신회원 : 단국대학교 컴퓨터학부 교수
choijm@dankook.ac.kr

논문접수 : 2008년 4월 22일
심사완료 : 2009년 1월 23일

정보과학회논문지: 시스템 및 이론 제36권 제3호(2009.6)

마이크로 커널 구조의 장점을 취하고 있다. 리눅스의 모듈은 동적으로 적재(Load)하거나 제거(Un-load) 될 수 있으며, 적재된 이후에는 운영체제의 특권(Privilege) 모드에서 수행되므로 모든 자원에 대해 완벽한 제어 권한을 가진다. 디바이스 드라이버(Device Driver), 파일 시스템(File System), 실행 도메인(Executable Domains), 바이너리 포맷(Binary Format), 네트워크 레이어(Network Layer)등 운영체제의 다양한 기능을 모듈로써 제공할 수 있다[3].

리눅스는 전체 커널 소스 중 약 70%정도가 모듈 형태로 작성되어 있으며[4], 특히 임베디드 기기에 많이 적용되고 있는 리눅스의 특성상 수많은 개발자들이 자신만의 디바이스 드라이버를 모듈 형태로 제작하여 사용하고 있다. 이에 따라 검증되지 않은 모듈의 사용으로 인한 다양한 문제점이 야기 되고 있다[5]. 예를 들어 메모리와 같은 자원에 대한 비정상 사용은 운영체제의 자원 누수(Leak)를 야기하며, 락(Lock)과 같은 자원에 대한 비정상 사용은 시스템의 데드락(Dead-Lock) 상태를 발생시키며, 심지어 모듈의 잘못된 포인터 사용은 시스템을 다운시키거나 패닉(Panic) 상태로 만들 수 도 있다.

따라서 본 논문에서는 모듈 기법을 사용하는 운영체제에서 모듈이 시스템의 자원을 사용할 때 발생시키는 문제를 최소화시키기 위한 커널 자원 관리자(Kernel Resource Protector: 이하 KRP)를 제안한다. KRP는 우선 다음 세 가지에 중점을 두고 구현되었다. 첫째, 커널 내부에서의 잘못된 할당/해제로부터 메모리 자원을 보호한다. 둘째, 디바이스 드라이버의 잘못된 등록/해제로부터 주 번호 자원을 보호한다. 셋째, 바톰 하프(Bottom Half)를 이용하는 인터럽트 핸들러의 비정상적 사용으로부터 워크 큐 자원을 보호한다. 제안된 기법은 리눅스 커널 2.6.18에 실제 구현되었으며, 실험을 통해 본 논문에서 제안한 KRP가 모듈이 발생시키는 시스템의 자원 관련 문제를 효율적으로 해결하고 있음을 보였다.

본 논문의 구성은 다음과 같다. 2절에서는 관련연구에 대해 소개하며, 3절에서는 리눅스의 모듈과 문제점에 대해 논한다. 4절에서는 커널 자원 관리자의 설계에 대해, 5절에서는 커널 자원 관리자의 구현에 대해 설명한다. 6절에서는 구체적인 실험결과에 대해 언급하며 마지막 7장에서는 결론 및 향후 연구 방향에 대해 소개한다.

2. 관련연구

소프트웨어의 신뢰성 향상을 위한 다양한 연구가 수행된 바 있다. 그중 첫 번째 형식은 가상 머신(Virtual Machine)이나 마이크로 커널(Micro Kernel)에 기반 한 연구이다. 참고문헌 [6]의 연구에서는 리눅스의 디바이스 드라이버의 신뢰성 향상을 위해 KVM위에 각 드라이버마다 VM Driver Stub를 두고 드라이버를 고립시키는 기법을 소개하였다. 그러나 이 기법은 인텔 처리기의 VT 지원이 있어야 한다는 단점을 가지고 있다. 또한 [7]의 연구에서는 디바이스 드라이버의 수정 없이 가상 머신 모니터를 사용하여 드라이버를 고립시키는 기법을 제안하였다. 이 기법은 별도의 하드웨어 지원이 있어야 하는 것은 아니지만 정보교환을 위해 메시지 패싱을 사용하므로 많은 오버헤드가 존재하는 단점이 있다. 또한 [8]의 연구에서는 가상화 기술의 일종인 Xen을 이용한 기법을 소개하였다. Xen 상위에 일반 Guest OS와 논문에서 소개된 Proxos OS를 동시에 수행시키고, 특별한 보호가 필요 없는 일반 소프트웨어는 일반 Guest OS에서 수행시키며, 보안이 필요하거나 신뢰성을 높이기 위한 별도의 노력이 필요한 소프트웨어는 Proxos OS상에서 수행시키는 기법을 소개하였다. 한편 [9]와 같은 마이크로 커널 기반 보호 기법도 제안된 바 있다. 그러나 윈도우나 리눅스처럼 일반적인 운영체제에는 적용이 불가능하다는 단점이 있다.

두 번째는 신뢰성과 보안성 제공을 위한 새로운 운영체제에 대한 연구이다. [10,11]은 Choices와 Singularity는 신뢰성과 보안을 위한 새로운 운영체제를 제안하였다. 그러나 이들은 기존 운영체제의 신뢰성을 높이기 위해 사용될 수 없다는 문제점이 있다. 또한 [12]의 연구에서는 사용자 레벨이 존재 하지 않는 SPIN운영체제를 개발하였다. 시스템은 모두 커널 영역으로 구성되고, 모든 태스크를 실시간 태스크로 간주하고 빠른 수행을 가능케 한다. 이러한 구조에서 안정적인 동작을 위해 SPIN 운영체제는 modular3 라는 언어를 통해 태스크를 작성한다.

이외에도 [13]의 연구는 Nooks라는 기법을 제안하였다. 이 기법은 fault tolerance가 아닌 fault resistance 기법을 제안하였다. 구체적으로 각 드라이버를 light-weight kernel protection domain에서 수행시키며, 메모리의 직접 접근을 제한한다. 커널에서 디바이스 드라이버에 접근하는 경우나 혹은, 디바이스 드라이버에서 커널에 접근하는 경우 그 경로를 제한함으로써 안정성을 얻을 수 있는 기법을 제안하였다. 구체적으로 XPC (Extension Procedure Call)와 쉐도우 드라이버(Shadow Driver)를 소개하였다. 모든 디바이스 드라이버는 XPC를 통해 정보를 주고받으며, 디바이스 드라이버에 문제가 발생 한 경우 쉐도우 드라이버를 통해 안정 상태로 복원을 시도한다. 약 98%의 linux fault를 자동으로 복구해내지만, 악의적인 코드에 의해 쉽게 우회 가능하므로 악의적인 드라이버의 동작을 막을 수 없으며 드라이버를 수행시켜주는 커널 모드 extensions이 오염된 경

우 이를 막을 방법이 없다는 단점을 가진다. 한편 [14]의 연구에서는 프로그램의 특정 위치에 소프트웨어 가드를 삽입하여 프로그램 수행을 제어한다. 또한 프로그램을 정적으로 분석하여 정상적인 프로그램의 흐름 그래프를 유지한다. 그러나 존재하는 운영체제의 모든 코드를 분석하여 소프트웨어 가드를 삽입하는 것은 대단히 시간 소모적인 작업이다.

