응용프로그램 특성을 고려한 모바일 플랫폼의 동적 메모리 관리기법

유 용 덕† · 박 상 현†† · 최 훈†††

요 약

모바일 다비이스는 시스템 자원이 매우 제한적이기 때문에 응용프로그램을 실행시키기 위해서는 자원들을 효율적으로 관리하여야 한다. 특히 제한적인 메모리에 대한 동적 관리기법은 모바일 디바이스의 운영체제 및 플랫폼에서 매우 중요한 요소이다. 그러나 기존 동적 메모리 관리기법들은 응용프로그램의 실행 스타일과 사용되는 객체의 라이프 타임(life time), 객체 종류 및 종류 분포를 고려하지 않음으로써 효율적으로 메모리를 관리할 수 없으며, 응용프로그램의 실행 속도를 저하시킨다. 따라서 본 논문에서는 모바일 응용프로그램의 실행 특성을 분석하고, 분석한 결과를 토대로 모바일 디바이스용 응용프로그램의 실행 시 메모리를 절약하고, 실행 속도를 향상시키는 새로운 동적 메모리관리기법을 제안 및 개발하였다. 기존 동적 메모리 관리 모듈과의 응용프로그램 실행 속도를 비교한 결과, 제안한 동적 메모리 관리기법은 테스트용 응용프로그램을 실행할 때 링크드 리스트[11]에 비하여 6.5배, Doug. Lea 메모리 관리기법[13]에 비하여 2.5배, Brent 메모리 관리기법[15]에 비하여 10.5배 빠른 실행 속도를 보였다.

키워드 : 모바일 디바이스, 런타임 라이브러리, 동적 메모리 관리기법, 모바일 응용프로그램 특성, 모바일 플랫폼, 위피, 위피 에뮬레이터

Dynamic storage management for mobile platform based on the characteristics of mobile applications

Yong-Duck You† · Sang-Hyun Park†† · Hoon Choi†††

ABSTRACT

Performance of the mobile devices greatly depends on the efficient resource management because they are usually resource-restricted. In particular, the dynamic storage allocation algorithmis very important part of the mobile device's operating system and OS-like software platfom. The existing dynamic storage allocation algorithms did not consider application's execution style and the type, life-time, and characteristics of memory objects that the application uses. Those algorithms, as a result, could not manage memory efficiently. Therefore, this paper analyzes the mobile application's execution characteristics and proposes anew dynamic storage allocation algorithm which saves the memory space and improves mobile application's execution speed. The test result shows that the proposed algorithm works 6.5 times faster than the linked-list algorithm[11], 2.5 times faster better than the Doug. Lea's algorithm[12] and 10.5 times faster than the Brent algorithm[14].

Key Words : Mobile Device, Runtime Library, Dynamic Storage Allocation, Mobile Application Characteristic, Mobile Platform, WIPI(Wireless Internet Platform for Interoperability), WIPI Emulator

1. 서 론

최근 정보통신 및 반도체 기술이 급격히 발전함에 따라 다양한 모바일 디바이스(mobile device)들이 개발되고 있다. 대부분의 모바일 디바이스는 시스템 자원이 매우 제한적이기 때문에 응용프로그램을 실행시키기 위해서는 자원들을 효율적으로 관리하여야 한다. 특히 제한적인 메모리에 대한 동적 관리기법은 모바일 디바이스의 운영체제 및 플랫폼에서 매우 중요한 요소로서 자바 가상머신 및 런타임 라이브러리 영역에서 오래 전부터 폭넓게 연구되고 있다[1-5].

자바 가상머신을 통한 동적 메모리 관리기법은 응용프로그램 개발자에게 자동적인 메모리 관리 기능을 제공함으로써 응용프로그램을 개발할 때 메모리 관리에 따른 부담을 덜게 해준다[6]. 그러나 자바 가상머신은 자바 바이트코드 실행을 위해 높은 사양의 메모리와 프로세서 수행 능력을 요구하기 때문에 모바일 디바이스와 같은 자원 제한적인 실

† 준 회 원: 충남대학교 컴퓨터공학과 분산이동컴퓨팅연구실 박사과정
†† 정 회 원: 국가보안기술연구소 선임연구원
††† 종신회원: 충남대학교 전기정보통신공학부 부교수
논문접수: 2006년 9월 6일, 심사완료 : 2006년 10월 24일

행환경에서는 성능이 저하된다.

이와 비교하여 C 언어 기반 런타임 라이브러리에서 응용프로그램은 바이너리 이미지로 실행되므로 자바 가상머신보다 실행 속도가 빠르다. 그러나 응용프로그램을 실행할 때, 메모리 할당/해제가 빈번히, 반복적으로 일어남에 따라 메모리는 조각나기 쉽고, 조각난 메모리로 인하여 새롭게 메모리를 할당할 때 미사용 메모리 블록을 검색하기 위하여 큰 비용이 소요되며, 사용 중인 메모리를 해제할 때도 매번 인접한 메모리 블록의 사용 유무를 검사하여 미사용 메모리간의 병합을 수행해야 하기 때문에 많은 오버헤드가 따른다[7].

따라서 위와 같은 문제점들은 해결하기 위하여 다양한 동적 메모리 관리기법들이 개발되었다[8, 9]. 그러나 현재까지 개발된 동적 메모리 관리기법들은 일반적으로 데스크톱 이상의 기기에서 실행될 때를 고려하여 개발되었으며, 실행되는 응용프로그램의 특성을 고려하고 있지 않다. 모바일 디바이스와 같은 자원제약적인 실행환경에서 응용프로그램이 실행될 때 발생되는 오버헤드는 응용프로그램의 실행 특성인 실행 스타일과 사용되는 객체의 라이프 타임(life time), 객체 종류 및 종류 분포와 관련이 높다[10]. 따라서 실행되는 응용프로그램의 특성을 고려하지 않은 메모리 관리기법은 메모리 할당하거나 해제할 때, 메모리에 대한 효율적인 관리 기능을 제공하지 못한다.

본 논문에서는 자원제약적인 실행환경에서의 응용프로그램의 실행 특성에 대하여분석하였다. 분석한 결과를 토대로 모바일 디바이스용 응용프로그램의 실행을 위한 새로운 동적 메모리 관리기법을 제안하고 이를 적용한 동적 메모리 관리 모듈을 개발하였으며, 기존 동적 메모리 관리 모듈과의 성능 분석을 수행하였다.

기존 동적 메모리 관리 모듈과의 응용프로그램 실행 속도를 비교한 결과, 제안한 동적 메모리 관리기법은 테스트용 응용프로그램을 실행할 때 링크드 리스트[11]에 비하여 6.5배, Doug. Lea 메모리 관리기법[13]에 비하여 2.5배, Brent 메모리 관리기법[15]에 비하여 10.5배 빠른 실행 속도를 보였다.

본 논문의 구성은 다음과 같다. 2장에서는 기존 동적 메모리 관리기법에 대하여 기술하고, 3장에서는 모바일 디바이스용 응용프로그램의 특성에 대하여 기술한다. 4장에서는 본 연구에서 제안하는 새로운 동적 메모리 관리 알고리즘에 대하여 기술한다. 5장에서는 개발한 동적 메모리 관리 모듈을 탑재한 CNU 위피 에뮬레이터와 테스트용 응용프로그램을 이용하여 기존의 동적 메모리 관리기법과의 성능 비교 및 분석에 대하여 기술한다. 끝으로 6장에서 결론을 맺는다.

2. 동적 메모리 관리(Dynamic Storage Allocation) 기법

플랫폼이 시작된 후 응용프로그램이 계속적으로 수행되어 메모리의 할당 및 해제가 반복적으로 일어나면, 메모리의 단편화(fragmentation)가 발생된다. 메모리의 단편화로 인해 전체 사용 가능한 메모리의 크기가 충분함에도 불구하고 큰 메모리 블록의 할당이 어려워지며, 적절한 미사용 메모리 블록을 찾기 위한 시간 비용의 증가를 초래하여 메모리를 할당할 때 시간 비결정성이 발생한다. 그리고 할당되는 메모리의 크기 및 미할당 메모리의 크기가 작을 경우 오히려 메모리 관리를 위한 오버헤드가 상대적으로 증가하게 된다. 따라서 동적 메모리 관리기법은 메모리 단편화의 최소화, 메모리 할당에 있어서의 시간 결정성 보장 및 메모리 관리를 위한 오버헤드의 최소화를 위하여 다양하게 연구되고 있다.

