포스트 모템 디버깅과 프로세스 덤프※

박주항, 김영식*
한국산업기술대학교 산업기술경영대학원 디지털엔터테인먼트학과**
juhang3@daum.net, kys@kpu.ac.kr

Post Mortem Debugging And Process Dump

Ju-hang Park, Young-sik Kim*
Course for Industrial Technology(Digital Entertainment)
Graduate School of Industrial Technology & Management
Korea Polytechnic University**

요 약

프로그램 개발에 있어 디버깅은 매우 중요한 요소이다. 개발 과정에서 디버깅을 통해 버그를 대부분 잡을 수 있지만 제품이 출시된 이후에도 프로그램 버그를 잡을 수 있는 시스템이 필요하다. 이를 위해 포스트 모템 디버깅에 대해 살펴보고 프로세스 덤프에 대해 논의한다. 또한 프로그램이 다운되었는데 프로세스 덤프가 생성되지 않는 상황에 대해 원인을 분석해 보도록 한다. 아울러 검증된 예외 핸들러라 하더라도 프로세스 덤프를 남기지 못하는 상황을 이해하고 이런 상황을 타개하기 위해 EHModule(Exception Handler Module)를 구현한다. 구현한 EHModule의 핵심 루틴을 설명하고 해당 모듈의 특징 및 효용성에 대해 언급한다.

ABSTRACT

Debugging is very important element in programming development. We can find a lot of bug of program we made, and we can fix it. but after a product is released, we need system we can catch bugs as soon as possible. For this, we need to know post-mortem debugging. We will look at post-mortem debugging, and we will talk about process dump. Also when process get the problem, we should get the process dump, but we can have situation process dump was not generated. We will consider this case, and we will examine programming technique which this case can be retrieved. Finally, I will introduce EHModule(exception handler module)

Keywords : post-mortem debugging, process dump, exception handler

접수일자 : 2011년 02월 09일 일차수정 : 2011년 03월 25일 심사완료 : 2011년 04월 13일
교신저자(Corresponding Author) : 김영식, e-mail : kys@kpu.ac.kr
※ 이 연구는 2011 한국콘텐츠진흥원 산업계 맞춤형 인력양성사업 지원을 받았음

1. 서 론

프로그램의 버그로 인한 다운 및 강제 종료는 프로그래머가 피해갈 수 없는 문제 중 하나이다. 이러한 문제를 해결하기 위해서는 디버깅 작업이 필요하며 디버깅 작업은 제품 출시전과 출시후로 나누어진다. 제품 출시 전에는 프로그램의 문제점을 디버깅 작업으로 해결할 수 있지만 제품이 출시된 이후에는 문제의 원인을 바로 확인할 수 없는 이상 프로그램의 결함을 바로 수정하는 것이 힘들다. 이런 경우 개발사는 문제의 원인을 수집하고 분석할 수 있어야 하는데 이때 필요한 프로그래밍 기술이 포스트모템 디버깅이다.

포스트모템 디버깅은 프로그램의 소스코드가 없어도 디버깅을 가능하게 해준다. 이를 가능하게 하기 위해서 필요한 것이 프로세스 덤프이다. 프로세스 덤프는 프로그램에 이상이 생겨 종료할 때 파일로 남게 되며 이런 프로세스 덤프에는 프로그램이 비정상적으로 종료할 때의 상황을 기술해 주는 정보를 담고 있다 프로그래머는 이 프로세스 덤프를 수집해서 프로그램의 문제의 원인을 진단하고 수정한다.

[1]은 포스트모템 디버깅을 하기 위해 필요한 여러 툴을 설명하고 있다. 윈도우용 디버깅 툴로 cdb, ntsd, windbg 등이 있는데 windbg는 유저 인터페이스를 지원하기 때문에 가장 많이 애용된다. 이런 툴들을 이용해서 프로세스 덤프를 분석할 수가 있게 되는 것이다.

그런데 어떤 경우 프로그램이 다운되었는데 프로세스 덤프가 남지 않는 경우가 종종 발생한다. 프로세스 덤프를 남도록 하기 위해서는 프로그램에 예외 핸들러를 설치해야 되는데 검증된 예외 핸들러를 설치했다 하더라도 프로세스 덤프가 남지 않는 경우가 발생하는 것이다.

본문에서는 이렇게 프로세스 덤프가 생성되지 않는 상황을 이해하도록 하며 검증된 예외 핸들러가 한계를 가질 수 밖에 없는 상황을 파악하도록 한다. 아울러 본인이 제작한 EHModule의 동작원리를 이해하고 이 모듈의 목표와 특이성을 소개하도록 한다.

