실시간 운영체제 iRTOS™ 상에서 KVM 메모리 관리 체계 설계 및 구현

백대현[0], 안희중, 성영락[†], 이철훈
충남대학교 컴퓨터공학과, [†] 국민대학교 전자정보통신공학부
(dhbaek[0], hjahn, chlee)@ce.cnu.ac.kr, [†] yeong@mail.kookmin.ac.kr

Design and Implementation of KVM Memory Management Facility on Real-Time Operating System, iRTOS™

Dae-Hyun Baek[0], Hee-Joong Ahn, Yeong Rak Seong[†], and Cheol-Hoon Lee
Dept. of Computer Engineering, Chungnam National Univ.
[†] School of Electrical Engineering, Kookmin Univ.

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 적용하기 위해 연구가 계속되고 있다. 임베디드 시스템이나 모바일 시스템과 같이 자원이 제한적인 다양한 기기들에는 자바 가상 머신을 경량화한 최소 크기의 자바 플랫폼에 대한 Configuration 인 CLDC(Connected, Limited Device Configuration)에서 정의하고 있는 K 가상 머신(K Virtual Machine: KVM)을 탑재한다. 본 논문에서는 실시간 운영체제로 iRTOS™을 사용하는 기기에서 KVM 을 탑재할 때 필요한 메모리 체계를 설계하고 구현한 내용을 설명한다.

1. 서론

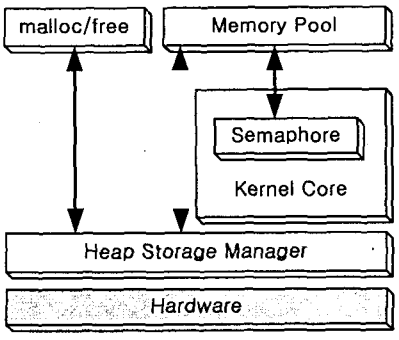
증가하는 임베디드 시스템에 대한 운영체제와 각각에 맞는 응용 프로그램들을 매번 다시 개발하는 것은 개발자에게는 매우 번거로운 일이다. 그러나 각각의 임베디드 시스템에 맞는 자바 가상 머신(Java Virtual Machine: JVM)이 탑재되어 있다면 하나의 응용 프로그램을 작성하더라도 모든 플랫폼에서 사용할 수 있게 된다[1]. 그러나 정보 가전기기와 같이 리소스가 제한적이며 네트워크 연결 능력이 있는 장치들에 JVM 을 탑제하기엔 JVM 은 너무 큰 용량을 차지하게 된다. 이를 해결하기 위해 작은 footprint Java™ Platform 을 개발할 필요성이 생기게 되었다. 그 결과 리소스가 제한적인 기기들에는 표준 자바 가상 머신에서 정의하고 있는 몇 가지 기능들을 제외하고, 필요한 최소한의 기능을 추가한 작은 크기의 자바 플랫폼에 대한 Configuration 인 CLDC (Connected, Limited Device Configuration)에서 정의하고 있는 KVM 을 이용하게 된다. 이러한 KVM 에서는 속도의 향상 뿐만아니라 리소스가 제한적이기 때문에 리소스 사용을 최소화 해야 할 필요가 있다. 본 논문에서는 실시간 운영체제 iRTOS™ 상에서 KVM 이 메모리를 효율적으로 관리할 수 있는 기법에 대해 기술한다.

본 논문에서는 2 장에서 실시간 운영체제 iRTOS™의 특성과 메모리 관리 체계에 대한 연구를, 3 장에서 KVM 메모리 관리 체계 설계 및 구현을, 4 장에서 가비지 콜렉션 구현을, 5 장에서 테스트 환경 및 결과를, 6 장에서 결론 및 향후 과제를 기술한다.

2. 관련 연구

2.1 iRTOS™ 특징 및 메모리 관리

iRTOS™는 실시간 운영체제의 핵심이라 할 수 있는 우선순위 기반의 멀티태스킹 환경 및 태스크간 통신할 수 있는 ITC(InterTask Communication) 환경을 지원한다. 그리고 효율적인 자원관리 및 태스크간 통신을 위해 세마포어, 메시지 큐, 메시지 메일박스를 제공한다[1].