2.1 링크드 리스트 관리기법

링크드 리스트 관리기법은 전체 메모리 상에서 미사용 메모리 블록의 리스트를 유지하고, 메모리 할당 요청이 있을 때마다 미사용 메모리 블록 리스트를 검색하여 요청된 크기의 메모리를 할당할 수 있는 미사용 메모리 블록을 찾아준다. 메모리 해제 시 해제되는 메모리 블록은 미사용 메모리 리스트에 연결되며, 메모리 상에서 인접한 메모리 블록의 사용 여부를 확인하여 미사용일 경우 두 인접 미사용 메모리 블록을 병합하여 보다 큰 미사용 메모리 블록을 만든다. 이 방법은 메모리 단편화 발생 시 오버헤드가 크다. 메모리 할당 시 적절한 크기의 미사용 메모리 블록을 검색하는데 많은 시간이 소요되며, 사용 중이던 메모리를 해제할 때 인접한 메모리 블록의 사용 유무에 관계없이 미사용 메모리간 합병을 위하여 매번 리스트 전체를 검사하여야 하는 오버헤드가 있다[11].

2.2 Half Fit 알고리즘

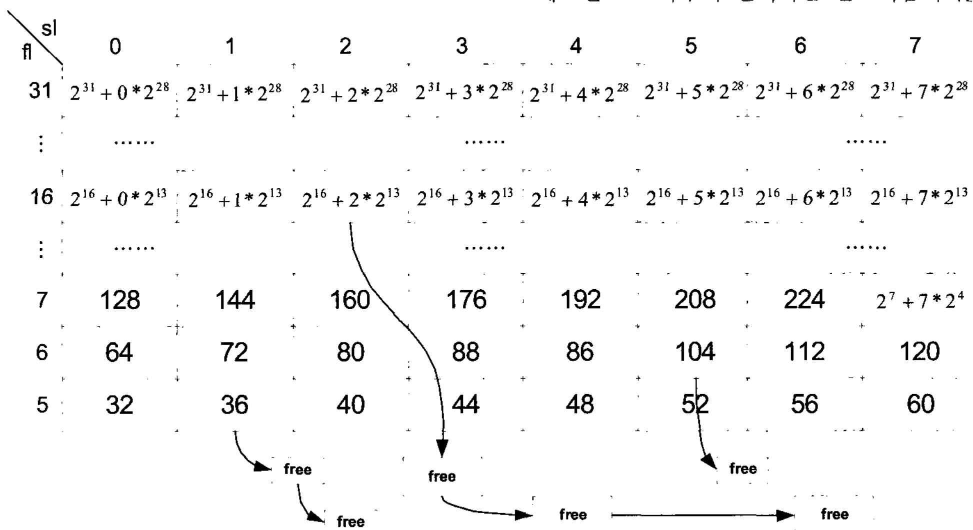
Half Fit 알고리즘은 T. Ogasawara[12]에 의해 제안된 알고리즘으로서 미사용 메모리 관리에 있어 메모리 블록의 크기에 따라 세분화된 여러 개의 리스트로 관리한다. 각 리스트가 관리하는 미사용 메모리의 크기는 2의 배수 형태로 유지되며, 리스트의 인덱스 값이 'i' 일 경우 해당 리스트에서 관리하는 미사용 메모리 블록의 크기는 2^i 이상 2^{i+1} 미만이다. 따라서 응용프로그램의 실행 시 메모리 블록을 인덱스 값 i인 리스트에서 할당할 경우, 할당할 수 있는 메모리 블록의 크기는 2^{i-1} 이상 2^i 이하의 크기이며, 할당되고 남은 1/2 크기 이상의 미사용 영역은 다시 크기에 맞는 리스트에 재연결된다. 따라서 할당에 소요되는 시간 비용이 O(1)로 매우 적다. 사용 중이던 메모리를 해제할 때 링크드 리스트 메모리 관리기법과 같이 인접 메모리 블록의 사용 유무를 파악하고 병합하여 큰 미사용 메모리를 만들며 이를 해당 리스트에 재연결한다. Half Fit 알고리즘의 단점은 초기에 설정된 'i' 값에 따라 결정되는 최대 미할당 메모리 크기보다 큰 메모리를 할당할 수 없으며, 특정 크기 이상의 메모리 할당이 메모리 할당 전체에 있어 많은 비율을 차지할 경우 메모리 압축 기능을 제공하지 않음에 따라 전체 메모리 상에는 미할당 메모리가 있음에도 불구하고 메모리를 할당할 수 없는 경우가 발생할 수 있다. 또한 사용 중인 메모리의 해제 시 인접한 메모리 블록 미사용 유무에 관계없이 미사용 메모리간 합병을 위하여 매번 검사하여야 하는 오버헤드가 있다.

2.3 Doug. Lea의 메모리 관리기법

Doug. Lea 메모리 관리기법[13]은 현재 널리 쓰이고 있는 방법으로서 할당하고자 하는 메모리의 크기에 따라 각각 다른 메모리 관리기법을 적용한다. Doug. Lea 메모리 관리기법은 전체 메모리 영역을 두 영역으로 구분하고, 한 영역은 8byte 크기의 메모리 블록으로 구성된 메모리 풀을 구성하며, 다른 하나의 영역은 가변 크기의 메모리 블록들을 링크드 리스트 기법으로 관리한다. 메모리 풀 영역은 16byte에서 512byte 크기의 메모리를 할당할 경우 8byte 단위로 요청하는 메모리의 크기에 맞춰 할당하며, 링크드 리스트 관리 영역은 512byte 크기 이상의 메모리 블록을 Best-Fit 방식으로 할당한다. Doug. Lea 알고리즘의 단점은 응용프로그램을 실행할 때 중간 크기(512 byte) 이상의 메모리 할당이 메모리 할당 전체에 있어 많은 비율을 차지할 경우 성능이 저하된다는 점이다[13]. 또한 512byte 크기 이상의 메모리를 할당하는 영역은 Best-Fit 방식의 링크드 리스트 관리기법을 사용함에 따라 메모리의 사용 효율을 높일 수 있으나 적절한 크기의 미사용 메모리 블록을 찾기 위한 비용이 크며, 메모리가 단편화될 경우 기존의 링크드 리스트 관리기법과 같은 단점들을 갖게된다.

2.4 TLSF(Two Level Segregated Fit) 알고리즘

TLSF 알고리즘은 미사용 메모리 블록을 두 단계로 나누어 관리한다[14]. 첫 번째 단계는 관리하는 미사용 메모리 블록의 크기(8, 16, 32, 64, 128, etc)에 따라 클래스로 구분하며, 두 번째 단계는 각 클래스에서 유지하는 미사용 메모리 블록들을 다시 세분화된 크기에 따라 별도의 리스트로 관리한다([그림 1]).



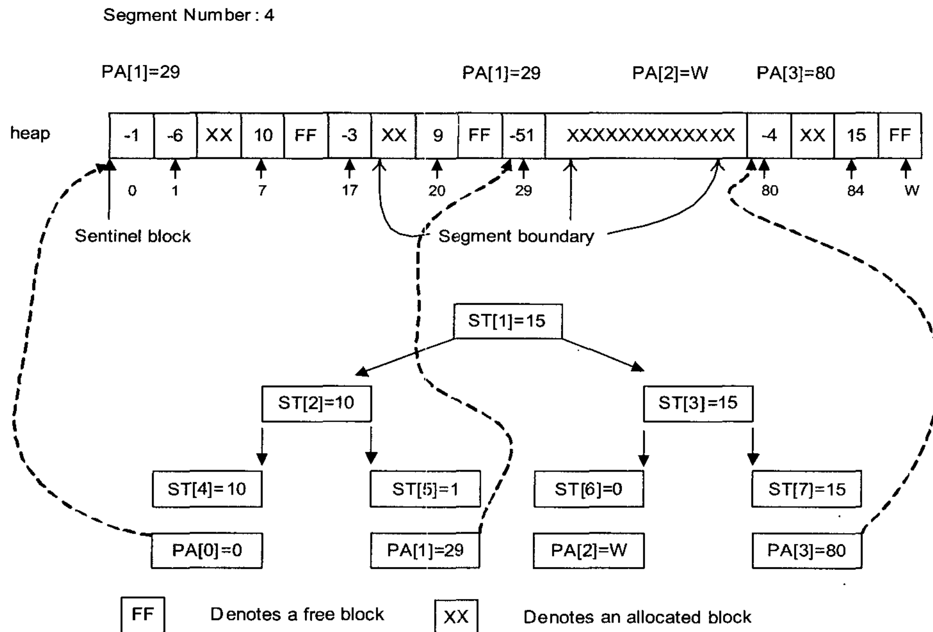
[그림 1] TLSF 알고리즘

예를 들어 [그림 1]에서 클래스값이 32일 경우 첫 번째 단계에서 관리하는 미할당 메모리의 크기는 32~63이며, 각 클래스는 미사용 메모리 블록의 크기에 따라 8개의 세부 영역으로 나뉜다. 각 세부 영역은 설정된 범위(32~35, 36~39, 40~43, … , 60~63)안에 포함된 미사용 메모리 블록을 포함하는 별도의 미사용 메모리 블록을 위한 링크드 리스트를 유지한다.

