Journal of The Korea Society of Computer and Information
Vol. 22 No. 4, pp. 1-8, April 2017

www.ksci.re.kr
https://doi.org/10.9708/jksci.2017.22.04.001

Fault Isolation for Linux Device Drivers

Sunghoon Son*

Abstract

In this paper, we propose a fault isolation system for device drivers of the Linux operating system. High availability systems impose stringent requirements upon Linux operating system. Especially device drivers can be a major source of operating system instability and many times contribute to system degradation and outages. The proposed fault isolation system identifies the occurrence of the memory-related faults in device driver and isolates it from the kernel. By operating at the early stage of the page fault handler in Linux kernel, the system detects which module causes fault and isolates it transparently from the remaining part of the kernel. By experiments, we show that the proposed system efficiently detects faults incurred by device driver, isolates the device driver and the process which accessed the driver module from the kernel.

▸ Keyword : Fault, Fault isolation, Linux device driver, High availability

I. Introduction

컴퓨터 시스템이 중단 없이 정상적으로 동작하도록 하는 것은 중요한 이슈 중의 하나이다. 그간 컴퓨터 시스템의 가용성을 높여 중단 없이 동작하도록 하기 위하여 여러 가지 고장 감내 기법들이 개발, 사용되어 왔다. 흔히 고장 감내 기법은 많은 비용이 소요된다.

컴퓨터 시스템이 정지되는 원인과 그에 따른 대처 방법은 정지 원인을 제공하는 컴퓨터 시스템 구성 요소에 따라 분류해 볼 수 있다. 이 중 하드웨어의 오동작으로 인한 정지는 대부분의 고장 감내 기법들에서 주요 하드웨어 구성 요소들을 이중화하는 방식으로 해결하고 있다. 반면 소프트웨어 오동작으로 인한 시스템 정지는 대상 소프트웨어가 응용 소프트웨어인지 시스템 소프트웨어인지에 따라 달라진다. 응용 소프트웨어의 경우 해당 프로세스만 종료 및 재시작 시킴으로써 전체 시스템은 정지되지 않고 정상적으로 작동을 이어나가도록 할 수 있다. 그러나 운영체제와 같은 시스템 소프트웨어 수준에서 발생하는 오류는 복구가 쉽지 않아 이로 인해 시스템이 정지되는 상황이 발생할 가능성이 크고, 자칫 큰 피해로 이어질 수도 있다. 특히 기업이 제공하는 서비스에 사용되는 웹 서버, 데이터베이스 서버인 경우 연속적인 서비스 제공에 지장을 초래하여 큰 손해로 이어질 수도 있다.

최근 리눅스 운영체제가 폭넓게 사용되고 있다. 리눅스 운영체제는 웹 서버, 파일 서버, 데이터베이스 서버뿐만 아니라 임베디드 시스템 분야에서도 널리 이용되고 있다. 리눅스 운영체제는 오픈 소스의 특징을 이용하여 커널 소스를 수정하거나 디바이스 드라이버, 파일 시스템 등을 커널 모듈의 형태로 커널에 추가할 수 있다. 이는 리눅스 운영체제의 커다란 장점이기도 하지만, 여러 개발자들이 커널 소스를 수정함에 따라 상업적인 운영체제에 비해 일부 문제점을 가지고 있다[1]. 특히 커널 모듈 등을 잘못 작성하는 경우 이는 시스템의 오동작으로 직결된다. 잘못된 코드가 커널에 존재하는 경우 이로 인한 비정상적인 동작으로 인해 서비스 자체가 정지되거나 관련 데이터의 무결성이 손상되기도 한다.

본 논문에서는 적은 비용으로 시스템의 정지 현상을 감소시킬 수 있는 커널 하드닝 기법에 대해 다룬다. 커널 하드닝 운영

• First Author: Sunghoon Son, Corresponding Author: Sunghoon Son
*Sunghoon Son (shson@smu.ac.kr), Dept. of Computer Science, Sangmyung University
• Received: 2017. 02. 23, Revised: 2017. 03. 07, Accepted: 2017. 03. 20.
• This research was supported by 2017 Research Grant from Sangmyung University.

체제에서는 커널 패닉이 발생한 경우, 단순히 시스템을 정지시키기 보다는 현재 상태가 복구 가능한지 여부를 판단하여 복구가 불가능하다면 기존의 방식대로 패닉 상태로 빠지게 되지만, 복구가 가능하다고 판단되는 경우 복구를 수행하여 시스템이 연속적으로 동작하도록 한다. 특히 본 논문에서는 리눅스 디바이스 드라이버에서 발생하는 오류를 대상으로 오류를 검출하고 이를 격리하는 기능을 제안한다. 디바이스 드라이버 모듈에서 오류가 발생하는 경우, 일반적으로 리눅스 커널은 커널 패닉으로 진입하여 전체 시스템이 정지하게 된다. 본 논문에서는 이러한 상황에서 커널 패닉에 진입하기보다 오류를 일으킨 디바이스 드라이버를 인지하여 이를 커널에서 격리시킴으로써 전체 시스템의 안정성을 향상시키는 것을 목표로 한다.

