※ 본 아티클은 CMP MEDIA LLC와의 라이선스 계약에 의해 국문으로 제공됩니다

Gamasutra.com

게임 개발을 위해 PC의 메모리 사용을 모니터링하기

Jelle van der Beek
2007년 5월 31일

http://www.gamasutra.com/view/feature/1430/monitoring_your_pcs_memory_.php

메모리 사용량에 대한 아티클 두 개를 작성한지 몇 년이 지났다. Gamasutura에 게시된 이 아티클의 제목은 “콘솔의 메모리 사용량 모니터링하기(monitoring your console’s memory usage)” 였다. 이 아티클에서 메모리의 단편화와 메모리 소모, 메모리 누수를 감시하기 위한 툴을 어떻게 만들었는지에 대해 토론을 했었다. 그러나 아티클의 작성을 거의 끝냈을 때 툴의 컨셉을 굉장히 진보시킬 수 있는 아이디어가 떠올랐다.

얼마 후, 새 버전의 툴을 만들기 시작했고 몇 달 후에 완성 했다. 툴을 개발한지 몇 년이 지난 시점에서, 새 컨셉은 아직도 매우 강력하고 내가 작업했던 타이틀에서 유용하다는 것을 증명해냈다. 우린 심지어 Xbox API와 *TechCertGame*(기술적으로 완전히 보장된 게임의 Xbox 샘플)의 메모리 누수를 찾아내는 것까지 성공했다.

이 아티클에선 Playlogic 에서 근무할 때 만든 메모리 분석 툴의 컨셉에 대해 이야기할 것이다. 놀랍게도 이 툴의 이름은 MemAnalye 2.0 이라 지었다.(멋진 이름을 짓는데 소질이 없다.)

처음은 컨셉에 대해서 시작하여 툴의 함수를 살펴보고 마지막으로 까다로운 실행 세부에 대해 착수할 것이다. 이전 아티클은 Xbox 와 PS2 에서의 실행에 대해 다뤘었으니 이번 시간에는 PC 의 메모리를 감시하는데 있어서의 함정들에 대해 알아보겠다.

MemAnalyze 의 컨셉

예전 컨셉에 대해 간단히 요악해 보겠다. 원한다면 이전의 아티클에서 상세 내용을 읽을 수 있겠지만 이 아티클을 이해하는데 이전 아티클의 내용을 전부 이해해야 할 필요는 없다.

우선은 게임의 모든 메모리 할당을 간섭하는 것부터 시작할 것이다. 메모리 할당이 수행될 때마다 메모리에는 일반적으로 호출 스택(callstack) 정보와 같은 메타 데이터(metadata)가 저장되며 사용자는 언제든지 모든 메모리 할당 데이터를 파일에 기록할 수 있다. 이 출력 파일은 MemAnalyze 의 입력 데이터로 작동한다. 툴은 데이터를 분석할 수 있으며 여러 개의 데이터 보기(Multiple view)를 지원한다.

하지만 이 컨셉에는 몇 가지 약점이 있다.

- 메모리 사용을 감시 하는 것은 게임의 메모리 통계 스냅샷이기 때문에 한 순간만 가능하다.
- 감시하고 있는 응용 프로그램의 메타 데이터를 저장하는 것은 메모리 통계에 악영향을 미친다. 평범한 규모의 게임인 제 테스트 환경에선 최고 할당 수는 거의 50 만 이었다. 그 중 대략 80,000 개는 메모리 할당 연산자(Operator New)에 의한 것이었다. 할당된 블록의 메타 데이터 크기는 호출 스택 길이의 차이로 인해 블록마다 다르지만 테스트 중 하나는 할당된 블록당 평균 126 바이트가 필요한 것으로 나타났다. 이 테스트에서 32 비트 주소들이 호출 스택에 보관 된 것을 참고하라. 이것은 블록마다 모든 메타 데이터를 수용하는데 약 77 메가바이트의 추가 메모리가 필요하다는 것을 뜻한다. 만약 메모리 할당 연산자에 의한 할당만 저장되었다면 약 9.5 메가바이트만 필요했을 것이다.

두 제한을 없애기 위해 할당 데이터를 네트워크를 통해 직접 전송하기로 했다. 툴이 PC 에서 실행 되면 할당 뮤테이션(Mutations)을

모으고 내부 메모리 맵을 유지한다. 그렇게 함으로써 이 툴은 내부 메모리 맵을 실시간으로 분석할 수 있게 되었다.

제한이 사라져서 이제 실시간 분석을 할 수 있게 되었으며 게임의 메모리 사용을 실시간으로 감시 하는 것이 가능해 졌습니다. 게다가 메모리 뮤테이션을 기록할 수도 있게 되었다. 그러므로 제가 '논리적 메모리 누수(Logical memory leaks)'라고 부르는 것을 찾아낼 수도 있다.

이 부분은 본 아티클의 뒷부분에 있는 기록 보기에서 설명될 것이다. 그리고 메모리 제한이 사라졌으므로 전체 할당의 호출 스택을(나는 종종 메모리 제한 때문에 호출 스택 정보를 강제로 제한하기도 했다.) 전송할 수 있다. 하지만 기록 중에 방대한 양의 데이터를 네트워크로 전송하기 때문에 모든 데이터를 처리하려면 효율적인 전략이 필요하다. 그래서 데이터의 양을 알아보기 위해 다른 시험을 해보았다.

게임의 주 메뉴만 켜도 4 백만 개의 *패키지(Packages)*들이 보내졌다.(패키지는 모든 할당, 재할당, 사용되지 않는 메모리 블록으로 구성) 평균적으로 이 패키지들의 크기는 93 바이트이다. 그러므로 주 메뉴에 들어갈 때 거의 400 메가바이트가 툴로 보내진다는 뜻이다.

MemAnalyze 의 기능

이 부분에서는 MemAnalyze 의 현재 기능과 이것들이 어떤 방식으로 메모리 문제를 해결하는지 설명할 것이다.

현재 상태 감시

툴을 실행하고 다른 프로그램과 연결이 되면 항상 단순히 현재 상태를 추적한다. 그림 1 을 보면 전체 통계가 표시되어 있다. 이때 버튼을 클릭하면 다음과 같은 형태로 통계를 분석할 수 있다.

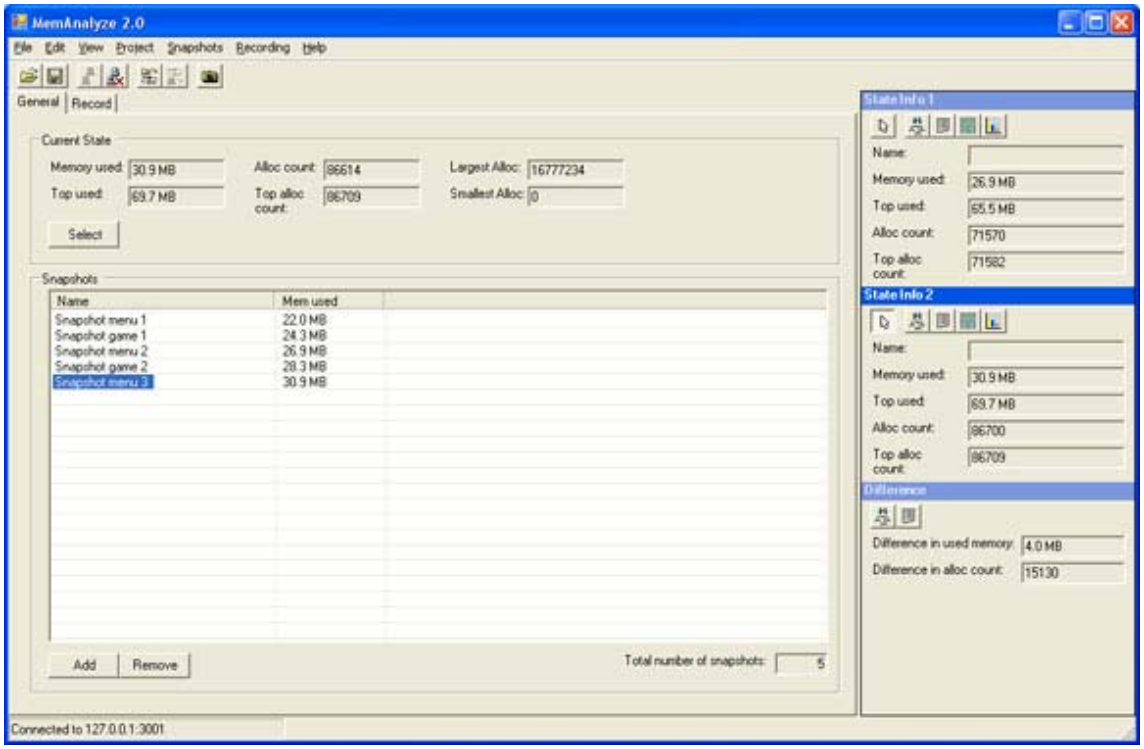

그림 1: 메인 화면. 상단에 전체 메모리 통계가 표시된다. 오른쪽에는 메모리 통계를 비교하기 위한 두 개의 메모리 상태 '슬롯'이 있다. 창의 나머지 부분에는 메모리 스냅샷 관리 목록이 표시된다.

