임베디드 자바가상기계를 위한 고정 크기 메모리 할당 및 해제

양 희 재
경성대학교 컴퓨터공학과
전화 : 051-620-4783 / 휴대폰 : 016-242-4783

Fixed-Length Allocation and Deallocation of Memory for Embedded Java Virtual Machine

Heejae Yang
Dept. of Computer Engineering, Kyungsung University
E-mail : hjyang@star.kyungsung.ac.kr

Abstract

Fixed-size memory allocation is one of the most promising way to avoid external fragmentation in dynamic memory allocation problem. This paper presents an experimental result of applying the fixed-size memory allocation strategy to Java virtual machine for embedded system. The result says that although this strategy induces another memory utilization problem caused by internal fragmentation, the effect is not very considerable and this strategy is well-suited for embedded Java system. The experiment has been performed in a real embedded Java system called the simpleRTJ.

I. 서론

자바가상기계(Java virtual machine)의 핵심 부분을 이루는 요소 중 하나로 메모리 관리 기능을 들 수 있다. 어떤 클래스의 인스턴스가 생성될 때마다 그 인스턴스를 위한 메모리가 할당되며, 그 인스턴스가 더 이상 사용되지 않으면 할당된 메모리는 쓰레기 수집기(garbage collector)에 의해 해제되어지게 된다. 또한 메소드가 호출될 때마다 새로운 자바 스택이 생성되기 때문에 이때도 메모리의 할당이 이루어지며, 메소드로부터 복귀하면 할당된 메모리는 해제된다 [1].

효율적 자바가상기계의 구축을 위해서는 효율적 메모리 할당 및 해제 알고리즘의 지원이 필수적이다. 특히 메모리의 크기 및 프로세서의 성능에 제약을 받는 임베디드 시스템에서 그 중요성은 더욱 커진다. 메모리 할당 및 해제를 어렵게 하는 이유 중 한가지는 할당되는 메모리의 양이 임의의 크기를 갖기 때문에 할당과 해제가 반복되면 심각한 외부 단편화 (external fragmentation) 현상이 발생하기 때문이다 [2].

이런 단편화 현상을 최소화하고 메모리를 효율적으로 관리할 수 있는 한가지 방법은 고정된 크기로 메모리를 할당하는 것이다. 페이징 시스템의 경우처럼 항상 고정 길이로 메모리를 할당하면 외부 단편화 현상을 피할 수 있으며, 따라서 밀집화(compaction) 등과 같은 고비용의 동작도 없앨 수 있게 된다. 그러나 이 경우 내부 단편화 현상, 즉 할당된 메모리가 모두 사용되지는 않음으로 인해 낭비되어지는 메모리가 있을 수 있다는 단점이 생긴다.

본 논문에서는 임베디드 자바가상기계에서 고정 크기로 메모리를 할당 및 해제를 하는 방법 및 그것의 영향 등에 대해 알아보았다. 실험 결과 인스턴스의 크기는 예외/오류 클래스들과 일반 클래스들 간에 큰 차이가 있으므로 내부 단편화에 따른 메모리의 낭비는 발생하지만 실제로 예외/오류 사건은 정상 상태에서 빈번히 일어나는 일이 아니므로 그것에 따른 영향은 무시할 만 하였다. 또한 자바 스택의 크기에도 큰 차이가 있지만 인스턴스의 경우와 달리 자바 스택은 메소드 리턴에 따라 즉시 메모리 해제가 일어나므로 그

영향 역시 크지 않았다. 즉 고정 크기 메모리 할당 정책은 임베디드 자바 시스템에서 적당한 대안으로 평가되었다. 이 실험은 원천코드가 공개되어져있는 simpleRTJ [3] 라고 하는 임베디드 자바가상기계 상에서 이루어졌다.