이를 위해 기존 리눅스 커널에 오류 격리 서브시스템을 추가하였다. 이 서브시스템은 기존 리눅스 커널의 페이지 폴트 처리 함수의 초기 단계에서 동작하여 등록된 커널 모듈 정보와 현재 커널 스택을 참조하여 어떤 디바이스 드라이버 모듈이 오류를 일으켰는지 찾아낸다. 이 과정을 통해 오류를 일으킨 모듈을 특정하게 되면, 오류 격리 서브시스템은 해당 모듈의 모듈 구조체에 접근하여 해당 모듈의 file operation을 변경하여 이후 이 디바이스 드라이버로의 접근을 차단하게 된다.

실험을 통하여 제안된 오류 격리 서브시스템이 효과적으로 동작함을 확인하였다. 이를 위하여 잘못된 메모리 주소를 참조하는 시험용 모듈을 제작하고, 이를 동작시키는 과정에서 발생한 오류에 대해 해당 모듈이 제대로 탐지되고, 시험용 모듈에 접근한 프로세스가 제대로 제거되는지, 해당 모듈이 적절히 제거되는지를 확인하였다. 또한 잘 알려진 파일 시스템 모듈을 대상으로 오류주입기를 사용하여 kmalloc 관련 오류를 강제로 발생시킨 후, 마찬가지로 오류 모듈의 탐지 및 제거와 이에 접근한 프로세스에 대한 종료가 제대로 이루어지는지 확인하였다.

본 논문의 구성은 다음과 같다. 우선 2장에서는 기존 고장 감내 시스템과 커널 하드닝 기법 등 관련 연구를 살펴본다. 3장에서는 논문에서 제안한 오류 격리 서브시스템을 자세히 소개하고, 4장에서는 오류 격리 서브시스템에 대한 실험 결과를 설명하며, 마지막으로 5장에서 결론을 맺는다.

II. Related Works

그간 컴퓨터 시스템의 가용성을 높이기 위한 다양한 연구들이 이루어져 왔다. 이중 고장 감내 컴퓨팅(Fault-tolerant computing)은 다양한 오류 발생에도 정상적으로 동작을 이어나갈 수 있는 컴퓨터 시스템을 만드는 기술을 의미한다. 이 분야의 기술들은 크게 하드웨어 고장 감내 기법과 소프트웨어적인 고장 감내 기법으로 구분할 수 있다. 고장 감내 컴퓨팅의 대부분의 연구는 주로 임의의 하드웨어적 오류 발생으로부터 자동적인 복구가 가능한 컴퓨터 시스템을 설계하는 것을 목표로 하고 있다. 이를 위해 컴퓨터 시스템을 여러 개의 오류 방지 영역(fault-containment region)의 역할을 하는 모듈 단위로 구분하고, 각 모듈의 하드웨어를 이중화함으로써 오류에 대처하는 방식을 취하고 있다. 반면 소프트웨어 고장 감내 기법의 경우 소프트웨어의 설계 오류 또는 프로그래밍 오류에 대처하기 위해 동일한 기능을 하는 여러 개의 모듈을 만든 후, 한 모듈이 실행 중 오류가 발생하면 정적 또는 동적으로 다른 모듈이 이를 대체하게 하는 기법들이 사용되고 있다. N-version programming 기법[2] 등이 대표적인 소프트웨어 고장 감내 기법이다.

상용화된 대표적인 고장 감내 시스템들은 은행의 전산 시스템이나 항공사 예약 시스템과 같은 온라인 트랜잭션 처리 시스템들에 주로 적용되었다. Tandem Computers사는 이 분야에서 최초의 메이저 회사이다[3]. Tandem 사의 OLTP 시스템은 일종의 약결합 형태의 분산 시스템으로 정교한 이중화 시스템을 사용하고 있다. 모든 실행 중인 프로세스는 주 프로세스와 다른 컴퓨터 노드에 이에 대응하여 동작하는 백업 프로세스로 구성된다. 주 프로세스는 일정한 체크포인트마다 자신의 상태를 디스크에 저장하게 된다. 만일 고장이 발생하게 되면 가장 최근의 체크포인트 이후로부터 백업 프로세스가 주 프로세스 대신 동작하는 방식으로 오류에 대처한다. Stratus Technologies 사는 이 분야의 또 다른 주요 기업이다[4, 5]. Stratus는 duplex self-checking 이라는 개념을 사용한다. 모든 컴퓨터 시스템은 내부적으로 복제되어 두 컴퓨터가 동시에 동일하게 동작하는 구조로, 한 컴퓨터에 고장이 발생하는 경우 다른 컴퓨터가 이를 대체하여 지연 없이 연속적인 서비스가 가능하게 한다. IBM 사의 전통적인 메인프레임 시리즈들도 고가용성을 보장하기 위해 internal checking, 명령어 재실행(instruction retry), 이중화를 기반으로 한 백업 장비(redundant unit)로의 자동 전환 등의 고장 감내 기법을 광범위하게 적용해왔다[6].

서버 시장은 인터넷 상의 중단 없는 서비스가 강조되면서 고장 감내 기능에 대한 요구가 커진 분야이다. 대부분의 서버 제조사들은 프로세서, 디스크, 파워 등을 이중화하고, 고장 발생 시 백업 유닛으로의 자동 전환을 통해 중단 없는 서비스가 가능한 제품들을 내놓았다. SUN의 ft-SPARC이나 HP의 Continuum 400 등이 대표적인 예이다. 다른 제조사들은 클러스터 시스템을 기반으로 한 고장 감내 시스템을 내놓기도 했는데 마이크로소프트사의 MSCS 기술이 그 예이다.