특정 하드웨어의 지원을 통해 신뢰성을 높이려는 시도도 진행되었다. [15]에서는 X86 구조의 컴퓨터와 리눅스 운영체제에서 드라이버를 고립시킬 수 있는 방법을 제안하였다. 구체적으로 디바이스 드라이버는 링 레벨 1, 2에서 수행시키며, 커널 모드 메모리에 대한 접근을 막는다. 이를 위해 드라이버와 커널간의 별도 API를 제공한다. 그러나 기존 드라이버가 수정된 커널과 지장 없이 돌기 위해서는 재 컴파일을 필요로 하며, Nooks처럼 드라이버 자체가 오염된 경우 이를 막기 어렵다. 한편 [16]의 연구에서는 x86 구조의 세그먼트 기법을 이용하여 call stack을 보호할 수 있음을 보였다. 그러나 이 기법 역시 적절한 하드웨어의 지원을 필요로 하는 단점을 가지고 있다.

그리고 [17]의 연구는 소프트웨어의 버그로 인해 비정상 결과가 초래되는 경우 환경변수를 변경한 후 재실행시킴으로서 소프트웨어의 저항력(resistance)을 높이는 기법을 소개하였다. 그러나 커널 레벨 소프트웨어(모듈)에 대한 고려가 없다. [18]의 연구는 기본적인 보호 정책의 강화를 위해 디자인된 시스템으로써 비교적 높은 성능을 보여준다. 그러나 아직 메모리 레이아웃이나 시그널, 멀티쓰레딩과 같은 다른 시스템의 관점에서 본다면 여러 가지 제약사항을 만족시키기 어렵다.

[19]의 연구에서는 소프트웨어가 발생시키는 문제를 특정 소프트웨어로 제한하기 위한 기법을 소개하였다. 구체적으로 Fault Domain에 코드와 데이터를 두고, Fault Domain 밖으로 실행을 옮기거나, 외부의 데이터를 접근하지 못하도록 하였다. Fault Domain 간의 통신은 RPC를 이용하였다.

[20]의 연구에서는 커널 내 일관성을 향상시켜 신뢰도를 높이기 위한 연구를 소개하였다. 대용량의 메모리를 할당받은 뒤 악의적인 목적으로 해제하지 않는 모듈로 인한 시스템의 메모리 누수를 해결하기 위한 기법을 소개하였다. 본 논문은 [20]의 연구를 확장/개선한 것이다. [20]의 연구에서는 메모리 누수 문제만을 해결하였으나 본 논문에서는 디바이스 드라이버 등록 과정의 비일관성 문제와 워크 큐 실행 시 발생되는 일관성 문제까지 해결하였으며, 이를 통해 커널 자원 보호 기법이 다른 자원에 대해서도 적용될 수 있음을 시사한다는 점에서 기존 연구와의 차이점이 있다.

3. 리눅스의 모듈과 문제점

리눅스의 모듈은 커널이 실행되고 있는 중에 동적으로 적재 혹은 제거 될 수 있는 오브젝트 파일(Object File)이다. 모듈의 적재는 'insmod' 명령어를 이용하거나, 'modprobe'를 통해 필요시 자동으로 로딩 할 수 있으며 모듈의 제거는 'rmmod' 명령어를 이용한다.

'insmod' 명령어는 바이너리 프로그램으로써 내부적으로는 리눅스가 제공하는 sys_init_module() 함수를 호출하여 사용자가 원하는 모듈을 커널에 적재시켜준다. sys_init_module() 함수는 해당 모듈을 커널 내부에 적재하기 위한 공간을 할당 받고, 모듈을 관리하기 위해 필요한 커널 내부 자료구조를 구축한다. 그런 뒤 모듈에서 참조하는 다양한 심볼(커널 내부 함수나 자료구조 등)에 대한 재배치(Relocation) 작업을 수행하며, 마지막으로 모듈 작성자가 작성한 초기화 함수를 호출한다.

'rmmod' 명령어 역시 바이너리 프로그램으로써 내부적으로는 리눅스가 제공하는 sys_delete_module() 함수를 호출하여 모듈을 제거시킨다. sys_delete_module() 함수는 해당 모듈을 사용 중인 태스크가 있는지 검사한 후, 제거해도 되는 경우 커널 내부에 유지되고 있던 자료구조에 대해 적절한 처리를 하고, 모듈 개발자가 작성한 종료 함수를 호출한다.

적재된 모듈은 커널 내부에 존재하는 모든 자원을 사용할 수 있다. 예를 들어, kmalloc()이나 vmalloc() 함수를 사용하여 메모리 자원을 할당 받아 사용할 수 있고, request_irq()등의 함수를 통해 IRQ 자원을 사용하는 것이 가능하다. 또한 디바이스 드라이버인 경우 register_XXXdev() 같은 함수를 통해 시스템 내에 고정 개수 존재하는 주 번호 자원을 사용할 수도 있으며, 동기화를 위해 락 자원을 획득하여 사용할 수 도 있다.

적재된 모듈은 커널의 일부로서 특권 모드에서 동작하며 일체의 보호 장치 없이 모든 자원에 접근한다. 이로 인해 자원에 대한 잘못된 사용은 자원의 소실을 가져오며, 심지어 시스템의 전체의 불안정을 가져올 수 있다. 예를 들어 메모리 할당 함수를 호출하여 메모리를 할당 받은 뒤 정상 반환하지 않은 경우 이는 시스템의 메모리 누수를 유발한다. 디바이스 드라이버인 경우 register_XXXdev() 함수를 통해 주 번호 자원을 획득한 후, unregister_XXXdev()함수를 통해 적절히 반납하지 않은 경우 해당 주 번호는 시스템의 재부팅 전까지 사용 불가능 하게 되어 특정 디바이스 드라이버를 사용할 수 없게 만든다. 한편 인터럽트 핸들링 중 바톰 하프(Bottom Half)를 이용해 추후 작업을 진행하기 위해 리눅스는 워크 큐(Work Queue)를 메카니즘을 제공한다. 이때 워크 큐에 등록해 놓은 함수가 수행되기 이

전에 해당 모듈이 제거되는 경우 시스템은 패닉 상태에 빠진다.