1) CallGraph 분석. CallGraph를 표시한다. 계층 구조 표시(Hierarchy view)와 비슷한 것으로써 이전 아티클에서 이미 소개되었다. 각각의 기능을 위해 할당량, 전체 크기, 게임의 전체 할당 크기가 확률로 표시된다. 그러므로 매우 쉽게 메모리 소비의 주요 경로를 볼 수 있다. CallGraph의 아무 곳이나 클릭하면 소스 코드가 포함된 파일과 줄 정보가 표시된다.(그림 2 참조)

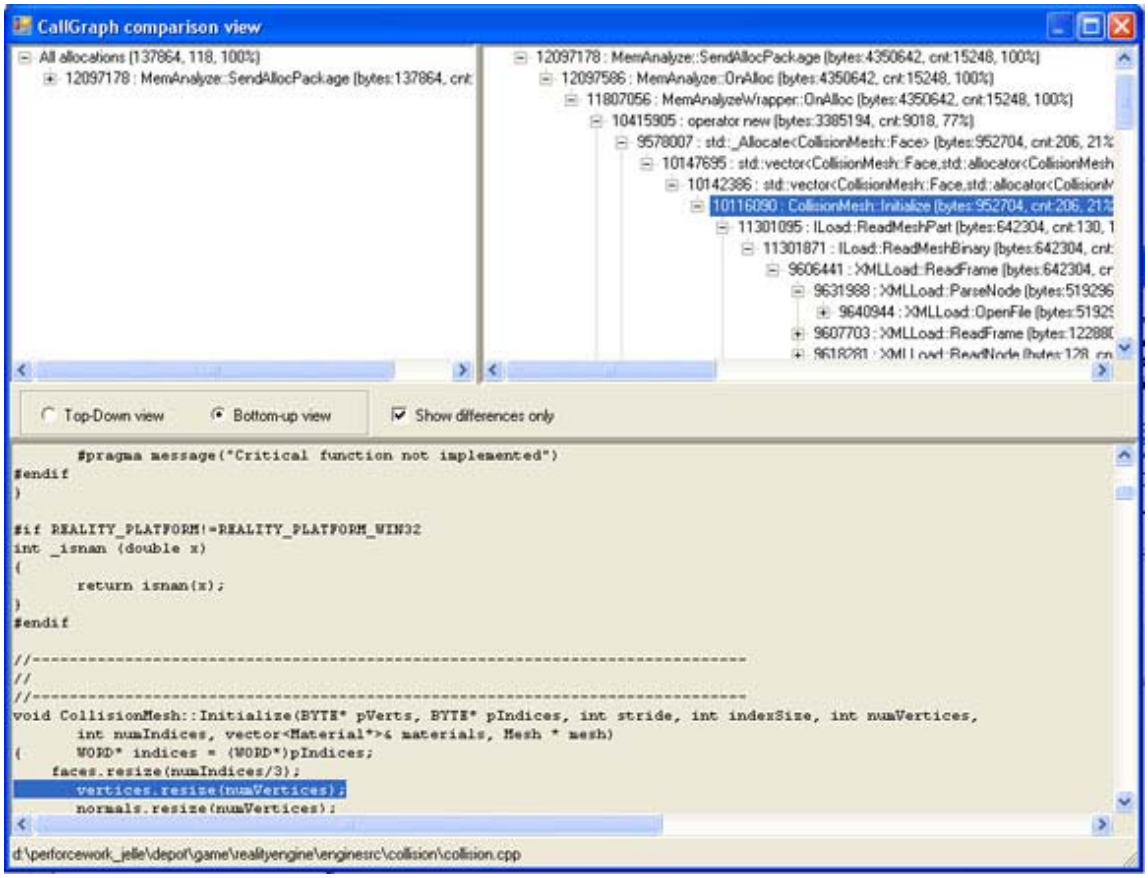

그림 2: CallGraph 보기.

2) TopX 분석. Xbox 프로파일러(Profiler)의 이름을 따 지었다. 직접 혹은 간접적으로 할당된 메모리를 이름, 할당 횟수, 총 할당량을 게임의 전체 할당량에 대한 비율로 표시된다. 이 모드에서는 어떤 방식으로든 자료를 정렬시킬 수 있다. 함수를 클릭하면 소스 코드가 표시된다.(그림 3 참조)

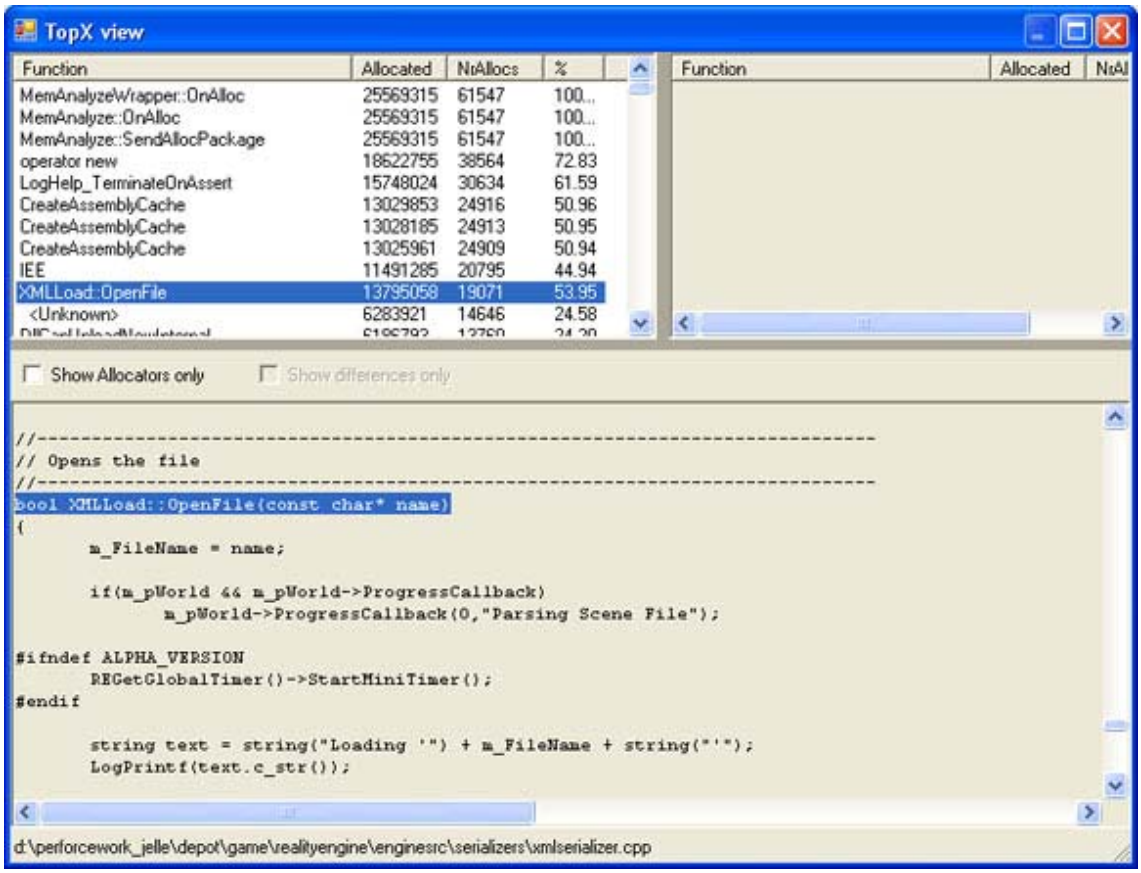

그림 3: TopX 보기.

3) **단편화(Fragmentation) 분석.** 물리/가상 메모리 블록들을 표시한다. 블록을 클릭 하면 블록의 호출 스택이 표시된다. 다른 보기 모드와 마찬가지로 함수를 클릭하면 소스 코드가 표시된다.(그림 4 참조)

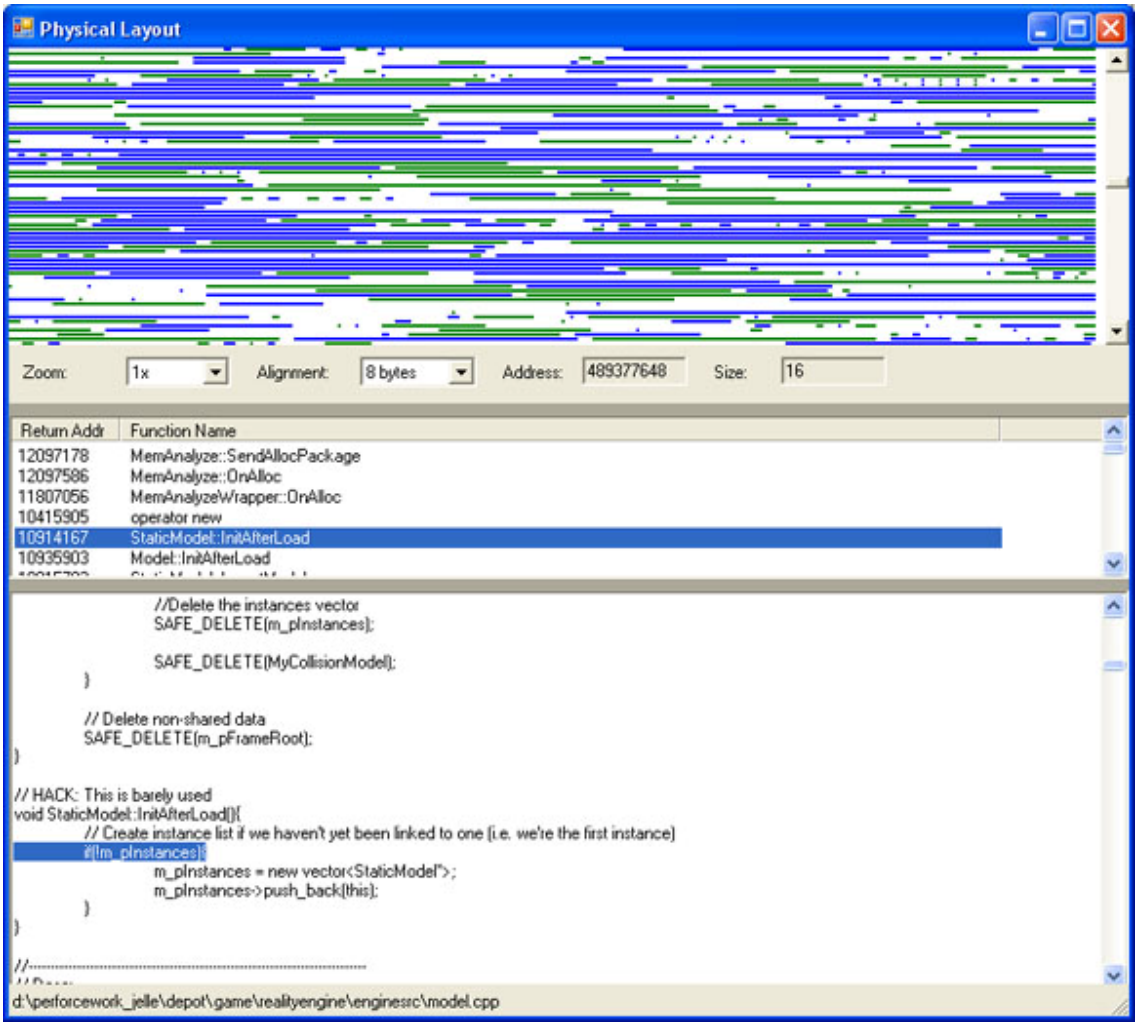

그림 4: 물리적 레이아웃(Physical layout) 보기.

