Gama Network Presents:

Gamasutra.com

Monitoring Your Console's Memory Usage, Part One
(콘솔 메모리 상태 모니터링 제 1 장)

Jelle van der Beek

04 년 4 월 14 일

http://www.gamasutra.com/features/20040414/vanderbeek_01.shtml
http://www.gamasutra.com/features/20040414/vanderbeek_02.shtml
http://www.gamasutra.com/features/20040414/vanderbeek_03.shtml

▶ 콘솔 메모리 상태 모니터링(제 1 장), 젤레 밴 데르 비크 저술[04 년 4 월 14 일] Xbox 와 PS2 게임 개발시, 가장 어려운 작업 중의 하나는 콘솔의 메모리 요구조건을 만족시키는 것이다. 이와 같은 문제를 해결할 상용툴이 현재 출시되어 있지 않으므로, 젤레 밴 데르 비크가 직접 개발했다.

제 1 장

콘솔게임을 개발할 때 가장 어려운 부분은 콘솔메모리 요구조건을 만족하는 것으로, 메모리 부분은 게임 개발 중 계속해서 문제를 발생시킨다. 메모리 요구조건을 만족시키기 위해 많은 노력을 기울려 조정해두어도, 1 주일이 지나서, 게임 내용이 바뀌면 모든 작업을 다시 시작해야 한다. 따라서, 개발자에게 필요한 정보를 신속하게 제공해줄 툴이 매우 요구된다. 나의 요구조건을 만족하는 상용툴을 아직 찾지 못했기 때문에, 내가 직접 스스로 개발하기로 했다. 본 칼럼을 2 회에 걸쳐 연재되며, 이번 회에서는 내가 Xbox 와 PS2 와 같은 이기종 환경하에서 사용할 수 있는 툴 개발에 대해 소개한다.

여기서 소개되는 방법은 캐시 실패(cache misses) 혹은 페이지 실패(page misses)와 같이 메모리 성능을 모니터링 하는 것이 아니며, 그대신, 메모리 사용에 있어 3 가지의 주요 관점에 대해 중점적으로 다루고 있다.

- (사용하는 언어에 따른) 응용프로그램의 메모리 사용량
- 단편화된 메모리 조각 표시용 메모리 구조도
- 프로그램 응용시의 메모리 릭 현상 발견 및 원인 분석

메모리와 성능문제는 내가 작업 시 항상 고민하는 문제이지만, 상기의 세 가지 기본사항을 모니터링 하는 콘솔프로그램 개발용 장치를 이제까지 발견하지 못한 것은 좀 이상하기도 하다. 마이크로소프트가 내놓은 Xbox 개발 툴인 XbMemdum 는 툴 개발을 위한 시작으로서 의미를 가지며, 이는 물리적인 페이지 구조도를 모니터상에 보여준다. 하지만, 이는 ASCII 코드 출력용 커맨드 라인 툴로 아주 기초적 장치이다. 이와 같은 메모리 문제 해결을 위해 Metrowerks 에서 개발한 CodeTEST 같은 툴이 있기도 하다. 하지만, 게임 콘솔에는 이용할 수는 없다[참조 7]. 메모리에 관련된 다른 툴은 Boundschecker[참고 8]이다. 이 툴은 메모리 릭을 찾아내며, 7.1 버전부터는 메모리와 자원을 조회할 수도 있다. 하지만, 이 제품도 콘솔 프로그램 제작에는 사용할 수 없다. 내가 MemAnalyze 이라고 부르는 툴을 어떻게 개발했는지 보여주겠다. 이 툴에 의해 Xbox 와 PS2 에 있어 상기에 언급한 세 가지 메모리 문제에 대해 모니터링 할 수 있다. (플레이로직(www.playlogicinternational.com)은 게임큐브를 지원하지 않기 때문에, 게임큐브에 대해서는 언급하지 않는다.) 본 사설을 읽고 나면, 여러분들은 다른 플랫폼에 대해서도 모니터링툴을 어떻게 사용할지에 대한 아이디어를 얻을 수 있을 것이다. 본 사설은

모니터링툴에 대한 개요를 제공하며, 게임 메모리에 대한 스냅샷(snapshot)을 어떻게 하면 만들 수 있는지 보여준다. 제 2 장에는(이번 금요일에 Gamasutra 에 발간 예정), 데이터를 어떻게 해석하는지에 대해 설명하겠다.

개요

이론

게임 시작 후 버튼을 누르게 되면, 현재 메모리 상태, 즉 메모리에 현재 할당된 모든 블럭의 주소와 크기가 저장된 파일이 생성된다. 콘솔에 하드디스크가 있는 경우, 파일은 거기에 저장될 것이다. 그렇지 않고, PC 에 연결해서 게임을 하는 경우는 PC 의 하드 디스크에 저장할 수 있다. 본 사설은 파일을 생성하는 방법에 대해서는 별도로 언급하지 않는다. 단지 파일에 저장할 올바른 내용을 어떻게 모으는지에 대해 설명한다. 어디에 저장할 지는 독자들께서 추가로 할 작업이다.

본 칼럼에서는 할당되는 블럭 정보 외에 실시간 콜스택 추적(callstack tracing)기능을 사용해서, 각각에 할당된 블록에 대한 콜스택(callstack)을 제공할 것이다. 각 플랫폼 별로 실시간 콜스택 추적(callstack tracing) 기능이 필요한데, 그 이유는 각각의 함수는 이전 단계로 항상 복귀해야 하며, 따라서 돌아갈 주소는 어딘가에 어떻게 해서든지 저장되어 있어야 한다. 우리는 이제 각 플랫폼에서 이 데이터를 어떻게 되찾을지에 대해서 고민해야 한다.

만약 자체 할당(allocation) 혹은 힙매니저(heap manager)를 직접 프로그래밍해보았다만, 메모리 덤프를 위해 정확한 정보를 수집하는 작업은 좀더 쉬운 일일 것이다. 아마도 이미 대부분의 정보를 가지고 있을지도 모른다. 툴을 사용하기 위해서는 다음의 데이터가 필요하다.

- 각각 할당된 블럭의 주소
- 할당된 크기
- 함수주소의 행렬에 해당하는 각 할당에 대한 콜스택.(정확한 표현은 아니지만, 다음에 다시 설명하겠다.)

본 툴은 PC 의 Offline 상에서 메모리 덤프된 내용 및 맵 파일(map file)이나 프로그램 데이터베이스로부터 심볼 정보를 읽는다. 심볼 정보는 함수 주소를 함수 이름으로 바꿀 때 사용된다. 이 툴에 의해 이 데이터를 여러 시점에서 실행시켜 보게 된다.

플랫폼의 독립성

플랫폼 의존성과 플랫폼 독립성의 차이는 무엇인가? 이러한 차이 여부는 개발자에게 달려있다. 플랫폼에 특정한 정보는 다음을 포함한다.