이러한 문제의 원인은 크게 두 가지로 구분해 볼 수 있다. 첫째, 모듈 개발자의 사소한 실수로 인한 것일 수 있다. 둘째, 정상적으로 작성된 모듈이라 할지라도 현재 시스템의 상태에 따라 비정상적인 결과를 유발할 수 도 있다. 이 경우 그 심각성은 더욱 크다 하겠다.

4. 커널 자원 관리자 설계

본 논문에서는 모듈 기법을 사용하는 운영체제에서 모듈이 시스템의 자원을 사용할 때 발생시키는 문제를 최소화시키기 위한 KRP를 제안 한다. 현재 KRP의 설계는 우선 메모리 자원, 주 번호 자원 그리고 워크 큐 자원의 일관성 문제를 주로 고려하였다.

4.1 메모리 자원 보호를 위한 설계

모듈에서 주로 사용하는 메모리 자원 할당 함수는 kmalloc()과 vmalloc()이다. kmalloc() 함수는 할당된 공간이 물리적으로 연속임을 보장하는 반면, 한 번에 할당 할 수 있는 최대 크기는 128KB이다. 한편 vmalloc()은 할당된 공간이 물리적으로 연속임을 보장하지 않지만 최대 할당 크기에 대한 제약이 없다. 본 논문에서 제안된 KRP는 kmalloc()함수와 vmalloc()함수 모두 아무 차이 없이 지원가능하다. KRP 설계를 설명하기 위해 우선 kmalloc()함수를 예로 들도록 한다.

구체적인 설계 내용은 그림 1과 같다. 모듈에서 kmalloc()함수를 통해 메모리 자원 할당을 요청 한다(그림 1의 ①). 그러면 슬랩 할당자는 메모리 할당을 요청한 모듈이 누구이며, 얼마만큼의 공간을 할당하는지에 대한 정보를 기록한다(그림 1 ②). 할당받은 공간을 해제 하기 위해서 모듈은 kfree()함수를 호출한다. 이때 커널은 현재 해당 메모리에 대한 해제작업을 수행한다(그림 1 ③). 그런 뒤 메모리를 할당할 때 유지했던 정보를 삭제한다(그림 1 ④).

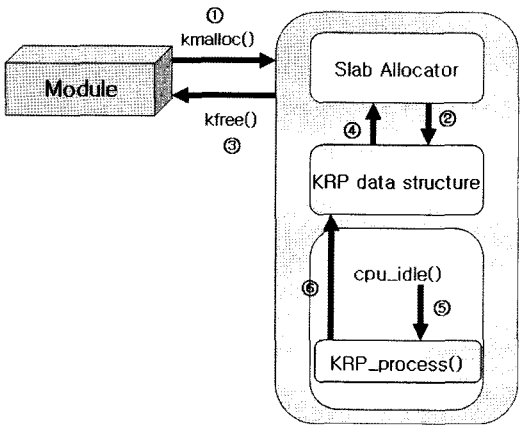

그림 1 KRP의 메모리 자원 보호 구조

모듈에서 kfree()함수를 통해 메모리를 해제하지 않았거나 혹은 모듈의 버그 혹은 시스템의 문제 등으로 인해 정상적인 해제 작업이 수행되지 않는 경우 메모리 누수가 발생한다. 이를 해결하기 위해 시스템이 유휴(idle) 상태일 때 KRP_process()함수를 호출하도록 하였다(그림 1 ⑤). KRP_process()함수는 kmalloc()과 kfree() 함수 호출 시 유지해 오던 정보를 바탕으로 현재 활성화 되어 있지 않은 모듈이 메모리 자원을 사용하는 것으로 판단되는 경우 할당되어있는 메모리를 해제 시켜 메모리 누수를 방지 한다.

4.2 주 번호 자원 보호를 위한 설계

그림 2는 디바이스 드라이버가 주 번호 자원을 사용하는 일반적인 예이다. 모듈 형태로 제작된 디바이스 드라이버가 커널 내에 적재되면 각 장치에 적합한 등록 함수를 호출하여, 커널 내에 유지되고 있는 자료구조에 등록을 하게 된다. 그림 2에서는 흔히 사용되고 있는 문자(Character) 장치 디바이스 드라이버를 등록하는 예를 들었다.

문자 디바이스 드라이버는 register_chrdev() 함수를 통해 커널 내의 chrdevs 테이블에 필요한 정보를 등록한다. 이 함수는 첫 번째 인자로 주 번호를 받는데 이는 chrdevs 배열의 인덱스 값이며 0으로 지정된 경우 커널은 비사용 중인 인덱스 번호에 드라이버의 정보를 등록하고 이 번호를 반환한다. 이후 드라이버의 사용이 완료되어 적재 돼 있는 모듈을 제거하면 chrdevs 배열에 등록되어있던 디바이스 드라이버 정보 역시 제거 해주어야 한다. 그림 2에서는 chrdevs 테이블에 등록하였으므로 unregister_chrdev()함수를 호출하여 커널 내에 있는 해당 자료구조 내용을 제거해야 한다. 이를 통해 해당 주 번호 자원은 다른 디바이스 드라이버에서 다시 사용할 수 있게 된다.