4) 할당 전체 보기. 모든 할당을 크기나 횟수를 기준으로 정렬해 그래프로 나타낸다. 가로 축은 바이트 단위의 크기를 나타내고 세로 축은 할당 횟수나 총 할당 크기를 나타낸다.(그림 5 참조) 커스텀 메모리 관리자(Custom Memory Manager)를 작성하려 한다면 이 정보는 매우 도움이 될 것이다. 커스텀 메모리 관리자는 특정 할당 크기를 위해 최적화될 수 있다. 예를 들어 Boost의 memory pool 구현[1]이나 Windows XP와 Windows Server 2003 이 *낮은 단편화 힙*(*Low Fragmentation heap*)을 사용해 메모리 단편화를 줄이는가[2]를 보라.

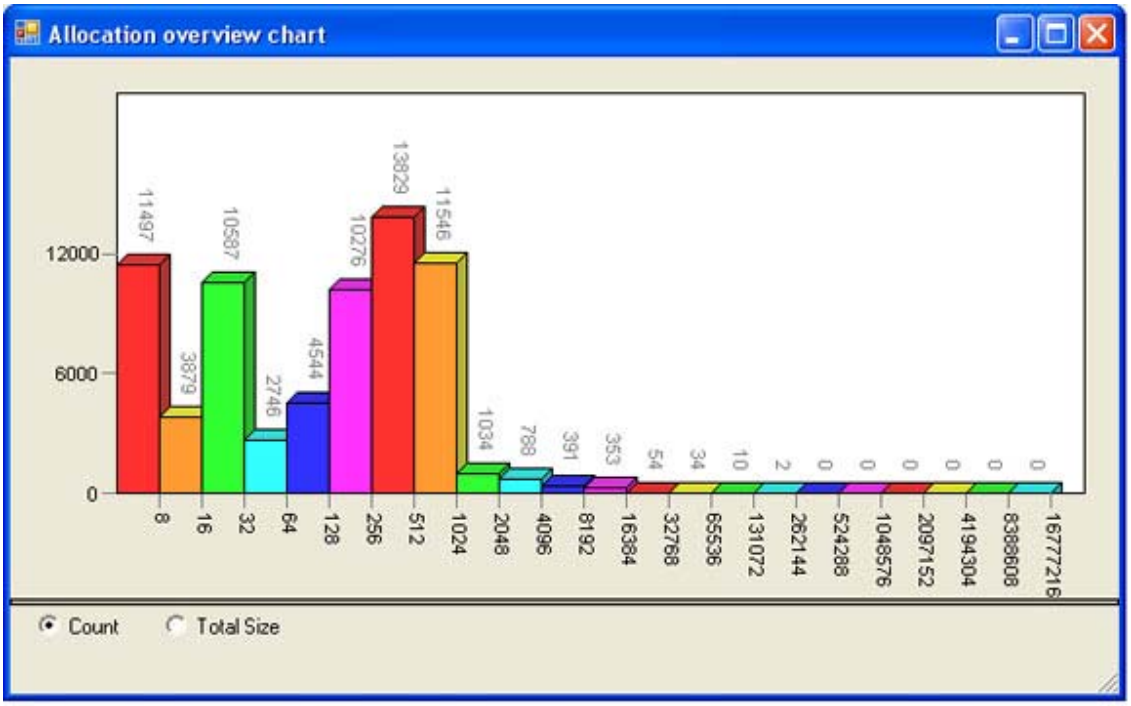

그림 5: 할당 전체 보기.

현재 상태의 스냅샷 만들기

툴이 게임에 연결되어 있을 때에는 언제나 버튼만 누르면 현재 메모리 상태를 스냅샷으로 만들 수 있다. 이 스냅샷은 목록에 보관된다.(그림 1 참조) 스냅샷은 현재 상태를 분석하는 방법과 동일한 방법으로 분석할 수 있다.

스냅샷은 특히 메모리 누수(Memory Leaks)를 찾는데 효과적이다. CallGraph 와 TobX 보기는 편리하게 서로의 상태를 비교하는데 효과적이다. 이 보기들은 듀얼 스크린(Dual Screen)을 지원하며 왼쪽 화면은 스냅샷 A 의 상태를 보여주고 오른쪽 화면은 스냅샷 B 의 상태를 보여준다.

이것으로 메모리 누수를 어떻게 찾을까? 이 보기들은 두 상태의 차이점만을 나타낸다.(그림 2 참조) 스냅샷들이 게임의 메모리 레이아웃이 동일해야 할 순간에 만들어 졌다면 스냅샷을 비교하여 스냅샷 B 에는 포함되어 있지만 스냅샷 A 에는 포함되지 않는 메모리 할당을 찾을 수 있다. 게임의 주 메뉴는 스냅샷을 비교하는데 안성맞춤이다. 이전 아티클에서 비교 방법에 대해 설명 했었다.

스냅샷들은 또한 게임에 의해 프로그램처럼 조작될 수 있다. 게임은 MemAnalyze API 를 통해 툴에 스냅샷을 만들도록 명령할 수 있으며 몇몇 스냅샷들은 게임에서 조작하는 것이 더 편리한다. 예를 들어 게임 종료 직전에 스냅샷을 만드는 것이 도움이 될 수도 있다.

메모리 상태 기록

메모리 상태를 기록하면 시간에 따른 메모리의 동작을 살펴볼 수 있다.(그림 6 참조) 기록의 어느 곳에나 커서를 둘 수 있으며 스냅샷이나 현재 상황과 같이 4 가지 방식의 분석할 수 있다. 이것만으로도 충분히 강력하지만 스냅샷을 비교하는 것처럼 기록을 비교할 수도 있다. 이 보기에서는 메모리 누수를 찾을 수 없을지도 모르지만 대신 매우 흥미로운 정보를 발견할 수도 있다. 다음의 두 가지 분석을 위해 이 기능을 사용한다.

- **런타임 할당(Runtime allocations)**. 콘솔 개발 경험을 가진 사람들은 런타임 할당이 가장 피해야 할 것임을 알 것이다. 할당에는 성능 오버헤드(Performance overhead)가 있으며 메모리 단편화를 증가시킬 수 있다. 이것은 기록 보기에서 쉽게 찾을 수 있다.

- **논리적 메모리 누수(Logical memory leaks).** 메모리 누수는 게임 중에 삭제되지 않은 적을 상상해 보라. 하지만 그런 것은 게임이 종료될 때 정상적으로 제거되는 경우가 많다. 예를 들어 전체 게임 객체 목록이 게임이 종료될 때 파괴되는 것처럼 말이다. 고전적인 메모리 관리자는 이런 종류의 누수를 추적할 수 없고 게임 후에도 객체가 할당된 것을 보고하지 못한다. 하지만 게임을 오랫동안 한다면 메모리 사용량은 메모리를 전부 사용할 때까지 증가한다. 이런 문제들은 해결하기 매우 힘들다.(나의 경험을 토대로 이야기 하는 것이다.) 이 보기는 메모리 크기가 늘어나는 것과 그 원인을 바로 찾아낼 수 있다. 예를 들자면 기록된 두 상태를 CallGraph로 비교할 수 있다.

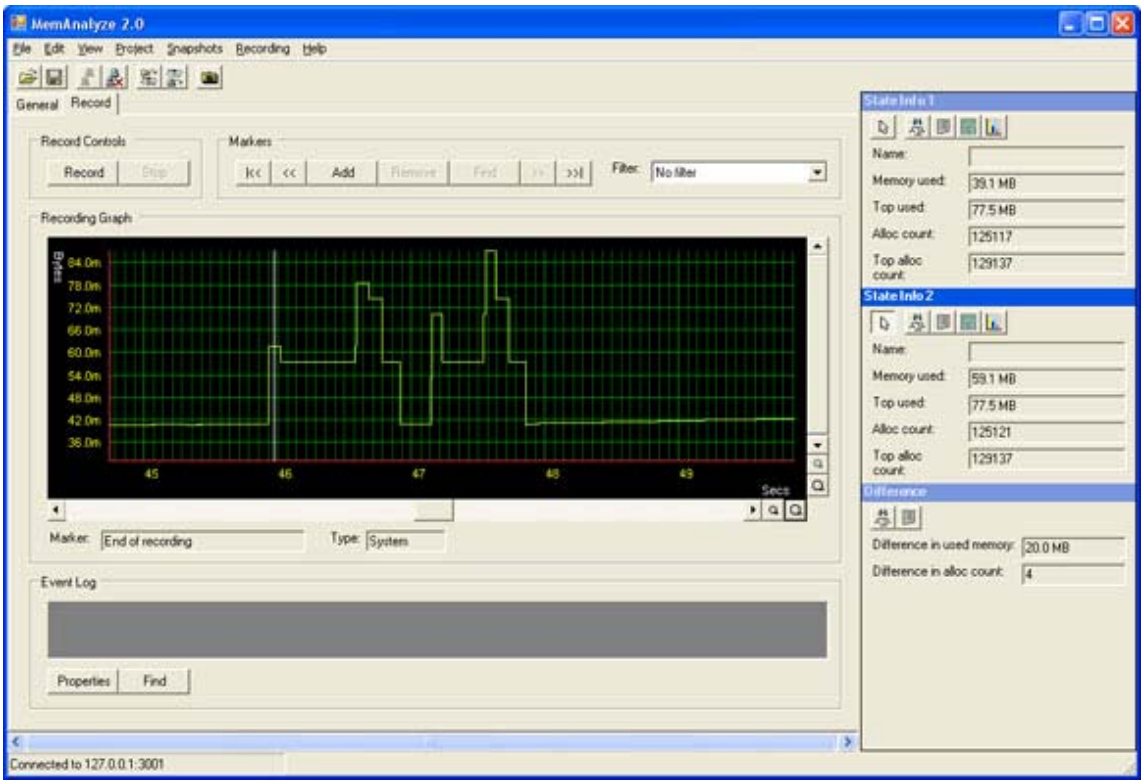

그림 6: 기록 보기. 이 부분은 게임 중에 기록되었다. 몇몇 부분은 매우 큰 라이트맵(Lightmaps)을 생성할 때 소모적인 할당을 보여준다. 이것은 최적화에 매우 효율적이다.

