제 18 회 한국정보처리학회 추계학술발표대회 논문집 제 9 권 제 2 호 (2002.11)

임베디드 시스템을 위한 가상머신 분석 및 설계

백대현, 정명조, 안희중, 박희상, 이철훈
충남대학교 컴퓨터공학과
e-mail : dhbaek@ce.cnu.ac.kr

Analysis and Design of Virtual Machine for Embedded System

Dae-Hyun Baek, Myung-Jo Jung, Hee-Jung Ahn, Hee-Sang Park, Cheol-Hoon Lee
Dept. of Computer Engineering, Chung-Nam University

요 약

최근들어 IT 산업이 급속도로 발전하면서, 리소스가 제한된 작은 기기들의 사용이 비약적으로 증가하는 추세에 있다. 이들 기기들에 플랫폼 독립성(Platform Independency), 보안성(Security), 이동성(Mobility) 등의 장점을 포함하고 있는 자바 환경을 적용하려는 연구가 계속되고 있는데, 자바 환경의 핵심인 자바가상머신(Java Virtual Machine: JVM)은 임베디드 시스템이나 모바일 시스템과 같이 작고, 자원이 제한적인 장치에 탑재하기에 너무 큰 용량(footprint)을 차지한다. 이를 해결하기 위해 좀 더 경량화한 가상머신이 필요하였다. 본 논문에서는 네트워크 연결 능력이 있고 적은 리소스를 가진 다양한 기기들에 적합한 최소 크기의 표준 자바 플랫폼에 대한 Configuration 인 CLDC(Connected, Limited Device Configuration)에서 정의하고 있는 K 가상머신(K Virtual Machine: KVM)에 대해 분석하고 설계한 내용을 기술하고 있다.

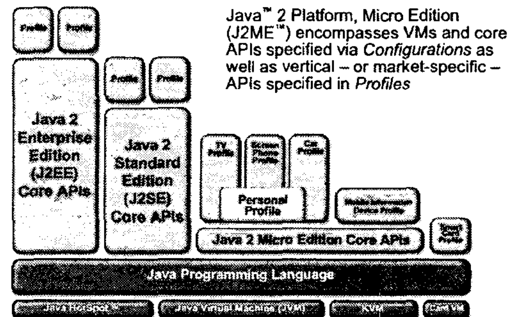
1. 서론

증가하는 임베디드 시스템에 대한 운영체제와 각각에 맞는 응용 프로그램들을 매번 다시 개발하는 것은 개발자에게는 매우 번거로운 일이다. 그러나 각각의 임베디드 시스템에 맞는 자바가상머신(Java Virtual Machine: JVM)이 탑재되어 있다면 하나의 응용 프로그램을 작성하더라도 모든 플랫폼에서 사용할 수 있게 된다[1]. 그러나 작고 리소스가 제한적이며 네트워크 연결 능력이 있는 장치들에 JVM 을 탑재하기엔 JVM 이 너무 큰 용량을 차지하게 된다. 이를 해결하기 위해 작은 footprint Java™ platform 을 개발할 필요성이 생기게 되었다. JCP(Java Community Process)들에 의해 표준화 작업이 진행되었고, JSR-30 문서를 통해 표준이 발표되었다. 그 결과로 발표된 표준이 -네트워크 연결 능력이 있고, 적은 리소스를 가진 다양한 기기들에게 적합한 최소 크기의 자바 플랫폼에 대한 Configuration - CLDC(Connected, Limited Device Configuration)이다. CLDC 는 KVM 을 기본 가상머신으로 채택하고, 여기에 코어 API (Application Programming Interface)에 대한 정의를 포함하고 있다.

본 본문은 2 장에서 CLDC 에 대한 관련연구를, 3 장에서는 KVM 분석 및 설계를, 4 장에서는 결론 및 향후 과제에 대하여 기술한다.

2. CLDC 란 무엇인가?

CLDC 는 기본 가상머신으로 KVM 을 채택하고, J2SE 의 코어 API 의 서브셋을 포함한 J2ME Configuration 이다.



[그림 1] 자바 2 플랫폼

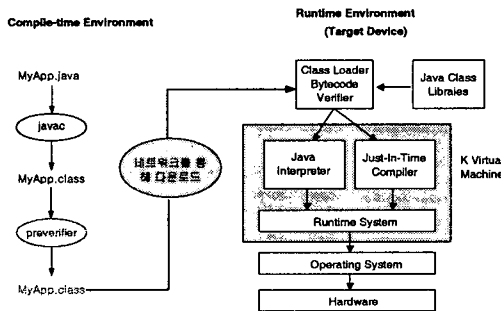
CLDC 의 명세서에는 VM 과 자바 언어의 특성, 코어 자바 라이브러리, 입출력 및 네트워크, 보안과 국제화 등에

대해서 정의하고 있다. 또한 CLDC 명세서에는 장치들이 128 ~ 256KByte 의 free 메모리와 16/32-bit 의 RISC 또는 CISC 방식의 마이크로 프로세서를 가지며, 전력을 적게 소모하고, 네트워크 연결성을 보장해야 한다고 정의하고 있다[2][3].