하지만 모듈이 커널 내에 삽입되어 디바이스 드라이버 등록 과정을 거치고 난 뒤, 버그가 발생하거나 혹은

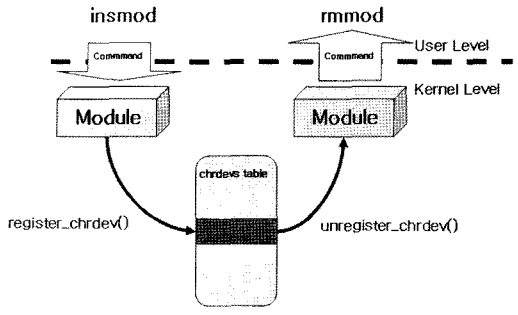

그림 2 디바이스 드라이버의 주 번호 자원 사용 예

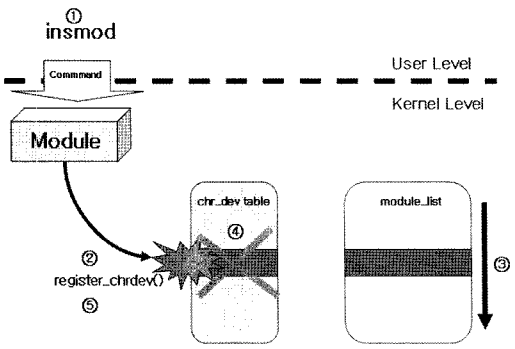

그림 3 KRP의 주 번호 자원 보호 구조

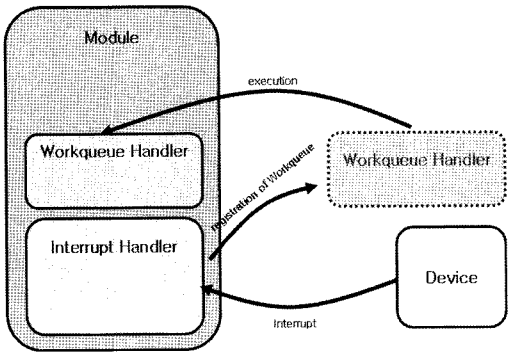

그림 4 워크 큐 자원 사용 예

시스템 문제로 인해 정상적인 등록 해제 과정을 거치지 못하게 되면 사용 중이던 디바이스 드라이버의 주 번호 자원은 시스템의 재부팅 이전까지 다른 어떤 모듈도 사용할 수 없게 된다. 이 문제를 해결하기 위해 본 논문에서는 디바이스 드라이버를 등록하는 과정에서 문제가 발생한다면 문제의 발생 원인을 확인하고, 이 문제가 비일관성 문제인지를 검토 후 디바이스 드라이버를 등록할 수 있도록 설계하였다.

구체적인 설계 내용은 그림 3과 같다. 모듈 형태로 제작된 디바이스 드라이버를 커널 내에 삽입한다(그림 3 ①). 이후 디바이스 드라이버 등록 함수를 통해 등록되는 과정에서 사용 불가능한 주 번호 자원을 요청하는 경우 함수 호출 과정에서 에러가 발생한다.(그림 3 ②). 이때 에러가 디바이스 드라이버 등록 과정에서 발생한 문제인지 확인한다(그림 3 ③). 확인하는 방법은 시스템에 적재되어 있는 모든 모듈이 이중 연결 리스트(Double Linked List)로 유지되고 있는 module_list를 검사하여 등록하려는 테이블에서 비일관성 문제를 야기 시키는 요소가 있는지 확인한다. 만약 시스템에 적재되어 있지 않는 모듈이 해당 주 번호 자원을 사용하고 있는 것으로 판단되면 해당 요소를 제거한다(그림 3 ④). 이후 등록하려했던 드라이버를 등록시켜줌으로써 문제를 해결한다(그림 1 ⑤).

4.3 워크 큐 자원 일관성 보호를 위한 설계

인터럽트 핸들러 내에서 어떠한 작업들을 바로 수행하고(Top Half), 어떠한 작업들을 지연된 작업(Bottom Half)으로 수행 시킬지는 인터럽트 핸들러 작성자에 의해 결정된다. 리눅스는 지연된 작업을 위한 다양한 메커니즘을 도입하였으나 현재 커널 버전 2.6부터는 워크 큐로 통합 되어 가고 있다. 그림 4는 그 사용 예를 보이고 있다.

그림 4는 모듈 형태의 디바이스 드라이버가 인터럽트를 처리하고 있다고 가정한다. 이때 장치에서 인터럽트가 발생하였고 이 인터럽트를 처리하기 위해 기 지정된 디바이스 드라이버의 인터럽트 핸들러가 수행된다. 인터럽트 핸들러가 실행되는 동안 지연시켜 처리할 작업을 워크 큐에 넘기게 되고 특정 시간의 지연 뒤 워크 큐에 등록시킨 함수(Workqueue Handler)가 실행되면서 인터럽트 시에 다 처리하지 못한 지연된 처리를 마저 실행하게 된다.

하지만 워크 큐의 사용은 다음과 같은 문제를 발생시킬 수 있다. 드라이버에 의해 등록된 워크 큐 함수는 커널 쓰레드의 형태로 실행 대기 상태가 된다. 이때 버그로 인해 혹은, 시스템의 문제로 인해 디바이스 드라이버 모듈이 커널 내에서 제거 된다면 워크 큐 함수가 가리키는 주소는 무효한 주소가 되므로 전체 시스템이 다운되는 상황을 초래하게 된다. 이 문제를 해결하기 위해 본 논문에서는 모듈 형태의 디바이스 드라이버가 워크 큐에 지연된 작업을 등록 한 후 시스템에서 제거되는 상황을 차단함으로써 워크 큐 함수가 원활히 동작할 수 있도록 설계하였다.

구체적인 설계 내용은 그림 5와 같다. 우선 커널 내부에서 모듈을 관리하기 위해 모듈 별로 할당/유지되는

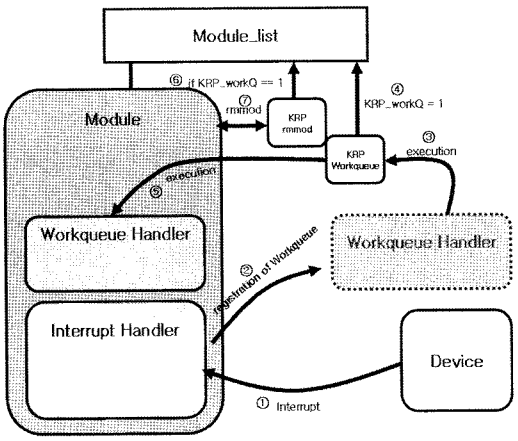

그림 5 KRP의 워크 큐 자원 보호 구조

구조체에 새로운 변수 하나를 추가한다. 이 변수의 이름은 KRP_workQ 이고 이 변수는 차후 모듈이 불시에 제거되는 상황을 막게 되는 역할을 한다.

인터럽트를 사용하는 모듈형태의 디바이스 드라이버가 적재되는 상황을 예로 들어보자. 장치에서 인터럽트가 발생하면 기 등록되어 있는 디바이스 드라이버의 인터럽트 핸들러가 동작 된다(그림 5 ①). 인터럽트 핸들러는 필수적인 작업(Top Half)을 우선 실행한다. 그리고 나머지 작업들을 지연된 처리로 워크 큐에 등록시킨다(그림 5 ②). 특정 시간이후 워크 큐가 실행 될 때 모듈에 존재하는 워크 큐 핸들러를 실행 시키는 것이 아니라 KRP_Workqueue 루틴을 실행한다(그림 5 ③). KRP_Workqueue 함수는 module_list에 의해서 관리되고 있는 module 구조체의 KRP_workQ 변수 값을 변환해준다(그림 5 ④).