또한 MemAnalyze API 를 사용하면 툴에 의해 전송되는 스냅샷처럼 마커(Markers)를 보낼 수도 있다. 마커는 현재 상태를 보관하지 않는다는 점에서 스냅샷과 다르다. 대신 간단히 기록 보기에서 삽입할 수 있다. 각 프레임마다 마커를 툴로 보낼 수 있으며 마커를 통해 메모리 변화를 관찰할 수 있다.

구현(Implementation)

이전 아티클에서 이미 이 버전에 적용된 대부분의 기능을 PS2 와 Xbox 의 구현 세부 사항으로 다뤘었다. 이 아티클에선 다음과 같은 항목을 다루겠다.

- 모든 할당을 가로채는 방법.
- 호출 스택 정보를 얻는 방법.
- 심볼(Symbol) 정보를 읽는 방법.

이 부분에선 각 주제들을 PC 를 위해 다시 살펴볼 것이다. 몇몇 해결책들은 Xbox 개발에도 적용할 수 있다. 하지만 PC 에서만 검증이 이루어진 방법이다.

우선 모든 할당을 가로채는 방법부터 시작하겠다. 그리고 이전 아티클에서 설명한 StoreCallStack 함수의 개선된 버전에 대해 이야기하고 마지막으로 심볼 불러오기를 다시 살펴보겠다.

툴은 C#와 .NET framework 를 이용해 제작되었다. C#의 세부적인 내용을 사용할 수 있을 경우 해당 내용을 표기해 놓았다.

모든 할당 가로채기

메모리 사용량을 좋은 모양새로 보려면 가능한 많은 할당들을 가로채야 한다. 가장 좋은 방법은 할당 연산자를 새로 작성해 오버로딩(Overloading)하는 것 이다. 게임의 생성(new)과 삭제(delete) 할당을 간섭해도 malloc, LocalAlloc, GlobalAlloc 혹은 HeapAlloc 과 같은 직접 할당은 포함되지 않는다. 큰 덩어리의 게임

메모리 기록은 저수준 할당(Low level)을 사용하는 DirectX 로 구성되었을 수도 있다. 아시다시피 PC 는 이 저수준 할당을 가로채는 것을 지원하지 않는다. Xbox 는 XbMemAlloc 덕분에 할당 추적을 단순화할 수 있다. PC 에서 할당을 가로채는 최선의 방법은 위에서 살펴본 생성과 삭제 연산자를 오버로딩해서 가로채는 방법으로 보인다.

모든 할당을 가로채는 방법이 존재하기는 하지만 해결책은 명확하지 않는다. 이 해결책은 저수준 할당 함수을 가진 DLL 파일을 후킹(Hook)하는 것 이다. 이 방법의 가장 좋은 점은 해당 DLL 을 사용하는 모든 실행 중인 프로세스에 간섭할 수 있게 된다는 것이다. 말 그대로 실행 중인 프로세스의 모든 할당을 가로챌 수 있게 된다.

DLL파일을 후킹하는 방법은 매우 복잡하지만 다행히도 이 정보들을 라이브러리에 적어둔 사람들이 있다. 첫 번째 것은 *Microsoft Research*[3]의 Detours이다. 좀 더 빨리 시작하려면 tracemem의 샘플을 살펴 보라. 이 샘플은 HeapAlloc 함수에 후킹을 했다. 이 라이브러리는 비상업적 목적을 위해선 무료이다.

두 번째 방법은 *Matt Conover*[4]의 라이브러리를 사용하는 것이다. 이 라이브러리는 사용하기가 좀 더 어렵기는 하지만 두 가지의 장점이 있다. 첫째로 이 라이브러리는 상업적으로 사용하더라도 무료이며 둘째로 Matt이 자신의 소프트웨어가 제대로 작동하도록 작성했다는 것이다. 패키지에 포함되어 있는 *HeapHook* 샘플을 살펴 보라.

불행하게도 후킹을 사용하는 방법은 매우 위험하며 위의 것만 가지고는 충분하지 않다. 위의 방법들을 사용하면 프로그램에 의해 수행되는 *모든* 할당들이 기본 할당 함수가 아닌 우리가 새로 작성한 할당 함수에 의해 수행된다는 것을 명심하라. 새로 작성한 힙(heap) 콜백(callback) 함수 내부에서 힙(heap)에 메모리를 할당하는 함수를 호출하는 것과 같은 행동을 하는 날에는 무한 루프(Endless loop)에 빠지게 될 것이며 이런 경우는 소켓(Sockets)을 사용할 때 흔히 일어날 수 있는 일이다. 아마도 소켓을 사용할 때 얼마나 할당이 빈번하게 일어나는지를 알게 되면 놀라게 될 것이다. 이런 무한 루프를 방지하려면 힙 콜백 함수 내부에서 재귀(Recursion) 횟수를 확인하면 될 것이다. 이것으로 문제가 끝일까? 아니다. 이제 더 심각한 경우를 살펴보도록 하겠다.

어떤 스레드(Thread)가 소켓을 사용하고(비동기[Asynchronous] 방식을 사용해서) 해당 소켓이 어떤 작업을 하는 와중에 메모리 할당을 하는 경우를 살펴보자. 그러면 새로 작성한 할당 함수가 호출되며 또 다른 새로 작성된 소켓이 작업을 시작할 것이다. 이럴 경우에는 보통 소켓 라이브러리의 내부 상태를 심각하게 망가뜨리게 된다. 게다가 이런 경우에는 콜백 함수 내부에서 재귀 횟수를 확인한다고 해도 도움이 안되므로 이런 경우에 다른 해결책이 필요한다.

지금 아주 위험한 영역을 다루고 있다는 점을 떠올려 볼 때 아무래도 제가 처음으로 이 문제의 해결을 시도했을 때 일어났던 일을 말씀드리는 것이 좋을 듯하다. 나의 해결책은 할당 함수를 어떤 종류의 모듈이 호출했는지를 확인하는 것이었다. 여기서 소켓 DLL 중의 하나가 할당 함수를 호출했다면 어떤 소켓 작업도 하지 않는 것이었다. 개인적으로 이 해결책이 마음에 안 들었던 것은 둘 째 치더라도 제대로 작동하지도 않았다. 새로 작성한 콜백의 반환 주소(Return Address)를 소켓 프로세스의 주소 영역(Address Range)과 맞추려고 했었다. 하지만 DbgHelp 라이브러리를 사용해 소켓 프로세스에 대한 정보를 얻어오려고 하자 프로그램이 충돌(Crash)을 일으켰다. 충돌이 일어난 시점을 알아보니 DLL 을 불러오면서 응용 프로그램이 할당을 수행하는 시점이었다. DbgHelp 는 새로운 프로세스를 불러오는 도중에 프로세스에 대한 정보를 얻어올 수 없어서 어쩔 수 없이 다른 해결책을 찾아 나섰다.

문제를 해결할 수 있는 방법은 의외로 상대적으로 간단한 방법이었다. 툴과 서로 통신을 할 수 있는 일꾼 스레드(Worker thread)가 있으면 되는 것이었다. 할당이 수행될 때마다 메타 데이터를 수집해서 버퍼(Buffer)에 저장한 후에 일꾼 스레드를 생성하고 이렇게 생성된 일꾼 스레드가 클라이언트(Client) 프로그램에게 데이터를 전송하는 것이다. 이 방식을 사용하면 어떤 경우의 재귀 호출도 일어나지 않는다. 게다가 원한다면 새로 작성한 소켓에서 할당을 수행하는 것까지도 모니터링할 수 있다.

이제 문제는 해결되었지만 주의해야 할 점이 한 가지가 더 있다. 그림 7 에 나타난 것과 같은 이벤트의 연쇄(Chain) 작용으로 인해 데드락(deadlocks)이 발생할 수도 있다는 것이다. 이런 경우는 이중 버퍼링(Double Buffering)을 사용하고 이중 버퍼링 구조에만 크리티컬 섹션(Critical Sections)을 사용해 보호하는 것으로 피할 수 있다.

그러면 할당 콜백 함수가 전송 작업이 완료될 때까지 대기해야 할 필요성이 사라지게 된다. 이 방법의 단점은 데이터 버퍼(Databuffer)가 여러 개의 할당 이벤트의 데이터를 저장할 수 있을 만큼 커야 한다는 것이다. 장점으로는 이런 방식의 일괄처리(Batching)를 사용하면 전송 작업의 속도가 빨라질 수도 있다는 것이다.