2. 본 론

2.1 EHModule 개발 동기

게임서버 운영에서 가장 중요한 요소는 서버 안정화이다. 게임서버의 다운이나 비정상 종료는 유저의 원성을 사고 이는 매출과 직결되기에 개발자는 한층 조심해서 코딩해야 한다. 하지만 프로그래밍은 사람이 하는 일이기에 실수가 존재하기 마련이다. 이런 실수를 줄이기 위해서는 단위 테스트 같은 사전 검사가 필요하지만 프로그램이 다운되었을 때 원인을 빨리 파악하는 것도 매우 중요하다. 이를 위해 개발자가 필요로 하는 것이 프로세스 덤프이다. 프로세스 덤프란 프로세스의 메모리 상황이나 레지스터 정보 등을 저장한 파일을 의미한다. 보통은 프로그램이 다운되었을 때 이런 파일이 생성되는데 프로그램이 이런 기능을 수행하기 위해서는 예외 핸들러를 설치해야 한다. 이런 예외 핸들러는 신뢰할 수 있는 회사인 구글 등에서 이미 모듈화 해놓았기 때문에 굳이 따로 예외 핸들러 모듈을 만들 필요는 없다. 그러나 실제 서비스를 해보면 여러 가지 상황에 의해 검증된 모듈이라고 해도 덤프를 남기지 못하는 경우가 허다하다. 이런 문제 때문에 프로세스 덤프가 남지 않는 상황을 연구하게 되었으며 몇 가지 새로운 아이디어를 추가하여 모듈화한 것이 EHModule이다.

2.2 포스트 모템 디버깅 예비 지식

2.2.1 프로세스 덤프

앞에서 언급한 바와 같이 프로세스 덤프는 프로세스의 레지스터 정보나 메모리 값들을 그대로 저장한 파일이다. 상황에 따라 프로세스가 차지한 메모리 전체를 덤프해야 경우도 있으나 보통은 레지

스터나 콜 스택(Call Stack) 같은 최소한의 정보만을 저장하기도 한다. 이런 정보는 프로세스가 작업을 수행 중일 때에 얻을 수도 있으며 프로세스가 다운되었을 때에도 얻을 수 있다.

2.2.2 예외 핸들러(SEH)

[2]에서는 프로세스에서 문제가 발생했을 때 윈도우즈 운영체제가 에러를 다루는 개념을 잘 설명해 주고 있다. X86 구조에서는 프로그램에서 예외가 발생했을 경우 해당 프로그램에서 예외를 처리할 수 있는 기회를 부여해 준다. 프로그램은 이런 예외를 처리하기 위해 예외 핸들러를 구현해야 한다. 프로그램이 오동작하면 해당 예외 핸들러에서 예외를 처리하기 위한 루틴이 실행되며 예외 처리가 구현되어 있지 않으면 디폴트 예외 핸들러로 제어를 옮기게 된다.

2.2.3 예외 핸들러에 구현해야 될 사항

예외 핸들러로 제어를 옮겼다는 것은 프로그램의 비정상적인 동작을 감지했다는 것을 의미한다. 당연히 왜 이런 비정상적인 동작이 발생했는지에 대한 원인 규명을 위해 예외 핸들러는 프로세스 덤프를 남기도록 루틴을 작성해야 할 것이다.

2.2.4 프로세스 덤프 생성

프로세스 덤프를 생성하는 방법에 대해서는 [3]에서 자세히 기술하고 있다. 윈도우즈 운영제제는 DbgHelp.dll 이라는 모듈을 제공하고 있는데 이 모듈은 MiniDumpWriteDump 함수를 익스포트하고 있다. 이 함수를 이용해서 프로세스 덤프를 생성할 수 있으며 해당 루틴을 예외 핸들러에 구현해야 한다.

2.3 생성되지 않는 프로세스 덤프

게임서버가 비정상 종료되면 원인을 찾기 위해 대부분의 개발자는 프로세스 덤프를 확인하려고 할 것이다. 개발자들은 신뢰할 수 있는 예외 핸들러 모듈을 사용했기에 프로세스 덤프가 남지 않으면 당황한다. 본인도 똑같은 경험을 하면서 곤혹을 치른바 있다. 이를 기점으로 프로세스 덤프가 생성되지 않은 원인에 대한 조사를 하게 되었다.

2.3.1 포스트 모템 분석과 프로세스 덤프

프로세스 덤프가 생성되지 않는 원인을 조사하기에 앞서 지금 취하는 행동이 소프트웨어 엔지니어링에 있어서 어떤 영역에 포함되는지를 확인할 필요가 있다고 생각되었다.

포스트 모템 디버깅은 결국 포스트모템 분석(PMA)의 영역에 속한다고 볼 수 있다. [4]에서는 포스트 모템 분석의 절차에 대해 설명하고 있다. 포스트 모템 분석은 경험적인 방법론이다. 이 방법론의 이점은 프로젝트가 완료된 이후 많은 해결책을 발견할 수 있다는 데에 있다. 포스트모템 분석은 프로젝트의 규모에 따라 적용 기술이 달라지는데 예를 들면 프로젝트의 범위나 응집도를 예로 들 수 있겠다. 프로젝트 리뷰에 대한 방식은 [5]에서 확인할 수 있다.