- 힙 정보: 하드웨어 콘솔에 저장됨
- 심볼 정보: 맵 파일 혹은 프로그램 데이터베이스파일의 형태임
- 이미지 위치 정보(필요한 경우)

플랫폼에 독립적인 데이터 구조상에서 게임에 통해 힙(heap)에 접근하거나, 이를 처리하고 파일에다가 덤프를 하게 할 수 있다. 게임 자체에서 심볼을 분석하고 함수이름으로 메모리 덤프시 주소를 금방 바꿀 수 있게 만들 수 있다. 이러한 방법으로 플랫폼에 무관한 파일을 만들 수 있으며, MemAnalyze 을 플래트폼과는 완전히 독립적으로 활용하게 된다. 이와 같은 방식은 장점도 있으나, 몇 가지 단점이 있다.

- 각각의 덤프된 콜스택함수에 함수이름을 추가하는 것은 함수이름이 여러 번 복사되므로, 큰 부하를 발생시킨다.
- 함수 주소가 이름으로 바뀌는 경우, 데이터를 처리(비교, 무효화) 하기 위해 CRC32 와 같은 특정한 값으로 이름을 바꾸어야 한다.
- 심볼 정보 분석시 콘솔 라이브러리에 제한을 받고 있다. 어떤 플랫폼의 경우, 이는 문제가 될 수도 있다. 프로그램 데이터베이스에서 심볼을 분석하고자 한다면, 자체적으로 PDB 분석기를 만들 수도 있으나, 이는 매우 복잡하며 플랫폼 버전이 바뀌는 경우, 수정이 쉽지 않다.
- 심볼 정보를 로딩해야 한다. 이 데이터는 메모리 분석시 표시될 것이다. 심볼 정보를 각 메모리 덤프에 다시 로딩함으로써 이 문제를 부분적으로 해결할 수 있다.

그대신, 함수 이름을 제외하고는 게임시 플랫폼에 종속적인 파일을 출력하도록 선택했다. PS2 의 경우, 전체 힙부분을 파일에 덤프 시킨 후, 완전히 오프라인 상태에서 힙을 검색함으로써, 힙상태를 확인한다. 이와 같은 작업으로, 핵심적인 문제가 발생한 경우 파일에다가 PS2 메모리를 덤프 할 수 있으며, 이에 따라 MemAnalyze 에서 나중에 디버깅 작업을 할 수 있다. 또한 MemAnalyze 를 이용하여 메모리의 내용을 출력하고 비교할 수 있다. 분명, 이것은 플랫폼에 종속적인 코드 일부를 모니터링 툴로 옮긴다.

이 툴은 플랫폼에 종속적인 두 경우의 코드를 포함한다.

- 플랫폼에 특정한 메모리 덤프 내용을 읽고 이것을 플랫폼에 독립적인 내부 데이터 구조로 전환한다.
- 플랫폼에 특정한 심볼을 읽고 이것을 플랫폼에 독립적인 내부 데이터 구조로 전환한다.

이러한 관점에서 모든 것은 다종 플랫폼에 적용 가능해야 한다.

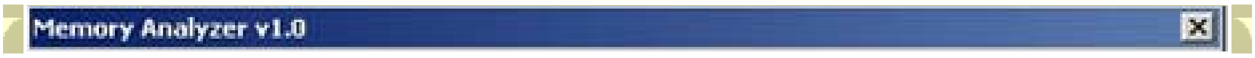
MemAnalyze

마지막에는 데이터를 세 가지의 다른 관점에서 보게 된다. 그래픽으로 볼 때 난 단지 2 개만 실행했다. 하나는 메모리 레이아웃을 표현하는 것이고, 또 다른 하나는 각 함수가 얼마나 많은 메모리를 할당 했는지 보여주는 것이다. MemAnalyze 를 이용하면, 다중윈도우(multiple windows)에서 다중메모리(multiple memory) 덤프 내용을 볼 수 있다. 세 번째 관점은 다중 메모리 덤프를 비교함으로써 메모리 릭(leaks)를 나열하는 다이얼로그 화면이다.

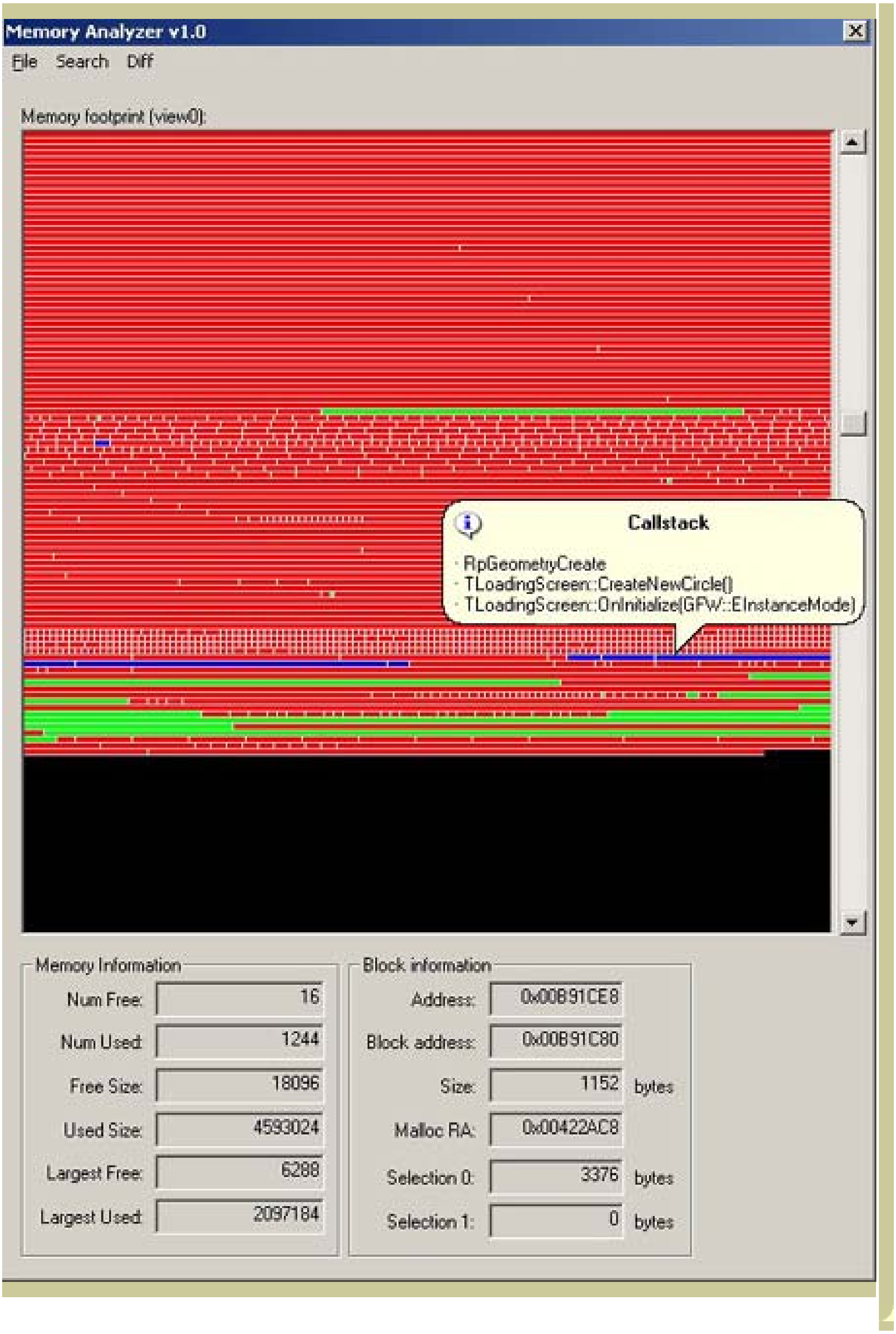
아직까지 계층구조 측면에서는 아직 실행하지 못했다. 계층구조도는 메모리를 할당하는 함수의 계층구조를 표현할 것이다. 이러한 시점으로 보면, 메모리 사용에 대한 개략적인 내용에 대해 더 잘 확인할 수 있으며, 할당되는 주요 위치에 대해 조명할 수 있게 된다. 여기에 대한 자세한 정보는 2 장에서 설명하겠다.