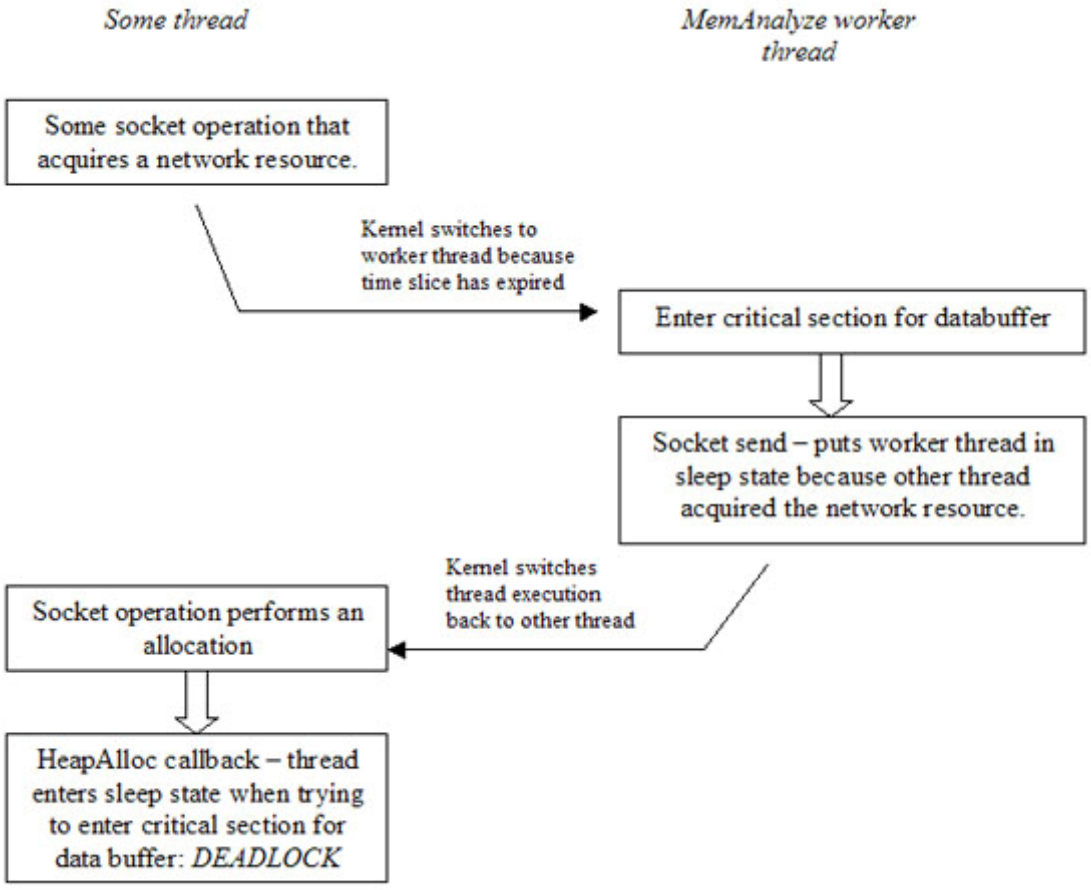

그림 7: 데드락 상황을 야기시킬 수 있는 이벤트의 연쇄 작용.

지금쯤이면 눈치챘겠지만 이 방법은 그리 쉽지만은 않은 방법이다. 생성과 삭제 연산자 오버로딩부터 해보는 것을 권한다. 생성과 삭제 연산자를 제대로 오버로딩하는 방법을 알고 싶으시다면 Scott Meyer씨의 Effective C++ [10] 중 제 8 장을 읽어보라.

PC를 위한 향상된 호출 스택 추적

이전의 아티클은 몇몇 플랫폼을 위한 호출 스택 추적에 대한 해결책을 포함하고 있었지만 PC 를 위한 개선 방법이 발견 되었다. PC 버전은 정상적으로 작동 하는 저수준 구현을 갖고 있지만 한가지 약점이 있다. 모든 모듈이 컴파일러 최적화 설정인 '프레임 포인터 생략(omit frame pointer)'을 꺼야 한다는 것이다. 자신이 직접 제작하는 프로젝트에서는 해당 기능을 사용할 수도 있겠지만 다른 회사의 프로젝트를 구입해서 사용하는 경우에는 어쩔 수가 없다. 이런 제한으로 인해 종종 스택 프레임이 알 수 없는 메모리를 가리켜서 아래와 같은 버전의 StoreCallStack 함수가 충돌을 일으키기도 한다.

목록 1 에 표기된 버전의 StackWalk64 함수는 Windows Platform SDK 의 일부인 *DbgHelp* 라이브러리에서 발췌한 것이다.

```
unsigned int StoreCallStack(
            unsigned int* returnAddresses,
            unsigned int nrReturnAddresses)
{
   unsigned int nrItemsWritten = 0;
   STACKFRAME64 stackFrame;

   unsigned int* ebpReg = 0;
   __asm mov [ebpReg],ebp

   if(ebpReg != NULL)
   {
      unsigned int frame = 0;
      unsigned int address = 0;
      frame = ebpReg[0];
      address = ebpReg[1];

      memset(&stackFrame, 0, sizeof(stackFrame));
      stackFrame.AddrPC.Offset = (DWORD64)address;
      stackFrame.AddrPC.Mode = AddrModeFlat;
      stackFrame.AddrFrame.Offset = (DWORD64)frame;
      stackFrame.AddrFrame.Mode = AddrModeFlat;

      while(nrItemsWritten < nrReturnAddresses)
      {
         if(StackWalk64(
            IMAGE_FILE_MACHINE_I386,
            GetCurrentProcess(),
            GetCurrentThread(),
            &stackFrame,
            NULL,
            NULL,
            SymFunctionTableAccess64,
            SymGetModuleBase64,
            NULL))
         {
            address = (unsigned int)stackFrame.AddrPC.Offset;
            if(address == NULL)
            {
               returnAddresses[nrItemsWritten] = 0;
               break;
            }
            else
            {
               returnAddresses[nrItemsWritten] = address;
            }
            nrItemsWritten++;
         }
         else
         {
            break;
         }
      }
   }

   return nrItemsWritten;
}
```

Listing 1: The new StoreCallStack function for x86.

차후에 *Jochen Kalmbach* [5]이 제작한 샘플을 소개하겠다. 이것은 목록 1 에 표기된 것보다 훨씬 광대한 것이다. StackWalk64 가 사용한 알고리즘이 이전 아티클의 StoreCallStack 함수보다 훨씬 느리다는 것을 알아두라. Microsoft 포럼에서 소개된 기사[11] 에 따르면 *DIA SDK* 의 stackwalk 함수는 DbgHelp 라이브러리의 stackwalk보다 빠르다고 하니 한 번 살펴볼 가치가 있다.

심볼 불러오기

계속 하기 전에 함수 주소에 대해 간단히 요약해 보겠다. 호출 스택 정보를 얻을 때는 항상 함수의 절대 주소(Absolute Address)를 얻어온다.(해당 시점의 메모리 상의 함수 주소) 이미지(image), 예를 들어 실행 파일이나 DLL 은 실행 시 메모리의 다른 장소에 적재될 수도 있다. 메모리 상에서 이미지의 위치는 *이미지 베이스(Image base)*라 불린다.

함수의 절대 주소에서 이미지 베이스 주소를 빼면 이미지 베이스와의 상대 주소를 얻을 수 있으며 이 상대 주소는 상대적인 가상 주소(Relative Virtual Address, RVA)라고도 불린다. RVA 는 프로그램 데이터베이스(Program database, PDB)와 같은 것에서 심볼 정보를 얻는데 사용할 수 있다.

DLL 이 사용되면 좀 더 복잡해진다. 호출 스택에 DLL 의 경계를 넘어서는 주소가 담겨있을 수도 있기 때문이다. 이점은 나중에 몇 가지 예를 들어 설명하겠다.

처음 시작하는 사람들을 위해 말하지만 RVA 를 툴 밖으로 전송 하는 것은 불가능하다. 그렇게 했다가는 다른 상대 주소들과 충돌이 발생할 수 있기 때문이다. 서로 다른 DLL 에 들어 있는 서로 다른 함수의 경우에 절대 주소가 서로 다르더라도 서로 동일한 상대 주소를 가지고 있을 수도 있다. 이럴 경우에 RVA 를 전송하면 툴의 분석 알고리즘과 심볼 참조 알고리즘 모두를 망가뜨릴 것이다.

그리므로 다양한 모듈을 사용하는 플랫폼에선 절대 주소를 보내야 한다. 그 후에는 어떤 방식으로 툴이 심볼 정보를 불러올 때 사용할 어떤 이미지가 불러져 있고 해당 이미지들의 베이스 주소를 알아낼

것인지의 문제가 발생한다. 이제부터 이 두 가지 문제에 대한 해결책을 알아보겠다.

해결책 1: SymInitialize를 통해 심볼 정보 불러오기.

PC 에서는 *DbgHelp* 라이브러리를 사용해 심볼 정보를 불러올 수 있다. 그리고 첫 번째 해결책으로 사용할 것은 *DbgHelp* 라이브러리의 SymInitialize 함수를 사용하겠다.

```
BOOL SymInitialize(

HANDLE hProcess,

PCTSTR UserSearchPath,

BOOL fInvadeProcess

);
```

첫 번째 매개변수는 프로세스의 핸들(Handle)이다. 적합한 프로세스의 핸들은 넘겨주면 SymInitialize 함수는 해당 프로세스가 불러온 모든 모듈을 나열하고 심볼 파일을 찾는다. 심볼 파일을 찾을 때는 기본 경로를 검색하게 된다. 하지만 두 번째 매개변수인 UserSearchPath 를 사용하면 기본 경로 외에 추가적인 경로에서도 파일 검색을 할 수 있다.

이 방법에는 다음과 같은 장점과 단점이 있다.

장점:

- 매우 간편하고 강력하다. 이 함수는 자체적으로 불러온 모든 모듈에 대한 심볼 정보를 찾는다.
- 이 함수는 심볼 정보를 불러올 이미지 베이스 주소를 알고 있다.

단점:

- 툴은 이 방법을 게임이 네트워크를 통해 진행되지 않을 때에만 사용할 수 있다.
- 툴은 이 방법을 게임이 실행 중일 때만 사용할 수 있다. '프로젝트'를 저장하고 오프라인 분석을 할 때에는 프로세스가 실행 중이 아니기 때문에 이 방법을 사용할 수 없다.