본 논문의 구성은 다음과 같다. 2장에서는 자바가상기계의 메모리 사용에 대한 일반적 내용을 소개하며, 3장에서는 고정 크기 메모리 할당에 대해 인스턴스를 위한 힙 메모리와 자바 스택을 위한 프레임 메모리로 나누어 설명하였다. 4장에서는 simpleRTJ 상에서 실제로 실험한 내용 및 분석 결과를 설명하고 5장에서 결론을 맺는다.

II. 자바가상기계의 메모리 사용

자바가상기계에서 메모리의 사용은 매우 중요한 부분을 차지한다. 그림 1은 자바가상기계 내부의 데이터 영역, 즉 메모리 사용에 대한 개념을 보여주고 있다.

이 그림에서 알 수 있듯이 메모리의 사용은 크게 네 가지로 나뉘어진다 [1]. 먼저 클래스 영역이 있는데, 이곳에는 클래스들의 코드와 각종 상수들이 놓인다. 코드와 상수들은 프로그램 실행 시 변함이 없으므로 동적 메모리 관리와는 무관하다.

둘째로 클래스의 인스턴스들을 저장하기 위한 힙(heap) 영역이 있다. 각각의 인스턴스들은 클래스 선언에서 정의된 필드들을 저장하기 위해 메모리 슬롯을 필요로 하는데, 각 슬롯들은 각기 하나의 필드를 저장한다. 상속성의 원리에 의해 자신의 필드 외에 상위 클래스들이 가지는 필드들을 저장할 메모리 슬롯도 가져야 한다. 프로그램 실행 시 새로운 인스턴스들이 계속해서 만들어짐에 따라 힙 영역이 계속 증가되어지게 되며, 더 이상 사용되지 않는 인스턴스들이 갖고 있는 슬롯들은 쓰레기 수집기에 의해 회수되어진다.

세 번째는 자바 스택 영역이다. 자바에서는 메소드

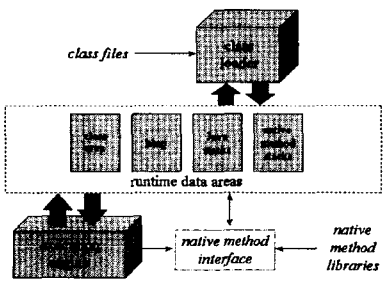

그림 1. 자바가상기계의 메모리 사용

가 호출될 때마다 자바 스택(Java stack)이라는 데이터 공간이 만들어진다. 스택 프레임은 연산의 대상이 되는 오퍼랜드(operand)들이 놓여지게 되는 오퍼랜드 스택과 파라미터 전달이나 지역변수 저장을 위한 목적으로 사용되는 지역변수배열(local variable array) 등으로 구성된다. 하나의 메소드가 호출될 때마다 새로운 자바 스택이 만들어지며, 이 메소드가 종료되고 원래 이 메소드를 호출했던 메소드로 복귀할 때마다 프레임은 사라지게 된다.

마지막 네 번째는 네이티브 메소드 스택으로서 C 등으로 작성된 네이티브 메소드가 실행될 때 필요로 하는 스택 영역이다.

첫 번째의 클래스 영역은 클래스 적재 후 일정 크기로 고정되어 있으며, 네 번째의 네이티브 메소드 스택은 일반 컴퓨터의 스택 영역과 크게 다르지 않다. 따라서 본 논문에서는 특히 두 번째와 세 번째의 힙 영역 및 자바 스택 영역을 위한 메모리 사용에 대해서 고찰해 본다.

III. 고정 크기 메모리 할당

3.1 힙 메모리의 할당

먼저 힙 메모리의 할당에 대해 알아보자. 힙은 각 클래스의 인스턴스를 저장하는 목적으로 사용되며, 인스턴스의 주요 정보는 필드들이다. 클래스들의 필드 개수는 일반적으로 서로 다르므로 할당되는 힙 메모리의 크기도 서로 다르다.