메모리 레이아웃도

아래 그림은 콘솔에 물리적으로 나타나는 메모리 블럭을 보여준다. 이것은 *Microsoft Defrag* 방식의 표현방법이다. 블록 위에 마우스 커서를 움직이면 할당한 함수의 전체 콜스택을 보여주는 풍선도움말이 나타나게 된다. PS2 에서는 메모리 단편화가 매우 중요한 이슈이므로, 이 방식이 매우 편리하다. 아마도 개발자를 골치 아프게 하는 흩어진 작은 블록들을 확인할 수 있게 것이다.



Memory Analyzer v1.0



Callstack

- RpGeometryCreate
- TLoadingScreen::CreateNewCircle()
- TLoadingScreen::OnInitialize(GFW::EInstanceMode)

Memory Information

Num Free: 16

Num Used: 1244

Free Size: 18096

Used Size: 4593024

Largest Free: 6288

Largest Used: 2097184

Block information

Address: 0x00B91CE8

Block address: 0x00B91C80

Size: 1152 bytes

Malloc RA: 0x00422AC8

Selection 0: 3376 bytes

Selection 1: 0 bytes

그림 1. 레이아웃도. 첫 번째 그림은 RpGeometry 에 의해 단편화문제가 야기하는 상황을 표현한다. 아래 그림을 통해 우리는 이 문제를 해결했다. 작은 블록들의 흐트러짐이 훨씬 덜함을 알 수 있다.

불행하게도, 본 레이아웃도는 Xbox 게임에는 별로 쓸모가 없다. Xbox 에서는 가상메모리 주소를 사용하며, VMM 상에서 힙 단편화 문제의 대부분을 해결한다. VMM 은 대형 가상 할당부분을 각각 4KB 의 물리적 페이지로 분리할 수 있다. 우리 프로그램에서 사용하는 주소는 가상 주소이며, 여러 개의 물리적 페이지 상에 연결될 수 있고, 가상 주소를 모니터링 할 수 도 있다. 하지만, 가상주소공간에 있어서의 단편화는 실제로는 문제가 아니면, 이는 4GB 주소 공간에 64MB 을 매핑(mapping) 수 있기 때문이다.

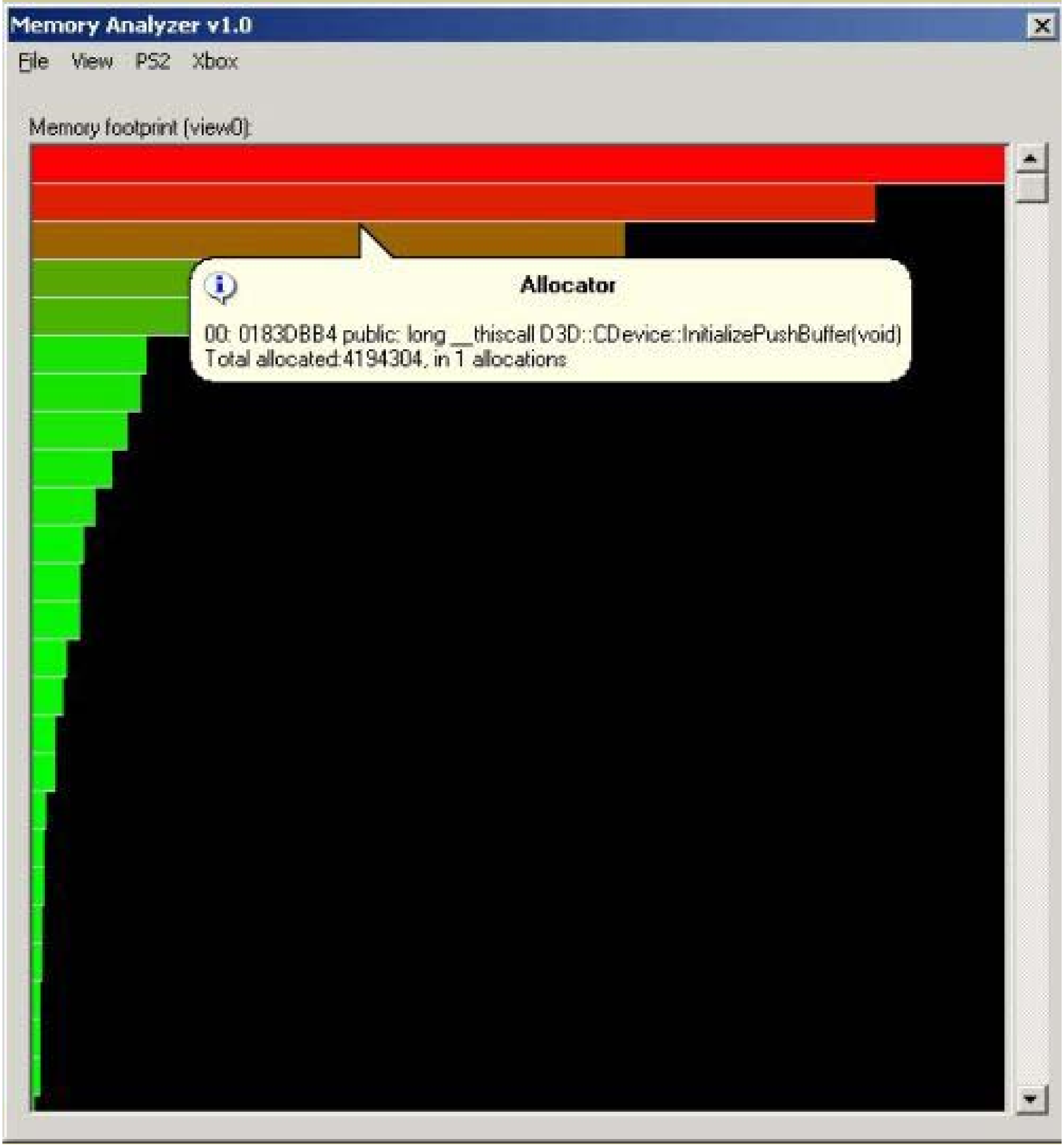
각각의 가상적 힙 할당에 대해 실제 물리적 페이지를 추적하는 것이 가능한지 모르겠다. 그럴 경우, 아마도 가상적으로 할당된 부분의 물리적 매핑상태를 실제로 확인할 수도 있다. 하지만, 이것이 별로 유익한 정보인 것 같지는 않다.