리눅스 운영체제를 대상으로 커널과 디바이스 드라이버에서의 오류 발생과 그 대처 기법들에 대한 연구도 이루어졌다. 우선 [7]에서는 리눅스 운영체제를 대상으로 커널이 오류에 대해서 어떤 행동 양식을 보이는지에 대한 광범위한 실험 결과를 소개하고 있다. 또한 리눅스 상에서 모듈 형태로 동작하는 디바이스 드라이버 등의 오류 발생에 대처하는 기법에 대한 연구도 이루어졌다. 인텔사에서는 고가용성 시스템에 리눅스가 도입되

는데 가장 큰 걸림돌 중 하나가 디바이스 드라이버라는 인식 하에, 단순히 잘 작성된 디바이스 드라이버 코드를 넘어, 다양한 하드웨어적/소프트웨어적 문제들이 발생하는 경우 이에 적절히 대처하여 디바이스 드라이버에 발생한 문제가 커널 전체로 전파되지 않도록 하는 디바이스 드라이버를 설계하는 연구를 수행하였다[8]. [9]에서는 리눅스 운영체제의 디바이스 드라이버의 오류를 탐지하고 복구할 수 있는 섀도우 드라이버(shadow driver) 개념을 소개하고 있다. 리눅스 시스템의 각 디바이스 드라이버마다 섀도우 드라이버를 두고, 디바이스 드라이버에서 오류가 발생하면 해당 드라이버를 섀도우 드라이버가 대체하도록 하여 리눅스 시스템의 신뢰성을 높이고 있다. [10]에서는 리눅스 시스템의 커널 모듈에서 발생하는 오류를 리눅스 커널로부터 제거할 수 있는 커널 자원 보호기를 제안하였다.

리눅스 운영체제의 커널 하드닝과 관련해서는 몬타비스타(Montavista)사에서 많은 연구, 개발이 이루어지고 있다. 특히 몬타비스타는 커널 하드닝 기능이 포함되어 있는 CGE(Carrier Grade Edition) 버전 리눅스 운영체제를 상업적으로 판매하고 있다[11, 12]. 몬타비스타의 CGE 리눅스는 기존의 리눅스 운영체제 커널에 코드 리뷰 (code review), 패닉 제거 (panic removal), 오류 주입 테스트 (fault injection test) 등의 커널 하드닝 기법을 적용한 것이다. 코드 리뷰 기법은 기존 리눅스 커널의 소스 코드에 대해 지속적인 검토를 통해 소스 코드 수준에서 커널 코드의 오류를 제거하는 기능이다. 패닉 제거 기법은 운영체제 코드를 검사한 후, 시스템을 중지(panic)시킬 것인지 아니면 단순히 관련 프로세스를 종료시킬 것인지를 결정하는 기능이다. 오류 주입 기법은 동작 중인 리눅스 커널에 강제로 인위적인 오류를 발생시켜 리눅스 커널이 복구할 수 있는지 없는지에 대해 검사하는 일종의 블랙박스 검사이다[13, 14].

본 논문에서 제안하는 커널 내 오류 격리 기능은 리눅스 운영체제의 디바이스 드라이버 모듈에서 잘못된 코드의 실행으로 인하여 커널 전체가 정지되는 것을 정상적으로 복구함으로써 시스템이 정상 동작되도록 한다. 본 논문에서 제안하는 오류 격리 서브시스템은 시스템의 가용성을 높이고, 안정된 시스템 사용을 보장할 수 있다. 이 오류 격리 서브시스템은 기존의 고장 감내 시스템과는 달리 많은 비용을 들이지 않고도 적용이 가능하다. 따라서 앞으로 많은 리눅스 시스템에서 이 오류 격리 기능을 통해 시스템의 안정성을 높일 수 있을 것으로 기대한다.

III. Fault Isolation System

본 논문에서 제안하는 오류 격리 서브시스템(Fault isolation subsystem)은 리눅스 시스템 동작 중 오류가 발생한 디바이스 드라이버 모듈을 (재부팅 등 없이) 자동으로 검출하고 이를 커널로부터 격리시켜 전체 시스템은 동작을 이어나가도록 하는 서브시스템이다. 본 장에서는 이러한 리눅스 기반의 오류 격리 시스템의 설계와 관련된 구체적인 사항을 기술한다. 우선 제안하는 오류 격리 시스템이 추구하는 최종 목표는 다음과 같다.

오류 대상의 수정이 아닌 커널 수준의 오류 격리 오류를 검출하거나 격리하기 위해 오류를 유발한 디바이스 드라이버 모듈의 소스 코드를 수정해야 한다면 확장성이나 활용성이 떨어진다. 오류 격리 서브시스템은 문제를 일으키는 디바이스 드라이버 모듈을 직접적으로 수정하지 않고, 시스템 동작 중에 커널 레벨에서 오류를 유발한 디바이스 드라이버 모듈을 검출하여 격리한다.