3. KVM(K Virtual Machine) 분석 및 설계

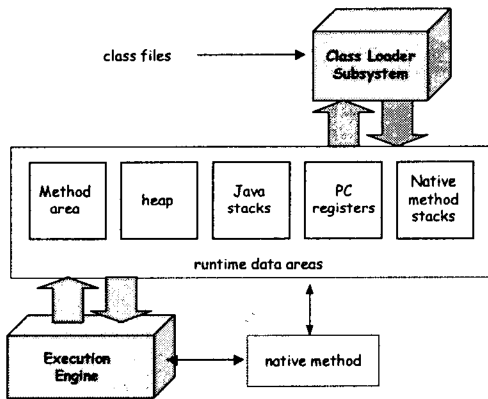
3.1 KVM 전체 구성

[그림 2]는 자바 프로그램을 컴파일하고 검증과정을 거쳐 실행되는 단계에 대한 설명이다.



[그림 2] 자바 프로그래밍 환경

[그림 3]에서와 같이 KVM 의 내부에는 클래스 파일을 메모리에 로드해주는 클래스 로더 시스템과 로드된 클래스 파일을 바이트코드 단위로 읽으면서 수행하는 Execution Engine 이 가장 핵심적인 부분이며, 런타임 데이터 영역은 메소드 영역, 힙, 자바 스택, PC registers, native 메소드 스택으로 이루어진다.



[그림 3] KVM의 내부 구조

3.1.1 클래스 로더 시스템

타입들을 찾고 로딩하는 JVM 구현의 일부분으로 JVM 은 두 종류의 클래스 로더(부트스트랩(bootstrap) 클래스 로더와 사용자 정의 클래스 로더)를 포함하는데 KVM 에서는 Java API 의 클래스들을 포함한 신뢰성있는 클래스들을 어떻게 로드하는지 나타내는 부트스트랩 클래스 로더만 지원한다[2][4].

클래스 로더 시스템은 클래스들을 위한 바이너리 데이터를 배치하거나 임포팅(importing)하며, 임포트된 클래스들의 정확성을 검증한다. 클래스 변수를 저장하기 위해 메모리를 할당하고 초기화하며, 심볼릭 레퍼런스의 resolution 을 돕는다. 이러한 과정은 로딩, 링킹, 초기화 과정을 통해 수행되는데 이에 대해서는 다음 절에서 설명한다.

3.1.2 런타임 데이터 영역

KVM 은 프로그램이 실행되는 동안 사용되는 여러 개의 런타임 데이터 영역을 정의한다. 이런 데이터 영역의 일부는 KVM 이 시작될 때 할당하며 KVM 이 종료될 때 회수된다. 다른 데이터 영역은 쓰레드가 생성될 때 할당되고 쓰레드가 종료될 때 회수된다[2][4].

3.1.2.1 메소드 영역

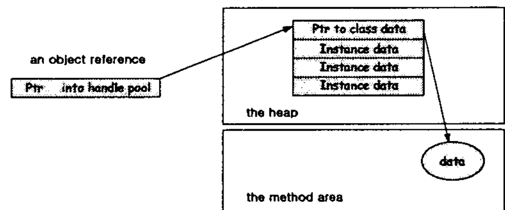
로드된 타입들에 대한 정보는 메소드 영역이라 불리는 논리적 메모리 영역에 저장된다. KVM 은 타입을 로드하고, 클래스 파일을 배치하기 위해 클래스 로더를 사용한다. 클래스 로더는 클래스 파일(바이너리 데이터의 순차적 스트림)을 읽어 KVM 에 전달하며, KVM 은 바이너리 데이터로부터 타입에 대한 정보를 추출하여 메소드 영역에 정보를 저장한다.

각각의 타입을 로드하기 위해 KVM 은 메소드 영역에 타입의 fully qualified name, 타입의 수퍼 클래스의 fully qualified name, 타입이 클래스 인지 인터페이스 인지 구별하는 정보, 타입의 모디파이어(public, abstract 또는 final 등), 수퍼 클래스의 fully qualified name 에 대한 리스트 정보를 저장한다[4].