반면에 물리적인 할당상태는 툴 상에서 모니터링 하기에 유용한 정보라고 본다.

TopX view

아래 그림은 메모리를 할당한 각 함수 별로 여러 개의 바(bar)를 보여준다. 즉, 만약 바 위로 마우스를 움직이면, 풍선도움말이 할당된 정확한 크기 및 그 갯수와 함께 함수의 이름을 표시한다. 우리는 함수를 몇 가지의 흥미로운 방식으로 분류할 수 있다.

- 각각에 할당된 전체 크기
- 각각에 할당된 갯수
- 함수 이름

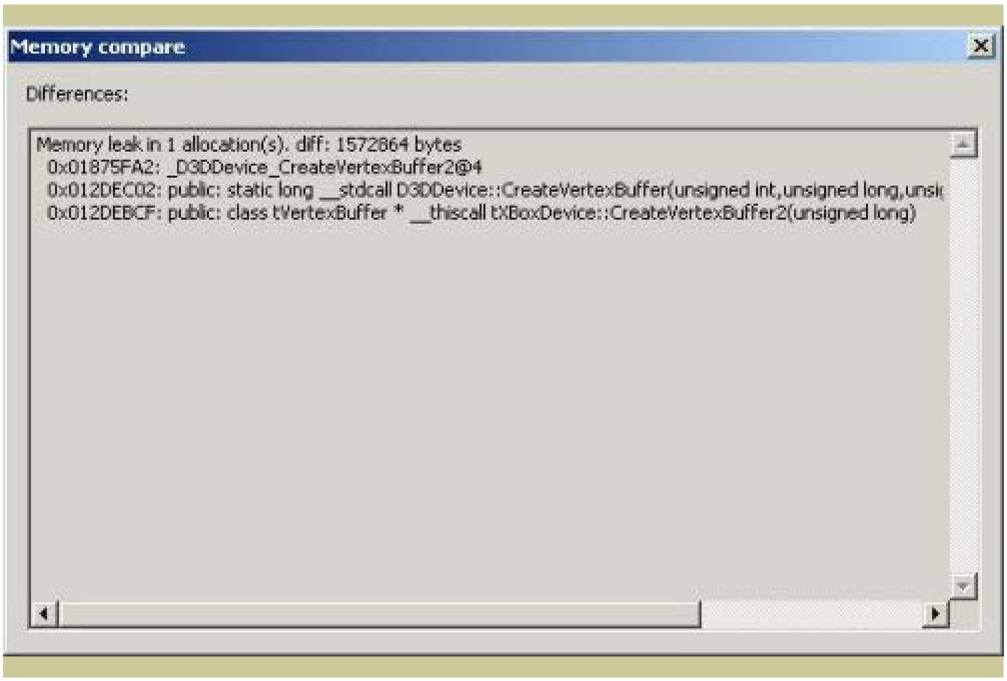


00: 0183DBB4 public: long __thiscall D3D::CDevice::InitializePushBuffer(void)
Total allocated:4194304, in 1 allocations

Figure 2: The TopX view, sorted on total size allocated.

메모리 릭 화면

그림 3 에서와 같이 메모리 릭 화면은 두 개의 덤프를 비교하고, 그 차이를 다일로그 박스에 출력한다.



```
Memory leak in 1 allocation(s). diff: 1572864 bytes
  0x01875FA2: _D3DDevice_CreateVertexBuffer2@4
  0x012DEC02: public: static long __stdcall D3DDevice::CreateVertexBuffer(unsigned int,unsigned long,unsi
  0x012DEBCF: public: class tVertexBuffer * __thiscall tXBoxDevice::CreateVertexBuffer2(unsigned long)
```

그림 3. 메모리 릭을 통한 메모리 비교 결과

Making a memory dump

Xbox

Xbox 프로그래밍을 즐겁게 해본 사람들은 마이크로소프트가 게임 프로그래밍에 관련해서 거의 모든 것을 생각했다는 것을 알 수 있었을 것이다. 운이 좋게도, XbMemDump 라는 툴에 의해 콜스택을 포함한 모든 할당된 메모리 블럭을 덤프할 수 있다. 만약 XbMemDump가 사용하기 불편하다면, 콜스택 정보를 저장하여 힙에서 수동으로 작업할 수 있는 디버깅 함수 시리즈를 이용하면 된다.

마지막으로, Xbox는 단일화된 메모리 구조를 가지고 있다. 따라서 음향과 비디오에 사용된 메모리를 포함해 모든 메모리를 모니터링 할 수 있다.

XbMemdump를 사용해서 자동적으로 메모리 덤핑 하기

얼핏 보면, XbMemdump에 우리가 필요한 모든 것이 있어 보인다. 사실, 많은 장점이 있다. 이것은 커널 단계에서 메모리 트래킹을 할 수 있도록 지원하여, 어떤 할당여부도 빠뜨리지 않도록 한다. 각 할당별로 32등급까지 콜스택정보를 표시할 수 있다. 따라서, 그런 할당된 정보를 추적하려고 노력할 필요가 없다.

하지만, 반년 전 MemAnalyze를 사용하여, 할당여부에 대한 트래킹이 활성화되었을 때 XbMemdump는 엄청나게 느리게 작동했다. 레벨을 로딩할 때 계속해서 시스템이 다운되었으며, 그렇지 않는 경우라도 끝마치는데 1시간 반이 걸렸다. 결국에 메모리를 모두 덤프하게 되었을 때, 그냥 반환주소만 표시되었고, 작업할 심볼정보는 얻을 수 없었다.