커널 수행 중 발생하는 오류의 검출 리눅스 커널이 수행하는 중에 디바이스 드라이버 모듈에서 오류가 발생하는 경우, 일반적으로 커널 패닉 상태로 진입한다. 오류 격리 서브시스템은 커널의 오류 처리 과정 중에 오류를 유발한 디바이스 드라이버 모듈을 검출한다.

오류를 유발한 디바이스 드라이버 모듈의 격리 한번 오류를 유발한 디바이스 드라이버 모듈은 흔히 지속적으로 문제를 일으킬 가능성이 높다. 오류 격리 서브시스템은 일단 오류를 격리 시킨 후 다른 프로세스가 오류를 발생시킨 디바이스 드라이버 모듈로 접근하지 못하도록 한다.

제안한 오류 격리 시스템에서는 현재 리눅스 커널의 페이지 폴트 예외 처리기 (Page fault exception handler)인 함수 do_page_fault()에서 오류를 검출한다. 함수 do_page_fault()는 기본적으로는 가상 메모리 시스템 하의 페이지 부재를 처리하는 함수이다. 하지만 예외 발생 주소가 특정 프로세스의 주소 공간에 해당하는지의 여부, 메모리 접근 유형이 무엇인지, 예외가 어떤 모드(사용자 또는 커널 모드)에서 발생했는지의 여부 등에 따라 매우 복잡하고 다양한 경우들을 처리하도 설계되어 있다. 특히 커널 내에서 발생한 메모리 관련 오류의 경우도 이 함수에서 처리하도록 되어있는데, 본 논문의 디바이스 드라이버의 메모리 오류도 이 경우에 해당한다. 따라서 현재 검출 가능한 오류는 메모리 관련 오류들로 한정된다. 특히 검출 가능한 오류의 유형은 (1) 잘못된 메모리 참조 오류와 (2) 오류주입기를 통한 kmalloc 오류의 두 가지로 한정한다.

본 논문에서 제안하는 오류 격리 서브시스템은 크게 오류 탐지 블록 (Fault detection block)과 모듈 격리 블록 (Module isolation block)으로 구성된다. 오류 탐지 블록은 오류가 발생한 시점에 해당 오류에 연루된 디바이스 드라이버 모듈을 검출하는 동작을 수행한다. 모듈 격리 블록은 일반 프로세스가 오류를 발생시킨 디바이스 드라이버 모듈로 접근하지 못하도록 해당 모듈을 커널로부터 격리시키는 동작을 수행한다. 또한 오류 탐지 블록과 모듈 격리 블록 간에 오류 발생 모듈에 대한 정보를 주고받을 수 있는 모듈 정보 블록이 존재한다. 오류 격리 서브시스템의 전체적인 구조, 기존 리눅스 커널과의 관계, 그리고 개괄적인 동작은 Fig. 1과 같다[15, 16].

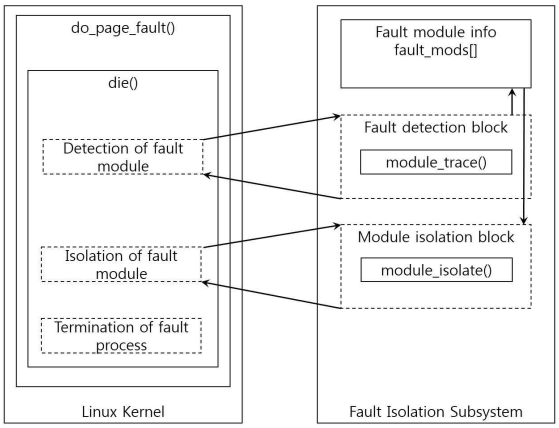

Fig. 1. System Architecture

Fig. 1에서 보는 바와 같이 디바이스 드라이버 모듈에서 오류가 발생하면 오류 탐지 블록이 오류를 검출하여 디바이스 드라이버 모듈을 검출하여 그 정보를 fault module info에 저장한다. 모듈 격리 블록은 fault module info의 정보를 토대로 해당 디바이스 드라이버 모듈을 격리한다. 오류를 유발한 프로세스의 종료와 할당된 동적 메모리 반환은 기존 리눅스 커널의 die() 함수에서 수행된다.

오류 격리 서브시스템의 각 블록을 구성하는 함수의 구조, 기능, 기본 동작, 관계, 그리고 주요 자료 구조는 다음과 같다.

1. Fault module information

배열 fault_mods[]은 오류 탐지 블록이 수행 과정 중에 오류와 관련된 디바이스 드라이버 모듈의 이름을 저장하는데 사용하는 자료 구조이다. module_trace() 함수에서 초기화하고, module_trace_address() 함수에서 정보를 업데이트한다. 수집된 정보는 module_trace() 함수에서 커널 로그에 기록하기 위해 사용되거나, 모듈 격리 블록에서 격리할 모듈을 결정하기 위해 사용된다.

2. Fault detection block

오류 탐지 블록은 오류가 발생한 시점에 해당 오류와 관련된 디바이스 드라이버 모듈들을 검출하는 동작을 수행한다. 오류 탐지 블록은 Fig. 2와 같은 수행 과정을 거친다.

