퍼스널 가상머신위한 프로세스 종료기법에 관한 연구

유홍식, 조유섭, 정민수, 경남대학교 컴퓨터공학과

A Study on Process Finalization algorithm for Personal JAVA Virtual Machine

Hong-sik Yoo, You-sub Cho, Min-soo Jung, Dept of Computer Engineering Kyungnam University
E-mail : yoohongsik@hanmaile.net,yoseph@mail.csc.ac.kr,msjung@hawk.com.kyungnam.ac.kr

요 약

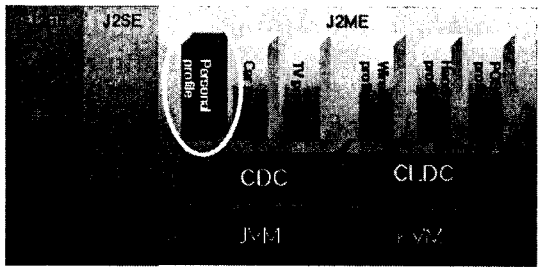
소형 플랫폼에 자바 언어를 구동하기 위한 PJAVA VM(이하 PVM)이 많은 관심을 끌고 있다. PVM 역시 JVM과 같이 클래스 로더, 서브시스템, 런타임 데이터 영역 (메소드 영역, 힙 자바 스택, PC레지스터,원시 메소드 스택), 실행 엔진으로 구성되어 있다. 본 논문의 서론은 PDA및 set-top등을 소개하고 JAVA 언어를 구동하기 위해 필요한 플랫폼을 소개하고, J2ME와 PJAVA를 관계를 소개하였다. 관련연구는 PVM의 메소드 구조에 대해 연구하였다. 그리고 본론으로 PVM에서 프로세서 마무리 처리과정을 분석하고 이 과정에서 프로세서의 종료 작업 알고리즘을 개선하였다.

1. 서론

컴퓨터의 응용 기술이 중대형 컴퓨터 기술로 점차적으로 소형 컴퓨터로 초점이 이동하고 있다. 소비자들은 집에서 뿐만 아니라 여행 중에도 자신의 컴퓨터를 사용하기를 원하며, 자신의 컴퓨터로 언제, 어디서나 인터넷에 접속하여 각종 정보를 얻기를 원한다. 이러한 소비자의 욕구를 만족하기 위해 PDA와 인터넷 폰이 등장 하였다.

최근 컴퓨터 응용 분야에서 인터넷상에 사용하는 컴퓨터 언어 중에서 자바가 많은 주목을 받고 있다. 비록 마이크로 소프트에서 자바와 경쟁하기 위해 .net & C#을 개발하여 보급하기 시작하였지만, 그들의 핵심적인 소스는 공개되지 않으므로 결국에는 라이센스에 걸리게 되어있다. 하지만 자바는 완전 공개 소스이므로 이런 문제에 걸리지 않는다.

자바 언어가 실행되기 위해서는 자바 가상 기계(이하 JVM)이라는 별도의 실행장치가 필요하다.



<그림 1-1 J2ME와 PJAVA>

자바는 <그림 1-1> 같이 환경에 따라 J2EE, J2SE, J2ME으로 구분할 수 있다. J2EE는 서버 시스템

본 연구는 정보통신부 대학 기초 연구(2001-149-2)지원으로 수행되었음.

환경의 자바, J2SE는 표준 시스템 환경의 자바, J2ME (JAVA 2 Micro Edition)는 소형 시스템의 자바를 이용하기 위한 환경에 사용된다. 따라서 가상 머신(이하 VM)도 각 환경에 따라 다르다. <그림 1-1>에서 J2ME만 보면 용량이 32KB ~ 512KB에선 KVM을 사용하고, 1MB ~ 10MB에선 JVM을 사용한다. 본 논문에서는 CDC에 분류되어 있는 PJAVA에 대해 고찰하고 프로세스의 종료작업을 개선하여 보다 효율적인 메모리 관리를 달성하고자 한다.