이제 반년이 지났고, XbMemDump 를 다시 시험해보았다. 성능상 문제는 없으며, 심볼도 꽤 잘 로딩되었다. 내가 이에 대해 질문했을 때, 마이크로소프트는 2002년 12월 이후로 XbMemdump에 어떤 수정도 하지 않았다고 한다. 하지만, 프로그램을 돌려보면 여러분들도 확인할 수 있다. 물론 내가 그 이후 다른 할당방식을 적용했기 때문에 제대로 동작하는 것인지도 모른다. 만약 여러분의 프로그램상에서 XbMemdump 가 제대로 작동하지 않거나, 혹은 내가 이 문제에 대해 어떻게 해결했는지 알고 있다면, 다음의 Xbox 메모리를 수동으로 덤프하는 방식을 보라.

메모리를 수동으로 덤프하기

할당된 영역 가로채기

먼저 모든 할당된 영역을 가로채야(intercept) 한다. 이건 꽤 힘든 작업이다. Xbox에는 두 가지의 할당방식이 있다. 첫 번째는 PhysicalAllocs로서, 이는 비디오 버퍼나 음향데이터 같이 연속적인 메모리를 할당할 때 사용하며, 다른 것은 HeapAllocs 이다.

Xbox는 XMemAlloc라는 글로벌함수를 제공하는데, 이 함수는 과부하(overloaded) 될 수도 있다. XMemAlloc는 (거의) 모든

할당종류를 지원한다. 모든 제품은 할당을 위해서 XMemAlloc을 사용해야 하며, 따라서, 게임 개발자는 할당을 가로챌 수가 있다. 만약 툴 개발자가 32바이트 정렬이나 그 이상의 것 등 XMemAlloc 가 지원하지 않는 기능이 필요하다면, 콜백함수(callback function)가 활용될 가능성을 위해 할당함수를 위한 래퍼(wrapper)가 개발되어야 한다. 이러한 방식에 의해 응용프로그램은 모든 할당내용을 처리할 수 있다.

불행하게도, 모든 제품이 이러한 법칙에 맞는 것은 아니다. 마이크로소프트 조차 이 법칙을 무시했다. 2003년도 12월 SDK 까지 XACT 및 XMV 모듈은 XMemAlloc를 사용하지 않았다.(물론, 이제는 사용한다.)

일단 모든 할당 혹은 필요한 모든 할당을 해석하고 나면, 콜스택 정보를 저장할 수 있다.

실시간 콜스택 추적

마이크로소프트는 "Dm"이란 접두어의 디버깅 함수를 연속적으로 내놓았다. 이 함수군을 사용하기 위해서는 디버그 라이브러리인 XbDm.lib을 연결해야 한다. DmCaptureStackBackTrace 함수는 콜스택정보를 저장하기 위해 사용한다. (만약 인텔기반 컴퓨터에서 콜스택 추적에 대해 좀더 알고자 한다면, Chavdar Dimitrov의 저서[참고문헌 2] 읽어보기를 권한다). 리스팅1에는 만약 컴파일러상태에서 프레임 포인터의 생략기능이 비활성화한 경우, 어떤 IA-32 기반구조(그리고 상위기반)에서도 작동하는 콜스택 추적함수를 보여준다.

```
unsigned int StoreCallStackCPP(
                        unsigned int* pArray,
                        unsigned int nCount
                        )
{
    struct CStackFrame
    {
        CStackFrame*    pPrevFrame;
        unsigned int    nReturnAddress;
    };
    CStackFrame*    pStackFrame;
    unsigned int    nResult = 0;

    if(pArray != NULL)
    {
        _asm mov [pStackFrame], ebp
        // Point to the previous frame: the frame of the caller
        pStackFrame = pStackFrame->pPrevFrame;

        for(unsigned int i=0; i <nCount; + +i)
        {
                        pArray[i] = pStackFrame>-nReturnAddress;
            // If return address is zero, we have reached the
            // end of the callstack
            if(pArray[i] == 0)
            {
                        break;
            }
            pStackFrame = pStackFrame->pPrevFrame;
        }
        // Store the number of succesful items
        nResult = i;
    }

    return nResult;
}
```

```
unsigned int __declspec(naked) StoreCallStackAsm(
                                        unsigned int* pArray,
                                        unsigned int nCount
                                        )
{
    __asm
    {
        // Note: this function has no prolog/epilog code

        mov ebx, ebp                    // use ebp directly =                                   //
framepointer of                                          // previous function

        mov ecx, dword ptr [esp +8] // Load nCount
        mov eax, ecx

        xor edi, edi                    // Fill edi with zero for                               //
NULL pointer comparison
        mov esi, dword ptr [esp +4] // Load pArray
        cmp    esi,    edi                                     //    Check    for    pArray    NULL
// pointer
        jz done

store_items:
        cmp    ebx,    edi                                        //    Check    for    framepointer
// NULL pointer
        jz done

        mov edi, dword ptr [ebx +4] // Offset +4 from
                                         // framepointer                                    // = return
address
        mov dword ptr [esi], edi      // Store RA
        mov ebx, dword ptr [ebx]      // Load the previous
                                         // framepointer
        add esi, 4                      // Inc the array
        loop store_items
done:
        sub eax, ecx                    // Store the number of                              //
successful items

        ret
    }
}
```

리스팅 1. 인텔 기반 콜스택 추척.

본 함수를 선행하는 함수의 시작주소에 대한 정보를 얻지 못했다는 것을 명심해야 한다. 그대신 반환주소를 얻게 되었다. 이 주소는 caller 함수의 시작과 끝 주소 사이에 존재한다.

StoreCallStackAsm 와 StoreCallStackCPP 함수는 배열상에 놓인 성공적인 아이템의 갯수를 반환한다. 리스팅2는 StoreCallStack이 어떻게 사용되는지 보여준다.

```
const unsigned int STACK_DEPTH = 3;
const unsigned int EXTRA_ALLOC_TAG = 0xCAFEBABE;

class CExtraAllocHeader
{
public:
    unsigned int        tag;
    unsigned int        RA[STACK_DEPTH];

    CExtraAllocHeader()
    {
        tag = EXTRA_ALLOC_TAG;
        memset(RA, 0, sizeof(RA));
    }
};

void Foo3()
{
    CExtraAllocHeader header;
    int                 nrItemsCPP;
    int                 nrItemsAsm;

    nrItemsCPP =     StoreCallStackCPP(header.RA, sizeof(header.RA) /sizeof(int));
    nrItemsAsm =     StoreCallStackAsm(header.RA, sizeof(header.RA) /sizeof(int));
}

void Foo2()
{
        Foo3();
}

void Foo1()
{
        Foo2();
}
```

```
int _tmain(int argc, _TCHAR* argv[])
{
        Foo3();

        return 0;
}
```

리스팅 2. Callstack 추적의 예.