만약 워크 큐가 실행되기 전에 버그 혹은 임의로 현재 핸들러 코드가 들어간 모듈을 제거 하는 것을 방지하기 위해 'rmmod' 명령이 호출하는 sys_delete_module() 함수 내부에 KRP_rmmod() 함수를 호출하도록 수정하였다. KRP_rmmod()함수는 현재 모듈이 안전한 상태인지를 검사하게 된다(그림 5 ⑦). 구체적으로 검사하는 내용은 현재 제거 하려는 모듈이 워크 큐 비일관성 문제를 갖고 있는 모듈인지 아닌지를 검사한다(그림 5 ⑥). 만약 KRP_worqueue를 통해 KRP_workQ 변수의 수정이 이루어졌다면 모듈을 안전하게 제거 시켜준다. 하지만 만약 KRP_workQ 변수의 수정이 이루어지지 않았다면 현재 모듈의 제거에 문제가 있다는 경고를 하고 더 이상 모듈이 제거되는 루틴을 실행하지 못하게 함으로써 커널의 자원을 보호 한다.

5. 구 현

본 논문에서 제안하는 KRP는 리눅스 커널 2.6.18상에서 구현되었다. 그리고 제안된 기법의 구현은 자세한 설명을 위해 3절로 구분하여 설명한다.

5.1 메모리 자원 보호 구현

모듈이 처음 적재될 때 그림 6과 같이 내부적으로 sys_create_module()을 호출하며, 이는 다시 sys_init_module()을 호출하게 된다. 이때 본 논문에서는 각 모듈이 갖는 고유한 아이디를 초기화 하도록 하였다. 그리고 include/linux/slab.h를 수정하여, 모듈의 아이디와 메모리를 할당해 사용할 경우 할당받은 포인터를 저장할 필드를 가지는 KRP_M 구조체를 커널 내부에 추가하였다. 한편 mm/slab.c를 수정하여, 모듈이 kmalloc()으로 메모리를 할당받을 때 모듈의 이름과 할당받은 메모리의 포인터를 위 KRP_M 구조체에 저장해 놓도록 한다. kfree()시엔 기 작성한 구조체에서, 해제하는 메모

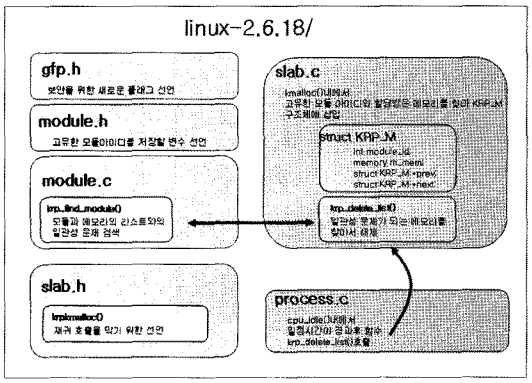

그림 6 KRP의 메모리 자원 보호 구현

리에 해당되는 부분을 삭제하도록 한다. 다음으로 커널이 idle state가 되어 arch/cpu별디렉토리/kernel/process.c 내의 cpu_idle() 함수를 호출하게 되면, 작성해 두었던 구조체를 검색해 나간다. 현재 적재 되어 있지 않은 모듈의 이름이 구조체 상에 존재한다면, 이는 정상적으로 kfree()되지 않는 메모리가 있음을 의미한다. 따라서 해당 메모리를 kfree()시켜준 뒤 구조체를 삭제한다. cpu_idle()이 호출될 때 마다 불필요하게 여러 번 구조체 검색이 생기는 것을 막기 위해 변수를 선언하여 이 변수가 특정 값, 즉 cpu_idle()이 호출된 뒤 일정 시간이 지난 뒤에만 작업이 이뤄지도록 구현하였다.

5.2 주 번호 자원 보호 구현

모듈 형태의 드라이버는 적재 시 디바이스 드라이버를 등록하는 함수를 호출한다. 현재의 구현에서는 문자 장치를 대상으로 하였으며 추후 동일한 메커니즘으로 다른 종류의 디바이스 드라이버에 적용하는 것도 가능하다. 문자 장치를 등록하는데 사용하는 register_chrdev() 함수가 사용된다. 이 함수가 호출될 때 본 논문에서는 fs/chr_dev.c를 수정하여 장치 등록 시 발생되는 문제를 해결하였다. 모듈에서 호출하는 register_chrdev() 함수는 내부적으로는 chr_dev.c 내에 __register_chrdev_region() 함수를 호출함으로써 디바이스 드라이버를 등록하는 과정을 거친다. 이때 그림 7과 같이 장치가 등록되는 루틴 중에 다음과 같이 두 함수가 실행되도록 수정하였다.

첫째, KRP_unregister_chrdev() 함수는 현재 등록하려는 모듈이 일관성 문제가 있는 디바이스 드라이버인지를 찾게 된다. 구체적으로 일관성 문제가 있는 모듈인지를 확인하기 위해 커널의 전역 변수인 chrdevs 배열과 역시 전역 변수인 module_list를 검색한다. 만약 chrdevs 테이블 내에 있는 모듈이 module_list에 없다면 이는 일관성 문제를 의미한다. 둘째, 일관성 문제가 확인되면 KRP_init_chrdev_entry() 함수를 호출하게 되

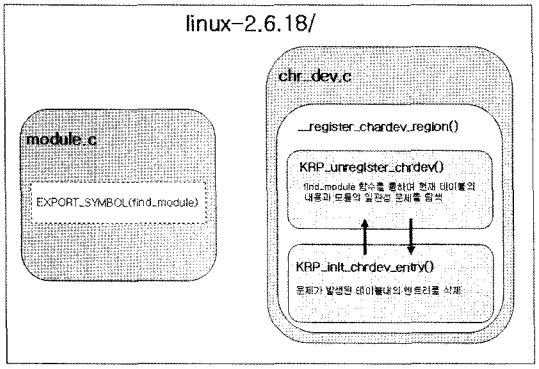

그림 7 KRP의 주 번호 자원 보호 구현

는데 이 함수에서는 문제가 발생된 문자 장치 테이블 엔트리를 삭제한다.

5.3 워크 큐 자원 일관성 보호 구현

디바이스 드라이버에서 워크 큐를 등록할 때는 보통 workqueue.c 파일 내에 존재하는 schedule_delayed_work()함수를 호출한다.

이때 워크 큐 자원을 보호하기 위한 본 논문의 구현은 그림 8과 같다. 본 기법에서는 schedule_delayed_work()함수의 실행 시 KRP_check_point 변수를 '0'으로 세팅한다. 이 변수는 include/module.h안에 module 구조체 내에 선언되어 있다. 이 변수는 추후 모듈이 제거 될 때 제거하여도 안전한지를 판단하는 기준이 된다. 모듈은 제거될 때 sys_delete_module()함수를 호출하는데, 이때 krp_check_work_setting() 함수를 호출하도록 하여 현재 제거 되려는 모듈이 제거해도 안전한지 아닌지를 판단한다. 만약 안전하다면 제거 루틴을 계속 실행하고 안전하지 않다면 모듈 제거 루틴을 취소한다.