프로젝트 리뷰의 이점은 프로젝트 구성원에게 서로의 시각과 견해 또는 이해를 공유할 수 있고 개인이나 팀이 얻은 지식을 통합할 수 있다는 데에 있다. 또한 프로젝트의 숨겨진 문제점을 발견할 수 있고 다음 프로젝트 진행시 좋지 않은 방법을 반복하지 않을 수 있도록 좋은 습관과 문제점 등을 문서화 할 수 있다는 것이다. 뿐만 아니라 자신의 작업에 관해서 사람들에게 피드백을 줄 수 있고 이는 작업 만족도를 증가시켜 어떤 경우에는 프로젝트의 비용 절감을 가져올 수 있다.

포스트모템 디버깅을 포스트모템 분석의 영역에 놓을 지는 개인차가 있겠지만 디버깅을 통해 프로젝트의 문제점을 발견하고 다음 프로젝트에서는 그런 문제를 반복하지 않을 것이란 점에서 포스트모템 디버깅은 포스트모템 분석의 한 영역이라고

생각한다.

2.3.2 휘발성 메모리 데이터 덤프

[6]은 라이브 시스템의 휘발성 데이터, 즉 메모리 데이터를 덤프하여 분석하는 툴에 관해 설명하고 있다. 프로세스를 정지시키지 않은 채 프로세스를 분석하는 방법을 설명하고 있으며 관련 툴은 현재 웹이 폐쇄되어 다운받을 수 없다. 프로그램의 비정상적인 종료에 의해 생성된 프로세스 덤프의 분석보다는 코드 인젝션, 예를 들면 dll 인젝션 같은 공격에 의해 프로세스가 변형되는 것을 감지하기 위해 프로세스 덤프를 생성하고 메모리를 조사하는 방법을 기술하고 있다. 본 논문은 외부 공격에 의한 프로세스의 변형에 대한 탐색 부분을 대상으로 삼고 있지 않지만 프로그램의 상태를 조사한다는 측면에서 [6]은 프로세스 덤프 분석의 여러 줄기 중 하나라고 할 수 있겠다.

2.3.3 프로세스 덤프가 남지 않는 케이스

visual studio 6.0같이 오래된 컴파일 프로그램으로 생성된 프로그램은 해당 컴파일 프로그램의 한계에 따라 오류도 많이 발생하며 유지보수하기도 어렵다. 이러한 연유로 컴파일 프로그램을 상위버전으로 마이그레이션(migration)을 하게 되는데 이 과정에서 프로그램은 예전에는 발생하지 않았던 많은 문제를 안게 되는데 그 중 하나가 프로세스 덤프 생성과 관련이 있다.

기존에는 우리가 설치된 예외 핸들러 루틴이 실행되어야 했다. 하지만 위와 같은 이유로 컴파일 프로그램이 업그레이드되면서 STL이나 CRT 관련 보안 속성이 증가되었다. 이에 의해 STL이나 CRT 관련 에러가 발생하면 예외 처리는 우리의 예외 핸들러를 타지 않고 STL이나 CRT가 선언한 예외 핸들러로 흐름을 타게 된다.

[12]는 클라이언트 오류처리 관련 사항을 소개하고 있다. 우리가 설정한 예외 핸들러가 무용지물이 되는 상황이 존재하기 때문에 별도의 예외 핸들러를 설정해야 함을 언급하고 있다. [표 1]은 이런 별도의 예외 핸들러 몇 가지를 보여준다.

[표 1] 별도의 예외 핸들러

예외 핸들러	설명
set_purecall_handler	순수가상함수 처리
set_invalid_parameter_handler	CRT 함수에 잘못된 인자가 들어올 경우

[9]에서는 왜 수많은 상황에서 자신이 등록한 예외 핸들러가 제대로 동작하지 않는지를 잘 설명해 주고 있다. Visual Studio 2005 이후의 변경점에 대해 [표 2]는 그 내용을 기술하고 있다.

[표 2] 예외 핸들러 작동 불가 이유

VS2005에서 자신의 예외 핸들러가 제대로 동작하지 않는 이유
_abort-behaviur 속성이 _CALL_REPORTFAULT 속성으로 선언되었을 경우
security checks에 의해 감지된 실패
CRT 함수 호출시 잘못된 인자값이 넘어왔을 경우

2.3.4 프로세스 덤프가 남지 않는 케이스 2

앞 절의 내용은 덤프 생성되지 않는 이유가 자신이 설정한 예외 핸들러의 예외 처리 능력과는 무관한, 컴파일러나 라이브러리가 보안 등의 문제를 강화하기 위해 여러 동작을 수행하면서 자신의 예외 핸들러가 동작하지 못하게 해서 프로세스 덤프를 생성하지 못하게 한 경우였다. 그렇다면 예외 핸들러의 기능 자체가 많이 떨어져서 덤프를 생성하지 못하는 경우는 없을까라는 생각에 도달하게 되었고 이 부분 관련 조사에 들어가게 되었다.