할당 방식은 필요한 메모리의 크기에 가장 적합한 미할당 메모리를 찾는 Best-Fit 방식으로 사용하며, 사용 메모리의 해제 시 인접 미사용 메모리와의 병합 기능을 제공한다. TLSF는 미사용 메모리의 관리에 있어 정확한 크기 값을 통해 관리함으로써 메모리를 할당할 때 메모리의 낭비가 적으며 메모리에 대한 사용/미사용 여부를 비트맵 정보를 이용함으로써 할당하고자 하는 메모리에 대한 검색 및 할당 비용이 상대적으로 적다. 그러나 TLSF 알고리즘은 메모리 압축 기능을 제공하지 않으며, 사용 중인 메모리의 해제 시 미사용 메모리간의 병합 기능도 추가적인 오버헤드를 발생시킬 수 있다. 예를 들면 사용 중인 메모리 블록의 해제로 인하여 발생하는 병합은 새롭게 생성된 미사용 메모리의 클래스 변경을 초래할 수 있으며, 이후 변경된 클래스에서의 연속적인 미사용 메모리간의 병합을 발생시킬 수 있다. 또한 TLSF 알고리즘은 초기 설정한 클래스를 수정할 수 없으며, 설정된 클래스 및 세부 영역도 임베디드 디바이스용 응용프로그램의 실행 특성을 고려할 때 사용되지 않는 것이 대부분을 차지한다.

2.5 Brent 알고리즘

Brent 알고리즘은 메모리 할당/해제에 따른 메모리 블록의 사용 유무에 대한 정보를 관리하기 위하여 트리 구조를 이용하며, [그림 2]와 같이 전체 메모리를 가변적인 개수의 세그먼트로 나누어 관리하는 알고리즘이다[15].



[그림 2] Brent 알고리즘

따라서 미사용 메모리에 대한 정보를 세분화된 세그먼트와 트리 구조를 통하여 관리함으로써 메모리를 할당할 때 적절한 크기의 미사용 메모리 블록을 찾는데 소요되는 시간 비용은 메모리 블록의 개수(F)에 대하여 O(log(F))값을 갖는다. 그러나 Brent 알고리즘은 사용 중이던 메모리를 해제할 때 인접 미사용 메모리와의 병합과 세그먼트별 최대 미사용 메모리에 대한 정보 갱신을 위한 세그먼트 내의 연산 비용이 크며, 할당되는 객체의 크기가 다양할 경우 오히려 링크드 리스트를 이용한 관리보다 비용이 크다. 또한 Brent 알고리즘은 메모리 관리에 있어 연속적인 미사용 메모리의 확보를 위한 메모리 압축 기능을 제공하지 않는다.

3. 모바일 디바이스용 응용프로그램 특성

지금까지 여러가지 동적 메모리 관리기법들이 연구되어 왔으며, 실제 많은 시스템에 각 동적 메모리 관리기법들이 적용되어 사용되고 있다. 그러나 이러한 동적 메모리 관리기법들은 대부분 데스크톱 컴퓨터에서 응용프로그램을 수행할 경우 우수한 성능을 보이지만 자원 제약적인 모바일 시스템에서는 좋은 성능을 보이지 못한다. 따라서 응용프로그램의 실행환경을 고려한 효율적인 메모리 관리기법에 대한 연구가 필요하며 이를 위해 모바일 디바이스용 응용프로그램의 특성을 파악하는 것이 중요하다. 다음 <표 1>, <표 2>는 모바일 디바이스용 응용프로그램에서 사용하는 객체 및 객체의 특성을 나타낸다[16].

<표 1>은 모바일 디바이스용 자바 응용프로그램인 MP3 플레이어[17], 주소록 프로그램, ICQ 프로그램[18]의 실행 시 할당되는 객체의 종류와 종료 시까지 각 객체당 메모리 사용량을 나타낸다. <표 1>에서 MP3 플레이어 프로그램은 보통 3~5분 정도의 음악 파일을 재생하는 프로그램으로서 음악 재생을 위한 MP3 파일의 표준 디코딩(decoding)을 위하여 상수(int) 테이블과 오디오 신호 생성을 위한 객체(char, byte)를 할당하며, 이는 할당되는 객체들의 90% 비중이다. 주소록 프로그램은 개인들의 신상정보를 기록하고 이를 디스플레이하는 프로그램으로서 각 개인의 이름, 주소, 전화번호 저장을 위한 객체(char, string)를 할당하며, 이는 할당되는 객체들의 81% 비중이다. 마지막으로 ICQ 프로그램은 인스턴스 메신저로서 메시지 전송을 위하여 반복적인 창의 생성을 위한 객체(object)와 메시지를 위한 객체(string, char)를 할당하며, 이는 할당되는 객체들의 82% 비중이다.

<표 2>는 모바일 디바이스용 응용프로그램인 MP3 플레이어, 주소록 프로그램, ICQ 프로그램의 종료 시까지 메모리에서 해제되는 개체별 개수를 나타낸다. <표 2>에서 MP3 플레이어 프로그램의 경우 메모리에서 다른 객체에 비하여 상대적으로 많이 해제되는 객체(char, string, int)는전체 해제되는 객체의 77%를 차지한다. 주소록 프로그램은 객체 할당에 있어서 많은 부분을 차지한 char, string 객체가 종료 시까지 해제되는 객체에 있어서도 100%를 차지한다. ICQ 프로그램은 객체 할당에 있어서 많은 부분을 차지한 char, string, objece 객체가 전체 종료 시까지 해제되는 객체의 82%를 차지한다.

<표 1>, <표 2>와 같이 모바일 디바이스용 응용프로그램은 크기가 작고 고정된 크기의 단순한 객체(int, char, string)들을 주로 할당하며, 이러한 객체들은 다른 크고 복잡한 객체에 비하여 라이프 타임이 짧다. 따라서 이러한 응용프로그램의 실행 특성을 메모리 관리기법의 설계 시 고려하게 되면 메모리 관리의 효율성을 높일 수 있을 것이다.

〈표 1〉 응용프로그램 실행 시 객체별 메모리 사용량

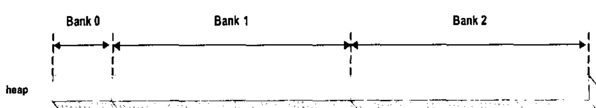
객체 타입	MP3 플레이어		주소록 프로그램		ICQ	
	메모리량(byte)	백분율	메모리량(byte)	백분율	메모리량(byte)	백분율
int	231,000	17%	0	0%	1,248	1%
float	73,000	5%	0	0%	0	0%
char	215,000	16%	1,520	41%	76,000	40%
byte	780,000	57%	0	0%	128	0%
double	0	0%	0	0%	0	0%
object	23,000	2%	260	7%	21,000	11%
string	16,000	1%	1,512	40%	58,000	31%
AccessControlContext	108	0%	180	5%	0	0%
Hashtable Entry	0	0%	108	3%	24,000	13%
class	23,000	2%	156	4%	8,424	4%
stringbuffer	3,836	0%	0	0%	28	0%
총 계	1,364,944	100%	3,736	100%	188,828	100%