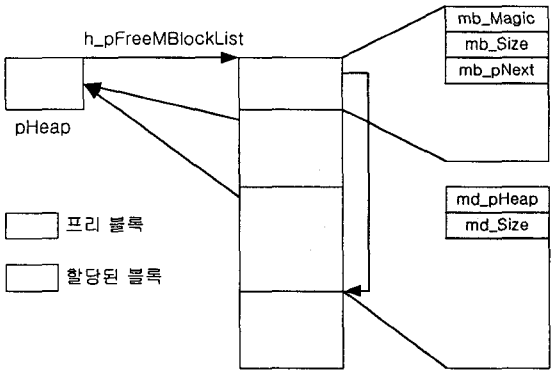
[그림 1] 메모리 관리

· 본 논문은 한국과학재단이 지정한 지역협력연구센터(ARC)인 충남대학교 소프트웨어 연구센터의 지원으로 수행된 과제의 결과입니다.

iRTOS 커널은 동적 메모리 관리를 위해 두 단계의 메모리 관리 기법을 제공한다. 하위 단계는 가변 크기의 메모리를 할당, 해제할 수 있는 힙 스토리지 메니저(Heap Storage Manager)이고, 상위 단계는 고정 크기의 메모리를 할당, 해제할 수 있는 메모리 풀이다. [그림 1]은 메모리 관리의 구성을 보여준다. 하드웨어상의 메모리는 힙스토리지 메니저에 의해 관리되며, 메모리 풀은 이미 생성된 힙 스토리지 메니저에서 시스템 초기화 시에 미리 할당 받아 여러 개의 고정크기의 버퍼로 관리한다.

2.2 힙 스토리지 메니저 (Heap Storage Manager)

[그림 2]는 힙 스토리지 메니저가 힙 영역의 메모리를 리스트로 관리하는 구조를 보여주고 있다. 힙 스토리지 메니저에서 할당되지 않은 프리 블록(Free Block)은 블록 앞에 12 바이트의 프리 블록을 관리하기 위한 구조체가 선언되어 있고, 이를 통해 프리 블록을 관리한다. 힙 스토리지 메니저에서 할당된 메모리는 사용자가 원하는 메모리 크기에 추가적으로 8 바이트의 구조체를 할당된 블록 앞에 선언하고, 이를 통해 할당된 블록을 관리한다. 할당된 블록에 선언된 구조체는 자신의 힙 스토리지 메니저와 할당된 크기를 보관하며, 메모리의 해제 시에 이 구조체를 사용하여 쉽게 메모리를 해제할 수 있다.

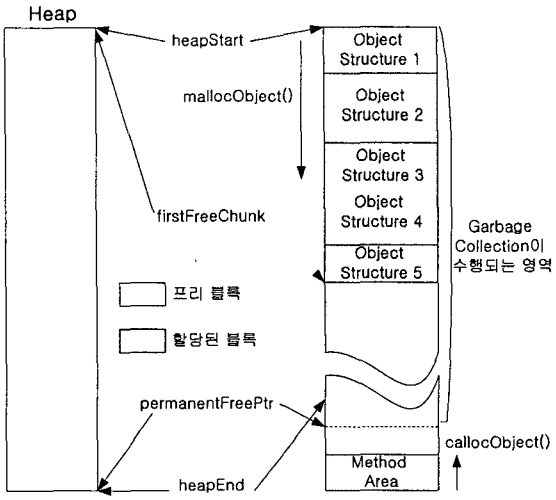


[그림 2] 힙 스로트리 메니저 구조

3. KVM 메모리 관리 체계 설계 및 구현

[그림 3]은 KVM 이 힙 메모리를 어떻게 관리하고 있는지 보여주고 있다. 실행중인 자바 애플리케이션에서 클래스 객체나 배열을 생성할 때 필요한 메모리 공간을 할당 받기 위해 mallocObject() 함수를 이용한다. mallocObject() 함수를 이용하여 힙 메모리를 할당 할 경우 firstFreeChunk 위치부터 시작하는 프리 리스트(Free List)에서 퍼스트 핏(Firtst Fit) 알고리즘에 따라 힙 메모리를 할당하고, 할당되지 않은 공간은 다시 프리 리스트에 추가한다. 프리 공간이 남아 있지 않다면 CHUNK 구조체를 프리 리스트에서 삭제한다.