[11]의 글은 프로세스 덤프가 콜 스택을 정보를 얻을 수 있도록 예외 핸들러를 구현하는 방법을 설명해 주고 있다.

함수의 리컬시브 호출 등에 의해 스택 오버플로

우(Overflow)가 발생하면 더 이상 스택에 로컬 변수 등을 할당할 수 없는 상황이 발생한다. 이런 상황에서 예외가 발생하면 예외 핸들러가 에러를 남기기 위해 덤프를 생성하려고 하지만 스택 공간이 부족해 예외 핸들러 마저 정상 동작을 못하고 만다. 따라서 이런 경우 별도의 쓰레드를 생성해서 예외처리를 하면 쓰레드 생성에 따라 별도의 스택 공간이 생성되므로 이 문제를 해결할 수 있게 된다. [표 3]은 프로세스 덤프 관련 스택 오버플로우 처리에 관한 의사 코드이다.

[표 3] 스택 오버플로우 처리 의사코드

```
int ExceptionHandler(예외정보)
{
        if(예외정보 == 스택 오버플로우)
       {
        쓰레드 생성
        예외정보와 함께 쓰레드 실행
        별도 쓰레드에서 프로세스 덤프 생성
       }
       else
       {
       일반적인 프로세스 덤프 처리
       }
}
```

[8]은 try_except 구문의 예외 필터 안에서 예외를 던지는 경우에, 즉 예외처리를 잘못한 경우에 OS가 JIT 디버거를 띄울 기회를 얻지 못하는 상황에 대해 언급하고 있다.

이 경우는 디버거를 붙인 상황에서 프로그램을 실행하면 예외 처리가 가능하지만 그렇지 않은 경우에서는 조용히 프로그램이 종료가 될 것이다.

이 외에도 덤프가 생성되지 않는 문제 중의 하나로 메모리 오버플로우(Memory Overflow)를 들 수 있다. 메모리 오버플로우를 통해 프로그램이 다운되면 어떠한 프로세스 덤프도 생성되지 않음을 확인하였다. 이 문제를 해결하기 위해서는 프로그램이 다운될 때 덤프를 생성하는 것이 아니라 특정 메모리 공간 이상을 어플리케이션이 점유했을 때, 즉 프로세스가 동작하고 있을 때 미리 덤프를 생성하는 수밖에 없다고 판단이 된다.

[7]에서는 ProcessDump라는 프로그램에 대해 설명하고 있는데 이를 이용하면 프로세스가 수행중에 있을 때 덤프를 남기는 것이 가능하다.

2.3.5 프로세스 덤프가 남지 않는 경우 정리

[표 4]는 프로세스 덤프가 남지 않는 원인들에 대한 설명을 하고 있다.

[표 4] 프로세스 덤프 생성 실패 원인

에러타입	내 용
스택 오버플로우	스택이 오버플로우된 상태이기 때문에 덤프를 남길 때 스택을 사용하면 문제가 된다.
CRT 에러	버퍼 오버 플로우 등 런타임 함수에 인자값에 문제가 있을 경우 예외가 발생할 경우 CRT에서 자체적으로 해당 문제를 해결할 수 있는 함수를 호출한다.
메모리 오버플로우	메모리가 부족하여 덤프를 남길 수 없는 상황.
순수 가상함수 호출	순수 가상 함수를 호출하는 경우. 프로그램이나 컴파일러의 오류에 순수가상함수가 호출되는 상황이 발생한다.
힙 손상	객체의 잘못된 타입 캐스팅을 통해 객체 메모리 크기 이상의 힙 영역에 접근하는 경우

2.3.6 프로세스 덤프 생성 해결책

스택 오버플로우에 관해서는 [표 3]에서 설명을 했으며 CRT 에러의 경우 관련 예외 핸들러를 오버라이딩(Overriding)하면 된다.

메모리 오버플로우의 경우 메모리 공간이 부족하기에 더 이상의 코드를 실행시키는 것이 불가능하고 이 때문에 프로세스 덤프 생성 코드를 실행할 수는 없지만 콜스택을 출력하는 것에는 문제가 없었으며 확인 결과 콜 스택이 정상적으로 출력됨을 확인하였다. 콜 스택만 출력해도 메모리 오버플로우 문제를 해결할 수 있는 이유는 콜 스택을 확

인하면 어떤 메모리 연산 부분에서 메모리가 증가되었는지에 대한 함수 호출 흐름을 알 수 있기 때문이다.