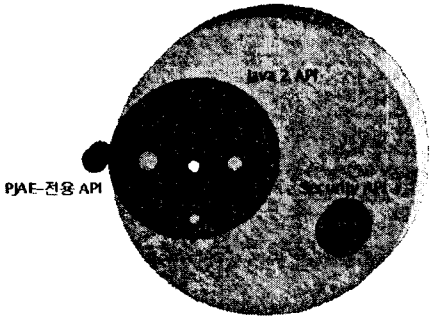
2. 관련연구

2.1 PJAVA

PJAVA는 개방형 시스템 운영기술로 PDA,웹 TV 디지털 셋탑 박스 등에 맞는 애플리케이션에 활용할 수 있다. 즉 네트워크 연결성이 있는 소형 기기를 위한 소규모의 자바 실행 환경이다. PJAVA는 JDK 1.1.8 기반이므로 서로의 대부분의 애플릿을 구동할 수 있다. PJAVA를 구동하기 위한 메모리는 VM과 클래스 라이브러리를 위한 2MB ROM과 실행을 위한 1 ~ 2 MB RAM이 필요하다. 다른 자바 가상기계와 비교해서 상대적으로 적은 메모리를 사용하기 때문에 프로그램의 이식성과 유연성, 그리고 재 사용성이 아주 좋다.

2.2 Personal Java Application Environment(이하 PJAE)

(1) PJAE는 자바 언어로 프로그래밍 된 소프트웨어를 실행하는 환경을 말한다.



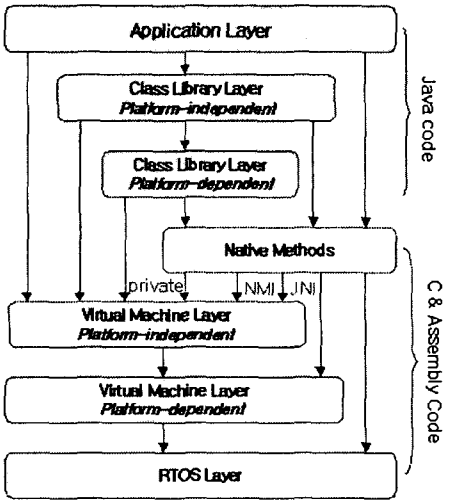
< 그림 2-1 PJAE 개념도 >

(2) PJAE는 JDK 1.1로부터 파생된 클래스 라이브러리와 VM이 있고 추가적으로 JDK 1.2의 보안 관련 API들이 있다.

(3) PJAE 구조는 솔라리스와 윈도우 NT 와 같은 데스크탑 플랫폼상에서 작동하는 JRE와 유사하다.

A. 클래스 라이브러리 계층은 C나 C++로 쓰여진 네이티브 메소드와 자바 프로그래밍 언어로 구현 되어 있다.

B. PJAE 구조를 플랫폼 독립과 플랫폼 종속적인 나누어 볼 때 PJAE내의 클래스와 패키지 플랫폼 독립적이다. 따라서 타 환경에 포팅할 때 수정할 필요가 없다. 하지만 낮은 단계에서는 플랫폼 종속적이므로 포팅 작업에 해당된다.