각각의 로드된 타입의 정보를 얻기 위해 메소드 영역에 타입을 위한 constant pool, 필드 정보, 메소드 정보, 상수를 제외한 타입에서 선언된 모든 클래스 변수, 클래스 ClassLoader 의 레퍼런스, 클래스 Class 의 레퍼런스에 대한 정보를 저장한다. 모든 쓰레드가 같은 메소드 영역을 공유하게 되기 때문에 메소드 영역의 데이터 구조는 쓰레드에 안전하게 디자인되어야 하며, 이 영역은 가비지 콜렉터에 의해 관리된다[4].

3.1.2.2 힙(Heap)

자바 애플리케이션을 실행할 때 새로운 객체를 저장하는 메모리로 모든 쓰레드에 의해 공유되기 때문에 쓰레드 안전을 보장하기 위한 방법이 필요하며, 힙 영역의 관리는 가비지 콜렉터가 담당한다.



[그림 4] 한 영역에서 객체를 참조

3.1.2.3 자바 스택

새로운 쓰레드가 실행될 때 KVM 은 쓰레드를 위한 자바 스택을 생성하며, 자바 스택에 직접적으로 필요한 두개의 연산(pushing 과 popping frames)만 수행한다. 자바 스택에는 메소드가 호출될 때 생성되는 세 영역(local variable, operand stack, frame data)으로 구성된 스택 프레임을

push 한다[4].

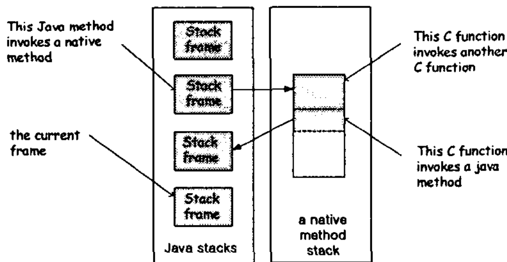
- **local variable**
  메소드의 파라미터와 지역 변수들을 저장한다.
- **operand stack**
  pushing 과 popping 연산에 사용되는 값을 저장한다.
- **frame data**
  Constant pool resolution, 메소드 리턴, 예외 처리를 수행하기 위한 데이터를 저장한다. 이러한 데이터는 KVM 이 constant pool 에 있는 엔트리를 참조하는 명령을 수행할 때 정보를 얻기 위해 사용된다.

3.1.2.4 PC registers

실행중인 프로그램 각각의 쓰레드는 그것이 시작될 때 생성된 자신의 Program Counter(PC) register 를 가진다. PC register 는 한 word 크기이며, native 포인터와 리턴 주소를 가진다. 쓰레드가 자바 메소드를 실행할 때, PC register 는 쓰레드에 의해 실행되는 현재 명령의 주소를 포함하고 있다.

3.1.2.5 Native 메소드 스택

쓰레드가 native 메소드를 호출할 때 생성되는 스택으로 KVM 구조나 보안 제약을 받지 않는다. [그림 5]는 자바와 native 메소드를 호출하는 쓰레드를 위한 스택 구조이다.



[그림 5] 자바와 native 함수를 호출하는 쓰레드를 위한 스택

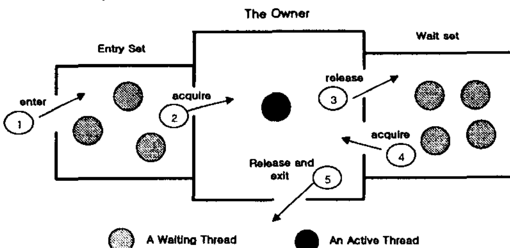
3.1.3 Execution engine

KVM 구현의 핵심부분으로 execution engine 의 수행은 명령 집합으로 구성된다. 각각의 명령을 수행하기 위해 설계명세서에는 명령어에 따른 동작을 자세하게 정의한다.

3.1.3.1 명령어 집합

메소드의 바이트코드 스트림은 KVM 을 위한 명령 순서이고 Execution engine 이 바이트코드를 한번에 하나씩 수행한다.

3.1.3.2 쓰레드

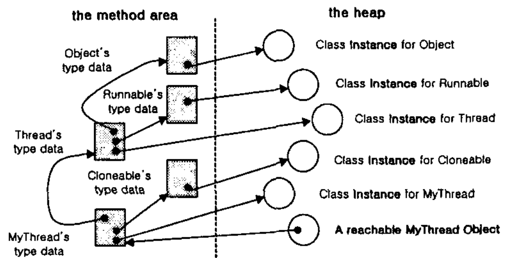


[그림 6] 자바에서 모니터 기법

KVM 에서는 쓰레드 그룹과 데몬 쓰레드는 지원하지 않으며, 쓰레드 구현은 동기화의 두가지 측면(객체 락킹과 쓰레드 wait, notify)을 제공한다[2][3]. KVM 에서 객체 락킹을 제공하기 위해서 [그림 6]과 같은 모니터 기법을 이용한다.