힙 손상의 경우는 힙 손상이 발생한다고 해서 그것이 프로그램 비정상 종료와 반드시 연결되지는 않기에 문제점을 발견하기가 힘들다. 이런 경우는 Windows Debugging Tool에 포함된 gflag를 이용하거나 Application Verifier를 통해서 사전에 문제점을 발견하는 것이 좋다.

2.3.7 프로세스 덤프 생성 관련 궁극적 해결책

[표 4]에서 알 수 있듯이 다양한 상황에서 프로세스는 덤프를 남기지 못한다. 또한 위의 사례 외에도 덤프가 남지 않는 상황은 존재한다. 이런 문제가 발생하는 이유는 예를 들면 CRT 라이브러리에 속한 함수 중 보안이 강화된 함수가 인자 체크를 하고 인자 값에 문제가 있으면 예외 핸들러를 오버라이딩시켜 우리가 설정한 예외 핸들러가 호출되지 않게 하기 때문이다. 부연 설명하면 프로그래머가 커스트마이즈된 예외 핸들러를 추가하기 위해 SetUnhandledExceptionFilter라는 API를 사용하는데 CRT 에러의 경우 CRT 자체가 이 API를 호출해서 우리가 설정한 예외 핸들러를 무효화시킨다. 그래서 덤프가 생성되지 않는 것이다. CRT의 경우는 보안상의 이유로, 예를 들어 버퍼 공격에 따른 에러를 예외 핸들러에서 다시 다룰 수 있게 하면 프로세스의 취약성을 공격할 수 있기에 그렇게 할 수 없도록 기존의 예외 핸들러를 무효화 시킨다.

그렇지만 게임 서버 같은 경우 외부에 노출되지 않기에 이런 기능은 불필요하다. 또한 이런 에러를 다루기 위한 핸들러를 모두 구현하는 것은 무리이다. _set_invalid_parameter_handler 핸들러라던지 _set_purecall_handler등 관련 핸들러 수가 너무 많다.

따라서 이런 상황에서는 무조건 우리가 설정한 예외 핸들러가 호출되도록 구현하는 것이 바람직하다. 그럼 어떻게 하면 우리가 설정한 예외 핸들러를 무조건 실행 가능하게 할 수 있을까?

SetUnhandledExceptiomFilter 함수를 호출해서 예외 핸들러 등록을 한 뒤 이 함수를 후킹을 해서 어떠한 동작도 하지 않도록 구현한다면 우리의 예외 핸들러가 항상 실행됨을 보장받을 수 있을 것이다.

즉 다음에 SetUnhandledExceptiomFilter 함수를 호출하는 상황을 만나도 이 함수는 아무런 작업을 하지 않기 때문에 우리가 설정한 예외 핸들러는 항상 유효하다. 따라서 운영체제로부터 에러 보고를 받을 수 있으며 또한 프로세스 덤프를 생성하는 것 또한 가능해진다.

3. EHModule(Exception Handler Module)

3.1 EHModule 개발 이슈

서론에서 EHModule의 개발 동기에 대해 간략히 설명하였다. EHModule은 어떠한 비정상적인 프로그램 종료 상황에서도 프로세스 덤프를 남기자라는 취지하에 만들어진 모듈이다. 이 모듈이 어떻게 동작하는지에 대해 설명하기에 앞서 현재 대표적으로 쓰이는 예외 핸들러를 소개하고자 한다. [표 5]는 대표적 예외 핸들러를 소개하고 있다.

[표 5] 대표적인 예외 핸들러

핸들러	특 징
버그트랩 (BugTrap)	덤프 생성시 서버로 내용을 전송하는 기능이 있으며 UI창을 제공해서 문제의 상황을 직관적으로 이해할 수 있게 해준다.
브레이크 패드 (BreakPad)	스택 오버플로우 관련 예외에 대해서도 프로세스 덤프를 생성한다.

브레이크 패드는 [14]에서, 버그트랩은 [15]에서 확인 가능하다. 보통 브레이크 패드는 서버측에서, 버그트랩은 클라이언트 측에서 사용을 하는 편이

다. 이 두 예외 핸들러는 검증된 모듈이기 때문에 대부분의 예외 상황에서 프로세스 덤프를 생성할 수 있다. 일단 EHModule에서는 해당 모듈이 처리하지 못하는 예외들을 처리할 수 있도록 루틴을 추가해 놓았다. 이 예외 처리들은 게임서버를 운영하면서 발생한 문제들을 해결하기 위해서 추가해 놓은 것이며 EHModule의 핵심은 기존의 예외 핸들러를 사용하면서도 동시에 개별적으로 추가한 예외 핸들러 모듈을 사용할 수 있다는 데에 있다. 일반적으로 프로세스에는 한번에 하나의 예외 핸들러만 설치할 수 있다. 하지만 EHModule은 예를 들어 버그트랩과 개별적으로 추가한 예외 핸들러를 동시에 사용할 수 있다. 즉 개별적으로 추가한 예외 핸들러가 예외를 처리하지 못하면 제어권을 버그트랩으로 옮겨서 예외 처리하게 하는 것이 가능한 것이다. 이것이 어떻게 가능한지 살펴보도록 하자.