툴을 실행 중인 프로세스의 디버거(Debugger)로 사용할 때와 같이 위의 문제점이 별로 문제가 되지 않는다면 심볼 정보를 불러올 때 이 방법을 사용하라. 목록 2 는 이 방법을 사용하는 방법을 보여주는 샘플 프로그램이다.

```
#include "stdafx.h"
#include <windows.h>
#include <iostream>

_declspec(noinline) unsigned int GetProgramCounter()
{
   unsigned int result;
   _asm mov eax, dword ptr[ebp + 4]
   _asm mov [result], eax
      return result;
}

_declspec(noinline) void Foo()
{
   unsigned int programCounter = GetProgramCounter();
   std::cout << "Function address: 0x" <<
      std::hex << programCounter << std::endl;
}

int main(int argc, char* argv[])
{
   int processID = GetCurrentProcessId();
   std::cout << "process id: 0x" <<
      std::hex << processID << std::endl;

   Foo();

   Sleep(INFINITE);

   return 0;
}
```

Listing 2: The SomeDummyProcess program.

코드를 한 번 보자. 이 프로그램은 현재 실행 중인 프로세스의 Id와 Foo라는 함수의 주소 영역을 출력하는 매우 간단한 Win32 콘솔 응용 프로그램이다. 직접 명령 포인터(Instruction Pointer, EIP)를 읽어올 수가 없기 때문에 반환 주소(Return Address)를 이용해 명령 포인터를 얻어오는 *Dan Moulding*의 편법을 사용하겠다. *Dan Moulding*의 아티클은 [6]으로 가면 볼 수 있다. 또 다른 방법으로는 Microsoft 전용이기는 하지만 x86 이외의 아키텍처(Architectures)에서도 작동하는 ReturnAddress 표기(intrinsic)를 사용하는 방법이다.

출력값을 출력한 후에는 프로세스가 종료되는 것을 막기 위해 프로그램은 수면 상태(Sleep State)로 들어간다.

프로그램의 출력값은 현재 프로세스 Id와 Foo라는 함수의 주소이다.

process id: 0xe9c

Function address: 0x411553

목록 3은 프로세스 Id와 함수 주소를 활용하는 프로그램이다.

```
#include "stdafx.h"
#include <windows.h>
#include <iostream>
#include <stdexcept>
#include <sstream>
#include <psapi.h>
#include <DbgHelp.h>

using namespace std;

string FindFunction(DWORD64 address, HANDLE process)
{
   string result = "Unknown";

   ULONG64 buffer[
      (sizeof(SYMBOL_INFO) +
         MAX_SYM_NAME * sizeof(char) +
         sizeof(ULONG64) - 1) /
         sizeof(ULONG64)];
      PSYMBOL_INFO pSymbol = (PSYMBOL_INFO)buffer;
      DWORD64 displ = 0;

      pSymbol->SizeOfStruct = sizeof(SYMBOL_INFO);
      pSymbol->MaxNameLen = MAX_SYM_NAME;

      if(::SymFromAddr(process, address, &displ, pSymbol))
      {
         result = pSymbol->Name;
      }
      else
      {
         DWORD error = ::GetLastError();
         ostringstream str;
         str << "SymFromAddr returned error " << error;
         throw exception(str.str().c_str());
      }

      return result;
}

void Init(HANDLE process)
{
   ::SymSetOptions(
      SYMOPT_UNDNAME |
      SYMOPT_DEFERRED_LOADS |
      SYMOPT_LOAD_LINES);

   if(!::SymInitialize(process, NULL, TRUE))
   {
      throw exception(
         "Could not initialize symbol loader!");
   }
}

void Exit(HANDLE process)
{
   ::SymCleanup(process);
}

int main(int argc, char* argv[])
{
   HANDLE processHandle = NULL;
```

```
  try
  {
     if (argc != 3)
     {
        throw invalid_argument(
           "Usage: FileLineInfo <processId> <absFunctionAddress>");
     }

     DWORD processId;
     istringstream iss1(argv[1], ios_base::in);
     iss1 >> hex >> processId;

     processHandle = ::OpenProcess(
        PROCESS_QUERY_INFORMATION|PROCESS_VM_READ,
        FALSE, processId);
     if(processHandle == NULL)
     {
        throw invalid_argument(
           "Process ID does not refer to a running process!");
     }
     char fileName[MAX_PATH];
     ::GetModuleFileNameExA(
        processHandle,
        NULL,
        fileName,
        sizeof(fileName));
     cout << "Process refers to: " << fileName << endl;

     Init(processHandle);

     DWORD64 address;
     istringstream iss2(argv[2], ios_base::in);
     iss2 >> hex >> address;

     string name = FindFunction(address, processHandle);
     cout << "Function name: " << name << endl;
  }
  catch(exception& ex)
  {
     cerr << "Error: " << ex.what( ) << endl;
  }

  if(processHandle != NULL)
  {
     Exit(processHandle);
     ::CloseHandle(processHandle);
  }

  return 0;
}
```

Listing 3: The FindFunctionInProcess program.

이 프로그램은 프로세스 Id 와 함수의 절대 주소의 두 가지 매개변수를 입력 받는다. 다른 프로그램이 실행 중이라면 목록 3 의 프로그램에 첫 번째 프로그램이 출력한 출력값을 매개변수로 사용할 수 있다. 코드를 보시면 프로세스의 핸들을 얻기 위해 OpenProcess 를 어떻게 사용했는지를 볼 수 있다.

이렇게 얻은 핸들은 모든 *DebugHlp* 함수를 호출할 때 사용된다. 특히 SymInitialize 를 호출할 때 fInvadeProcess 매개변수를 TRUE 로 설정하는 부분을 눈 여겨 보라. 이것은 프로세스가 사용하는 모든 모듈에서 자신들의 디버그 정보를 불러올 것이라는 것을 확실하게 보장한다. 디버깅을 위해 GetModuleFileNameEx 함수를 이용해 실행 중인 프로세스의 파일 이름을 출력한다.

위의 두 가지 프로그램을 컴파일(Compile)하고 실행하려면 Win32 콘솔 프로젝트를 생성한 후에 각각의 *main.cpp* 파일에 위의 코드를 붙여 넣으시고 링크 설정(Link Settings)에서 *DebugHlp.lib* 와 *psapi.lib* 가 프로그램에 링크되도록 설정하라. psapi.lib 는 GetModuleFileNameEx 함수 링크에 필요한 라이브러리이다. 본 코드들은 Visual Studio 2005 에서 시험되었다.

참고: 실행 전에 DbgHelp.dll이 최신 버전인지 확인하라. DbgHelp.dll은 Debugging Tools for Windows[7]에 포함되어 있다.

다음의 프로그램 출력값은 Id 와 일치하며 Foo 라는 이름을 가진 함수를 포함하고 있는 프로세스를 찾았다는 것을 나타낸다.

```
Process refers to:
f:₩work₩tests₩somedummyprocess₩debug₩somedummyprocess.exe

Function name: Foo
```

이 샘플에서 볼 수 있듯이 프로세스의 핸들을 얻기 위해서는 프로세스의 Id 가 필요하다. 프로세스의 핸들은 심볼 정보를 불러올 때 필요한 것이다. 그렇다면 툴에서는 어떻게 프로세스의 Id 를 얻을

수 있을까? 물론 게임이 GetCurrentProcessId 를 사용해 자신의 프로세스 Id를 얻은 후에 그것을 툴로 전송하면 된다.

이 방법은 작동은 잘 되지만 네트워크 부분에서 그리 반갑지 않은 특정 플랫폼 전용 구현을 야기시키게 된다. 하지만 툴이 실행 중인 모든 프로세스들을 조회한 후에 사용자가 심볼 정보를 얻어올 프로세스를 선택할 수 있도록 대화 상자에 출력하는 방법도 있다. 이 방법은 Visual Studio 에서 디버깅을 위해 프로세스에 연결하는 방식과 같다. 이 해결책도 물론 특정 플랫폼에 종속적이기는 하지만 어떤 방식을 사용하건 간에 심볼을 불러오는 것은 플랫폼에 종속적일 수밖에 없다. 그러면 이제 어떻게 구현할 것인지를 살펴보겠다.

프로세스를 조작하는 것은 *프로세스 상태 도우미(Process Status Helper)* API 인 *PSAPI* 라이브러리를 사용하면 된다. 또한 *PSAPI* 는 Windows Platform SDK 의 일부이다. 이미 목록 3 에서 이 라이브러리에 속해있는 함수인 GetModuleFileNameEx 를 사용했었다. 다음의 목록 4 는 프로세스를 열거하고 프로세스 핸들을 얻어오는 방법을 보여준다.

```
#include "stdafx.h"
#include <windows.h>
#include <stdio.h>
#include <tchar.h>
#include <iostream>
#include <string>
#include "psapi.h"

std::string GetFileName(const char* fullPath)
{
   char file[_MAX_FNAME], ext[_MAX_EXT];
   _splitpath_s(fullPath, NULL, 0, NULL, 0,
      file, sizeof(file), ext, sizeof(ext));
   strcat_s(file, sizeof(file), ext);

   return file;
}

void PrintProcessNameAndID(DWORD processID)
{
   char processName[MAX_PATH] = "<unknown>";

   HANDLE processHandle =
      ::OpenProcess(PROCESS_QUERY_INFORMATION |
               PROCESS_VM_READ,
               FALSE, processID );

   if(NULL != processHandle)
   {
      char imageName[MAX_PATH];
      ::GetProcessImageFileNameA(
         processHandle,
         imageName,
         sizeof(imageName));

      std::cout << GetFileName(imageName) <<
         "  (Process Id: 0x" <<
         std::hex << processID << ")" << std::endl;

      ::CloseHandle(processHandle);
   }
}

int main(int argc, char* argv[])
{
   DWORD processes[1024], nrBytesWritten, nrProcesses;

   unsigned int i;
   int result = -1;
```

```
   if(::EnumProcesses(
      processes, sizeof(processes),
      &nrBytesWritten))
   {
      nrProcesses = nrBytesWritten / sizeof(DWORD);

      for(i = 0; i < nrProcesses; i++)
      {
         PrintProcessNameAndID(processes[i]);
      }

      result = 0;
   }

   return result;
}
```

Listing 4: The EnumerateProcesses program.

프로그램의 출력값은 실행 중인 프로세스의 목록과 Id를 나타낸다.