6. 실험결과

6.1 메모리 자원 보호 실험 결과

우선 현재 시스템의 메모리 상황을 확인하기 위해 그림 9와 같이 cat명령어를 통해 /proc/meminfo 엔트리를 확인 해 보았다.

그런 뒤 그림 10과 같이 KRP의 메모리 자원 보호 기능의 동작을 실험하기 위한 'TestDrv.ko' 모듈을 커널 내에 삽입하였다. 이 모듈은 약 1KB의 메모리 할당을 3000번 반복한 뒤 해제하지 않는 테스트 용도의 모듈이다. 그런 뒤 시스템의 메모리 상태를 확인해 보았다.

이때 동적으로 메모리를 할당 받은 모듈을 제거하게 되면, 커널 내에 모듈에서 할당 받은 동적 메모리는 해제되지 않은 상태이고 따라서 메모리 누수를 발생 시키

```
[root@localhost Paper]# uname -a
Linux localhost.localdomain 2.6.18 #39 SMP Mon Mar 10 06:48:25 KST 2008 i686 i68
6 i386 GNU/Linux
[root@localhost Paper]# cat /proc/meminfo
MemTotal:       659900 kB
MemFree:        282040 kB
Buffers:         46728 kB
Cached:         179596 kB
SwapCached:          0 kB
Active:         201596 kB
Inactive:       149328 kB
HighTotal:           0 kB
HighFree:            0 kB
LowTotal:       659900 kB
LowFree:        282040 kB
SwapTotal:     1052248 kB
SwapFree:      1052248 kB
Dirty:               4 kB
Writeback:           0 kB
AnonPages:      124604 kB
Mapped:          35360 kB
Slab:            14524 kB
PageTables:       2696 kB
NFS_Unstable:        0 kB
```

그림 9 모듈 적재 전 메모리 자원 사용 상황

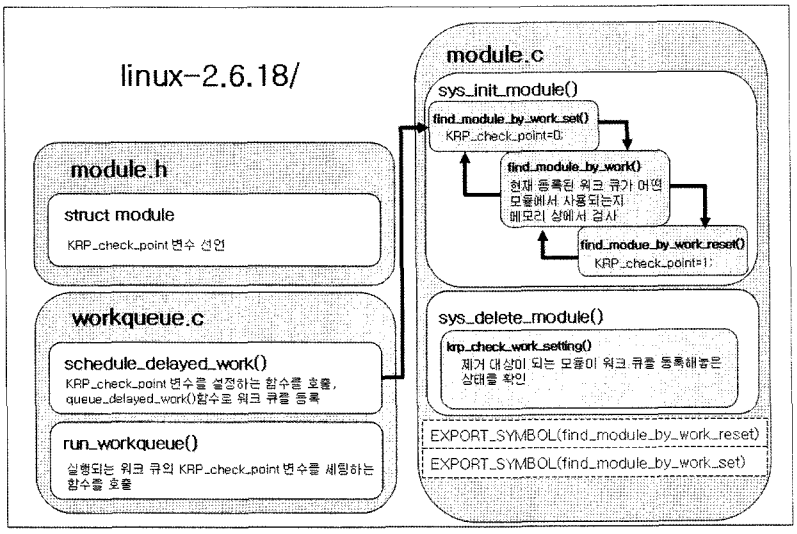

그림 8 KRP의 워크 큐 자원 일관성 보호 구현

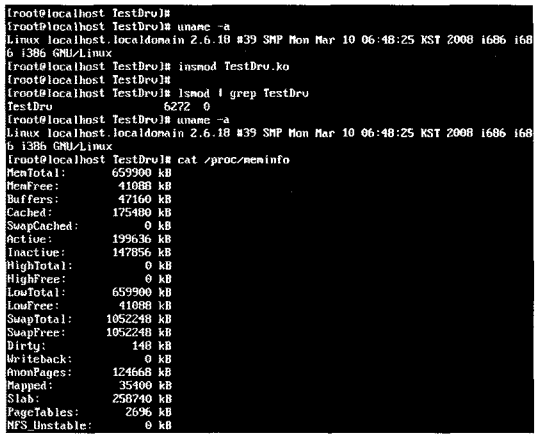

그림 10 동적 메모리 할당 후 커널 내의 메모리 용량

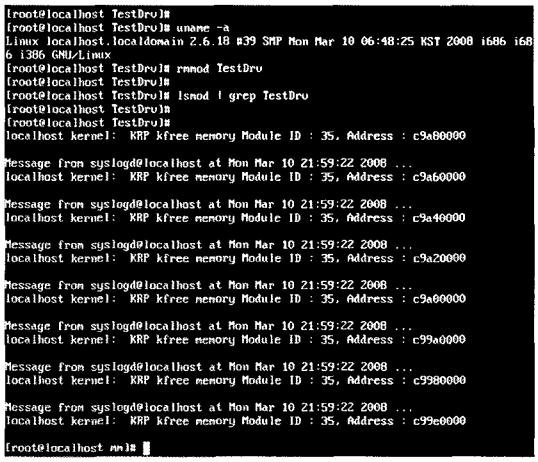

그림 11 모듈 제거

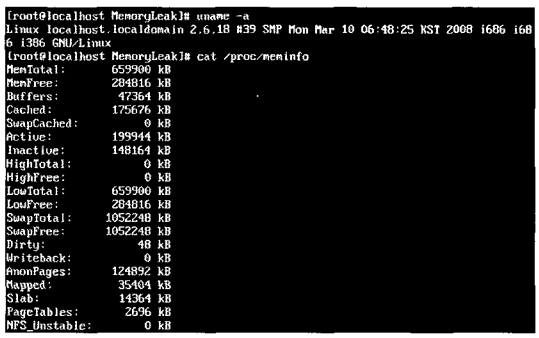

그림 12 메모리 복구 후 결과

게 된다. 그러나 본 논문에서 제안한 KRP를 통해 메모리 누수를 복구하는 결과를 그림 11에 보였다. 임의의 시간이 지난 뒤 cpu_idle()함수가 수행되고 결과적으로 본 논문에서 구현한 KRP가 동작되어 동적으로 메모리를 복구하게 된다.

