실시간 자바가상기계를 위한 효율적인 메모리 할당 기법†

김세영*, 양희재
경성대학교 컴퓨터공학과
e-mail: *sykim@conet.ks.ac.kr, hjyang@star.ks.ac.kr

An Efficient Memory Allocation Scheme for Real-Time Java Virtual Machine

Seyoung Kim*, Heejae Yang
Dept. of Computer Engineering, Kyungsung University

요 약

내장형 실시간 시스템에서는 메모리관리시스템의 구현에 있어 메모리 단편화와 시간 결정성의 문제를 해결하기 위한 방법 중의 하나로 고정크기의 메모리를 할당하는 기법이 사용된다. 내장형 자바가상기계에서도 객체를 관리하는 메모리 구조인 힙에 이를 적용하여 활용할 수 있으며 구현된 예로는 simpleRTJ 가 있다. 고정크기 메모리 할당 기법은 구현이 간단하기 때문에 시스템이 단순해지고 실행시 오버헤드의 발생이 줄어드는 장점이 있다. 하지만 실제 구현에 있어서 구현의 단순화에 초점을 맞추어 프리 리스트가 구성되지 않아 메모리할당에 있어 예상할 수 없는 시간 지연이 있을 수 있으며 이는 실시간성의 보장에 치명적인 결함이 될 수 있다. 또한 배열과 문자열의 구성에 있어 실제 크기를 그대로 사용함으로써 고정크기 할당방식의 장점을 완전히 이용되지 못하고 있다. 본 논문에서는 실질적인 고정크기 할당방식의 장점을 최대한으로 이용하기 위해 객체와 메소드의 구조, 그리고 힙을 재구성하였다.

1. 서론

내장형 시스템의 경우 실시간 특성을 요구하는 경우가 많지만 기존의 내장형 자바가상기계의 구현들은 대부분 자바의 기능을 내장형 시스템에 맞게 축소하였을 뿐 스케줄링이나 메모리 관리 측면에서 실시간 특성에 대한 고려가 되어있지 않다. 때문에 자바에 실시간 특성을 지원하기 위해 RTSJ(Real-Time Specification for Java)[1] 가 제안되었다. 특히 메모리 관리에 있어 RTSJ는 scoped memory[2] 기법을 사용하여 동적인 메모리 관리에서의 예측할 수 없는 지연시간을 줄이고 있다. 하지만 구현이 복잡하고 요구되는 메모리를 예측하여 미리 할당하여야 하기 때문에 낮은 프로세서 성능과 적은 메모리가 요구되는 임베디드 시스템에 그대로 적용하기에는 무리가 있다.

일반적인 내장형 실시간 시스템의 경우 메모리 관리를 위해 고정크기 메모리 할당 방식을 사용하는 경우가 있으며 구현으로는 microC/OS[3], iRTOS[4] 등이 있다. 고정크기 메모리 할당방식은 구현이 쉽고 메모리의 할당과 해제가 빠르기 때문에 내장형 실시간 시스템의 메모리 관리를 위한 좋은 대안이라 할 수 있으며 고정크기 메모리 할당방식을 사용하는 내장형 자바가상기계로는 simpleRTJ[5] 가 있다.

simpleRTJ의 고정크기 할당 기법은 클래스 파일들로부터 ROMizer[6]에 의해 만들어지는 롬 이미지를 통해 알게 되는 가장 큰 객체의 크기와 메소드 프레임(method frame)의 크기를 이용하여 객체와 메소드 프레임을 할당하게 된다. 하지만 simpleRTJ의 경우 메모리 할당에 있어 선형적인 메모리 탐색에 의해 비어있는 메모리를 검사하기 때문에 실시간 시스템에 적용하기 어렵다. 또한 배열을 위한 메모리 할당에서는 실제 배열의 크기가 가장 큰 객체의 크기보다 클 경우 배열의 크기만큼의 메모리 블록을 할당하기 때문에 단편화의 문제가 발생할 수 있다.

본 논문에서는 기존의 simpleRTJ의 메모리 할당 알고리즘을 수정하여 실시간 시스템의 시간 결정성을 보장하기 위한 추가적인 데이터 구조와 알고리즘에 관해 기술한다.

본 논문의 구성은 2장에서 논문의 기반 시스템이

† 이 논문은 2003년도 정보통신부 지원 정보통신기초기술연구지원 사업에 의해 연구되었음.

되는 simpleRTJ의 ROMizer와 메모리 구성에 관해 알아보고, 3장에서 메소드 프레임, 4장에서 객체와 배열의 메모리 할당 알고리즘에 관해 설명하며 5장에서는 이러한 기법의 장단점과 함께 기대되는 성능에 관해 분석해보며 6장에서 결론을 맺는다.