3.2 기존 예외 핸들러 모듈 성능 개선

우선 기존 예외 핸들러의 처리 능력을 살펴보고 EHModule에서 추가한 예외 처리를 살펴보도록 한다.

[표 6]은 기존 모듈 그자체로만 사용했을 때의 예외 처리 결과이다.

[표 6] 기존 예외 핸들러의 예외 처리 능력

예외 종류	버그트랩	브레이크패드
스택 오버플로우	X	O
메모리 오버플로우	X	X
잘못된 순수 가상함수 호출	X	X
힙 손상	X	X
CRT 에러	X	X

검증된 모듈임에도 예외 처리를 못해서 프로세스 덤프가 생성되지 않음을 알 수 있다. 검증된 모듈인 이상 [표 6]에 나온 대부분의 예외는 처리가 되도록 구현되어져 있었을 것이다. 앞에서 언급한 바와 같이 컴파일러 버전이 새 것으로 바뀜에 따라 발생한 문제이다.

한편 [표 7]은 기존 예외 핸들러를 설정하고 SetUnhandledExceptionFilter 함수를 무효화시킨 후 살펴 본 결과이다.

[표 7] SetUnhandledExceptionFilter 적용

예외 종류	버그트랩	브레이크패드
스택 오버플로우	X	O
메모리 오버플로우	X	X
잘못된 순수 가상함수 호출	O	O
힙 손상	O	O
CRT 에러	O	O

[표 7]을 보면 검증된 모듈의 덤프 생성 능력이 매우 향상되었음을 알 수 있다. 대부분의 경우에서 덤프를 생성하지만 메모리 오버플로우에서는 덤프를 생성하지 못함을 알 수 있다. 결국 메모리 오버플로우의 처리는 콜 스택을 출력해서 해결하는 것으로 노선을 선회하였으며 std::set_new_handler 핸들러를 오버라이딩해서 콜 스택을 출력하도록 하였다. EHModule에서는 콜 스택 관련내용이 파일로 저장된다.

[표 8]은 메모리 오버플로우 문제까지 해결한 결과이다.

[표 8] 메모리 오버플로우 해결

예외 종류	버그트랩	브레이크패드
스택 오버플로우	X	O
메모리 오버플로우	O	O
잘못된 순수 가상함수 호출	O	O
힙 손상	O	O
CRT 에러	O	O

두 모듈을 이렇게 개선해서 실서버에 적용하였더니 대부분의 경우 에서 프로세스 덤프 생성이 되었고 많은 문제들을 해결할 수 있었다.

3.3 버그트랩의 스택 오버플로우 문제

앞 절에서 버그트랩과 브레이크패드 모듈을 비교하였다. 둘을 비교한 이유가 어떤 모듈이 더 우수한지를 대조해 보기 위함은 아니다. 일단 성능상으로는 브레이크패드가 우수한 것으로 보인다. 버그트랩은 스택 오버플로우 문제를 해결하지 못하기 때문이다. 하지만 버그트랩은 브레이크패드에 비해 우수한 점이 너무나도 많다. 앞에서 언급한 바 있지만 생성된 프로세스 덤프를 서버에 전송하는 기능, 다이얼로그 창으로 덤프 정보 보여주기 등 매력적인 기능이 많다. 따라서 예외 처리 성능이 아주 우수하더라도 반드시 그 모듈을 쓰지는 않을 것이다. 그렇다 하더라도 스택 오버플로우 처리를 못하는 문제를 방치할 수도 없다. 그럼 이 문제를 어떻게 해결하면 좋을까?

직관적으로는 버그트랩 소스코드의 예외 처리 코드에 스택 오버플로우 처리를 구현하면 될 것이다. 여기서 문제가 발생하는데 신뢰된 모듈을 수정하면 어떤 문제가 발생할 지 모른다는 것이다. 오히려 더 많은 부작용을 초래할 지도 모른다. 또한 어떤 경우 소스 코드 자체가 제공되지 않을 수도 있다. 따라서 기존의 검증된 모듈은 수정하지 않고 약점만 보강하는 편이 유지보수 면에서도 편하고 깔끔하다고 할 수 있겠다.

하지만 한 프로세스에서는 하나의 예외 핸들러 밖에 등록을 하지 못한다. 버그트랩을 등록했으면 버그트랩밖에 예외 핸들러로 쓸 수 없는 것이다. 그래서 생각 끝에 한번에 두가지 핸들러를 동시에 쓸 수 있는 방법을 EHModule에 구현하였다.

3.4 동시에 두가지 예외 핸들러 사용하기

동시에 두가지 예외 핸들러를 사용하기 위해서는 다음 두가지 조건을 극복해야 한다.