이때 KRP가 모든 메모리 누수를 복구하고 난 뒤, 안정화된 상태의 메모리 사용량을 그림 12에 보였다. 실험을 시작하기 이전의 상태대로 메모리를 복구하였음을 확인 할 수 있다.

6.2 주 번호 자원 보호 실험 결과

KRP를 통해 주 번호 자원이 보호 되고 있는지를 실험하기 위한 디바이스 드라이버를 모듈 형태로 제작하여 적재시켰다. 작성한 모듈은 문자 디바이스 드라이버로써 장치의 이름, 주 번호는 'calldev', 240번으로 할당하여 실험하였다. 이 모듈은 KRP가 주 번호 자원을 효율적으로 보호하는지 확인하기 위한 테스트 모듈로써, 모듈 제거 시 unregister_chrdev() 함수를 통해 적절한 종료 작업을 수행하지 않는다. 그림 13에서는 이 테스트 모듈(pseudoDrv.ko)을 커널 내에 삽입하였다가 잠시 뒤 드라이버를 제거하는 모습을 보여준다.

이후 디바이스 드라이버의 사용이 요구되어 다시 드라이버를 적재하려 시도하였지만 "-1 Device or resource busy"경고를 출력하고 커널 내에 드라이버를 삽입하는 과정을 중단하게 된다. 이는 주 번호 자원의 일관성 불일치로 인해 자원의 낭비가 되고 있음을 나타낸다.

그러나 본 논문의 KRP가 적용된 커널에서의 실험결과는 그림 14와 같다. 그림 13과 동일한 모듈을 적재->제거 후 재 적재 시도를 해 보았다. 실험결과 그림 13의 일반 커널과 달리 본 논문의 KRP 커널은 주 번호 자원의 일관성 문제를 효율적으로 해결하여 이를 복구해 주었음을 알 수 있다.

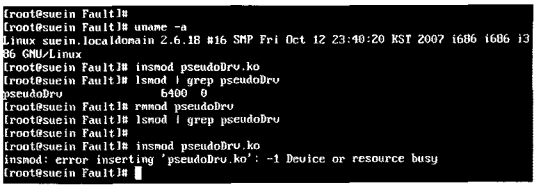

그림 13 주 번호 자원 문제

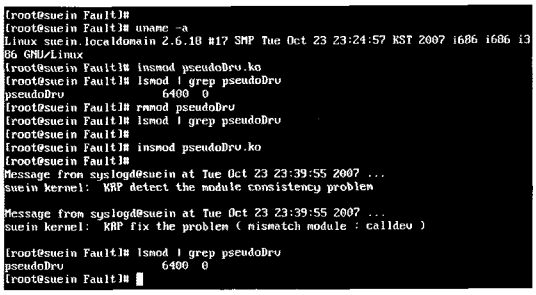

그림 14 주 번호 자원의 보호 구현 실험

6.3 워크 큐 자원 일관성 보호 실험 결과

그림 15는 워크 큐가 동작되는 일반적인 모습을 보이고 있다. 이 모듈에 의해 스케줄 된 워크 큐는 단순한 문자열 두 줄을 출력하고 별다른 동작을 하지 않는 테스트 용도의 모듈이다. 하지만 900ms시간동안 지연되어 실행되었으며, 올바르게 단순 문자열 두 줄이 출력되는 것을 확인할 수 있다. 이때 지연된 워크 큐 처리가 미처 끝나기 전에 시스템의 문제로 혹은 버그로 인해 드라이버가 제거된다면 시스템은 정지하게 된다.

이러한 문제점을 본 논문에서 제안한 KRP를 통해 해결한 결과를 그림 16에 보이고 있다.

워크 큐를 등록시킨 드라이버가 커널 내에 정상적으로 등록되어 있다. 이후 지연된 워크 큐가 등록된다. 하지만 지연된 워크 큐가 실행이 완료되기 전에 제거되려 한다. 그러면 KRP에 의해 제거되려 하는 모듈이 등록한 워크 큐가 완벽히 실행되었는지 검사한다. 하지만 워크 큐가 아직 실행되지 않은 상태이기 때문에 제거될 수 없음을 문자열 "Cannot remove Module"로 알리고 제거 루틴을 취소한다. 이후 등록된 워크 큐가 완벽히 실행되어 문자열을 문제없이 출력하는 것을 확인하였다.

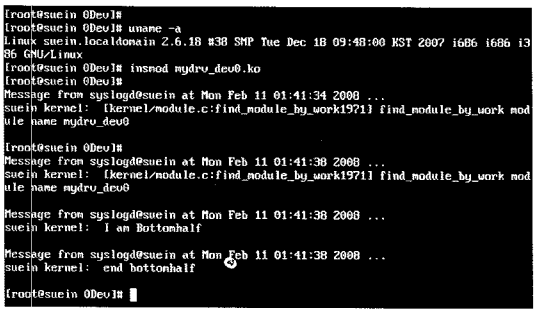

그림 15 큐의 실행

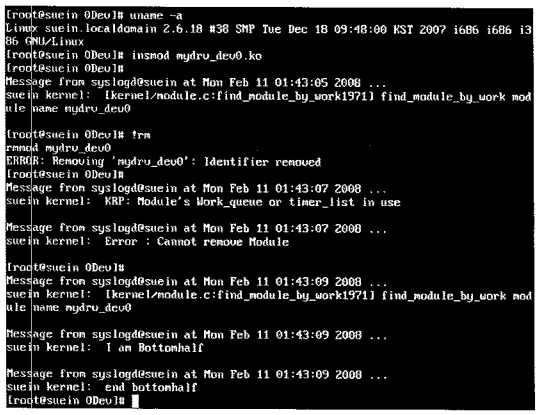

그림 16 워크 큐 실행 시 발생하는 비일관성 해결 구현

7. 결 론

본 연구는 모듈 기법을 사용하는 운영체제에서 모듈의 개념과 동작 구조에 대해 살펴보았다. 또한 모듈로 인해 발생 가능한 문제점에 대해 살펴보았으며, 이를 해결하기 위한 기법으로 KRP(Kernel Resource Protector)를 제안하였다. 제안된 기법은 리눅스 커널 2.6.18에 실제 구현되었으며 현재의 구현에서는 모듈로 인한 대표적 문제점들인 메모리 누수 문제, 디바이스 드라이버 등록 과정의 비일관성 문제, 워크 큐 실행 시 발생되는 일관성 문제를 주로 다루었다. 또한 실험을 통해 본 논문에서 제안한 KRP가 효율적으로 모듈에서 발생 가능한 문제점을 처리하여 커널의 자원을 보호하고 있음을 보였다. 모듈로 인해 발생 가능한 이러한 문제점들은 운영체제의 가용성을 저해하는 심각한 요소이므로 그 필요성이 더욱 크다 하겠다.