〈표 2〉 응용프로그램 종료 시까지 해제되는 개체별 개수

객체 타입	MP3 플레이어		주소록 프로그램		ICQ	
	개수	백분율	개수	백분율	개수	백분율
int	190	5%	0	0%	16	1%
float	1	0%	0	0%	0	0%
char	1584	44%	72	60%	778	38%
byte	84	2%	0	0%	11	1%
double	0	0%	0	0%	0	0%
object	383	11%	0	0%	296	15%
string	1037	28%	49	40%	583	29%
AccessControlContext	0	0%	0	0%	0	0%
Hashtable Entry	0	0%	0	0%	61	3%
class	0	0%	0	0%	0	0%
stringbuffer	361	10%	0	0%	251	13%
총 계	3,640	100%	121	100%	1,996	100%

4. 메모리 모듈 설계 및 구현

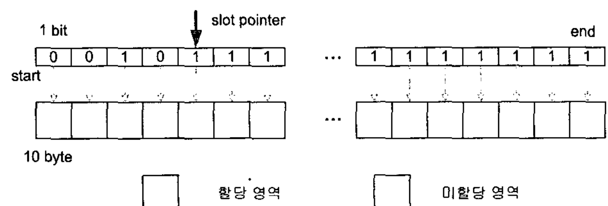
3장에서 기술하였듯이 모바일 디바이스용 응용프로그램은 크기가 작고 고정된 크기의 단순한 객체들을 주로 할당한다. 이러한 객체들은 다른 크고 복잡한 객체에 비하여 라이프 타임이 짧고, 다른 객체에 대한 참조를 제공하지 않으며, 또한 다른 객체에 대한 참조를 통해 발생할 수 있는 순환 참조도 없다[16]. 이러한 특성에 적합하도록 본 논문에서 개발한 동적 메모리 관리기법은 모바일 시스템에서 제공되는 전체 메모리를 세 개의 Bank로 구분한 후 각각의 Bank에 응용프로그램에 의해 사용되는 객체들의 라이프 타임, 크기, 종류, 분포 등의 특성을 고려하여 각기 알맞은 알고리즘을 적용함으로써 사용되는 메모리량을 효율적으로 절약하며, 응용프로그램의 실행 속도를 향상시켰다.



[그림 3] 시스템 메모리

4.1 Bank 0

응용프로그램이실행될 때 플랫폼과 응용프로그램은 다양한 이벤트를 처리한다. 이러한 이벤트로는 타이머 이벤트, 키 이벤트 등의 플랫폼 이벤트와 응용프로그램에서 발생되는 이벤트 등이 있다. Bank 0 영역은 이벤트 처리용으로 적합한 메모리 영역으로서 8바이트 고정 크기 메모리 블록의 풀로 구성되며, 각 메모리 블록의 사용 여부 표시는 비트맵을 이용한다. 이와같은 방법은 이벤트와 같이 빈번하게 할당 및 해제를 하여야 하는 객체에 대하여 포인터의 조작없이 단지 slot pointer의 인덱스 값을 증가시키기 때문에 메모리 풀을 스택을 이용하여 구현한 방법에 비하여 처리 속도를 높일 수 있다([그림 4]).



[그림 4] Bank 0 메모리 관리 방법

[그림 4]에서 비트 맵은 각 메모리 블록의 사용 여부를 플래그 값을 이용하여 나타낸다(할당0, 미할당 1). Bank 0 영역에서 slot pointer는 현재 검사할 메모리 블록의 비트맵 위치를 가리키며, 메모리를 할당할 때 'end' 방향으로 이동하며 미사용 메모리 블록을 찾아준다. 영역의 끝까지 slot pointer가 도달할 경우 다시 'start'로 이동하여 검색을 수행한다.

플랫폼 및 응용프로그램에서의 이벤트와 같이 빠르고, 빈번하게 할당 및 해제되는 작은 크기의 객체들은 메모리 영역을 단편화시키는 주된 요인이 되며, 이러한 객체들의 사용/미사용 여부를 관리하기 위해 소요되는 비용은 실제 사용되는 객체의 크기에 비하여 상대적으로 오버헤드가 크다. 또한 사용 중이던 메모리를 해제할 때, 연속적인 미사용 메모리의 확보를 위해 인접 미사용 메모리와 병합하거나 메모리를 압축하는 것은 이를 통하여 얻어지는 연속적인 미사용 메모리의 크기보다 연산에 따른 비용이 더 크다. 본 논문의 Bank 0 관리 방법은 이러한 문제점을 모두 해결한다.

본 연구에서는 이전에 개발한 위피 런타임 라이브러리에서의 플랫폼 이벤트 및 모바일 응용프로그램에서의 이벤트 특성을 고려하여 Bank 0에서 처리하는 메모리 블록의 크기를 설정하였다[19].

4.2 Bank 1

Bank 1에는 3장에서 기술한 모바일 디바이스용 응용프로그램의 특성을 고려하여 응용프로그램의 실행 시 60% 이상의 할당 및 해제를 차지하는 객체를 위한 동적 메모리 풀 관리기법을 적용하였다.

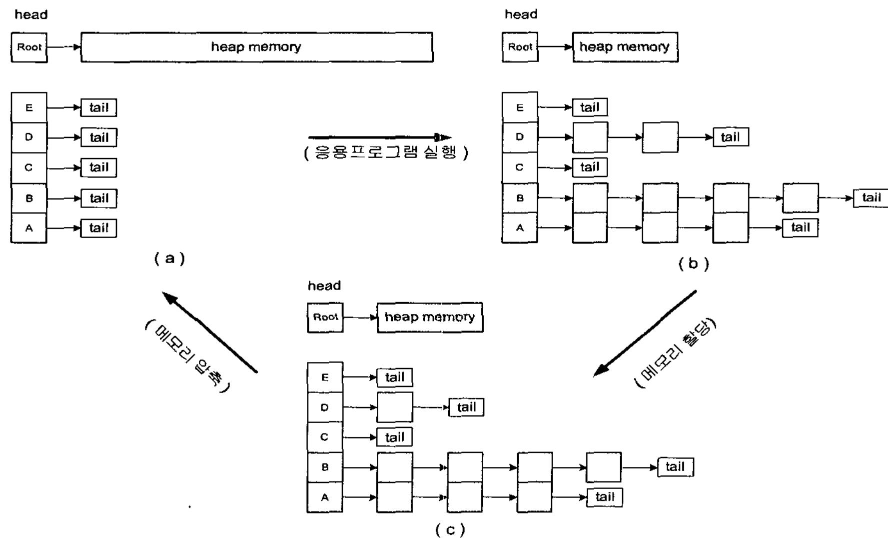
제안하는 동적 메모리 풀 관리기법은 일반적인 메모리 풀 관리기법과는 달리, 관리하는 미사용 메모리의 정확한 크기에 따라 별도의 싱글 링크드 리스트(single linked list)를 클래스 별로 관리하며, 각 싱글 링크드 리스트에 연결되는 미사용 메모리 블록의 개수는 응용프로그램이 실행됨에 따라 가변적으로 변하게 된다.

Bank 1은 미사용 메모리의 할당 시 Best-Fit 할당 방식을 사용하며, 연속적인 미사용 메모리 영역 확보를 위하여 메모리 압축 기능은 제공하지만 메모리 해제할 때 인접 미사용 메모리와의 병합 기능 제공하지 않는다. [그림 5]는 Bank 1에서의 메모리 할당, 해제 및 메모리 압축 과정을 나타낸다.

[그림 5]에서 각 싱글 링크드 리스트의 헤더에 있는 값들(A~E)은 각 클래스가 관리하는 메모리 블록의 크기를 나타낸다. 예를 들어 <표 1>의 char, int, string, object 등 자주 이용되는 객체 타입에 맞는 크기로 클래스를 정한다. [그림 5]의 (a)는 초기 메모리 상태로서 전체 메모리는 하나의 큰 메모리로 구성된다. 메모리 할당이 시작되면 우선 할당하려는 객체의 크기에 해당하는 클래스의 링크드 리스트를 검색한 후 미사용 메모리 블록이 있을 경우 이를 할당한다. 그렇지 않을 경우 Best-Fit 방식에 따라 상위 클래스의 링크드 리스트에서 할당하며 최종적으로 할당할 수 없을 경우, Root 리스트로부터 메모리를 할당한다. 사용 중이던 메모리가 해제되면 해제된 메모리 블록은 Root 리스트에 연결되지 않고 해당 클래스의 리스트에 연결된다.

[그림 5]의 (b)에서 모바일 응용프로그램이 자주 사용하는 객체들은 해당 클래스의 리스트에서 미사용 메모리를 쉽게 찾을 수 있으므로 할당 시 빠르다. [그림 5]의 (c)에서 계속적인 응용프로그램의 실행으로 미사용 메모리의 크기가 충분하나 새로운 메모리를 할당할 수 없을 경우 메모리 압축을 통하여 동적 메모리 풀의 재구성 기능을 제공한다.