(1) 무효화시킨 SetUnhandledExceptionFilter 함수 복구
(2) 예외 처리 상황에서 예외를 만들기

3.4.1 SetUnhandledExceptionFilter 복구

SetUnhandledExceptionFilter 함수를 무효화 시킨 이유에 대해서는 충분히 설명하였다. 이제 두 개의 예외 핸들러를 사용하기 위해서 이 함수를 원래대로 복구하여야 한다. 그렇게 해야 예외 핸들러를 다시 설정하는 것이 가능해지기 때문이다. 해당 함수를 복구시키는 루틴은 다음과 같다.

(1) kernel32.dll의 핸들을 얻어낸다.
(2) 해당 핸들에서 해당 함수 주소를 얻어낸다.
(3) 해당 주소의 5바이트 엔트리에 백업했던 값을 복사한다.

해당 함수를 무효화시키는 방법은 복구하는 절차와 유사하며 해당 주소의 엔트리에 새 함수 주수로 점프하는 코드를 복사하고 동시에 원래 5바이트를 변수에 저장한다.

3.4.2 예외 처리 상황에서 예외를 만들기

예외 핸들러에서 예외를 다시 발생시킬려면 RaiseException 함수를 사용하면 된다. 단 부작용을 피하기 위해 EXCEPTION_ACCESS_VIOLATION 값을 RaiseException 의 인자로 제공하도록 한다.

3.4.3 두 개의 예외 핸들러 동작

스택 오버플로우를 처리할 수 있는, 본인이 만든 예외 핸들러와 버그트랩을 어떻게 동시에 사용할 수 있는지 살펴보도록 하자. 의사 코드는 다음과 같다.

(1) SetUnhandledExceptionFilter 함수 호출시 자신이 구현한 핸들러 등록
(2) SetUnhandledExceptionFilter 함수 무효화
(3) 예외가 발생했을 때 자신이 등록한 예외 핸들러가 예외를 처리할 수 있으면 처리하고 프로그램 종료
(4) 예외를 처리할 수 없다면 SetUnhandledExceptionFilter를 복구

(5) SetUnhandledExceptionFilter 호출하여 버그트랩을 예외 핸들러로 등록
(6) 소프트웨어 예외 발생시킴(RaiseException)
(7) 예외 핸들러가 버그트랩으로 변경되어 버그트랩이 예외를 처리

3.4.4 EHModule의 기능 정리

[표 9]는 EHModule의 제작 동기와 효용성을 보여주고 있다.

[표 9] EHModule에 대한 정리

항 목	내 용
제작동기	어떠한 예외적 상황에서도 덤프를 남길 수 있도록 한다.
검증된 예외 핸들러 보강	기존에 사용되는 검증된 예외 핸들러 모듈은 대부분의 예외를 처리할 수 있다. 하지만 컴파일러의 버전업등으로 예전에는 존재하지 않았던 문제가 발생한다. 대표적인 예로 CRT 문제를 들 수 있으며 EHModule은 이를 해결한다.
두 개의 핸들러 동시 사용	기존 모듈을 수정하는 것은 유지보수 관점에서 좋지 않다. EHModule은 기존 예외 모듈을 수정하지 않으면서 해당 모듈이 처리하지 못하는 예외를 처리 가능하게 함으로써 해당 모듈의 생명력을 높여 준다.
예외 핸들러 관리	EHModule은 여러개의 검증된 핸들러를 동록할 수 있도록 구현해 놓았다. 서버에서 쓰는 모듈과 클라이언트에서 쓰는 모듈이 다를 수 있기에 편의성을 위해서 추가가 쉽도록 구조화하였다.

EHModule은 단순히 검증된 예외 핸들러 모듈을 모아서 기능을 약간 강화시킬려고 만든 모듈이 아니다. EHModule은 서두에서 밝힌바와 같이 어떤 예외적 상황에서라도 프로세스 덤프를 생성하기 위해 제작하였으며 검증된 예외 핸들러 모듈은 각각의 특징이 있기 때문에 상황에 따라서 선택할 수 있다. 클라이언트에서는 버그트랩을 서버에서는 브레이크패드를 선택할 수 있는 것이다. 그렇기에 여러 개의 검증된 예외 핸들러 모듈을 등록할 수 있도록 만든 것이며 이러한 모듈들은 성능이 뛰어나지만 실제 게임 서비스를 해보면 컴파일러 버전업과 같은 문제와 예외 처리 미구현 같은 문제로 프로세스 덤프가 생성되지 않는 문제에 직면하게 된다. 이 경우 프로그래머가 검증된 예외 핸들러 모듈을 수정할 수밖에 없었으나 EHModule은 그 한계를 극복하여 모듈을 수정하지 않고 한번에 두 개의 예외 핸들러를 사용할 수 있게 하여 유지보수 측면에서 편하게 사용할 수 있도록 제작된 것이다. 버그트랩보다 더 기능이 많고 사용자 편의성이 뛰어난 덤프 모듈이 있으면 EHModule에 등록시키고 특정 예외 처리가 미구현되어 있으면 두 개의 예외 핸들러를 사용할 수 있다는 장점을 이용하여 우리의 예외 핸들러 모듈에 그 예외처리를 구현하기만 하면 되는 것이다.