< 그림 2-2 PJAE Architecture >

C. 가상 머신 계층은 C로 대부분 구현되어있고 약간의 어셈블리 코드로 되어있다. 그 중 어셈블리 코드의 일부분은 인터프리터 루프에 포함된다. 가상 머신 계층은 4개의 프로그램 인터페이스를 가지고 있다.

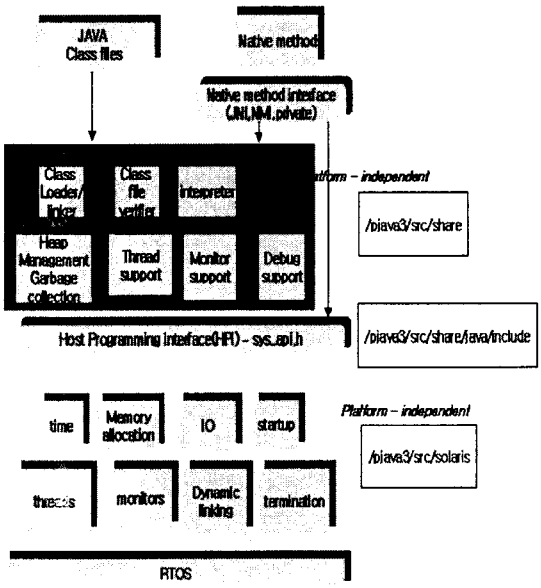
인터페이스	설명
Host Programming Interface (HPI)	VM Layer Platform dependent에 정의 된다. 이곳은 메모리 관리자와 쓰레드 관리자, I/O함수등을 포함
JAVA Native Interface (JNI)	JDK1.1에서 소개 오리지날 네이티브 메소드
JDK 1.0 native method interface (NMI)	클래스 라이브러리내의 네이티브 메소드들의 대부분 JDK 1.0 네이티브 메소드 인터페이스를 사용
VM 데이터 구조	라이브러리 내의 클래스들은 VM 데이터구조를 직접 접근한다.

<표 2-1 네이티브 메소드 계층>

(4) RTOS Layer는 PJAVA VM과 클래스 라이브러리를 위해 OS 서비스를 제공한다.

2.3 Personal JAVA 가상 머신 (이하 PVM) 구조

PVM 클래스 파일을 실행하고, 네이티브 메소드를 관리한다. 그리고, PVM는 JDK 1.1.8으로부터 파생되어있다. PVM역시 PJAE처럼 소스는 플랫폼 독립적이 부분과 플랫폼 종속부분으로 나눠져 있다.



< 그림 2-3 PJAVA VM 구조도 >

(1) 네이티브 메소드 계층: 낮은 계층의 클래스는 java.io, java.lang, java.util내에 존재한다. 이 클래스는 네이티브 메소드를 가지고 있으며 데이터 구조를 직접적으로 접근할 수 있다.

(2) 플랫폼 독립적인 계층: 일반적인 연관된 서비스를 제공한다. 예) 힙 관리자, 클래스 로딩 등의 역할을 한다. 이 소스의 위치는 src/share/java/runtime에 존재한다.

(3) 플랫폼 종속적인 계층: HPI를 수행하기 위한 함수를 제공한다.

(4) RTOS 계층: PVM을 위한 OS를 제공한다.

2.4 PJAVA VM 소스분류

(1) 플랫폼 독립적인 소스

pjava3/src/share/java/rumtime에 위치하고 있다.

분류	소스 파일
클래스 파일 로더 와 링커	classinitialize.c, classloder.c classresolver.c null_preloader.c preloafer.c
클래스 파일 검정기	check_class.c, check_code.c
인터프리터	classruntime.c, common_exceptions.c exception.c executeJava.c inline.c interpreter.c
힙 매니지먼트 / 가비지컬렉션	gc.c finalize.c
모니터	monitor.c monitor_cache.c
스레드	threadruntime.c threads.c
디버그	breakpoints.c debug.c jcov.c

<표 2-4 플랫폼 독립적인 소스>

(2) 플랫폼 종속적인 소스

pjava3/src/platform(name)/java/runtime에 위치하고 있다.