2. 배경연구

2.1 ROMizer

내장형 시스템의 경우 디스크와 같은 보조기억장치가 없는 경우가 대부분으로 필요한 클래스 파일들은 별도의 호스트 컴퓨터에서 개발되어 내장형 시스템의 ROM에 적재되는 방식을 취하고 있다.

ROM에 클래스 파일이 적재될 때에는 프로그램을 구성하는 클래스 파일들의 전체 크기를 줄이고 접근 속도를 높이기 위해 원래의 클래스 파일 구조를 변경하여 하나의 ROM image 파일로 만들며 이러한 작업을 수행하는 프로그램을 ROMizer라 한다.

ROMizer가 프로그램을 구성하는 클래스 파일들을 ROM image로 변환하기 위해 분석하는 단계에서 객체와 메소드 프레임의 구성요소가 되는 클래스파일의 필드와 메소드 정보를 분석하기 때문에 프로그램 동작에서 요구되는 객체와 메소드 프레임 각각의 가장 큰 크기를 알 수 있게된다.

2.2 힙(heap) 메모리 구성

simpleRTJ의 힙 메모리 구성은 그림 1과 같다. 자바 가상기계에서 사용하는 동적인 메모리 할당 및 해제가 필요한 객체, 배열 그리고 스택 프레임을 위한 영역이 object_start와 heap_end에 의해 구성된다.

메소드 프레임과 객체는 구조와 크기 그리고 사용되는 성격이 다르기 때문에 각각 heap_end (high memory address)와 objdata_start (low memory address)의 양 끝단에서부터 서로 분리되어 할당을

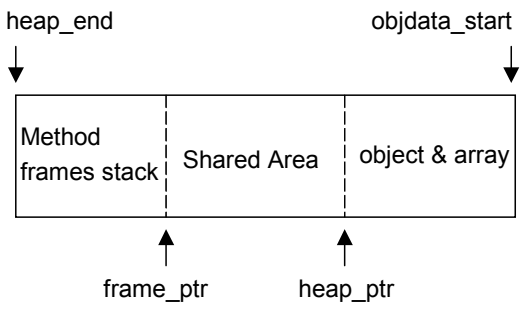

그림 1 simpleRTJ 힙(heap) 메모리 구성

```
Input: Frame_Pointer
Output: boolean FrameAllocOK
begin
  while true
    Mem ←heap_end - frame_size
    while Mem >= frame_ptr
      if Mem = FREE then
        frame_Pointer ← Mem
        FrameAllocOK ← true

      Mem ← Mem - frame_size

    if Mem < frame_ptr then
      if Mem <= heap_ptr then
        if GarbageCollector() = false then
          frameAllocOK ← false
        else
          continue

      frame_ptr ← Mem
      FrameAllocOK← true

end
```

그림 2 simpleRTJ 메소드 프레임 할당 알고리즘

시도하게 된다. frame_ptr과 heap_ptr은 각각 메소드 프레임과 객체의 할당되는 방향으로 가장 마지막 요소를 가리키게 되어 두 영역이 서로 충돌하는 일이 없도록 경계역할을 한다.

3 메소드 프레임 프리 리스트

기존의 simpleRTJ의 경우 새로운 메소드가 호출될 때는 먼저 그림 1의 heap_end에서부터 고정크기의 프레임 사이즈(frame_size)를 뺀 후 그 프레임이 비어있는지를 검사하게 된다. 만약 비어있다면 그 메모리를 할당해 주게 되며 비어있지 않다면 계속해서 프레임 사이즈만큼 뺀 후 비어있는지를 heap_ptr 포인터까지 검사하게 된다. heap_ptr 까지 검사해도 비어있는 공간이 나타나지 않으면 가비지 컬렉션을 이용하여 빈 공간을 얻을 수 있도록 시도하게 된다. 여기서 프레임 사이즈만큼 빼면서 검사하는 것은 프레임의 할당이 높은 메모리 주소 번지에서 낮은 번지로 순차적으로 이루어지기 때문이다. 그림 2는 이러한 알고리즘의 의사코드이다.

그림 2와 같은 방식의 시간 복잡도는 $O(n)$ 이며 이는 비어있는 메모리 슬롯을 찾기 위한 선형 탐색에 따른 것이다. 이 경우 비어있는 공간을 찾는데 있어 예측할 수 없는 시간이 필요하기 때문에 실시간 시스템에서는 문제가 된다. 따라서 시간 결정성을 보장하기 위해 그림 3과 같이 자료구조를 추가하였다.