본 예에서는 StoreCallStack 이 Foo2, Foo1 and _tmain함수 범위에서 명령어를 저장할 것이다. StoreCallStack: Foo3와 StoreCallStack 의 caller는 콜스택에 저장되어있지 않다.

데이터 저장

콜스택을 어딘가에 저장해야 한다. 힙할당을 위해서 본인은 16바이트에 의해 할당된 블럭을 크게 만들기로 결심했으며, 할당된 블럭의 뒷 부분에 정보를 추가하기로 했다. 또한 16바이트로 된 4바이트 태그를 제공한다. 태그 값으로는 0xCAFEBABE 와 같은 16진수 숫자를 선택하라. 태그 값은 나중에 힙을 검색할 때 사용될 것이다. heap walker는 processing 하고 있는 할당된 블럭이 콜스택 정보를 가지고 있는지를 확인해야 한다. 왜냐면, 우리가 추적하지 못한 할당이 항상 있기 때문이다. 첫 번째 레벨에 대한 테스트를 할 때, 거의 모든 할당에 대해 겨우 추적했다는 것을 알게 된다.

Heap summary: Total count=76162, of which: Tagged: 75756, Untagged: 406!

Heap summary: Total size=28244816 bytes, of which: Tagged: 26859088, Untagged: 1385728!

Xbox 메모리 메신저는 각 힙 할당을 16바이트 주소(16바이트 정렬)로 반올림했으며, 이 경우 크기는 항상 16바이트의 배수가 된다. 블럭에다가 데이터를 붙이려면, 이렇게 계산하면 된다. 할당된 아이템의 크기를 16바이트 배수로 반올림을 하고, 그리고 나서 데이터에 16 혹은 16의 배수를 더하라. 16바이트를 사용하면, 콜스택이 셋 정도의 함수를 저장할 수 있다. 그림 4는 Xbox의

24바이트가 할당된 레이아웃 예이다.

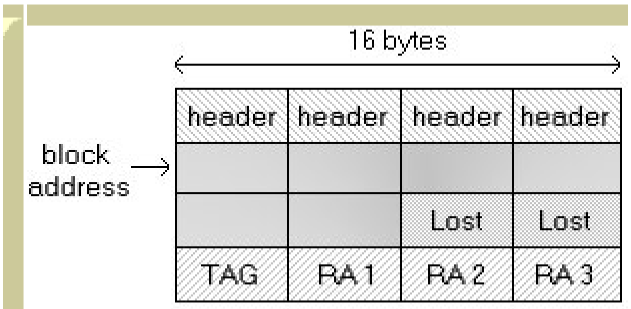


그림 4. 32 바이트로 반올림된 24 바이트의 할당상태. 릴리스 모드의 경우, 16 바이트 힙 헤더는 메모리 블록 상단에 위치한다. 추가적인 데이터는 블럭에 포함되며, 8 바이트는 손실된다. 주석: 많은 양을 할당하는 경우, Xbox 는 다른 형태의 리스트를 사용한다. 따라서, 이 경우 힙 헤더는 약간 크게 된다.

보는 바와 같이 8바이트가 손실되고 있다. 여기에 대해 우리가 손쓸 방법은 별로 없다. 힙을 검색하는 중에는, 할당 후에 블럭에 대해 요청되었던 원래의 크기를 회복할 수 있는 방법은 없다. 마지막 방법으로, 나타나있는 콜스택 레벨을 표시하는 블럭의 마지막에 한 바이트를 첨부할 수 있다. 이런 방식으로 3에서 6등급의 다양한 콜스택 레벨을 이용할 수 있으며, 이에 따라 사용되지 않는 바이트를 채울 수 있다.(물론 태그가 3바이트로 줄어들게 된다.)

비록 내가 이상에서 묘사한 방법을 사용했지만, 여기에는 몇 가지 단점이 있다.

- 각 할당 블록당 16바이트 오버헤드는 메모리 덤프 내에 찌꺼기를 남긴다.
- 각 블럭당 오버헤드를 더 추가하지 않으면 콜스택이 꽤 제한적이 된다.
- 메모리 블럭이 태그가 연결된 블록으로 인식되는 경우가 가끔 있다. 이는 사용될 태그가 유일하다는 것을 보증할 방법이 없기 때문이다. 이 현상은 그렇게 심각하지는 않다.

> 시스템이 망가지지는 않고, 단지 틀리거나 혹은 생소한 콜스택 함수가 포함된 몇 개의 블록을 단지 표시할 뿐이다.

긍정적인 면으로 보면, 이러한 불리한 점은 나에게는 결코 문제가 되지 않았다. 이 시스템은 실행하기 쉬우며, 더욱 중요한 것은 블럭이 할당되거나 프리(free) 되더라도 결과에 어떤 영향도 주지 않기 때문이다.

하지만, 여전히 나는 다른 접근방법을 제시한다. Xbox가 다중힙영역을 지원하기 때문에, 추가의 할당데이터를 저장하는 별도의 리스트를 만들어서 다른 힙에 저장할 수 있다. 이 방식의 장점은 메모리 스냅샷이 정상적인 메모리 상태를 정확히 표현한다는 것이다. 또한 XbMemDump 처럼, 좀더 큰 콜스택 레벨을 추적하기 용의할 것이며, 이로 인해 힙 사용을 훨씬 쉽게 한다. 이 모든 것은 단지 이 리스트를 다시 실행하면 된다. 이 경우 단점은 각각의 자유로워진 메모리 블럭이 추가의 데이터를 지우기 위해 이 리스트를 검색해야 한다는 것이다. 이로 인한 성능 저하를 막기 위해 해쉬(hash) 테이블이나 다른 최적화 알고리즘을 써야 한다.

물리적인 할당을 위해, 주소, 크기 그리고 반환 주소와 함께 별도의 리스트를 유지하는 것 이외에는 방법이 없다. 이는 Xbox에 hysicalAllocWalker와 같은 기능의 함수가 없기 때문이다. 일반적으로 HeapAllocs보다 PhysicalAllocs의 크기가 훨씬 적게 된다. 따라서 deallocation에 대한 리스트를 검색하는 데에 대한 성능 저하 정도는 그리 크지 않다. 첫 번째 단계에서의 테스트 실행에서, PhysicalAllocs 의 갯수는 :

**** 추적된 물리적 할당 수: 39, total size: 12601656 ****

할당된 모든 것을 덤핑하기

이제 메모리 스냅샷을 만들 수 있다. 만약 힙데이터를 각각의 힙에 설정한다면, 그 리스트를 한번 실행하면 된다. 만약 그렇지 않다면, 힙을 찾아야 하며, 각각의 아이템이 코드에 의해 추적되었는지 태그를 통해 확인해야 한다. 이러한 코드는 블록의 마지막에 저장한 추가할당 데이터를 출력한다. PhysicalAllocs의 경우, PhysicalAllocs의 리스트에

대해 실행해보면 된다.