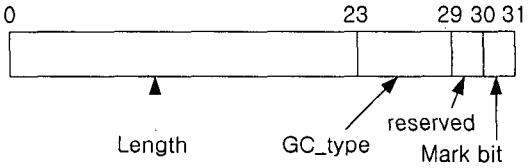
callocObject() 함수는 컨스턴트 풀 (Constant Pool)이나 필드 또는 메쏘드 정보 등과 같이 변하지 않는 클래스 정보를 저장하기 위한 공간을 할당 받을 때 사용한다. callocObject() 함수를 이용하여 힙 메모리를 할당 할 경우 요구하는 메모리 크기가 현재 사용가능한 메모리 공간 (permanentFreePtr ▯ heapEnd)보다 작을 경우 메모리 공간을 할당하고, 그렇지 않을 경우는 heapEnd 값을 재 설정한 다음에 메모리 공간을 할당하게 된다.



[그림 3] KVM 메모리 관리

3.1 Object Structure

오브젝트(Object) 구조는 오브젝트 헤더를 포함하여 클래스에 대한 포인터, 모니터와 해쉬코드에 대한 포인터, 클래스 객체에 필요한 데이터를 저장하기 위한 공간을 포함한다. [그림 4]는 오브젝트 구조를 보여주고 있다.



[그림 4] Object Header

-Length : 오브젝트 헤더 (1 byte)를 제외한 unsigned long(4 byte)을 하나의 크기로 가지는 길이를 나타낸다.

-GC_type : 가비지 콜렉션이 실행되는 동안 정확하게 오브젝트의 데이터 필드를 스캔하기 위해 사용되는 가비지 콜렉션 타입을 나타낸다.

-Mark bit : 가비지 콜렉션이 수행되는 도중에 오브젝트가 참조되고 있는지 마크하기 위한 비트이다.

3.2 Chunk Structure

힙 메모리 영역의 프리 블록에 대한 리스트를 관리하기 위한 데이터 구조이다. 처음에는 [그림 3]과 같

이 firstFreeChunk 포인터를 변경하며 힙 메모리를 할당한다. 더 이상 할당할 힙 메모리가 없을 경우 마크-회수 방법을 이용하여 가비지 콜렉션을 수행한 후, [그림 5-a]와 같이 프리 메모리 공간을 프리 CHUNK 리스트로 관리한다. 이 프리 리스트에서 퍼스트 핏 알고리즘에 따라 힙 메모리를 할당한다. [표 1]의 자료구조의 chunkStruct 는 연속된 메모리의 크기 및 다음 Chunk 를 가리키는 포인터로 구성된다.

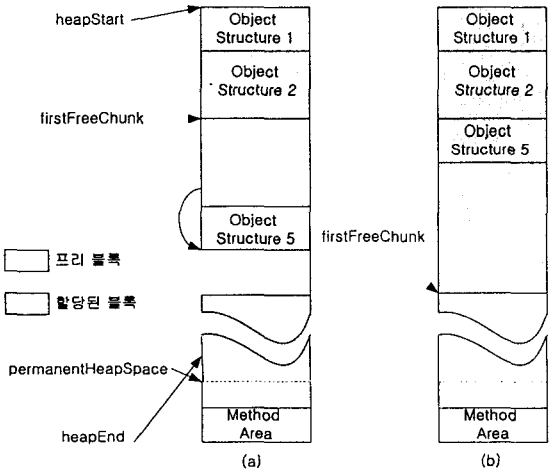
[표 1] CHUNK 구조

```
typedef struct chunkStruct* CHUNK;
struct chunkStruct {
    long    size;
    CHUNK next;
};
```