향후 본 연구는 모듈에서 발생 가능한 다양한 문제점을 정의하고 해결할 것이며, 나아가 모듈의 문제점을 그 유형에 관계없이 정형화시키고 이를 커널과 격리시킬 수 있는 방향으로 확장해 나갈 것이다.

참 고 문 헌

[1] W. Stalling, Operating Systems: Internals and Design Principle, 5th ed., Pearson Prentice Hall, 2004.

[2] J. Liedtke, "On Microkernel Construction," ACM SIGOPS Operating Systems Review, Vol.29, Issue. 5, pp. 237-250, 1995.

[3] D. P. Bovet and M. Cesati, Understanding the Linux Kernel, 3rd ed., O'Reilly, 2005.

[4] M. M. Swift, B. N. Bershad and H. M. Levy, "Improving the reliability of Commodity Operating System," Proceedings of the 19th ACM Symposium on Operating Systems Principles(SOSP), pp. 207-222, October 2003.

[5] B. N. Bershad, S. Savage, P. Pardyak, D. Becker, M. Fiuczynski, E. G. Sirer, "Protections is a Software Issue," Proceedings of the Fifth Workshop on Hot Topics in Operating Systems (HotOS-V), pp. 62-65, 1995.

[6] Lin Tan, Ellick M. Chan, Reza Farivar, Nevedita Mallick, Jeffrey C. Carlyle, Francis M. David, Roy H. Campbell, "iKernel: Isolating Buggy and Malicious Device Drivers Using Hardware Virtualization Support," Third IEEE International Symposium on Dependable, Autonomic and Secure Computing, pp. 134-142, 2007.

[7] J. LeVasseur, V. Uhlig, J. Stoess, S. Gotz, "Unmodified Device Driver Reuse and Improved System Dependability via Virtual Machines," In Proceedings of the 6th conference on Symposium

on Operating Systems Design and Implementation, pp. 17-30, 2004.
[8] Richard Ta-Min, Lionel Litty, David Lie, "Splitting Interfaces: Making Trust Between Applications and Operating Systems Configurable," In Proceedings of the 7th symposium on Operating Systems Design and Implementation, pp. 279-292, 2006.
[9] J. Liedtke, "On Microkernel Construction," ACM SIGOPS Operating Systems Review, Vol.29, Issue. 5, pp. 237-250, 1995.
[10] Roy H. Campbell, Nayeem Islam, Ralph Johnson, Panos Kougiouris, Peter Madany, "Choices, Frameworks and Refinement," In Proceedings of the International Workshop on Object Orientation in Operating Systems, pp. 9-15, 1991.
[11] G. Hunt et. al., "An Overview of Singularity Project," Microsoft Research Technical Report, http://research.microsoft.com/research/pubs/view.aspx?msr_tr_id=MSR-TR-2005-135, 2005.
[12] B. Bershad, S. Savage, P. Pardyak, E. G. Sirer, D. Becker, M. Fiuczynski, C. Chambers, S. Eggers, "Extensibility, safety and performance in the SPIN operating system," In Proceedings of the fifteenth ACM symposium on Operating systems Principles, pp. 267-283, 1995.
[13] M. M. Swift, B. N. Bershad, and H. M. Levy, "Improving the reliability of commodity operating systems," In Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles, pp. 207-222, 2003.
[14] U. Erlingsson, M. Abadi, M. Vrable, "XFI: Software Guards for System Address Spaces," In Proceedings of the 7th Operating System Design and Implementation, pp. 75-88, 2006.
[15] D. A. Kaplan, "RingCycle: A Hardware based approach to driver isolation," http://www.acm.uiuc.edu/projects/RingCycle/browser/RingCycle.pdf, 2006.
[16] M. Abadi, M. Budiu, U. Erlingsson, J. Ligatti, "Control-flow integrity: Principles, implementations, and applications," In Proceedings of the 12th ACM conference on Computer and communications security, pp. 340-353, 2005.
[17] Feng Qin, Joseph Tucek, Jagadeesan Sundaresan, Yuanyuan Zhou, "Rx: Treating Bugs As Allergies - A Safe Method to Survice Software Failures," In Proceedings of the 20th ACM Symposium on Operating Systems Principles, pp. 235-248, 2005.
[18] S. McCamant and G. Morrisett, "Evaluating SFI for a CISC architecture," In Proceedings of the 15th conference on USENIX Security Symposium, pp. 209-224, 2006.
[19] Robert Wahbe, Steven Lucco, Thomas E. Anderson, Susan L. Graham, "Efficient software-based fault isolation," In Proceedings of the 14th ACM Symposium on Operating Systems Principles, pp. 203-216, 1994.
[20] Jongmoo Choi, Seungjae Baek, Sung Y. Shin, "Design and Implementation of a Kernel Resource Protector for Robustness of Linux Module Programming," The 21st Annual ACM Symposium on Applied Computing, pp. 1477-1481, 2006.

김 병 진
2006년 단국대학교 정보컴퓨터학과 졸업(이학사). 2008년 단국대학교 대학원 정보컴퓨터 과학과(이학석사). 2008년~현재 벽산정보통신 연구원. 관심분야는 운영체제, 임베디드 시스템, 침입 감지 시스템 등

백 승 재
2005년 단국대학교 컴퓨터공학과 졸업(공학사). 2004년~현재 비트 컴퓨터 강사. 2007년 단국대학교 대학원 정보컴퓨터 과학과(이학석사). 2007년~현재 단국대학교 대학원 컴퓨터학과 박사과정. 관심분야는 운영체제, 임베디드 시스템, 차세대 저장장치 등

김 근 은
1990년 광운대학교 전자공학과 졸업(학사). 1997년~콜로라도대학교 대학원 전자공학과 졸업(석사). 2007년 부산대학교 대학원 컴퓨터 공학과 박사과정. 1990년~1997년 현대전자 선임연구원. 1997년~2003년 서울이동통신 책임연구원. 2006년~현재 한국정보사회진흥원 수석연구원. 관심분야는 지능형 로봇, 임베디드 시스템

최 종 무
1993년 서울대학교 해양학과 졸업(이학사). 1995년 서울대학교 대학원 컴퓨터공학과(공학석사). 2001년 서울대학교 대학원 컴퓨터 공학과(공학박사). 2001년~2003년 유비쿼스 주식회사 책임 연구원 2003년~현재 단국대학교 공과대학 컴퓨터학부 컴퓨터공학 전공 조교수. 2005년~2006년 UC Santa Cruz 방문 교수. 관심분야는 운영체제, 임베디드 시스템, 차세대 저장장치 등