모바일 디바이스용 응용프로그램은 실행 시 메모리를 절약하기 위하여 일련의 연속적인 객체 할당에 있어 같은 타입의 객체를 주로 할당한다. 이러한 객체들은 크기가 작고 고정된 크기의 단순한 객체들로 다른 크고 복잡한 객체에 비하여 라이프 타임이 짧다. 따라서 이러한 객체를 위한 메모리 할당 방식으로 주로 고정 크기 메모리 할당 방식을 사용하는 메모리 풀을 이용한다.



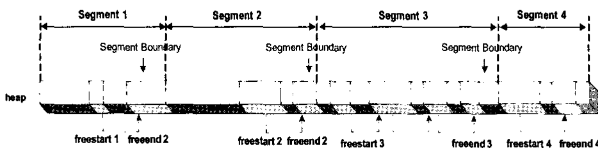
[그림 5] Bank1에서의 메모리 관리기법

그러나 기존 메모리 풀은 메모리 할당 및 해제를 위한 비용이 적지만 메모리 할당에 있어 비효율적인 할당 방식이며 (internal fragmentation), 전체적인 시스템 분석을 통하여 메모리 풀의 메모리 블록 크기를 설정해야 하는 오버헤드가 있다. 또한 메모리 풀에서 설정한 최대 메모리 블록의 크기보다 큰 크기의 메모리를 할당하거나 특정한 크기의 메모리 블록에 대한 할당이 전체 메모리 할당에 비하여 비율이 높을 경우 오히려 효율이 떨어지게 된다. 따라서 실행되는 응용프로그램의 특성을 따라 Bank 1과 같이 메모리 풀이 동적으로 재구성되면 모바일 응용프로그램의 실행 성능이 개선될 수 있다.

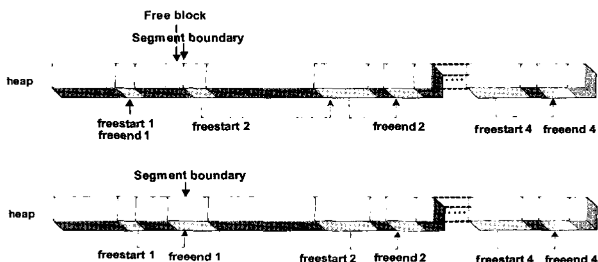
4.3 Bank 2

Bank 2는 다시 다수의 세그먼트로 구성된다. 각 세그먼트의 크기는 경계값(segment boundary)을 통하여 논리적으로 초기 설정에 좌우되지만 모바일 응용프로그램이 실행됨에 따라 가변적으로 변한다([그림 6]에서는 Bank를 4개의 세그먼트로 나누었음).

세그먼트화된 링크드 리스트 관리기법에서는 메모리 영역을 다수의 세그먼트로 나누어 미사용 메모리 블록들에 대한 관리를 하며, 경계값 이후의 첫번째 미사용 메모리 블록이 해당 세그먼트의 시작 미사용 메모리 블록이 되도록 유지한다. 따라서 세그먼트 수만큼의 미사용 메모리 블록 리스트가 생긴다. 각 세그먼트를 관리하는 리스트의 헤더에는 각 세그먼트들의 평균 미사용 메모리 블록 사이즈 정보를 유지하여 해당 세그먼트에서의 할당 가능성을 빠르게 파악할 수 있어 찾고자하는 미사용 메모리 블록에 대한 검색 시간을 단축한다. 또한 연속적인 미사용 메모리의 확보를 위한 메모리 압축 시 세그먼트별 압축이 가능하므로 압축에 대한 소요시간을 단축할 수 있다. [그림 6]에서 세그먼트 경계값으로 세그먼트를 구분하지만 경계값에 할당되는 메모리 블록의 사이즈가 유동적으로 변함에 따라 경계영역에서의 메모리 단편화를 줄일 수 있고 할당할 메모리 블록의 사이즈 제약을 받지 않는다.



[그림 6] Bank 2 메모리 관리기법: 세그먼트화된 링크드 리스트 관리

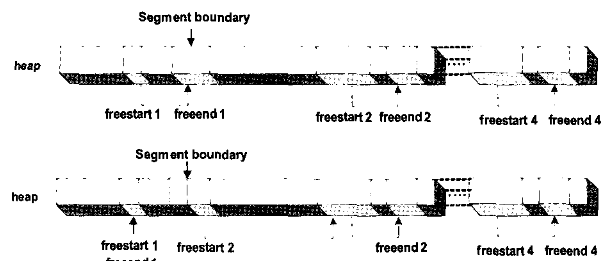


[그림 8] 메모리 해제

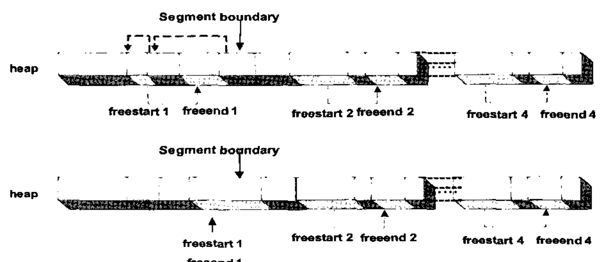
(1) 메모리 할당: 메모리 할당 요청이 들어오면 각 세그먼트의 미사용 메모리 블록 리스트의 헤더 정보내의 평균 미사용 메모리 블록의 사이즈를 검사하여 평균값이 큰 세그먼트를 선택한다. 할당할 세그먼트를 찾은 후, 해당 세그먼트의 첫 번째 미사용 메모리 블록부터 검색하여 First-Fit 방식으로 메모리를 할당한다. 이때, 할당할 수 있는 미사용 메모리 블록의 크기가 요청된 크기보다 크면 미사용 메모리 블록은 두 개로 나뉘어 한쪽 메모리 블록은 요청 사이즈에 맞게 할당하고 나머지 한쪽 미사용 메모리 블록은 경계값을 비교하여 어느 세그먼트의 미사용 메모리 블록 리스트 연결할 것인지 결정한 후 해당 리스트에 추가한다([그림 7]).

(2) 메모리 해제: 사용 중이던 메모리 블록이 해제될 때 해당 메모리 블록과 인접한 메모리 블록이 미사용일 경우 병합을 통하여 새로운 미사용 메모리 블록을 만든다. 이때 [그림 8]과 같이 해제되는 메모리 블록이 세그먼트의 경계에 있고, 인접한 미사용 메모리와의 병합을 통하여 생성된 미사용 메모리 블록이 세그먼트 경계에 있게 될 경우 생성된 미사용 메모리 블록은 이전 세그먼트의 미사용 메모리 블록 리스트에 연결하고, 다음 세그먼트의 미사용 메모리 블록 리스트의 시작 주소를 조정한다.

(3) 메모리 압축: Bank 2에서의 메모리 압축은 압축에 따른 오버헤드를 감소시키기 위하여 각 세그먼트 별로 수행된다. 메모리를 할당할 때 할당할 메모리의 크기와 각 세그먼트 별 미사용 메모리의 평균값을 비교한 후 '할당할 메모리 크기 > 평균 미사용 메모리 크기'인 세그먼트를 찾을 수 없으면 메모리 압축을 수행한다. 메모리 압축은 해당 세그먼트의 시작 메모리 블록에서부터 세그먼트의 경계값에 걸쳐 있는 메모리 블록까지 사용 중인 메모리 블록들을 연속적으로 한쪽으로 이동시킴으로써 수행된다([그림 9]).