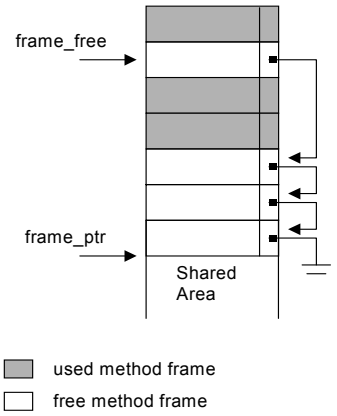

그림 3 메소드 프레임의 프리 리스트 구조

메소드 프레임 프리 리스트(free list)를 구성하기 위해 그림 3과 같이 frame_free 포인터가 추가되었으며 사용되지 않는 프레임들은 단일 연결 리스트로 연결되어있다. 그리고 리스트의 마지막은 NULL을 가리키고 있다. 이는 비어있는 슬롯이 없을 경우 frame_ptr에서부터 새로운 슬롯을 생성하기 위함이다. 프레임의 할당과 해제 작업에서 프리 리스트를 지속적으로 유지하기 위해 추가적인 작업이 이루어지게 된다. 하지만 이 경우에도 단순한 포인터 할당 연산만으로도 리스트 갱신이 가능하기 때문에 시스템의 전체 성능에는 큰 영향은 미치지 않게 된다. 실제 알고리즘은 그림 4에 나타난 방식으로 이루어진다.

그림 4에 나타난 알고리즘과 같이 새로운 메소드

```
Input: Frame_Pointer
Output: boolean FrameAllocOK
begin
    while true
        if frame_free != NIL then
            Frame_Pointer ⟵ frame_free
            Frame_free ⟵ frame_free.Next
            FrameAllocOK ⟵ true

        else if frame_ptr – frame_size > heap_ptr then
            frame_Pointer ⟵ frame_ptr – frame_size
            frame_ptr ⟵ frame_ptr – frame_size
            frameAllocOK ⟵ true

        else
            if GarbageCollector() = false then
                frameAllocOK ⟵ false

            continue
end
```

그림 4 프리 리스트를 이용한 프레임 할당 알고리즘

가 호출되면 frame_free 포인터에 의해 유지되는 리스트에서 메모리 블록을 할당받게 된다. 이때 프리 리스트의 블록들은 모두 동일한 크기를 가지기 때문에 first fit 할당방식을 이용하더라도 리스트에 비어있는 블록이 있는 경우 한번의 검색으로 메모리를 할당할 수 있게 되어 $O(1)$로 나타낼 수 있다. 만약 리스트에 메모리 블록이 없을 경우 그림1의 shared area에서 프레임 크기만큼의 메모리를 할당받아 사용하게 된다.

그림 4의 의사코드에 나타난 while 문은 할당할 메모리가 없을 경우 가바지 콜렉션을 수행한 후 다시 메모리 할당을 시도하기 위한 코드이다.

4. 객체 프리 리스트

4.1 객체 프리 리스트

simpleRTJ에서 객체를 위한 메모리 할당의 경우 메소드 프레임의 할당 방식과 비슷한 방법이 사용되며 할당되는 메모리 블록은 가장 큰 객체의 크기가 사용된다. 객체 할당 방식이 메소드 프레임의 할당 방식과 다른 점은 우선 단일 고정크기로 할당 하지만 배열의 경우 배열의 크기가 가장 큰 객체의 크기보다 큰 경우 배열의 실제 크기만큼 메모리에 할당하기 때문에 비어있는 메모리 블록을 탐색하는데 있어서 비어있는지 여부와 함께 사이즈 정보도 같이 검사하는 것이다. 또한 메소드 프레임의 경우 메소드의 호출과 복귀과정에서 메모리 블록의 사용여부를 바로 알 수 있기 때문에 메소드간의 컨텍스트 스위칭이 일어나는 시점에서 메모리 블록의 할당과 해제가 일어나지만 객체의 경우 할당은 객체의 생성시점에서 그리고 해제는 가비지 콜렉션이 호출되는 시점에서 수행된다.

객체의 메모리 할당방식은 메소드 프레임 할당방식과 마찬가지로 메모리를 그림 1의 objdata_start에서부터 선형탐색으로 이루어지기 때문에 메소드 프레임의 경우와 마찬가지로 $O(n)$의 시간 복잡도를 가지게 된다.

simpleRTJ에서 사용되는 객체의 메모리 할당 방식을 개선하기 위하여 메소드 할당방식의 개선시 사용한 프리 리스트 방식을 사용하게 된다. 이 경우 객체를 위해 할당하는 메모리 블록의 크기는 가장 큰 객체의 크기로 통일되기 때문에 first fit 할당 방식으로도 한번의 탐색으로 메모리를 할당할 수 있게 되며 $O(1)$로 나타낼 수 있다.

여기서 주의할 점은 배열의 경우 객체할당을 위한 메모리 블록과 크기가 다르다는 점인데 이를 위한 배열의 재구성은 4.2절에서 설명하였다.