SMSS.EXE (Process Id: 0x21c)

WINLOGON.EXE (Process Id: 0x278)

SERVICES.EXE (Process Id: 0x2a4)

LSASS.EXE (Process Id: 0x2b0)

ATI2EVXX.EXE (Process Id: 0x340)

SVCHOST.EXE (Process Id: 0x360)

SVCHOST.EXE (Process Id: 0x3ec)

ATI2EVXX.EXE (Process Id: 0x478)

CLI.EXE (Process Id: 0x9d4)

CLI.EXE (Process Id: 0xf8c)

NSCSRVCE.EXE (Process Id: 0x974)

VCExpress.exe (Process Id: 0x950)

dexplore.exe (Process Id: 0xca4)

mspdbsrv.exe (Process Id: 0xf0)

WINWORD.EXE (Process Id: 0xf18)

msmsgs.exe (Process Id: 0x65c)

EnumerateProcesses.exe (Process Id: 0x118)

해결책 2: EnumerateModules64 와 SymLoadModule64 를 사용해 심볼 정보 불러오기

지금까지는 SymInitialize 를 사용해서 심볼을 불러오는 방법에 대해 논의했다. 이전에 말했던 것과 같이 이 방법은 오직 로컬 프로세스(Local Process)를 디버깅할 때에만 적용할 수 있다. 이제는 그런 제한이 없는 해결책에 대해 논의를 해보겠다.

심볼에 대한 정보가 필요할 때 툴이 게임에게 불러온 모듈에 대한 정보를 요구하게 만들 수도 있다. 오직 단 하나의 이미지만 불러올 수 있는 콘솔 플랫폼에서는 단순히 이미지 베이스와 이미지 이름을 반환하면 된다. 하지만 다수의 이미지를 불러올 수 있는 플랫폼에서는 불러온 모듈의 목록과 함께 그에 해당하는 이미지 베이스를 모두 반환해야 한다. 그러므로 해야 할 일은 SymInitialize 의 작동을 흉내 내는 것이다. EnumerateModules64 를 사용하면 프로세스가 불러온 모든 모듈을 열거할 수 있다. 그리고 이 함수를 사용하면 불러온 모든 모듈의 이미지 베이스, 이름, PDB 파일의 위치까지도 찾을 수 있다. 목록 5는 그 방법을 보여준다.

```
#include "stdafx.h"
#include <iostream>
#include <windows.h>
#include <psapi.h>
#include <dbghelp.h>

std::string GetSymbolTypeString(SYM_TYPE symType)
{
   std::string result;

   switch(symType)
   {
   case SymCoff:
      result = "COFF symbols.";
      break;
   case SymCv:
      result = "CodeView symbols.";
      break;
   case SymDeferred:
      result = "Symbol loading deferred.";
      break;
   case SymDia:
      result = "DIA symbols.";
      break;
   case SymExport:
      result = "Symbols generated from a DLL export table.";
      break;
   case SymNone:
      result = "No symbols are loaded.";
      break;
   case SymPdb:
      result = "PDB symbols.";
      break;
   case SymSym:
      result = ".sym file.";
      break;
   case SymVirtual:
      result = "Virtual module.";
      break;
   }
   return result;
}

BOOL CALLBACK Enumerate(
                  PSTR moduleName,
                  DWORD64 moduleBase,
                  ULONG moduleSize,
                  PVOID userContext)
{
   HANDLE currentProcess = GetCurrentProcess();

   DWORD64 imageBase = ::SymLoadModule64(
      currentProcess,
      NULL,
      moduleName,
      NULL,
      moduleBase,
      moduleSize);
```

```
    IMAGEHLP_MODULE64 moduleInfo;
    memset(&moduleInfo, 0, sizeof(moduleInfo));
    moduleInfo.SizeOfStruct = sizeof(moduleInfo);
    ::SymGetModuleInfo64(currentProcess, imageBase, &moduleInfo);

    ::SymUnloadModule64(currentProcess, imageBase);

    std::string symbolTypeString = GetSymbolTypeString(moduleInfo.SymType);

    std::cout <<
       "name: " << moduleName << std::endl <<
       "   base: " << moduleBase << std::endl <<
       "   size: " << moduleSize <<std::endl <<
       "   symbol type: " << symbolTypeString.c_str() << std::endl <<
       "   pdb: " << moduleInfo.LoadedPdbName << std::endl <<
       std::endl;

    return true;
}

void EnumAllModules()
{
    char path[_MAX_PATH];
    ::GetModuleFileNameA(GetModuleHandle(NULL), path, sizeof(path));

    char file[_MAX_FNAME], ext[_MAX_EXT];
    _splitpath_s(path, NULL, 0, NULL, 0,
       file, sizeof(file), ext, sizeof(ext));
    strcat_s(file, sizeof(file), ext);

    ::EnumerateLoadedModules64(GetCurrentProcess(),
       (PENUMLOADED_MODULES_CALLBACK64)Enumerate, file);
}

int main(int argc, char* argv[])
{
    HANDLE currentProcess = GetCurrentProcess();

    ::SymInitialize(currentProcess, NULL, FALSE);

    EnumAllModules();

    ::SymCleanup(currentProcess);

    return 0;
}
```

Listing 5: EnumerateModuleInfo program.

프로그램의 출력값은 실행 중인 모듈에서 얻어온 정보들을 나타낸다.

name:
D:₩work₩Tests₩DisplayLoadedModules₩debug₩DisplayLoadededModules.exe

base: 4194304

size: 118784

symbol type: PDB symbols.

pdb: .₩DisplayLoadedModules.pdb

name: C:₩WINDOWS₩system32₩ntdll.dll

base: 2089811968

size: 745472

symbol type: Symbols generated from a DLL export table.

pdb:

name: C:₩WINDOWS₩system32₩kernel32.dll

base: 2088763392

size: 1040384

symbol type: Symbols generated from a DLL export table.

pdb:

name: D:₩work₩Tests₩DisplayLoadedModules₩debug₩dbghelp.dll

base: 50331648

size: 1134592

symbol type: Symbols generated from a DLL export table.

pdb:

이 정보는 툴로 바로 전송할 수 있다. 이런 식으로 툴이 이미지 정보를 요청하는 것은 쌍방향 통신을 지원하는 네트워크 연결이 필요하다.

이제 프로그램 데이터베이스의 이름을 얻었으므로 SymLoadModule64 를 사용해 불러올 수 있게 되었다. SymLoadModule64 를 사용해 정보를 불러오는 프로그램의 코드 조각은 이전 아티클에 포함되어 있다.

참고: 앞서 말한 DbgHelp.dll 의 최신 버전을 준비하란 것은 그냥 한 말이 아니다. 구버전의 DbgHelp.dll 의 버그를 잡는데 너무 시간을 많이 빼앗겨서 했던 말이다. 목록 5 에 포함된 SymGetModuleInfo64 와 같은 경우에는 DLL 의 버전이 5.1 일 때 항상 그저 0 을 반환할 뿐이다. 게다가 오류 처리도 그다지 잘되어 있지 않다. 지금 이 글을 쓰는 시점의 최신 버전인 6.6 같은 경우에는 무슨 짓을 하더라도 SymGetSymbolFile 함수를 정상적으로 작동시킬 수 없었다. 그러니 DbgHelp 라이브러리를 가지고 프로그래밍을 할 때에는 항상 주의하고 최신 DLL 을 사용하라.

C#으로의 전환

지금까지 살펴본 모든 샘플들은 C++로 작성된 것이었다. 하지만 툴은 C#으로 작성되었으므로 샘플 코드를 C#으로 이식하는데 몇 가지 중점사항에 대해 알아보겠다.

우선 .NET framework 이 강력한 프로세스 지원 기능을 가지고 있다는 것이 중요하다. Process 클래스(Class)에는 지역(local)과 원격(remote) 프로세스 모두를 지원하지만 DbgHelp 로 얻어낸 핸들을 이용해서 원격 프로세스의 핸들로 접속할 수 있는지는 의문이다. 이점을 한 번 알아보자.

목록 4 의 EnumerateModules 프로그램은 목록 6 과 같이 C#으로 몇 줄의 코드만으로 제작할 수 있다.

```
Process[] localProcesses = Process.GetProcesses();
foreach (Process process in localProcesses)
{
   m_ProcessesListBox.Items.Add(new ProcessItem(process));
}
```

Listing 6: Enumerating the processes in C#.