[그림 7] 메모리 할당



[그림 9] 메모리 압축

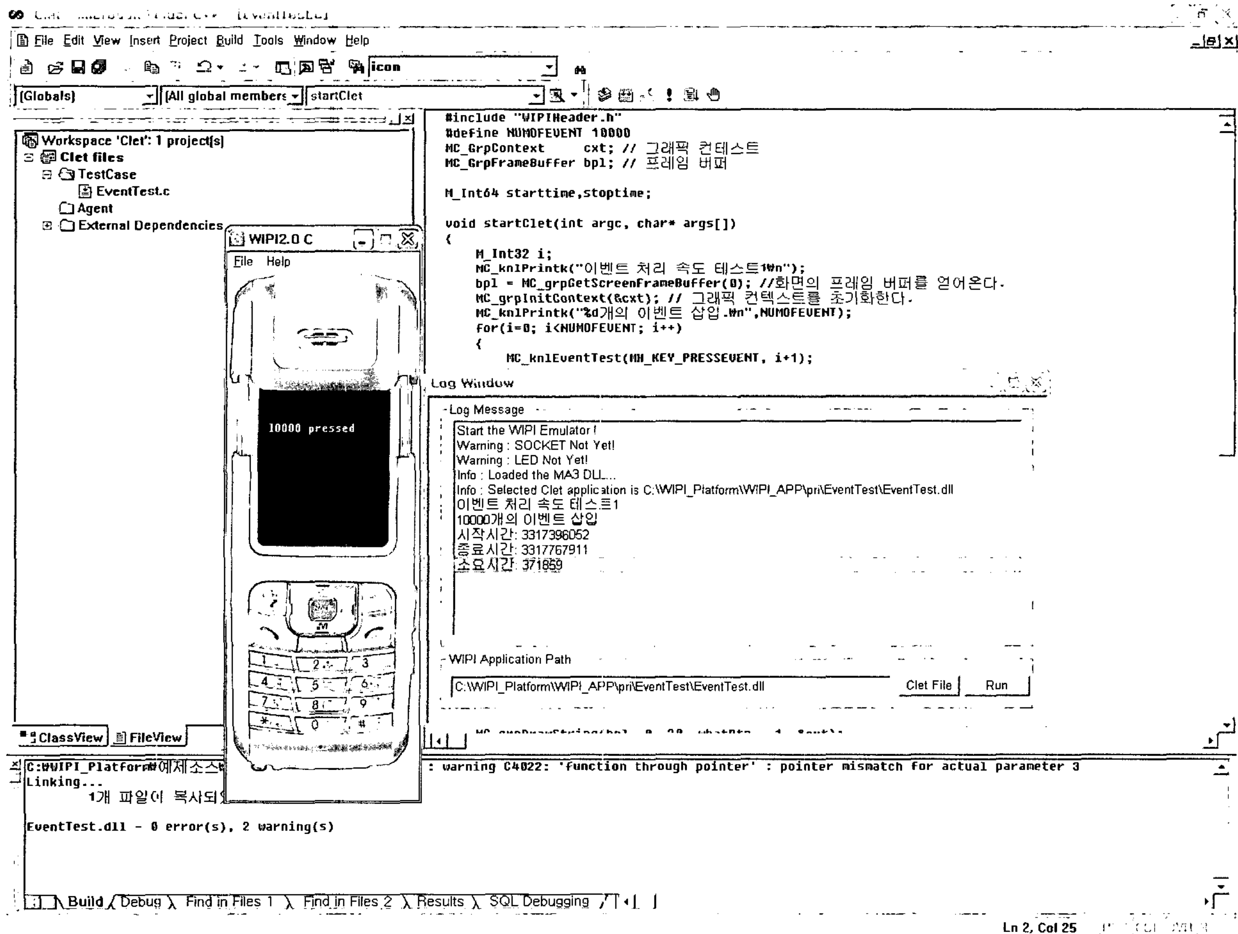
지금까지 런타임 라이브러리에서는 가변 크기 메모리 블록의 할당 및 해제 시 미사용 메모리 블록의 관리를 위하여 링크드 리스트를 이용한 방법이 가장 널리 쓰이고 있다. 그러나 링크드 리스트 관리 방법은 앞에서 언급한 것과 같이 많은 문제점들을 내포하고 있다. 이러한 문제점을 해결하기 위하여 가용 메모리 영역을 다수의 세그먼트로 나눈 후, 각 세그먼트에서의 미사용 메모리 블록의 관리를 위하여 링크드 리스트를 별도로 이용하는 방법들이 사용되고 있다. 이러한 방법은 미사용 메모리 블록에 대한 검색을 위한 시간을 단축할 수 있으며, 연속적인 미사용 메모리의 확보를 위한 미사용 메모리의 압축 시 메모리 전체를 압축하는 것보다 비용을 줄이는 장점이 있다. 그러나메모리의 물리적인 구분, 즉 고정 크기로 세그먼트를 나눔으로 인하여 메모리 할당의 최대 크기값이 결정되며, 영역을 구분하는 경계점에서 사용 메모리의 해제가 발생할 때 미사용 메모리간의 병합 기능을 제공하지 못하는 것이 단점이다. 따라서 영역간의 경계점에서 메모리 단편화가 발생할 수 있으며, 연속적인 미사용 메모리를 확보하기 위한 압축의 시점이 빨라질 수 있다. 본 연구에서 제안하는 방법은 세그먼트 구분이 논리적이며 세그먼트의 실제 크기가 가변적이기 때문에 기존 방식에서의 문제점들을 해결하였다.

5. 실험 환경 및 성능 비교

본 장에서는 개발한 메모리 관리 기법의 성능 측정을 위한 실험 환경에 대하여 기술하고, 현재 널리 쓰이고 있는 기존 동적 메모리 관리 기법과의 성능 비교 실험을 통한 결과 및 분석에 대하여 기술한다.

5.1 실험 환경

개발한 모바일 디바이스용 메모리 관리 모듈의 성능 분석을 위하여 Windows XP, CPU 3.2GHz, RAM 512메가의 환경에서 테스트하였으며, 모바일 디바이스 실행환경을 PC에서 에뮬레이팅하고, 위피 스펙을 준수하는 CNU 위피 에뮬레이터를 이용하였다[20]. CNU 위피 에뮬레이터는 본 연구팀이 2004년 11월에 개발한 위피 에뮬레이터로서 응용프로그램의 스케줄링, 이벤트 관리, 플랫폼 동적 재구성 및 동적 메모리 관리 기능을 제공하는 런타임 라이브러리와 핸드폰용 응용프로그램의 실행을 위한 API(Application Program Interface)들을 제공하며, 위피 스펙 변경에 따라 계속적인 업그레이드되고 있다. 실험을 위하여 CNU 위피 에뮬레이터의 런타임 라이브러리 내의 동적 메모리 관리 모듈을 본 연



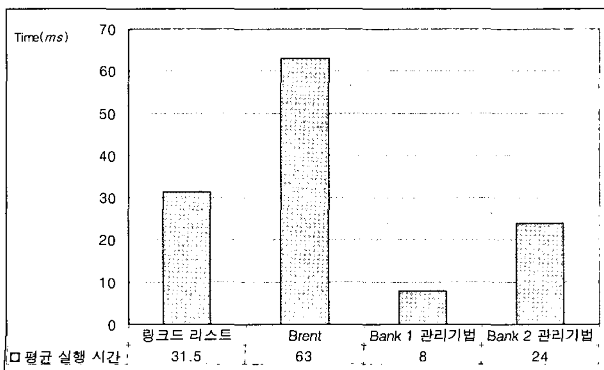
[그림 10] CNU 위피 에뮬레이터

구에서 새롭게 구현한 메모리 관리 모듈과 논문에서 소개한 메모리 관리 기법들을 구현한 메모리 관리 모듈로 교체한 후, 임베디드 디바이스용 응용프로그램의 특성을 고려한 테스트 응용프로그램을 작성하여 각각의 성능을 비교 분석하였다. [그림 10]은 CNU 위피 에뮬레이터의 실행 모습이다.

5.2 실험 1: 단일 메모리 관리기법 성능 비교

개발한 동적 메모리 관리기법의 성능을 비교 및 분석하기 위하여 링크드 리스트 관리기법, 메모리 압축 기능을 추가한 Brent 메모리 관리기법, 논문에서 제안한 동적 메모리 풀 관리기법(Bank 1 관리기법)과 세분화된 링크드 리스트 관리기법(Bank 2 관리기법)을 구현하였다. Half-Fit 메모리 관리기법은 본 논문에서 개발한 동적 메모리 풀 관리기법 및 TLSF 메모리 관리기법과 메모리 할당에 있어 소요시간이 O(1)로 동일하지만[13], 모바일 응용프로그램의 실행 보장을 위한 메모리 압축 기능을 제공하지 않기 때문에 다른 메모리 관리기법과의 비교대상에서 제외하였다. 또한 TLSF 메모리 관리 기법도 Half-Fit 메모리 관리기법과 같은 이유로 비교대상에서 제외하였다. 실험을 위하여 전체 메모리를 하나의 영역으로 보고 여기에 위에서 언급한 각 메모리 관리기법을 적용하여 동적 메모리 관리기법들의 성능을 비교하였다. Doug. Lea 메모리 관리기법은 여러 방법을 조합한 것이어서 5.3절에서 비교한다. 실험에 사용한 테스트용 응용프로그램은 8Byte에서 512Byte 사이의 크기값을 가지는 메모리 블록들을 랜덤(random)하게 할당 및 해제하며(회수: 12,000번), 이 과정에서 인접 메모리간 병합과 압축이 수행된다. 각 메모리 관리기법당 100회씩 실험을 반복한 후 평균 실행 시간을 구하였다([그림 11]).