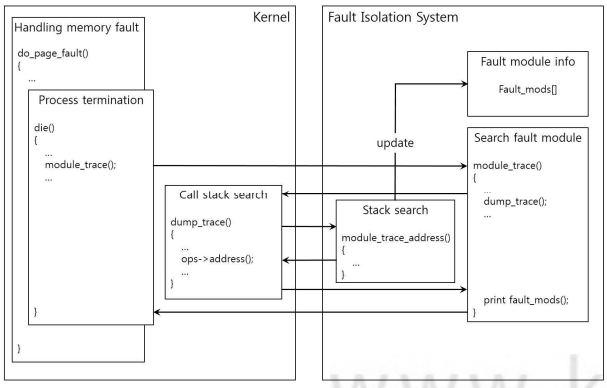

Fig. 2. Fault Detection Block

디바이스 드라이버 모듈에서 메모리 관련 오류가 발생할 경우 페이지 폴트 예외가 발생하고, 이를 처리하기 위해 리눅스 커널의 페이지 폴트 처리 함수인 do_page_fault()로 분기한다. 함수 do_page_fault()에서는 해당 오류를 유발한 프로세스를 종료하기 위해 함수 die()를 호출한다. 함수 die()는 소위 말하는 콘솔에 현재의 모든 CPU 레지스터 값을 출력하고(이를 kernel oops라고 함), 현재 프로세스를 종료시키는 역할을 한다. 따라서 오류 격리 서브시스템에서는 die() 함수에서 오류와 관련된 디바이스 드라이버 모듈을 탐색하기 위해 module_trace() 함수를 호출하도록 추가하였다. module_trace() 함수는 우선 오류 모듈의 정보를 저장할 배열인 fault_mods[]를 초기화한다. 그리고 커널의 dump_trace() 함수를 이용하여 오류와 관련된 디바이스 드라이버 모듈을 찾는다. dump_trace() 함수는 module_trace_address() 함수를 사용하여 오류가 발생한 시점에 연관된 디바이스 드라이버 모듈을 찾고 이를 fault_mods[]에 저장한다. module_trace() 함수는 마지막으로 fault_mods[]에 저장된 오류와 관련된 디바이스 드라이버 모듈 정보를 커널 로그에 기록한다.

함수 module_trace()는 오류와 관련된 디바이스 드라이버 모듈을 추적하고 fault_mods[] 배열에 저장하며 그 정보를 출력하는 함수이다. module_trace() 함수의 기능은 크게 (1) fault_mods[] 배열을 초기화, (2) dump_trace() 함수를 호출하여 오류와 관련된 모듈들을 fault_mods[] 배열에 저장, (3) 오류와 관련된 모듈이 존재할 때, fault_mods[] 배열로부터 모듈의 정보를 출력 등의 세 가지로 이루어진다.

dump_trace() 함수는 오류가 발생할 경우 현재의 문맥(context) 및 스택의 내용을 출력하는 리눅스 커널의 디버깅 함수이다. dump_trace() 함수는 오류가 발생한 시점에 스택의 내용을 확인하여 어떤 함수가 호출되었는지 추적한다. 이 때, 사용자 정의 오퍼레이션 함수를 통해 사용자가 의도하는 동작을 추가할 수 있다.

module_trace_address() 함수는 dump_trace() 함수에서 사용하는 사용자 정의 오퍼레이션 함수 중의 하나이다. module_trace_address() 함수는 스택에 저장된 메모리 주소를 통해 해당 주소에 위치한 모듈 이름을 찾고, 이를 fault_mods[] 배열에 저장한다.

3. Module isolation block

Module isolation block은 오류가 발생한 시점에 해당 오류와 관련된 디바이스 드라이버 모듈들을 격리하는 동작을 수행한다. Module isolation block은 그림 3과 같은 수행 과정을 거친다.

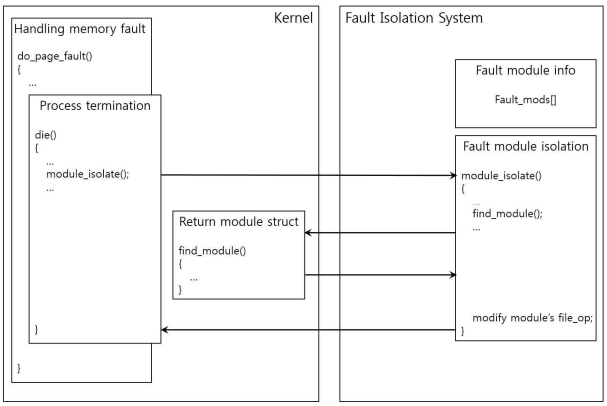

Fig. 3. Module Isolation Block

앞서 언급한 바와 같이 오류 탐지 블록에 의해 오류가 발생한 모듈에 대한 정보를 저장한 이후, 다시 die() 함수에서 모듈 격리 블록의 역할을 수행하는 module_isolate() 함수를 호출하도록 추가하였다. 함수 module_isolate()는 fault_mods[] 배열에 저장된 모듈의 이름을 find_module() 함수에 전달하여 오류 발생 모듈에 대한 구조체를 구하고, 해당 모듈의 file operation을 변경하여 해당 디바이스 드라이버 모듈로의 접근을 차단한다.

함수 find_module()은 모듈의 이름을 통해 해당 디바이스 드라이버 모듈의 정보를 담은 구조체를 구하는 함수이다.

함수 module_isolate()는 오류와 관련된 디바이스 드라이버 모듈들을 격리하는 함수이다. 함수 module_isolate()는 다음과 같이 동작한다.