목록의 나머지 부분은 상대적으로 쉽게 C#으로 이식할 수 있다. C#에서 DbgHelp.dll 과 같은 다양한 DLL 에 있는 함수를 호출하는 부분만 좀 까다로울 뿐이다. .NET framework 은 종종 *p/invokes* 라고 불리기도 하는 *플랫폼 인보크(Platform invoke)* 시스템을 사용한다. 이 주제에 대해 더 깊이 파고 들어가는 것은 본문의 주제를 벗어나는 일이지만 다음에 나타나는 목록을 보면 대략적인 것은 알 수 있을 것이다. 자신의 C# 코드로 함수를 가져와야(import)한다. 목록 7 은 다양한 샘플에 걸쳐 사용할 함수의 대부분을 가져오는 방법을 나타낸다. 이것을 자신의 C# 코드에 붙여 넣으면 C++ 코드를 C#으로 이식할 때 문제가 별로 발생하지 않을 것이다.

```
public class DbgHelp
{
 [Flags] public enum SymOpt : uint
 {
   CASE_INSENSITIVE       = 0x00000001,
   UNDNAME              = 0x00000002,
   DEFERRED_LOADS         = 0x00000004,
   NO_CPP               = 0x00000008,
   LOAD_LINES           = 0x00000010,
   OMAP_FIND_NEAREST      = 0x00000020,
   LOAD_ANYTHING          = 0x00000040,
   IGNORE_CVREC           = 0x00000080,
   NO_UNQUALIFIED_LOADS    = 0x00000100,
   FAIL_CRITICAL_ERRORS    = 0x00000200,
   EXACT_SYMBOLS          = 0x00000400,
   ALLOW_ABSOLUTE_SYMBOLS   = 0x00000800,
   IGNORE_NT_SYMPATH      = 0x00001000,
   INCLUDE_32BIT_MODULES    = 0x00002000,
   PUBLICS_ONLY          = 0x00004000,
   NO_PUBLICS           = 0x00008000,
   AUTO_PUBLICS          = 0x00010000,
   NO_IMAGE_SEARCH        = 0x00020000,
   SECURE               = 0x00040000,
   SYMOPT_DEBUG          = 0x80000000
 };

 [Flags] public enum SymFlag : uint
 {
   VALUEPRESENT    = 0x00000001,
   REGISTER       = 0x00000008,
   REGREL         = 0x00000010,
   FRAMEREL       = 0x00000020,
   PARAMETER       = 0x00000040,
   LOCAL          = 0x00000080,
   CONSTANT        = 0x00000100,
   EXPORT         = 0x00000200,
   FORWARDER       = 0x00000400,
   FUNCTION       = 0x00000800,
   VIRTUAL        = 0x00001000,
   THUNK          = 0x00002000,
   TLSREL         = 0x00004000,
 }

 [Flags] public enum SymTagEnum : uint
 {
   Null,
   Exe,
   Compiland,
   CompilandDetails,
   CompilandEnv,
   Function,
   Block,
   Data,
   Annotation,
   Label,
   PublicSymbol,
   UDT,
   Enum,
   FunctionType,
   PointerType,
   ArrayType,
```

```
    BaseType,
    Typedef,
    BaseClass,
    Friend,
    FunctionArgType,
    FuncDebugStart,
    FuncDebugEnd,
    UsingNamespace,
    VTableShape,
    VTable,
    Custom,
    Thunk,
    CustomType,
    ManagedType,
    Dimension
  };

  [StructLayout(LayoutKind.Sequential)]
  public struct SYMBOL_INFO
  {
    public uint SizeOfStruct;
    public uint TypeIndex;
    public ulong Reserved1;
    public ulong Reserved2;
    public uint Reserved3;
    public uint Size;
    public ulong ModBase;
    public SymFlag Flags;
    public ulong Value;
    public ulong Address;
    public uint Register;
    public uint Scope;
    public SymTagEnum Tag;
    public int NameLen;
    public int MaxNameLen;

    [ MarshalAs( UnmanagedType.ByValTStr, SizeConst=1024)]
    public string Name;
  };

  [StructLayout(LayoutKind.Sequential)]
  public struct _IMAGEHLP_LINE64
  {
    public uint SizeOfStruct;
    public uint Key;
    public uint LineNumber;
    public IntPtr FileName;
    public ulong Address;
  };

  public delegate bool SymEnumSymbolsProc(ref SYMBOL_INFO pSymInfo, uint
SymbolSize, IntPtr UserContext);

  [DllImport("dbghelp.dll", SetLastError=true)]
  public static extern bool SymInitialize(IntPtr hProcess, string UserSearchPath,
bool fInvadeProcess);

  [DllImport("dbghelp.dll", SetLastError=true)]
  public static extern uint SymSetOptions(SymOpt SymOptions);
```

```
  [DllImport("dbghelp.dll", SetLastError=true)]
  public static extern ulong SymLoadModule64(IntPtr hProcess, IntPtr hFile,
    string ImageName, string ModuleName,
    ulong BaseOfDll, uint SizeOfDll);

  [DllImport("dbghelp.dll", SetLastError=true)]
  public static extern bool SymEnumSymbols(IntPtr hProcess, ulong BaseOfDll,
string Mask, SymEnumSymbolsProc EnumSymbolsCallback, IntPtr UserContext);

  [DllImport("dbghelp.dll", SetLastError=true)]
  public static extern bool SymGetLineFromAddr64(IntPtr hProcess,
    ulong dwAddr, ref uint pdwDisplacement, ref _IMAGEHLP_LINE64 Line);

  [DllImport("dbghelp.dll", SetLastError=true)]
  public static extern bool SymFromAddr(IntPtr hProcess,
    ulong dwAddr, ref ulong pdwDisplacement, ref SYMBOL_INFO symbolInfo);

  [DllImport("dbghelp.dll", SetLastError=true)]
  public static extern bool SymEnumSymbolsForAddr(IntPtr hProcess,
    ulong Address, SymEnumSymbolsProc EnumSymbolsCallback, IntPtr
UserContext);

  [DllImport("dbghelp.dll", SetLastError=true)]
  public static extern bool SymUnloadModule64(IntPtr hProcess, ulong BaseOfDll);

  [DllImport("dbghelp.dll", SetLastError=true)]
  public static extern bool SymCleanup(IntPtr hProcess);
}
```

Listing 7: p/invoke signatures for some DbgHelp functions.

이 주제에 대한 더 자세한 내용은 [8]을 참조하라. 해당 문서에서 특히 유용한 부분은 .NET의 형식을 Windows 기본 형식(types)으로 맵핑(Type mappings)하는 방법을 설명한 페이지[9]이다. 형식 맵핑이란 알고 보면 p/invokes에 새로운 서명(Signatures)를 추가하는 그다지 어렵지 않은 작업이다. 마지막으로 www.pinvoke.net을 보면 p/invoke의 데이터베이스가 구축되어 있으며 심지어 데이터베이스에 접속할 수 있는 Visual Studio의 추가 기능(add-in)을 제공하기도 한다. 하지만 불행하게도 본 아티클을 작성할 때에는 해당 데이터베이스에 DbgHelp.dll의 함수들에 대한 내용이 하나도 없었다.

그 다음에는?

툴을 강력하게 만들 아이디어는 아직도 남아 있다. 예를 들어 단편화 보기 같은 경우에는 현재 시점에서 어딘지 기능이 떨어져

보인다. 현재로써는 특정 시점의 메모리 할당에 대한 물리적 위치 관계만을 볼 수 있을 뿐이다.

블록의 물리적 위치를 저장해놓았다가 애니메이션으로 재생할 수 있다면 좋을 것이다. 다양한 속도로 그것을 재생시켜 볼 수 있다면 프로그램에서 메모리 단편화를 가장 많이 유발하는 부분을 훨씬 더 쉽게 찾을 수 있을 거라고 생각한다. 그리고 단편화 보기에서 중요한 정보들인 가장 큰 메모리 블록과 같은 좀 더 상세한 통계 자료를 표시하는 것도 가능할 것이라고 생각한다.

결론

이제 PC 에서도 저수준의 할당을 가로챌 수 있다는 것을 알게 되었고 Xbox 와 Xbox 360 에서도 어느 정도의 범위 안에서는 저수준 할당 가로채기가 가능하다는 것을 알게 되었다. 하지만 실행 이미지를 불러오는 도중이나 서드 파티(third parties)에서 제작된 프로그램이나 커널에 의해 수행되는 할당과 같은 경우, 다른 몇몇 플랫폼에서는 가로채기가 매우 어렵거나 불가능하기도 하다.

플랫폼 제조사의 협조를 받으면 할당 가로채기가 훨씬 수월해진다는 사실은 제쳐놓더라도 MemAnalyze 와 같은 툴이 모든 표준 SDK 의 일부로 포함된다면 좋을 것이라는 생각도 하게 된다. 자신이 모든 게임을 특수한 디버그 모드로 실행할 수 있고 MemAnalyze 와 같은 툴로 모든 할당을 감시할 수 있다고 생각해보라. 그러면 게임 프로그래머가 추가적인 작업을 해야 할 필요가 없어질 것이다.

이런 시도는 이미 Xbox 의 XbMemDump 에서 이루어졌었다. 다만 조금만 더 신경을 썼더라면 정말 유용했을 법한 툴이었으므로 Microsoft 가 XbMemDump 에 더 노력을 기울이지 않은 것이 애석할 따름이다. 하지만 나는 그들이 올바른 방향으로 가고 있는 것 같다고 생각한다.

향후에는 게임이 사용할 수 있는 메모리의 양이 점점 더 증가될 것이다. 하지만 그렇다고 메모리 관리를 소홀히 해도 된다는 뜻은 아니다. 사실 정확하게 그 반대라고 말하고 싶다. 더 많은 메모리를 사용할 수 있게 되면 더 많은 양의 컨텐츠를 사용하게 될 것이므로 메모리를 효율적으로 관리하는 것이 더욱 중요해질 것이다. 그러므로

플랫폼 제조사들이 콘솔에서 메모리 관련 문제를 해결할 수 있는 더 좋은 툴과 API 의 개발에 힘을 쏟아주기를 요구한다.

더 좋은 대안이 발견되기 전까지 우리는 계속 자신만의 해결책을 만들어 나아가야 한다. 나는 머리 속에 있는 새로운 아이디어를 이용해 MemAnalyze 의 새 버전을 제작할 것 같다. 그리고 Windows Presentation Foundation 을 한 번 살펴봐야겠다.

감사의 말

Playlogic Game Factory, Tom van Dijck, Harm van Dinter, Arjan Janssen 의 수년간의 노고에 감사 드립니다.

참고 자료

[1] Boost pool implementation:

http://www.boost.org/libs/pool/doc/index.html

[2] Windows' Low Fragmentation Heap

http://msdn2.microsoft.com/en-us/library/aa366750.aspx

[3] Detours, a library by Microsoft Research to detour DLL functions:

http://research.microsoft.com/sn/detours/

[4] Matt Conover's solution to DLL hooking:

http://mconover.openrce.org

[5] Jochen Kalmbach's StackWalker:

http://www.codeproject.com/threads/StackWalker.asp

[6] Dan Mouldin's trick to obtain the instruction pointer:

www.thecodeproject.com/tools/visualleakdetector.asp

[7] The latest Debugging kit for Windows:

http://www.microsoft.com/whdc/devtools/debugging/default.mspx

[8] Marshalling data with platform invoke:

http://msdn2.microsoft.com/en-us/library/aa720284(VS.71).aspx

[9] Platform invoke data types:

http://msdn2.microsoft.com/en-us/library/aa720411(vs.71).aspx

[10] Scott Meyer's Effective C++, third edition. Published by Addison-Wesley, ISBN: 0-321-33487-6.

[11] Discussion on MSDN forums about StackWalk64 performance:

http://forums.microsoft.com/MSDN/ShowPost.aspx?PostID=396838&SiteID=1