실험 결과 Bank 1 메모리 관리기법은 테스트용 응용프로그램의 실행 속도가 Bank 2 메모리 관리기법에 비하여 3.0배, 링크드 리스트 관리기법에 비하여 3.94배, Brent 메모리 관리기법에 비하여 7.8배 빠름을 보였다. Bank 1 메모리 관리기법은 메모리 할당 시간이 O(1)으로 빠르며, 메모리 블록을 해제할 때도 해당 클래스의 리스트에 연결만 할 뿐 미사용 메모리간의 병합 및 메모리 압축을 하지 않음으로써 다른 메모리 관리기법에 비하여 높은 실행 성능을 보인다.



[그림 11] 동적 메모리 관리기법들의 성능 비교

Bank 2 메모리 관리기법은 전체 메모리를 세그먼트로 세분화함에 따라 검색 시간이 단축되어 링크드 리스트에 비하여 실행 속도에 있어 1.32배의 빠른 실행 속도를 보인다. 그러나 전체 메모리가 세분화되었음에도 불구하고 여전히 메모리 할당을 위한 리스트 검색 시간과 미사용 메모리간의 병합 및 메모리 압축에 따른 오버헤드가 있어 응용프로그램의 실행 속도가 Bank 1 메모리 관리기법에 비하여 30% 정도로 감소하였다. Brent 알고리즘은 세그먼트 개수(512개)를 크게 설정했음에도 불구하고 할당되는 객체의 크기가 다양함에 따라 세그먼트내의 메모리 검색 시간이 증가하여 다른 메모리 관리기법에 비하여 낮은 실행 성능을 보였다.

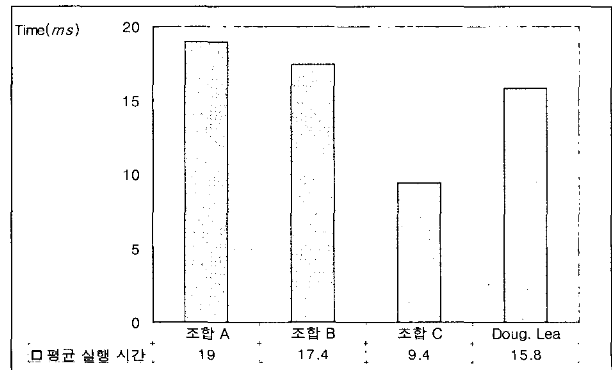
5.3 실험 2: 메모리 관리기법의 조합에 따른 성능 비교

메모리 관리에 있어 단일 메모리 관리기법을 적용하기 보다는 복수의 메모리 관리기법을 함께 적용하는 것이 응용프로그램의 실행 및 메모리 관리에 있어 효율적이다[15]. 따라서 메모리 관리기법의 조합에 따른 성능 비교 실험을 위하여 Doug. Lea 메모리 관리기법과 본 논문에서 제안하는 각 Bank에서의 메모리 관리기법들의 조합을 구현하였다.

전체 메모리를 두 개의 Bank로 구분하여 각 Bank별로 다른 메모리 관리기법을 적용하였으며, 조합에 따른 특징은 다음과 같다.

① 조합 A(고정 메모리 풀 + 링크드 리스트): 하나의 Bank에는 이벤트와 같은 객체의 할당/해제를 위하여 고정 메모리 풀 관리기법(본 논문 4.1절)을 적용하고, 다른 Bank에는 가변 크기의 객체를 위하여 링크드 리스트 관리기법(본 논문 2.1절)을 적용하였다. 기존에 링크드 리스트만 적용하였을 때와 비교하여 이벤트와 같은 객체의 처리를 위한 메모리 풀 영역의 효율성을 확인하고자 한다.

② 조합 B (고정 메모리 풀 + 세분화된 링크드 리스트): 조합 A에서 링크드 리스트 관리기법 대신 본 논문 4.3절에서 제안하는 세분화된 메모리 관리기법을 적용한 방법으로서 조합 A와 비교하여 세분화된 링크드 리스트의 성능을 확인하고자 한다.



[그림 12] 메모리 관리기법의 조합에 따른 성능 비교

③ 조합 C (동적 메모리 풀 + 세분화된 링크드 리스트): 두 개의 Bank에 각각 동적 메모리 풀 관리기법(본 논문 4.2절)과 세분화된 링크드 리스트 관리기법을 적용한 조합으로서 본 논문에서 제안하는 관리기법들의 조합에 따른 성능을 확인하고자 한다.

성능을 비교하기 위하여 이용한 테스트용 응용프로그램은 3장에서 분석된 응용프로그램의 실행 특성에 따라 작성되었으며, 작성 시 응용프로그램의 전체 실행에 있어 사용되는 객체들의 상대적 라이프 타임, 할당되는 객체의 종류 및 분포와 할당되는 객체들의 할당/해제 비율이 고려되었다. 각 조합별로 100회씩 실험을 반복하여 실행한 후 평균 실행 시간을 구하였다.

실험 결과 조합 B는 가변 크기 메모리 관리에 있어 Bank 2 관리기법을 사용하기 때문에 전체 메모리를 하나의 링크드 리스트로 관리하는 조합 A에 비하여 리스트 검색 시간 및 메모리 압축에 따른 오버헤드가 감소되어 응용프로그램의 실행 속도가 10% 빠르다. Doug. Lea 관리기법은 빈번히, 반복적으로 할당/해제되는 객체를 위한 관리에 있어 메모리 풀 방식을 사용함에 따라 응용프로그램 실행 속도의 향상을 보인다. 그러나 가변 크기 메모리 할당에 있어서는 링크드 리스트를 이용한 Best-Fit 방식을 사용하기 때문에 조합 A, B보다는 실행 속도가 빠르게 나타나지만(각각 17%, 11%) 조합 C의 실행 시간에 비해서는 실행 속도가 17% 줄었다. 조합 C는 모바일 응용프로그램 특성을 고려한 Bank 1 관리기법과 링크드 리스트의 오버헤드를 줄이는 Bank 2 관리기법의 조합으로서 실험 2에서 가장 빠른 실행 속도를 보였다. 따라서 메모리 관리기법들의 조합에 따라 응용프로그램의 실행 성능이 다르게 나타남을 확인할 수 있다.

5.4 실험 3: 제안하는 메모리 관리기법의 성능 비교

이번 실험은 본 연구에서 제안하는 메모리 관리기법(Bank 0: 고정 메모리 풀, Bank 1: 동적 메모리 풀, Bank 2: 세분화된 링크드 리스트)과 비교 대상인 Brent 관리기법, 링크드 리스트 관리기법 및 Doug. Lea 관리기법들을 실험 2에서 사용한 테스트용 응용프로그램을 이용하여 각 메모리 관리기법별로 100번씩 반복하여 실행한 후 각각의 소요 시간을 측정하였다.

<표 3>은 실험에서 사용한 메모리 관리 기법들에서 테스트용 응용프로그램을 실행할 때 소요되는 시간을 보여준다. 실험 결과 Brent 관리기법과 링크드 리스트 관리기법은 다른 메모리 관리기법에 비하여 테스트용 응용프로그램의 실행 소요 시간이 크고(31ms~78ms), 최소 소요시간과 최대 소요시간 간의 시간 차이도 크게 나타났다(16ms, 31ms). 또한 소요시간 중앙값과 평균 소요시간의 차이도 크게 나타났다(4.4ms, 6.1ms). 따라서 Brent 관리기법과 링크드 리스트 관리기법은 응용프로그램이 실행될 때 새로운 메모리의 할당 시 시간 소요가 크며, 할당에 소요되는 시간 역시 가변적이다. 이에 비하여 Doug. Lea 관리기법과 본 논문에서 제안한 메모리 관리 기법은 평균 소요시간이 각각 15.8ms, 6.2ms이며, 최소 소요시간과 최대 소요시간 간의 시간 차이는 3ms 미만이다. 따라서 Doug. Lea 관리기법과 제안한 메모리 관리 기법은 새로운 메모리의 할당 시 시간 소요가 작으며, 예측 가능한 시간 안에 새로운 메모리를 할당할 수 있다. 또한 제안한 메모리 관리기법은 Doug. Lea 관리기법에 비하여 테스트용 응용프로그램을 실행할 때 2.55배 빠르며, 최소 소요시간과 최대 소요시간의 차이도 1ms로 작다. 따라서 모바일 디바이스에서의 메모리 관리기법 설계 시 모바일 응용프로그램의 실행 특성을 고려할 경우, 사용되는 메모리를 절약할 수 있으며 모바일 응용프로그램의 실행 성능 속도도 향상시킬 수 있다.