분류	소스 파일
Time 함수	system_md.c
메모리 할당함수	gc_md.c memory_md.c

I/O	dirent.c fd_md.c io_md.c path_md.c system_md.c
Startup	javai.c
스레드	thread_md.c
모니터	moniter_md.c
동적 링킹	invokeNative_x86.asm, linker_md.c
종료	system_md.c

<표 2-5 플랫폼 종속적인 소스>

3. P VM 프로세스 종료기법

3.1 기존 JVM 프로세스 종료(Finalization) 작업

(1) 프로세스의 종료 작업은 두 가지로 되어 있다. 첫째는 가비지를 수집하여 자동적으로 프로세스를 종료하는 경우가 있다. 둘째는 가비지 수집기는 많은 불확실성이 있어서 객체들을 임의의 순서로 회수하며, 전혀 회수하지 않을 수도 있다. 이 때 finalize() 메소드를 호출하여 프로세스 종료한다

(2) 가상 기계의 프로세스 종료과정

단계 1 : 참조되지 않은 객체를 발견하여 프로세스 종료 선언 여부를 확인하여 선언된 프로세스를 종료한다.

단계 2 : 루트 노드로부터 시작해서 참조되지 않은 객체를 다시 발견한다.

마지막 : 단계 1 과 단계 2를 반복하여 참조되지 않은 객체를 발견하여 반환한다.

3.2 PVM 프로세스 종료 작업

프로세스의 종료작업이 시작되면 finalization thread를 생성하고 초기화한다. 그리고 동시에 발생된 finalization thread는 큐를 이용하여 지정된 우선순위에 맞추어 프로세스를 종료한다. 이 과정에서 runfinalization() 메소드를 수행한다.

(1) Finalization thread를 생성하고 초기화 소스

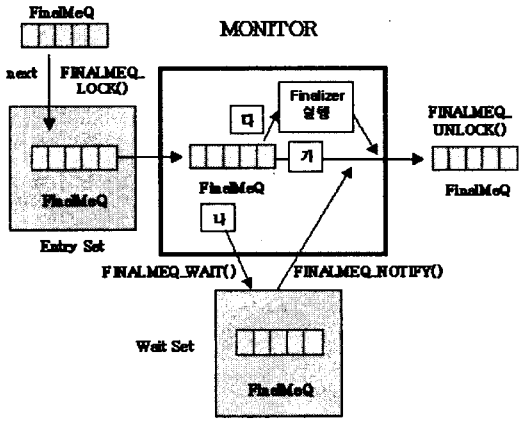
finalization thread는 신뢰한 코드를 실행할 수 있다. 그래서 그것은 일반적인 크기의 스택을 갖는다

```
void InitializeFinalizerThread()
{
(void) createSystemThread("Finalizer thread", MinimumPriority,
                                ProcStackSize, finalizer_loop);
}
```

(2) finalizeOnExit() : 모든 돌출된 객체를 종료한다. 첫번째는 가비지 수집과 종료작업을 동시에 관리한다. 이것이 일반적인 단계이다.

두번째는 runfinalization() 메소드를 수행하는 단계이다.

세번째는 위의 두 단계는 종료한 객체가 없는 때까지 반복된다. 외부 스택에서 STACK·REDZONE를 체크하고, gc(0,0) 메소드에서 가비지 수집(Garbage collection)을 실시한다. (첫번째 단계) runFinalization()에서 finalization의 진행 여부를 확인하다. (두번째 단계)



< 그림 3-3 runFinalization() 메소드 수행 과정 >

(3) runFinalizaion () 메소드 수행 과정

가. 만약 finalization 중이고 self-deadlock보다 좋다고 보증하면 FALSE를 반환한다. 이 때 결과를 통지할 필요가 없다.

나. 만약 finalization 중이 아니고 Wait set에서 대기하였다가 인터럽트가 발생되면, FALSE를 반환한다. 이 때 결과를 통지해야 한다.

다. 만약 finalization 중이 아니고 인터럽트가 발생하지 아니면, Entry Set에 다음 FinalizeMeQ를 받아

오고 execute_finalizer(handle) 메소드를 호출하여 프로세스 종료작업을 실행한다. 그리고 TRUE를 반환한다.