우리는 마이크로소프트의 디버그 함수인 HeapWalk을 사용하여 힙을 성공적으로 검색할 수 있다. 하지만, 이것이 디버그 라이브러리에서만 이용 가능하다는 것은 단점이다. 불가능한 것은 아니지만, XapiLibD.lib과 함께 연결되어 있는 동안에는 해제설정(release build)이 어렵다. 이것을 시도할 때면, 언제나 완전히 디버그 설정상태에서 끝낸다. HeapWalk이 디버그 라이브러리에 포함되어 있는 이유는 마이크로소프트가 낮은 수준의 heapwalk 기능으로 게임 작품의 수준도 떨어지기를 바라지 않기 때문이며, 이는 적절한 판단이다. 아마도, HeapWalk 함수를 XbDm 라이브러리에 포함시켰어야 할지 모른다. XbDm 라이브러리는 해제설정 시에 쉽게 링크될 수 있으나, 이 방식이 승인되지는 않았다.

디버그 설정시 하나의 중요한 단점은 데이터 구조가 아주 다르게 보일 것이라는 것이다. 디버그 모드에서는 메모리 매니저(memory manager)가 약간 다르게 작동한다. 예를 들면, 각 할당블럭에 대한 힙헤더가 더 크며, 메모리 초과실행(overruns)을 점검하기 위해 0xFF 태그를 더하게 된다. 마지막으로 언급하지만 결코 무시하지 못할 것은, 대부분의 경우 디버그 모드에서는 끔찍하게 느리게 움직인다.

슬프게도, 직접 heapwalker를 개발하지 않는 한, 해제 설정상에서 힙을 쉽게 검색하는 방법은 없다. 여러 가지로 많이 노력해보았지만, 별로 여러분께 알려드릴 만한 방법은 찾지 못했다.

Xbox kernel 은 너무 어려운 방법이며, 마이크로소프트에서 만든 모드를 사용하지 않는 것은 나쁜 습관이다. 하지만, PS2의 경우, 내 동료인 Tom van Dijck 가 heapwalker을 만들었다. 그의 PS2 heapwalker 에 대한 자세한 설명은 아래에 있다.

마지막에는 이미지 기반 주소(image base address)를 출력해야 한다. 우리는 DmWalkLoadedModules를 호출함으로써 이미지 기반 주소를 회수(retrieve)할 수 있다. 이 함수는 커널 및 디버깅 모듈을 포함하여 현재의 모든 로딩된 모듈을 반환할 것이다. 이 함수이름과 함께 모든 기반 주소(base address)를 출력해야 한다. 이미지 기반 주소(image base address)의 상세한 설명은 본 시리즈의 제 2장에서 설명할 예정이다.

앞에서 언급한 바와 같이, 메모리덤프에서의 함수이름을 출력하지 않으려고 개인적으로 결심했다. 만약 당신이 그렇게 하기를 원한다면, 심볼정보를 분석하고 함수이름에 따라 주소를 변환하기 위한 기능을 제공하는 “Dm”으로 시작하는 함수를 이용하라. Xbox 메모리 함수에 대한 더욱 상세한 정보는 Xbox central 에 대한 Forrest Trepte의 Xstream training session 을 확인하라[참고문헌 9].

PS2

플레이스테이션 2는 Xbox와 같은 멋진 디버깅 툴도, 동일화된 메모리 구조가 없다. 좋은 면은 힙시스템이 너무 간단해서 heapwalker를 쉽게 만들 수 있다는 것이다.

메모리 덤핑

모든 할당에 대한 가로채기

PS2에 있어서, Xbox에서처럼 가로채어질 수 있는 전역(global) 할당은 없다. 물론 새로운 오퍼레이터는 과다 로딩될 수 있지만, 어떤 다른 할당 함수를 가로챌 방법은 없다. 이것 때문에 단 하나의 옵션만 남으며, 이는 모든 할당을 랩(wrap)시키는 것이다. 게임을 위해 Renderware 그래픽을 사용했기 때문에, 이 할당 루틴을 단순히 사용했다. Renderware의 할당 루틴은 전환(redirect)될 수 있으며, 따라서, 이것들은 전용 할당(custom allocation) 루틴으로 전환했다. 모든 할당은 Renderware 및 우리의 전용 할당 함수를 통해서 이루어졌다.

새로운 오퍼레이터를 오버로딩하고 지우는 것은 훨씬 쉽다. 따라서, 우리 게임이 이 함수를 막 사용한 뒤에는, 몇 가지 예외를 제외하고는 대부분의 메모리가 가로채어질 것이다.

소니 런타임 라이브러리는 힙으로부터도 메모리를 할당한다. 좀더 명확하게는 설명하면, printf 와 atof 는 우리를 괴롭히는 두 개의 함수이다. 이 함수들은 필요할 때마다 단편화를 시키며 작은 메모리

블럭을 할당한다. 이것을 캡쳐 할 수 없는데, 이는 이 함수들이 malloc_r 을 바로 사용하기 때문이다. Malloc_r 은 런타임 라이브러리에서부터 나온 내부 할당 루틴이다. 마지막으로 응용프로그램을 시작할 때, printf 와 atof는 필요한 모든 메모리를 분명히 할당하기 위해 여러 차례 사용된다는 것을 확실히 해야 한다. 다음 코드에 이와 같은 방식을 사용하여, 게임 중에서 메모리 단편화가 발생되지 않는다.

```
float dummy = 0.0f;

dummy = atof("0.2123412341234");

dummy = atof("0");

dummy = atof("1e+6");

printf("%0.3f \ n", dummy);
```

리스팅 5. atof 와 printf 의 단편화 방지 방법.

이제 이러한 이슈가 해결되었으며, 게임에서 사용한 모든 다른 할당을 가로챌 수 있었다. 또한 우리는 16바이트의 데이터를 추가하여 콜스택정보에 저장할 수 있었다.

먼저, 이것을 각 블럭의 시작부분에 저장하고, 16바이트 주소를 추가로 반환하려고 했다. 하지만, Renderware DMA handler는 생각대로 제대로 작동하지 않았으며, 따라서 데이터를 약간의 오류가 발생된 메모리 블럭 마지막에 설정하면서 마무리 했다. :

“malloc(8)”을 실행할 때, 최소한 8바이트의 메모리 블럭을 획득했다. 하지만, malloc_usable_size () 을 사용해서 블럭크기를 회수할 때는 12 만큼의 사이즈로 데이터를 회수 하며, 이는 블록의 크기가 실제로 12바이트 임을 의미한다.

따라서, 우리가 16바이트를 덤으로 할당할 때는 우리의 정보를 "address+8;"에 저장하면 안되며, “address + malloc_usable_size(addres) – 16;”에 저장해야 한다. 그렇지 않는 경우, free, realloc, and heapwalker와 같은 함수 내에서 나중에 찾을

수 없기 때문이다.