3.2 타입의 생명주기

타입의 생명주기는 [그림 7]에서 보는 바와 같고 각각에 대한 설명은 다음 절에서 자세히 기술한다.



[그림 7] 도달 가능한 객체들을 통한 클래스 인스턴스

3.2.1 타입 활성화

클래스와 인터페이스의 로딩과 링킹의 시간은 유동적으로 구현되지만 초기화는 반드시 정해진 시간에 수행되어야 하며, 모든 클래스의 수퍼클래스는 자식 클래스의 초기화보다 먼저 수행되야 한다. 초기화 과정이 수행되는 경우는 다음과 같다[4].

(1) 바이트코드의 new 명령어를 수행할 때
(2) 클래스에 선언된 스태틱 메소드를 호출할 때
(3) 클래스나 인터페이스에 선언된 스태틱 필드를 사용하거나 할당할 때
(4) 클래스의 서브클래스를 초기화할 때
(5) KVM 이 시작할 때 초기화 클래스로써 클래스를 지시할 때

3.2.1.1 로딩

로딩은 특정한 이름을 사용하여 클래스나 인터페이스의 바이너리 형태를 찾는 과정으로 KVM 이 수행해야 할 세 가지의 기본적인 행동이 있다. 첫째, 타입의 fully qualified name 이 주어졌을 때 그 타입을 나타내는 바이너리 데이터 스트림을 생성한다. 둘째, 메소드 영역에 내부적 데이터 구조로 바이너리 데이터 스트림을 파싱(parsing)한다. 셋째, 타입을 표현하는 클래스 java.lang.Class 의 객체를 생성한다.

바이너리 데이터를 생성하기 위한 방법으로 로컬 파일 시스템으로부터 자바 클래스 파일을 로드하거나, 자바 클래스 파일을 네트웍을 통해서 다운받는 방법, JAR 파일에서 자바 클래스 파일을 풀어서 얻는 방법이 있다. 또한 클래스 로더는 타입을 로드하기 전 단계인 타입의 first active use (3.2.1 에서 설명한 5 가지 경우)까지 수행되지 않는다[2][4].

3.2.1.2 링킹

링킹 과정은 verification, preparation, resolution 의 3 단계로 수행된다.

3.2.1.2.1 Verification

J2SE 에서 사용되는 클래스 파일 검증기는 작은 장치에서 사용하기에 너무 큰 footprint 를 차지하기 때문에 KVM

에서의 검증기는 단순히 바이트코드를 순차적으로 검증하며 수행한다. 이러한 새로운 KVM 의 검증기에는 Off-device pre-verification 과 스택 맵을 이용한 런타임 검증이 있다.

- Pre-verification

자바 클래스 파일에 특별한 애트리뷰트를 추가하고 모든 subroutines 을 inline 화하며 클래스 파일에 있는 모든 jsr, ret, wide ret 바이트코드를 동등한 의미를 가지는 다른 바이트코드로 대체한다. 클래스 파일에 런타임 검증을 촉진시키기 위해 스택 맵 애트리뷰트를 사용하기도 한다[2].

- In-device verification

검증기는 모든 로컬 변수들의 타입과 주어진 메소드의 operand 스택 아이템을 저장하기 위한 충분한 메모리를 할당한다. 그 메모리 크기는 로컬 변수들의 최대값에 의해 결정되며, 스택의 깊이는 코드 애트리뷰트에 의해 결정된다. 다음 과정으로 검증기는 인스턴스 메소드, 변수 타입, empty operand 스택을 위한 this 포인터의 타입이 되는 derived 타입들을 초기화한다. 다음 과정으로 검증기는 각각의 명령어를 순차적으로 되풀이한다. 마지막 과정으로 검증기는 메소드에 있는 마지막 명령어가 unconditional jump(goto), return(ireturn), athrow, tableswitch, 또는 lookupswitch 인지 검사한다. 그렇지 않으면 검증 에러를 발생시킨다[2].

3.2.1.2.2 Preparation

KVM 은 클래스 변수를 위한 메모리를 할당하고, 디폴트 값으로 셋한다. 또한 실행중인 프로그램의 수행을 향상시키기 위해 데이터 구조를 위한 메모리(클래스 메소드를 위한 데이터를 포인트하는 메소드 테이블)를 할당한다.