(4) execute_finalizer() 메소드 실행과정

프로세스 종료과정은 runFinalization() 수행여부를 결정하고, <그림3-3>과 같이 execute_finalizer( ) 메소드에서 실질적인 프로세스의 종료작업을 실행한다.

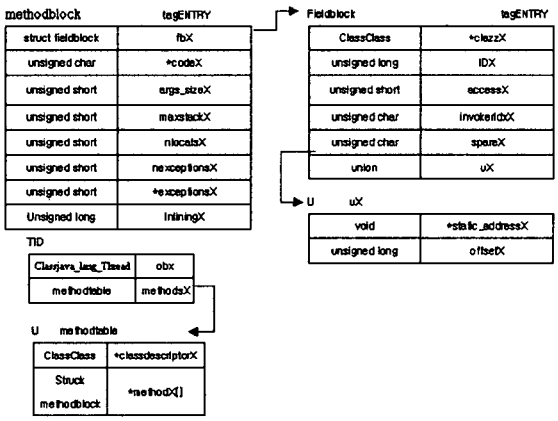
<그림 3-4>는 execute_finalizer( ) 메소드의 자료 구조이다. ClassClass 자료구조에서 파생된 구조이고, TID에서 식별자를 파악하여 TID의 해당하는 메소드 테이블을 슬롯의 개수만큼 가지고 온다. 그리고 <그림 3-4> 같이 자료구조로 메소드 블록을 메소드 테이블 메소드 개수 만큼 생성한다.

첫째는 do_execute_java_method에서 해당하는 변수 인수(argument)를 초기화 한다.

둘째는 do_execute_java_method_vararg 메소드를 실행한다. 이때 method_name과 signature는 "0"으로 설정한다.

셋째는 인수(argument)를 다시 설정한다.

do_execute_java_mehod() 메소드에서 위와 같은 매개변수를 주어서 최종 프로세스 종료 작업을 실행한다.



< 그림 3-4 execute_finalizer() 자료구조 >

```
mb = mt_slot(obj_methodtable(handle), FINALIZER_MB_OFFSET);
  //FINALIZER_MB_OFFSET 1
  // #define obj_methodtable(obj) ((obj)->methodsX);
  // #define mt_slot(methodtable, slot) (methodtablemethods(methodtable)[slot]);
  // #define methodtablemethods(mtab) ((mtab)->u.methodsX);
```

< 그림 3-5 메소드블럭 코드>

4. 결론 및 향후과제

본 논문에서 언급한 PJAVA 언어를 이용하여 소형기기의 사용하여 이동 중에도 보다 효율적인 인터넷을 사용할 수 있다고 소개하였다. 이 과정에서 PJAVA VM의 프로세스 종료과정을 개선하여 보다 빠르고 효율적인 프로세스를 관리하고, 효율적인 메모리를 관리하였다.

향후과제로는 현재 PJAVA VM을 솔라리스와 윈도우스 CE위에 장착할 수 있게 개발되었다. 이러한 PJAVA VM를 임베리드 리눅스 위에 설치하고자 한다 솔라리스는 소형 기기에 장착할 수가 없다. 따라서 PJAVA를 활용에 비효율적이다. 또한 MS의 윈도우스 CE에서 PJAVA가 잘 설치되고 아주 효율적으로 사용할 수 있지만 윈도우스 CE를 사용하면 중요한 API를 비싼 가격으로 구입하여야 한다. 즉 주요 API는 라이센스로 되어 있다. 그래서 당 연구실은 PJAVA VM를 더욱 연구 개발하여 임베이드 리눅스기반에 PJAVA VM을 설치 가능하도록 연구하고 있다.

[참고문헌]

[1] Personal Java Application Environment Porting Guide: www.java.sun.com

[2] Venners : " Inside the Virtual Machine "

[3] Venners : " Inside the Virtual Machine Second EditonBill"

[4] Richerd M,Stallman & Roland McGrath : "GNU Make"

[5] Engel: " 자바 가상 머신 프로그래밍"