다시, 데이터를 0xCAFEBABE와 같은 태그로 표시한다. 앞에서 언급한 바와 같이, 소니 라이브러리가 일부 메모리도 할당하기 때문에 필요하다. 이 태그를 사용함으로써, 메모리 블럭이 콜스택 정보를 가지고 있는지 아닌지를 확인할 수 있다.

실시간 콜스택 추적

MIPS machine에서의 콜스택 추적은 인텔기반의 기계보다 훨씬 더 복잡하다. 좀더 자세한 정보를 위해서는 “See MIPS run” [참고문헌1]을 읽어보라.

MIT X Consortium의 Keith Packard는 MIPS processors 사용을 위해 콜스택 추적 알고리즘을 제안했다. 이것은 Sony Developer Newsgroups 에서 찾아볼 수 있다[참고문헌 6]. 이것은 EE에 특정한 명령어를 일부 수정하는 것을 포함하며, 또한 활용하기 쉽게 작동한다.

heapwalker 사용

PS2 메모리 힙 부분의 레이아웃은 매우 분석하기 쉽다. 이것은 단지 연속적인 커다란 메모리 블럭이다. 먼저, 힙이 어디에서 시작하는지 찾아야 한다. 프로젝트를 시작하기 위해 CodeWarrior를 사용한다. 그리고 후속 코드는 GCC 혹은 ProDG와 같은 다른 컴파일러에 따라 다를 것이다. CodeWarrior의 경우, Linker Configuration Files (LCF)라고 불리는 특징이 있는데, 이것은 다른 것보다도 힙사이즈를 명확히 정의하는데 사용될 수 있다.

CodeWarrior linker는 코드에서 접근할 수 있는 일부 심볼을 정의한다. 소니의 기본 힙 실행도 이러한 심볼을 사용한다. 따라서, 힙이 어디서 시작하는지 정확히 찾을 수 있다.

```
typedef int __attribute__ ((mode (TI))) heap_size_type
__attribute__((aligned(16)));
```

extern heap_size_type _end;

파일에 상기의 두 라인코드를 추가함으로써 실행 가능한 데이터가 어디서 끝나는지에 대한 정확한 주소를 찾을 수 있다. 이것 역시 힙의 시작으로 보인다.

아래 코드를 사용하여, 힙의 시작부터 힙의 마지막까지 검색할 수 있다. 그렇게 함으로써, 모든 단일 메모리 블럭에 접근할 수 있다(리스팅 6).

```
void HeapWalk()
{
int currSize, i;
int currCode, nextCode;

int lastBlock = 0;
int freeBlock = 0;

int heapStart = (((int)&_end) + 0x10);
int* currHeader = ((int*)ms_HeapStart)-1;
int* nextHeader = NULL;

do
        {
                currSize  = (*currHeader) & 0xfffffff0;
                nextHeader        = currHeader + (currSize>>2);
                currCode= (*nextHeader) & 0x0000000f;

                lastBlock = (currCode == 0x09);
                freeBlock = (currCode == 0x00);

                currHeader += (currSize>>2);
        } while (!lastBlock);
}
```

리스팅 6. PS2 Heapwalker 함수.

변수	설 명
currSize	현 메모리 블록의 바이트 사이즈. 항상 16바이트의 배수임
currCode	현 메모리 블록의 플래그. 항상 다음 메모리 블록에 포함됨
nextHeader	다음 메모리 블록의 헤더 주소 저장함수
lastBlock	현 블록이 힙상의 마지막 블록임을 표시하는 연산함수
freeBlock	블록이 프리메모리인지 혹은 사용된 메모리인지를 나타냄

메모리 덤프

힙을 검색함으로써, 마지막 주소가 무엇인지 알아낼 수 있다.

이미 시작주소를 알고 있다. 따라서, 시작과 마지막 주소를 사용함으로써, 전체 힙을 파일에 덤프할 수 있다. 이렇게 함으로써 실제 메모리 내용을 알아낼 수 있다 리스팅 6 의 HeapWalk 함수를 일부 수정하여 offline 상에서 MemAnalyze 을 이용해 다시 힙을 검색할 수 있다. 리스팅 7 은 전체 힙을 파일에 어떻게 덤프하는지 보여준다.

```
void dumpHeap()
        {
int heapStart = (((int)&_end) + 0x10);
int heapEnd = GetHeapEndByWalkingTheHeap();
int fd = sceOpen("host0:heap.bin", SCE_CREAT | SCE_WRONLY);
if (fd >= 0)
{
        sceWrite(fd, (void*)heapStart, heapEnd - heapStart);
        sceClose(fd);
}
        }
```

리스팅 7. PS2 힙 전체의 덤핑.

다음은 무엇인가?

이제 PS2 및 Xbox 각 플랫폼상에서 힙데이타를 파일로 덤프할 수 있으므로, 이제는 툴을 확인할 차례이다. 제 2 장에서는 맵 파일 분석인 PDB 분석(parsing)에 대해 자세히 살펴보겠다. 또한 Xbox 이미지를 어떻게 메모리로 로딩하는지도 자세히 보겠다. 또한 몇 가지 흥미로운 관점과 함께 모니터링툴이 덤프된 메모리를 이용하여 어떻게 데이터를 처리하는지 살펴보겠다.

참고문헌

[1] *See MIPS run*, by Dominic Sweetman. Morgan Kaufmann Publishers, 1999 [ISBN: 1558604103]

[2] Playing with the stack, by Chavdar Dimitrov.
http://www.codeproject.com/tips/stackdumper.asp#xx324128xx

[3] XDK documentation: chapter "Xbox kernel memory management"

[4] Rob Wyatt's explanation on fragmentation and caching on Xbox

Xbox newsgroups: news.xds.xbox.com

Search for:

Matt Benic
D3D_AllocContiguousMemory question
08/12/2002

[5] Xbox Memory Architecture and Performance, by Mike Abrash.

Available in the XDK documentation and on Microsoft website:
https://xds.xbox.com/BPProgInfo.asp?Page=content/prog_wp_memoryarch.htm

[6] Keith Packard's algorithm for callstack tracing on MIPS processors

Sony Developer Newsgroups (news.ps2-pro.com)

Search for:

Phil Camp (SN Systems) <phil@snsys.com>
sce.dev.prog.ee
Tuesday, February 04, 2003 2:22 PM
Re: call stack trace for EE?

[7] Metrowerks' CodeTEST
http://www.metrowerks.com/MW/Develop/AMC/CodeTEST/CodeTEST+Memory.htm

[8] Compuware Boundschecker
http://www.compuware.com/products/devpartner/bounds.htm

[9] Forrest Trepte's training session on Xbox memory management
https://xds.xbox.com/media/Memory%20Management_files/default.htm