4. 가비지 콜렉션 구현

마크-회수 압축 방법을 이용하여 가비지 콜렉션을 구현했다. 객체를 할당하는데 필요한 메모리 공간은 mallocObject() 함수나 callocObject() 함수를 이용하여 할당한다. 그러나 더 이상 메모리를 할당하지 못할 경우 마크-회수 방법을 이용하여 가비지 콜렉션을 수행한다. 이 과정을 수행한 뒤에도 메모리를 할당하지 못한다면, 압축 방법을 이용하여 가비지 콜렉션을 수행하여 힙 메모리를 할당한다. 그러나 가비지 콜렉션이 수행된 뒤에도 할당할 메모리가 부족하다면 메모리가 부족하다는 메시지를 생성한다.

[그림 5-a] 마크-회수 방법을 이용하여 가비지 콜렉션을 수행한 후의 상태이고, [그림 5-b]는 압축 방법을 이용하여 가비지 콜렉션을 수행한 후의 상태이다.



[그림 5] 가비지 콜렉션 후의 메모리 상태

4.1 마크 과정

[그림 4]의 Mark bit 를 이용한다. 마크 알고리즘의 과정은 루트로부터 검사를 시작해서 마크되지 않은 객체 대해 그 객체가 참조중인지 판단하여 참조중인 객체에 대한 bit 를 ' marked' 하고, GC_type 에 따라 그 객체에 대해 참조중인 데이터를 모두 마크한다.

4.2 회수 과정

회수 알고리즘의 과정은 heapStart 로부터 검사를 시작하여 연속된 마크되지 않은 메모리 영역을 하나의 CHUNK 구조체를 생성하여 참조하고, 그 구조체를 프리 CHUNK 리스트에 추가하여 다른 객체에 의해 힙 메모리를 할당받을 수 있게 한다.

4.3 압축과정

마크-회수 과정을 수행한 후에도 할당해야 할 메모리 공간이 부족한 경우 수행된다. 압축 과정은 현재 참조되고 있는 모든 객체에 대한 포인터와 데이터에 대한 포인터를 재배치 시키고, 프리 CHUNK 리스트도 초기화 하여 firstFreeChunk 포인터만 남겨지게 된다. 압축 과정이 끝나면 [그림 5-b]와 같이 힙의 아래쪽 영역에 사용 가능한 메모리 공간이 존재하게 된다.

5. 테스트환경 및 결과

본 논문의 테스트 환경은 S3C2400X 보드에서 실시간 운영체제로는 iRTOS™를 사용했고, ARM STD v2.51 을 사용하여 컴파일을 했다. 그리고 디버그를 위해서 ARM MultiICE를 사용하였다. 아직 KVM이 완전히 구현되지 않았기 때문에 메모리 할당과 해제, 가비지 콜렉션에 대한 함수를 강제로 호출하여 메로리가 정확히 할당되고 해제되는지 확인하였다.

6. 결론 및 향후 과제

본 논문에서는 실시간 운영체제인 iRTOS™ 에서 힙 스토리지 메니저를 사용하여 KVM 에서 사용할 메모리를 할당 받고, 그 메모리를 어떻게 관리 할 것인가에 대한 메모리 관리 체계 설계 및 구현한 내용을 기술했다. 향후 연구과제로 힙 스토리지 메니저를 사용할 경우 실시간 운영체제 iRTOS™ 에서는 시간 결정성에 대한 문제가 발생할 경우가 생길 수 있다. 이를 해결하기 위해 CLDC 에서 제공되는 기본 클래스에 대한 로마이징(ROMing) 기법에 대한 연구가 계속 되어야 한다.

7. 참고문헌

[1] http://www.inestech.com
[2] Sun Microsystems, *JSR-30 J2ME Connected, Limited Device Configuration*
[3] Bill Venners, *Inside the Java Virtual Machine*, 1999
[4] Tim Lindholm, *The Java™ Virtual Machine Specification Second Edition*, 1999
[5] Sun Microsystems, *KVM Porting Guide, CLDC, Version 1.0.4*
[6] Sun Microsystems, *Java™ 2 Platform Micro Edition Technology for Creating Mobile Devices*