3.2.1.2.3 Resolution

런타임 constant pool 에 있는 심볼릭 레퍼런스로부터 동적으로 구체적인 값을 결정하는 과정이다. 이 과정에는 클래스와 인터페이스 resolution, 필드 resolution, 메소드 resolution, 인터페이스 메소드 resolution 과정이 수행된다.

3.2.1.3 초기화

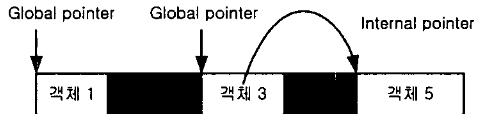
클래스나 인터페이스를 준비시키기 위해 요구되는 마지막 과정으로 적당한 초기값으로 클래스 변수를 세팅하는 과정이다. 자바 코드에서 적당한 초기값은 클래스 variable initializer 나 static initializer 에 의해 기술되고, 타입의 클래스 variable initializers 와 static initializers 은 자바 컴파일러에 의해 하나의 특별한 메소드(<clinit>())에 위치한다.

3.2.2 클래스 객체화

자바 프로그램에서 클래스가 객체화 되는 경우로 new 연산자를 수행할 때, java.lang.Class 에 있는 newInstance() method 를 호출할 때, 실행중인 클래스에서 clone() 메소드를 호출하는 경우가 있다. 이 밖에 소스 코드에서 String 객체를 사용할 경우에 클래스가 객체화 된다. KVM 은 클래스의 새로운 객체를 생성할 때 객체의 인스턴스 변수를 저장하기 위해 힙 영역에 메모리를 할당하며, 자바 컴파일러는 클래스마다 필요한 <init>이라 불리는 객체 초기화 메소드를 생성한다.

3.2.3 가비지 콜렉션

가비지 콜렉터는 프로그램에 의해 더 이상 참조되지 않는 객체들을 찾아서 그 객체들이 차지하고 있던 힙 공간을 재사용할 수 있게 한다. 이러한 가비지 콜렉터는 생산성을 높여주고, 프로그램 무결성을 보장하며, 자바 보안 전략에 중요한 역할을 한다[5][6]. KVM 에서 사용되는 가비지 콜렉션 방법은 마크-회수 알고리즘을 이용한다. 마크-회수 방법은 힙 영역을 순차적으로 검사하며 가비지 콜렉션을 수행하는 방법이다. 첫 번째 단계로 마크 단계에서는 root 로부터 시작하여 참조 트리를 순회하며 살아있는 객체에 대해 마크한다. 다음 단계인 회수 단계에서는 마크되지 않은 객체를 반환하여 그 객체가 차지하고 있던 힙 영역을 프로그램이 다시 사용할 수 있도록 한다.



[그림 8] 마크-회수 가비지 콜렉터

3.2.4 타입의 언로딩

프로그램에서 더 이상 참조되지 않는 클래스들을 언로드한다. JVM 에서는 사용자 정의 클래스 로더에 의해 로드된 클래스가 더 이상 참조되지 않을 경우 언로드 하지만 KVM 에서는 사용자 정의 클래스 로더를 지원하지 않는다.

4. 결론 및 향후 과제

리소스가 제한된 작은 기기들에 자바 환경을 적용하기 위해 JVM 을 탑재하기엔 리소스가 너무 부족하다. 이를 해결하기 위해 방법으로 본 논문에서는 임베디드 시스템이나 모바일 시스템과 같이 작고, 자원이 제한적인 장치에 탑재되는 작은 footprint Java™ platform 에 속하는 KVM 에 대해 분석 및 설계를 하였다. 향후 과제로 본 논문에서 분석 및 설계한 KVM 을 구현함은 물론이고, generation 알고리즘 방식을 적용한 좀더 효율적인 가비지 콜렉터의 구현, KVM 에서의 한글지원에 대한 문제, 네트웍을 통해 다운로드 받는 프로그램에 대한 보안사항 등을 좀더 연구해 나가야 할 것이다.

참고문헌

[1] http://www.inestech.com
[2] Sun Microsystems, " JSR-30 J2ME Connected, Limited Device Configuration(final Release)"
[3] Sun Microsystems, " Java™ 2 Platform Micro Edition (J2ME™) Technology for Creating Mobile Devices White Paper"
[4] Bill Veners, " Inside the JAVA2 Virtual Machine second edition"
[5] Richards Jones, " Garbage Collection Algorithm for Automatic Dynamic Memory Management" , p75-96, 1999
[6] Sun Microsystems, " The Java™ Virtual Machine Specification second edition