각 인스턴스는 헤더 부분과 데이터 부분으로 구성된다. 헤더 부분은 이 인스턴스의 크기, 동기화를 위한 열쇠의 개수, 상태를 나타내는 플레그, 이 인스턴스가 속한 쓰레드의 id 등을 나타내는데 사용되며, 데이터 부분은 실제 필드들이 저장되는 메모리 슬롯들이 놓이는 곳이다. 헤더의 길이는 고정이지만 데이터 길이는 가변적이다.

본 논문의 연구 목적은 모든 인스턴스를 위한 힙 메모리를 동일하게 일치시키고, 그것에 따른 효과를 조사하는 것이다. 즉 필드를 많이 필요로 하는 인스턴스나 적게 필요로 하는 인스턴스나 모두 일정한 양, 즉 가장 많은 필드를 필요로 하는 인스턴스가 요구하는 양 만큼의 힙 메모리를 할당하는 것이다. 이 경우 실제로 사용되지 않는 메모리, 즉 내부 단편화 현상이 발생하는데, 다음 장에서 그 정도를 분석해 본다.

3.2 자바 스택 영역의 할당

자바 스택도 인스턴스의 경우와 마찬가지로 헤더 부분과 데이터 부분으로 구성된다. 헤더 부분은 연결 리스트를 이루는 포인터, 메소드의 실행 위치를 나타내는 프로그램 카운터, 오퍼랜드 스택에서 최상위 스택 항목을 가리키는 스택 포인터, 기타 이 자바 스택을 사용하는 메소드와 이 메소드가 속한 클래스를 가리키는 포인터 등으로 이루어진다. 데이터 부분은 오퍼랜드 스택과 지역변수배열이 저장되는 메모리 슬롯들이 놓이는 곳이다. 헤더의 길이는 고정이지만 데이터 길이는 가변적이다.

본 연구에서는 모든 자바 스택을 위한 메모리를 동일하게 일치시키고, 그것에 따른 효과를 조사하고자 한다. 즉 오퍼랜드 스택과 지역변수를 많이 필요로 하는 자바 스택이나 적게 필요로 하는 자바 스택이나 모두 일정한 양, 즉 가장 많은 오퍼랜드 스택과 지역변수를 필요로 하는 메소드가 요구하는 양 만큼의 메모리를 모든 자바 스택에게 할당하는 것이다. 마찬가지로 내부 단편화 문제가 발생하는데, 그것의 영향을 다음 장에서 분석해 본다.

Ⅳ. 실험 및 분석

고정 크기 메모리 할당 및 해제 방식의 성능을 알아보기 위하여 simpleRTJ 자바가상기계에서 실험을 하였다. 고정 크기 메모리 할당 및 해제 방식은 이 시스템에 이미 적용되어 사용되고 있지만 효과 및 성능 등에 대한 논문은 아직까지 발표되지 않고 있다.

4.1 인스턴스 및 자바 스택의 크기

인스턴스를 위한 메모리는 3.1 절에서 살펴 본 바와 같이 헤더 부분과 데이터 부분으로 나뉜다. simpleRTJ 의 경우 헤더의 크기는 10 바이트이다. 데이터 부분은 필드의 개수에 따라 결정되는데, API 핵심 클래스들에 대한 조사 결과 예외/오류 클래스들은 모두 0개의 필드를 갖고 있었으며, 기타 일반 클래스들은 평균 4.53 개의 필드를 가진다. 전체 평균은 1.58 개다 [4]. 한개의 필드 크기는 4바이트이므로 이 값들은 각각 18바이트와 6바이트에 해당된다.