4.2 배열 구성

simpleRTJ의 경우 배열의 크기를 가장 큰 객체 크기와 비교하여 배열이 작다면 객체의 크기로 할당하고, 그렇지 않다면 배열의 크기만큼의 메모리 블록을 할당하도록 되어있다. 하지만 이러한 할당 방식은 결국 가비지 컬렉션이후 내부 단편화와 외부 단편화의 원인이 되며 단일 고정크기 할당방식의 장점을 제대로 살리지 못하게 된다.

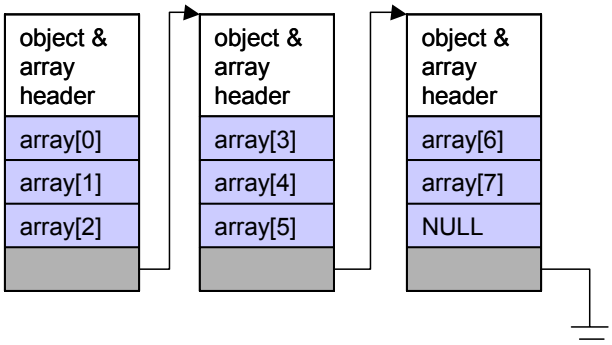

그림 5 배열 리스트 구성

그림 5는 배열을 위한 메모리 할당의 수정된 방식을 나타내고 있다. 가장 큰 객체 크기와 비교하여 배열이 작다면 객체의 크기로 할당하는 것은 기존의 방식과 동일하게 적용한다. 하지만 배열의 크기가 크다면 가장 큰 객체의 크기로 할당하며 할당한 메모리의 배열 마지막 슬롯을 포인터로 이용하여 다음 비어있는 메모리 블록을 가리키게 하고 다음 메모리 블록에 연속하여 배열정보를 저장하게 된다. 이러한 방법을 모든 배열의 요소를 초기화 할 때까지 계속하게 된다. 결국 모든 힙의 정보들이 가장 큰 객체의 크기 단위로 할당되어 외부 단편화 문제가 없어지게 된다.

4 성능 예측 및 현재 연구 진행상황

일반적인 동적인 메모리 할당에서의 가변 크기의 메모리할당의 경우 외부단편화 현상과 함께 메모리 할당을 위한 탐색에 예측할 수 없는 시간이 소요되기 때문에 특히 실시간성이 요구되는 시스템에서는 적용하기 어렵다. simpleRTJ의 고정크기의 메모리 블록을 이용한 메모리 할당의 경우 메소드 프레임과 객체가 각각 동일한 크기의 메모리 블록을 가지기 때문에 비어있는 메모리의 경우 블록의 크기검사를 하지 않고도 바로 할당할 수 있다. 하지만 이러한 경우에 있어서도 비어있는 메모리 블록을 검사가 선형적으로 이루어지기 때문에 $O(n)$의 시간이 요구되게 된다. 이러한 문제점을 해결하기 위해 프리 리스트를 추가하고 알고리즘을 수정하여 일정한 시간(constant time)내에 메모리를 할당함으로써 실시간 시스템의 시간 결정성을 만족시킬 수 있도록 하였다.

본 논문에서는 알고리즘을 개선하여 성능을 분석하였으며 실제 실험을 통하여 리스트를 유지하는 비용과 시간 결정성에 관한 성능 측정을 진행 중에 있다.

5 결론

본 논문에서는 내장형 자바가상기계인 simpleRTJ의 메모리 할당 알고리즘을 실시간 시스템에 적용하기 위해 개선하였고 이에 대한 분석을 하였다. 실시간 시스템을 위한 메모리 관리를 위해서는 메모리 할당뿐만 아니라 해제에 있어서도 실시간성을 보장하는 장치가 필요하며 이에 대한 연구가 진행되어야 할 것이다.

참고문헌

[1] Bollella, Gosling, Brosgol, Dibble, Furr, Hardin, and Turnbull; The Real-Time Specification for Java, Addison-Wesley, 2000.

[2] Greg Bollella and Kirk Reinholtz, "Scoped Memory", Fifth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing, April 2002

[3] Jean J. Labrosse, MicroC/OS-II: The Real Time Kernel, Second Edition, CMP BOOKS, 2002

[4] 박희상, 안희중, 김용희, 이철훈, “실시간 운영체제 iRTOS를 위한 메모리 관리 체계 설계 및 구현”, 정보과학회 춘계학술대회, pp.0058~0060, 2002

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

[6] 양희재, “simpleRTJ 임베디드 자바가상기계의 ROMizer 분석 연구“, 정보처리학회논문지 A 제10-A권, pp.397-404, 2003.