(1) fault_mods[] 배열에서 오류와 관련된 모듈의 이름을 구한다.

(2) find_module() 함수를 호출하여 해당 모듈의 모듈 구조체를 획득한다.

(3) 모듈에 대한 작업 요청을 모두 무시하도록 모듈 구조체의 file_operation을 초기화한다.

(4) (1) ~ (3)까지의 과정을 fault_mods[] 배열에 저장된 모듈 수만큼 반복한다.

IV. Experiments

본 장은 제안한 시스템의 기능이 정상적으로 동작하는지 확인하기 위해 시험을 설계하고 이를 진행한 결과에 대해서 기술한다. 또한 설계된 시험을 통해 결과를 도출한다.

시험을 수행하기 위한 최소한의 소프트웨어 환경은 다음과 같다. 운영체제로 대부분의 Linux 기반의 배포판은 모두 가능하며, 컴파일러는 GNU GCC 3.4.x 이상을 사용하였다. 시험에 사용한 리눅스 커널 버전은 2.6.24이고, 커널 컴파일 시 특히 kernel debuginfo 활성화 및 kprobe 활성화가 적용되었다. 또한 시험에 필요한 응용 프로그램으로는 Systemtap 버전 0.7 이상이 요구된다. 다음 Table 1은 앞 장에서 소개한 오류 격리 서브시스템을 위한 코드가 수정된 소스 코드를 보여준다.

Table 1. List of Modified Source Codes in Linux

Source file	Description
arch/i386/mm/fault.c	Invocation of fault isolation system in do_page_fault()
arch/i386/kernel/traps.c	- Invocation of module_trace() and module_isolate() in die() - Invocation of module_trace_address() in dump_trace()
kernel/module.c	- Invocation of module_isolate() in find_module() - Adding array fault_mods[] - Adding module_trace(), module_trace_address(), and module_isolate()

1. Isolation of invalid memory access fault

오류 격리 서브시스템의 동작을 시험하기 위하여 크게 두 가지 종류의 시험을 실행하였다. 첫 번째 실험은 잘못된 메모리 주소를 참조하도록 테스트 모듈을 만들고, 이를 실행하여 발생한 오류를 정상적으로 탐지하는지를 실험하였다. 구체적인 실험 내용은 Table 2와 같다.

Table 2. Isolation of Invalid Memory Access Fault

Test	Detection and isolation of invalid memory access fault
Procedure	① boots the kernel ② creates a device node for test module # mknod /dev/badtest c 240 32 ③ Loads test module which references invalid memory address # insmod /usr/src/bad_test/dev/bad_test_mod.ko ④ Compiles and runs application which accesses the test module # cd /usr/src/bad_test/app/ # gcc -o bad_test bad_test.c # ./bad_test
Verification	- Verifies that the outputs of the module are recorded in the kernel log # dmesg - Verifies that the process bad_test terminates normally # ps | grep bad_test - Verifies that the module bad_test can be unloaded normally # rmmod bad_test
Criteria for acceptance	- Is the name of fault module recorded in the kernel log? - Is the process killed normally? - Is the module bad_test unloaded safely?

이 실험의 수행 결과는 다음과 같다. 우선 “dmesg” 명령을 통해 커널 메시지에서 오류를 발생시킨 모듈의 이름과 해당 프로세스의 pid 정보를 확인할 수 있다 (Fig. 4).

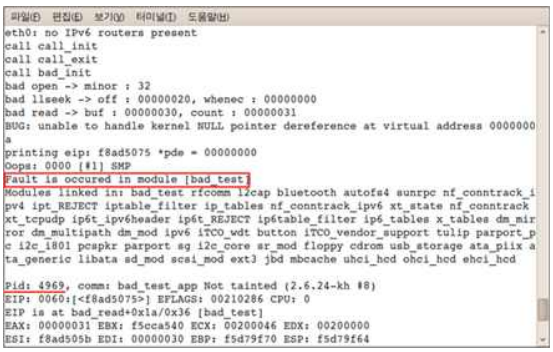

Fig. 4. Detection of Fault

두 번째 “ps” 명령을 통해 프로세스 정보에서 해당 오류가 발생한 프로세스의 이름과 pid 검색을 해보았으나 아무런 정보가 검색되지 않아 해당 프로세스가 정상 종료한 것을 알 수 있다 (Fig. 5).

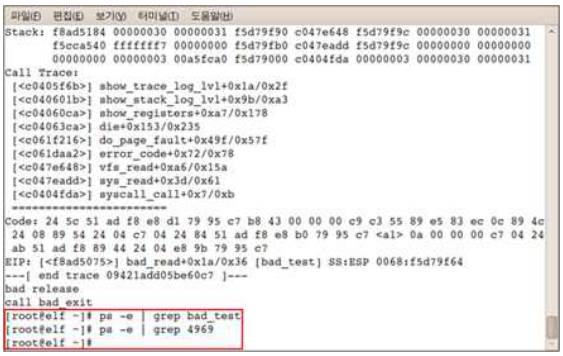

Fig. 5. Normal Termination of Process

해당 모듈의 제거 후 커널의 동작 중인 모듈 목록을 보면 해당 모듈의 제거가 이루어진 것을 알 수 있다 (Fig. 6).