4. 결 론

본 논문에서는 포스트모템 디버깅의 필요성과 프로세스 덤프에 대해 언급하였고 프로세스 덤프가 생기지 않는 상황을 극복하기 위해 구현한 EHModule을 소개하였다. 프로그래머에게 있어 디버깅은 프로그래밍의 50% 이상을 차지할 정도로 중요한 영역이며 디버깅 영역 중 포스트 모템 디버깅은 게임서버 운영에 있어 필수적으로 익혀야 할 영역이다. 프로세스 덤프는 프로그램의 버그나 크래쉬의 원인을 조사하기 위한 필수적인 자료이며 포스트 모템 디버깅의 기본이 된다. 하지만 프로그램이 다운될 경우 프로세스 덤프가 생성되지 않는 예외는 수없이 많으며 이런 상황을 없애기 위해 우리는 예외 핸들러에 몇 가지 작업을 해야 함을 알 수 있었다. 이러한 요구에 의해 프로세스 덤프를 반드시 생성하자라는 취지하에 만든 EHModule은 프로세스 덤프를 제대로 생성하는 것을 확인할 수 있었으며 검증된 예외 처리 모듈과 자신의 예외 처리 모듈 2개를 동시에 사용하는 방식을 쓰면 기존 예외 모듈을 수정하지 않고 예외 처리를 구

현할 수 있음을 알 수 있었다. 이 모듈은 궁극적으로 프로세스 덤프 관련해서 프로그래머가 신경쓰지 않도록 하기 위해 제작된 것이다.

참고문헌

[1] 실전 윈도우 디버깅 / Mario Hewardt Daniel Pravat

[2] Windows 구조와 원리 - OS를 관통하는 프로그래밍의 원리 / 정덕영

[3] MicroSoft Debugging Application .Net And Windows / 존 로빈스 저 서우석 역

[4] M. Myllyaho, O. Salo, J. Ka¨a¨ria¨inen, J. Koskela, A review of small and large post-mortem analysis methods,in: Proceedings of the ICSSEA, Paris, 2004

[5] Bonnie Collier: Project Review Process Web Site-PostmortemToolkit; http://www.projectreview.net/prtoolkit.asp [Retrieved March 28, 2004], 1996

[6] Tobias Klein. Process Dump Analyses: Forensical acquisition and analyses of volatile data; http://www.trapkit.de/img/pdf.gif

[7] ProcDump로 CPU 사용률 조건별 덤프 생성; http://laigo.kr/499?srchid=BR1http%3A%2F%2Flaigo.kr%2F499

[8] 예외를 만들지 않고, 그냥 사라지는 프로세스; http://serious-code.net/moin.cgi/CppDebuggingTips#head-ff430a953a7bcd6b78b3520757ea9c69149bc9c3

[9] “SetUnhandledExceptionFilter” and VC8; http://blog.kalmbachnet.de/?postid=75

[10] 덤프가 남지 않는 경우는 어떤 경우가 있을까요?; http://www.gpgstudy.com/forum/viewtopic.php?topic=23283

[11] Minidump stack overflow 검출; http://sweeper.egloos.com/2420307

[12] NDC 10 진선웅 클라이언트 오류처리 A to Z; http://ndc.nexon.com/150091248918

[13] Userdump.exe 도구를 사용하여 덤프 파일을 만드는 방법; http://support.microsoft.com/?scid=kb;ko;241215&x=14&y=15

[14] 구글 브레이크 패트; http://code.google.com/p/google-breakpad/

[15] 버그트랩; http://www.codeproject.com/KB/applications/BugTrap.aspx

박 주 항 (Juhang Park)

서강대학교 물리학과/컴퓨터학과 복수전공 학사
2006-2010 JCE 서버 프로그래머
2010.9-12 서강대학교 게임교육원 시간강사
현 재 한국산업기술대학교 대학원 디지털 엔터테인먼트 학과 4학기 재학

관심분야 : 게임서버 아키텍쳐, 소프트웨어 보안, 운영체제 제작 등

김 영 식 (Youngsik Kim)

1993 연세대학교 전산과학과 학사
1995 연세대학교 컴퓨터과학과 석사
1999 연세대학교 컴퓨터산업시스템공학과 박사
1995-1997 연세대학교 시간강사
1999-2005 삼성전자 System LSI 책임연구원
2005-현재 한국산업기술대학교 게임공학과 부교수

관심분야 : 3차원 그래픽 가속기(OpenGL ES, Ray Tracing), 물리엔진, 아이폰 프로그래밍