고정 크기 메모리 할당을 위해서는 응용 프로그램을 이루는 클래스들 중에서 가장 많은 필드를 가지는 (상위 클래스의 필드들도 포함) 클래스의 필드 크기로 일치시켜야 하는데, 그림 2에서 볼 수 있듯이 필드의 개수는 0-6개가 대부분을 차지한다. 예외/오류 클래스들은 실제로 0개의 필드를 사용하지만 메모리가 할당되므로 낭비가 된다. 그러나 일반적 응용 프로그램에서

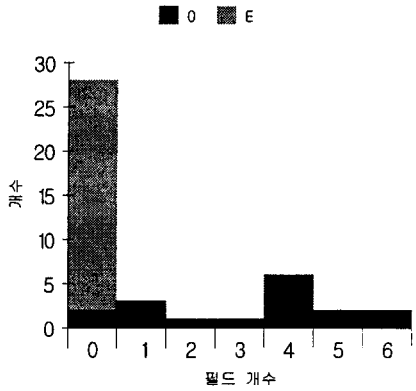

그림 2. 필드 개수 분포
O: 일반 클래스 E: 예외/오류 클래스

예외/오류 클래스의 인스턴스는 정상상태에서는 거의 생성되지 않으므로 그 효과는 크지 않다.

자바스택을 위한 메모리도 마찬가지로 헤더 부분과 데이터 부분으로 나뉜다. simpleRTJ 의 경우 헤더의 크기는 24바이트이다. 데이터 부분은 오퍼랜드 스택과 지역변수배열의 크기에 따라 결정되는데, API 핵심 클래스들에 대한 조사 결과 오퍼랜드 스택의 크기는 평균 2.25, 지역변수배열의 크기는 평균 1.90 이었다 [4]. 스택과 배열도 한 개당 4바이트를 차지하므로 이 값들은 각각 9 바이트와 8 바이트 정도에 해당된다. 따라서 데이터 부분의 평균 크기는 이들의 합인 17 바이트 정도에 해당된다.

고정 크기 메모리 할당을 위해서는 가장 큰 오퍼랜드 스택과 지역변수배열의 합으로 크기를 일치시켜야 하는데, 그림 3에서 볼 수 있듯이 스택의 크기는 1-6개, 지역변수배열의 크기는 1-9개이다. 최대값을 고려

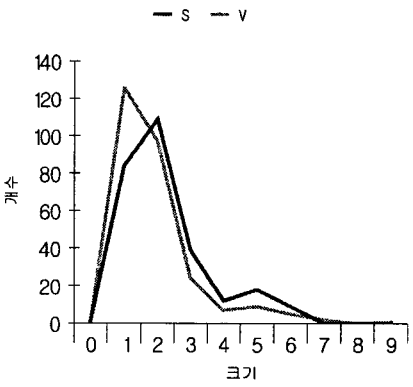

그림 3. 스택 및 지역변수배열 크기의 분포
S: 오퍼랜드 스택 V: 지역변수배열

한다면 15, 즉 60바이트까지 커질 수 있다. 이 값으로 통일할 경우 만일 1개의 스택과 1개의 지역변수배열을 사용하는 메소드라면 13개 크기, 즉 52바이트의 메모리 낭비가 있을 수 있다. 그러나 인스턴스와 달리 자바 스택은 메소드 호출시 만들어졌다가 메소드 리턴시 사라지므로 한 메소드가 다른 메소드를 부르고, 그 메소드가 또 다른 메소드를 부르는 등 메소드 호출 사슬의 길이가 크게 길지 않는 한 이것에 따른 메모리 낭비 역시 크지 않다.

4.2 null 프로그램에서의 분석

다음으로는 null 프로그램에 대해 분석해보았다. null 프로그램은 다음과 같이 아무런 유효 작업이 없는 프로그램으로 정의했다.

```
public class Test {
 public static void main() {
 }
}
```