〈표 3〉 메모리 관리 기법에 따른 소요 시간

단위: ms

	최소 소요시간	최대 소요시간	소요시간 중앙값 (median time)	평균 소요시간 (average time)
	시간 범위			
Brent 관리기법	62	78	70	65.6
	16			
링크드 리스트 관리기법	31	62	46.5	40.4
	31			
Doug. Lea 관리기법	13	16	14.5	15.8
	3			
제안 모델	6	7	6.5	6.2
	1			

6. 결 론

기존 동적 메모리 관리기법들은 응용프로그램의 실행 스타일과 사용되는 객체의 라이프 타임(life time), 객체 종류 및 종류 분포를 고려하지 않음으로써 효율적으로 메모리를 관리할 수 없으며, 응용프로그램의 실행 속도를 저하시킨다.

본 논문에서는 모바일 응용프로그램의 실행 특성을 분석하고, 분석한 결과를 토대로 메모리를 절약하며, 응용프로그램의 실행 속도를 향상시키는 새로운 동적 메모리관리기법을 제안 및 개발하였다. 기존 동적 메모리 관리 모듈과의 응용프로그램 실행 속도를 비교한 결과, 제안한 동적 메모리 관리기법은 테스트용 응용프로그램을 실행할 때 링크드 리스트에 비하여 6.5배, Brent 메모리 관리기법에 비하여 10.5배, Doug. Lea 메모리 관리기법에 비하여 2.55배 빠른 실행 속도를 보였으며, 다른 동적 메모리 관리기법들의 조합보다 빠른 실행 속도를 보였다.

향후 연구로서는 본 연구에서 개발한 동적 메모리 관리모듈의 실행 속도 향상 및 메모리 절약을 위한 연구를 계속적으로 진행할 예정이며, 아울러 에뮬레이터 환경이 아닌 실제 모바일 디바이스에 포팅(porting)할 예정이다.

참 고 문 헌

[1] P. R. Wilson, M. S. Johnstone, M. Neely, and D. Boles, "Dynamic Storage Allocation: A Survey and Critical Review," In H.G. Baker, editor, Proceedings of the International Workshop on Memory Management (IWMM'95), Kinross, Scotland, UK, Vol.986 of LNCS, Springer-Verlag, Berlin, Germany, 1995.

[2] I. Puaut, "Real-Time Performance of Dynamic Memory Allocation Algorithms," 14 the Euromicro Conference on Real-Time Systems (ECRTS'02), p.41, 2002.

[3] R. Jones and R. Lins, "Garbage Collection: Algorithms for Automatic Dynamic Memory Management," John Wiley and Sons, 1998.

[4] J. M. Chang and E. F. Gehringer, "A high-performance memory allocator for object-oriented systems,"IEEE Transaction on Computers, Vol.45, No.3, pp.357-366, 1996.

[5] M. Rezaei and K. M. Kavi, "A new implementation technique for memory management,"Proceedings of the IEEE Southeastcon 2000, pp.332-339, Apr., 2000.

[6] L. Dykstra, W. Srisa-an andJ.M. Chang, "An analysis of the garbage collection performance in Sun's HotSpotTM Java Virtual Machine,"Proceedings of 21st IEEE International on Performance, Computing, and Communications Conference, pp.335-339, Apr., 2002.

[7] M. S. Johnstone and P. R. Wilson, "The Memory Fragmentation Problem: Solved?," In Proceedings of the International Symposium on Memory Management(ISMM'98), Vancouver, Canada. ACM Press, 1998.

[8] L. Woo Hyong, J.M. Chang and Y. Hasan, "Evaluation of a high-performance object reuse dynamic memory allocation policy for C++ programs,"Proceedings of The Fourth International Conference/Exhibition on High Performance Computing in the Asia-Pacific Region, Vol.1, pp.386-391, May, 2000.

[9] C. Del Rosso, "Dynamic Memory Management for Software Product Family Architectures in Embedded Real-Time Systems," Proceedings of WICSA, 5th Working IEEE/IFIP Conference on Software Architecture, pp.211-212, Nov., 2005.

[10] Henry Lieberman and Carl Hewitt, "A Real-Time Garbage Collector Based on the Lifetimes of Objects,"Communications of the ACM, Vol.26, No.6, pp.419-429, Jun., 1983.

[11] A.C.K. Lau, N.H.C. Yung, Y.S. Cheung, "Performance analysis of the doubly-linked list protocol family for distributed shared memory systems," Proceedings of ICAPP 96, IEEE Second International Conference on Algorithms and Architectures for Parallel Processing, pp.365-372, Jun., 1996.

[12] T. Ogasawara, "An algorithm with constant execution time for dynamic storage allocation," 2nd International Workshop on Real-Time Computing Systems and Applications, pp.21-25, 1995.

[13] http://rtportal.upv.es/rtmalloc/allocators/dlmalloc/index.shtml, 2006.

[14] M. Masmano, I. Ripoll, A. Crespo, J. Real, ""TLSF: a new dynamic memory allocator for real-time systems," Proceedings of ECRTS, 16th Euromicro Conference on Real-Time Systems, pp.79-88, Jul., 2004.

[15] R. P. Brent, "Efficient Implementation of the First-Fit Strategy for Dynamic Storage Allocation," ACM Transactions on i'rograrrlming &guages and Systems, Vol.11, No.3, Jul., 1989.

[16] R. C. Krapf, G. Spellmeier and L. Carro, "A Study on a Garbage Collector for Embedded Applications," Proceedings of the 15 the Symposium on Integrated Circuits and Systems Design (SBCCI'02), pp.127-132, 2002.

[17] Javaplayer, Java MP3 Player,http://www.javazoom.net/javalayer/sources.html, 2006.

[18] ICQ Inc., ICQ Lite,http://lite.icq.com, Aug., 2006.

[19] 유용덕, 박충범, 최 훈, 김우식, "위피 응용프로그램 개발환경 설계 및 구현(Design and Implementation of Development Environment for WIPI Applications)," 한국정보처리학회 논문지 제12-C권 제5호, pp.749-756, 2005. 10.

[20] CNU(Chungnam National University) WIPI Emulator, http://strauss. cnu.ac.kr/research/wipi/research.html, Feb., 2006.

유 용 덕

e-mail : yyd7724@cnu.ac.kr

1999년 2월 충남대학교 컴퓨터공학과(학사)

2002년 2월 충남대학교 컴퓨터공학과(석사)

2002년 3월~현재 충남대학교 컴퓨터공학과 박사과정

관심분야 : 무선인터넷플랫폼, 임베디드 소프트웨어, 웨어러블 컴퓨팅

박 상 현

e-mai : sanghyun@etri.re.kr
1993년 2월 충남대학교 컴퓨터공학과 (학사)
1996년 2월 충남대학교 컴퓨터공학과 (석사)
2003년 3월~현재 충남대학교 컴퓨터공학과 박사과정
1996년 2월~2000년 11월 국방과학연구소
2000년 11월~현재 국가보안기술연구소
관심분야 : 무선인터넷플랫폼, 웨어러블 컴퓨팅, 컴퓨터 보안 등

최 훈

e-mail : hc@cnu.ac.kr
1983년 서울대학교 전자계산기공학과 (학사)
1990년 Duke Univ. 전산학과(석사)
1993년 Duke Univ. 전산학과(박사)
1983년 3월~1996년 2월 한국전자통신연구원 선임연구원
1996년 3월~현재 충남대학교 컴퓨터공학과 교수
관심분야 : 무선인터넷플랫폼, 임베디드 소프트웨어, 웨어러블 컴퓨팅, 분산시스템 등