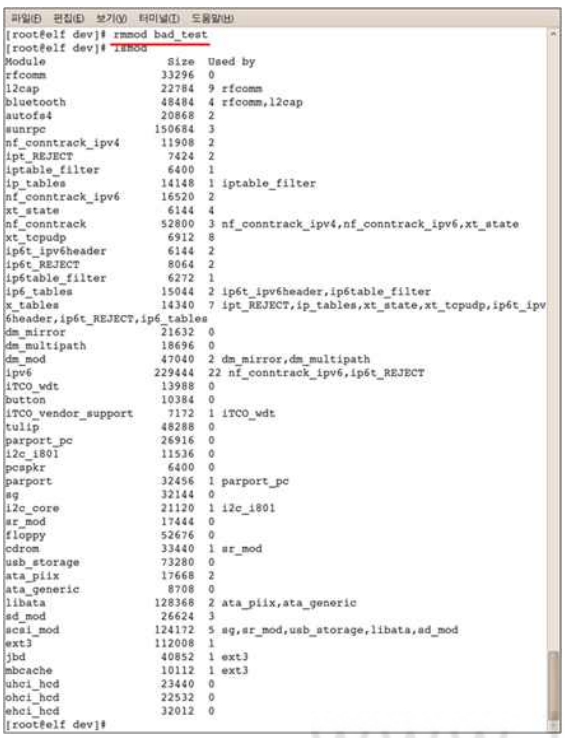

Fig. 6. Proper Unloading of Bad Module

2. Isolation of faults generated by fault injection

두 번째 실험은 오류 주입기를 통해 발생시킨 kmalloc 오류에 대한 탐지 및 격리 실험이다. 구체적인 실험 내용은 Table 3과 같다.

Table 2. Isolation of Invalid Memory Access Fault

Test	Detection and isolation of kmalloc fault by fault injector
Procedure	① boots the kernel ② compiles and loads tfat module # cd octovita/tester/tfat # make # make modules_install # modprobe tfat # mount -t tfat [test_disk_device_node] [test_mount_point] ③ inserts a kmalloc error using fault injector # faultinj -m tfat -t 1 -i 200 -s 200 -c 30000 /opt/octovita/bin/fs_workload [test_mount_point]
Verification	- Verifies that the outputs of the tfat module are recorded in the kernel log # dmesg - Verifies that the module tfat can be unloaded normally # rmmod -f tfat - Verifies that the process bad_test terminates normally # ps | grep faultinj
Criteria for acceptance	- Is the name of fault module recorded in the kernel log? - Is the process killed normally? - Is the module tfat unloaded safely?

이 실험의 수행 결과는 다음과 같다. 우선 “dmesg” 명령을 통해 커널 메시지에서 오류를 발생시킨 모듈의 이름과 해당 프로세스의 pid 정보를 확인할 수 있다 (Fig. 7).

Fig. 7. Detection of Fault Generated by Fault Injector

“ps” 명령을 통해 프로세스 정보에서 해당 오류가 발생한 프로세스의 이름과 pid검색을 해보았으나 아무런 정보가 검색되지 않아 해당 프로세스가 종료한 것을 알 수 있다 (Fig. 8).

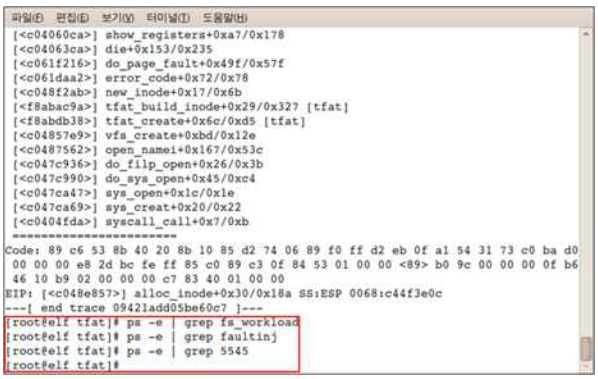

Fig. 8. Normal Termination of Process

"rmmod -f tfat" 명령을 통해 해당 모듈의 강제 제거 후 커널의 동작 중인 모듈 목록을 보면 해당 모듈의 제거가 제대로 이루어진 것을 알 수 있다 (Fig. 9).