이와같이 간단한 프로그램일지라도 다수개의 클래스로 구성된다. 어느 클래스라도 최상위 클래스에 해당되는 Object 클래스를 필요로 하며, 또한 쓰레드 실행을 위한 Thread 클래스를 필요로 한다. 그외에도 실행시 발생할 지 모르는 여러 가지 예외 및 오류 처리를 위한 클래스들도 있어야 한다. simpleRTJ 에서는 이 프로그램을 위해 Test 클래스를 포함하여 모두 23개의 클래스를 사용하고 있다. 이들 중 17 개는 예외 및 오류 처리를 위한 Throwable 및 그 이하의 계층에 위치한 클래스들이며 나머지는 Object, Thread, Runnable, String, 그리고 메인 클래스인 Test 등이다. 이들 클래스들의 인스턴스 크기는 필드 저장을 위한 16 바이트와 오브젝트 헤더 10 바이트 등 26 바이트로 통일되었다.

적재된 클래스들 중 실제로 인스턴스를 생성하는 것은 Thread 와 InternalError 등 11개의 예외 및 오류 클래스들이다. 따라서 main 메소드가 실행되는 시점에 힙 메모리는 312 바이트가 사용되었다 (26 바이트 × 12 인스턴스).

한편 자바 스택은 특정 클래스의 메소드가 호출 될 때 사용되다가 리턴될 때 다시 반납되어지므로 객체 생성과 달리 사용량이 많지 않았다. 메소드 실행을 위한 자바 스택의 크기는 데이터 저장을 위한 36 바이트와 프레임 헤더 24 바이트 등 60 바이트였다.

실험 결과 main 메소드가 실행되는 시점까지 최고 6개의 자바 스택 사슬이 만들어졌음을 알 수 있었다. 그러나 자바 스택은 메소드 리턴과 함께 반납되어지므로 이 시점에서 사용된 메모리는 180 바이트였다 (60 바이트 × 3 스택 크기). 또한 이 세 개의 자바 스택 프레임 중에서 main 메소드 실행 시점에서 실제로 사용 중인 프레임은 한 개에 불과했으며, 나머지 두 개는 재사용 가능한 상태로 비어있음을 확인할 수 있었다. 즉 단지 60바이트만 사용 중인 상태에 있다는 것이다.

V. 결론

본 논문에서 우리는 임베디드 자바 시스템에서 모든 인스턴스들이 요구하는 메모리의 양을 같은 크기로 통일하고, 또 모든 메소드들이 필요로 하는 자바 스택의 메모리 양을 같은 크기로 통일했을 때 기대되는 장점 및 단점들에 대해 조사해 보았다.

가장 큰 장점은 외부 단편화 문제가 해결되어지므로 보다 효율적인 메모리의 할당이 가능해진다는 점이다. 반면 가장 큰 단점은 내부 단편화 문제에 따른 메모리 활용률이 감소할 수 있다는 점이다. 본 논문은 실제로 사용 중인 임베디드 자바 시스템에 이 방식을 적용했을 때 발생되는 내부 단편화의 문제점을 실험을 통해 분석해 본 것이다. 핵심 API 클래스들에 대해 정적인 분석을 행하였고, 또한 실제 프로그램 실행 시 동적 인스턴스 생성과 메소드 호출에 따른 메모리 활용에 대해 조사해보았다. 분석 결과 내부 단편화가 발생하기는 하지만 그것에 따른 메모리 낭비는 미미한 수준이며, 따라서 고정 크기 메모리 할당 및 해제 방식은 프로세서의 성능이 크지 않은 임베디드 시스템을 위한 좋은 대안으로 판단된다.

참고문헌

[1] 양희재, 자바가상기계, 한국학술정보, 2001년 3월, ISBN 89-5520-342-4

[2] P. Wilson et al, "Dynamic Storage Allocation: A Survey and Critical Review," *Int'l Workshop on Memory Management*, 1995, pp.1-116

[3] RTJ Computing, *simpleRTJ: A Small Footprint Java VM for Embedded and Consumer Devices*, http://www.rtjcom.com

[4] 양희재, "simpleRTJ 클래스 파일의 형식 분석," 한국해양정보통신학회 2002 추계학술대회, 2002. 11., pp.373-377