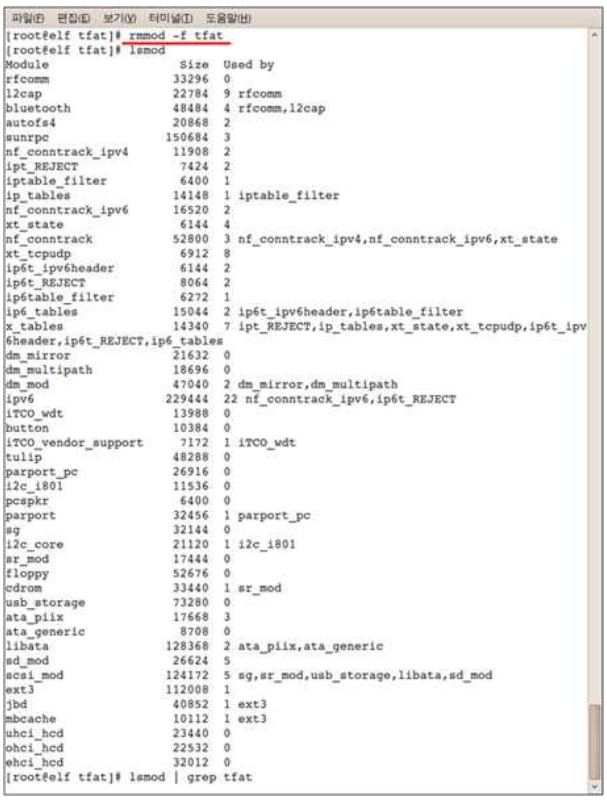

Fig. 9. Proper Unloading of Module tfat

V. Conclusions

리눅스 커널의 디바이스 드라이버 모듈에서 오류가 발생하는 경우, 일반적으로 리눅스 커널은 커널 패닉으로 진입하여 전체 시스템을 정지시키게 된다. 본 논문에서는 리눅스 디바이스 드라이버에서 발생할 수 있는 오류를 대상으로 발생한 오류를 검출하고 이를 격리하는 오류 격리 서브시스템을 제안하였다. 이 서브시스템은 기존 리눅스 커널의 오류 처리 함수의 초기 단계에서 동작하여 오류를 일으킨 모듈을 찾아내고, 이 모듈을 커널에서 격리함으로써 전체 시스템이 오류에도 동작을 이어나갈 수 있도록 하였다. 실험을 통하여 잘못된 메모리 접근을 시도하는 모듈과 오류 주입기를 통해 강제로 오류를 발생시킨 모듈에 대하여, 해당 모듈이 제대로 탐지되는지, 오류를 일으킨 모듈에 접근한 프로세스가 제대로 제거되는지, 해당 모듈이 적절히 제거되는지를 확인하였다. 앞으로 메모리 관련 오류 뿐 아니라 리눅스 커널 동작 중 발생할 수 있는 다양한 형태의 오류에도 오류 격리 기법을 확장하여 적용하는 연구를 진행하고자 한다.

REFERENCES

[1] M. Mitchell, J. Oldham, and A. Samuel, "Advanced Linux Programming," New Riders Publishing, 2001.

[2] L. Chen and A. Avizienis, "N-Version Programming: A Fault-Tolerance Approach to Reliability of Software Operation," Proc. of the 25th International Symposium on Fault-Tolerant Computing, pp. 113-119, Pasadena CA, USA, June 1995.

[3] Joel Bartlett, Jim Gray, Bob Host, "Fault Tolerance in Tandem Computer Systems," Tandem Technical Report TR-85.3, 1985.

[4] Stratus, *http://www.stratus.com*

[5] S. Webber and J. Beirne, "The Stratus Architecture," Proc. of the 21st International Symposium on Fault-Tolerant Computing, pp. 79-85, Montreal, Canada, June 1991.

[6] L. Spainhower and T. A. Gregg, "G4: A Fault-Tolerant CMOS Mainframe," Proc. of the 28th International Symposium on Fault-Tolerant Computing, pp.432-440, Munich, Germany, June 1998.

[7] W. Gu, Z. Kalbarczyk, R. K. Iyer, and Z. Yang, "Characterization of Linux Kernel Behavior under Errors," Proc. of the 2003 International Conference on Dependable Systems and Networks, San Francisco, USA, pp. 459-468, June 2003.

[8] L. Matassa, "Device Driver Hardening and Manageability," Intel Corporation White Paper, 2002.

[9] M. M. Swift, M. Annamalai, B. N. Bershad, and H, M. Levy, "Recovering device drivers," ACM Trans. on Computer Systems, Vol. 24, No. 4, pp. 333-360,

November 2006.

[10] J. Choi, S. Baek, and S. Y. Shin, "Design and implementation of a kernel resource protector for robustness of Linux module programming," Proc. of the 2006 ACM Symposium on Applied Computing, pp. 1477-1481, Dijon, France, April 2006.

[11] Carrier Grade Edition, *http://www.mvista.com/cge*

[12] John Mehaffey, "Montavista Linux Carrier Grade Edition," White Paper of Montavista Software Inc., April 2002.

[13] Dave Edwards and Lori Matassa, "An Approach to Injecting faults into Hardened Software," Proc. of the 2002 Ottawa Linux Symposium, pp. 146-157, Ottawa, Cananda, June 2002.

[14] T. Naughton, W. Bland, G. Vallée, C. Engelmann, and S. L. Scott, "Fault Injection Framework for System Resilience Evaluation," Proc. of Resilience '09, pp.23-28, Munich, Germany, June 2009.

[15] J. Corbet, A. Rubini, and G. Kroah-Hartman, "*Linux Device Drivers, 3rd Ed.*" O'Reilly, 2005.

[16] Daniel P. Bovet and Marco Cesati, "*Understanding the Linux Kernel 3rd Ed.*" O'Reilly, 2006.

Authors

Sunghoon Son received his B.S., M.S. and Ph.D. degrees in Computer Science from Seoul National University, Korea, in 1991, 1993 and 1999, respectively. Dr. Son joined the faculty of the Department of Computer Science at Sangmyung University, Seoul, Korea, in 2004. He is currently a Professor in the Department of Computer Science, Sangmyung University. He is interested in system software, embedded system, and virtualization.