실시간 시스템에서의 동적 스토리지 할당을 위한 빠른 수정 이진 버디 기법

이영재*, 추현승**, 윤희용**

Quick Semi-Buddy Scheme for Dynamic Storage Allocation in Real-Time Systems

Young Jae Lee*, Hyunseung Choo** and Hee Yong Youn**

Abstract

Dynamic storage allocation (DSA) is a field fairly well studied for a long time as a basic problem of system software area. Due to memory fragmentation problem of DSA and its unpredictable worst case execution time, real-time system designers have believed that DSA may not be promising for real-time application service. Recently, the need for an efficient DSA algorithm is widely discussed and the algorithm is considered to be very important in the real-time system. This paper proposes an efficient DSA algorithm called QSB (quick semi-buddy) which is designed to be suitable for real-time environment. QSB scheme effectively maintains free lists based on quick-fit approach to quickly accommodate small and frequent memory requests, and the other free lists devised with adaptation upon a typical binary buddy mechanism for bigger requests in harmony for the improved performance. Comprehensive simulation results show that the proposed scheme outperforms QHF which is known to be effective in terms of memory fragmentation up to about 16%. Furthermore, the memory allocation failure ratio is significantly decreased and the worst case execution time is predictable.

Key Words: Dynamic Storage Allocation, Real-Time System, Memory Allocation, Buddy System

* 성균관대학교 정보통신공학부

** 성균관대학교 정보통신공학부 교수

1. 서론

실시간 시스템이란 기존의 컴퓨터 시스템과 달리 시스템 동작의 정확성이 논리적 정확성뿐만 아니라 시간적 측면에 있어서도 좌우되는 시스템을 말한다. 실시간 시스템이 종료 시한(deadline)을 만족시키기 위해서는 고속의 계산을 요구하게 되지만, 고속의 계산이 실시간 시스템의 요구 조건을 만족하는 것은 아니다. 일반적으로 고속의 계산은 시스템의 평균 응답 시간을 최소화하지만, 실시간 시스템에서 요구되는 예측 가능성을 보장하지는 않는다. 여기에서 예측 가능성이란 시스템의 명세에 정의된 고장이나 작업 부하 조건에서 태스크의 종료 시한 만족을 보장하는 것을 의미한다.

전통적인 실시간 시스템에서는 최악의 경우에 실행시간을 사전에 예측할 수 있고 메모리 단편화가 적은 정적 할당(static allocation) 방법을 주로 사용하였다[1, 2]. 이것은 메모리 블록의 크기를 고정시키고, 이를 각 태스크에게 사전에 할당시키거나 시스템 초기화 시에 각 태스크가 사용할 메모리 블록을 미리 할당받는 방법이다. 이에 비해 동적 메모리 할당(dynamic storage allocation: DSA) 방법은 사전에 그 크기를 결정할 수 없을 경우에 효과적인 메모리 할당 기법이다.

기존의 실시간 시스템에서 정적 할당 방법을 사용한 이유는 동적 할당 방법의 불확실성에 기인한다. 대부분의 경성 실시간 시스템(hard real-time system) 개발자들은 DSA에 의해 시스템의 실행이 지속되면서 이용 가능한 메모리가 점점 작은 부분으로 분할되어 발생되는 메모리 단편화(memory fragmentation) 문제와 메모리의 할당하고 반환하는데 걸리는 최악의 경우 실행시간(worst-case execution time: WCET)을 사전에 예측할 수 없다는 문제 때문에 실시간 시스템에 적합하지 않다는 인식을 가지고 있었다. 이러한 이유 때문에 실시간 시스템을 위한 동적 메모리 할당 연구가 활성화되지 않은 것으로 알려진다[1, 2].

하지만 일반적인 시스템뿐만 아니라 실시간 시스템에서도 동적 메모리 할당은 중요한 시스템 구성 요소 중의 하나이며, 최근에 실시간 시스템에 적합하도록 WCET를 예측할 수 있는 동적 메모리 할당 알고리즘들이 연구되고 있다. 본 논문에서는 실시간 시스템에 대한 동적 메모리 할당 알고리즘의 적용 가능성을 확인하고, 실시간 시스템에 적합하도록 개선된 알고리즘을 제안한다. 제안하는 QSB(quick semi-buddy)는 작은 크기의 메모리 요구에 대해서 워드 크기별로 프리 리스트를 관리하고, 큰 크기의 메모리 요구에 대해서는 이진 버디 시스템을 실시간 시스템에 적합하도록 수정한 방식을 이용하여 관리한다. 이렇게 함으로써 메모리 조각율을 가능한 줄여 메모리 이용 효율성을 높이고, 외부 단편화를 감소시켜 할당 실패율을 개선하며, 할당과 반환에 소요되는 WCET가 예측 가능하도록 설계하였다.

본 논문의 구성은 다음과 같다. 제 2절에서는 기존에 연구된 동적 메모리 할당 알고리즘들에 대해 알아보고, 이들을 실시간 시스템에 적용할 때의 문제점을 살펴본다. 제 3절에서는 본 논문에서 제안하는 알고리즘을 설명한다. 제 4절에서는 시뮬레이션 결과를 통해 제안된 알고리즘과 기존의 알고리즘들을 비교하여 설명한다. 마지막으로 제 5절에서는 본 논문의 결론을 서술하고 앞으로의 연구 방향을 제시한다.

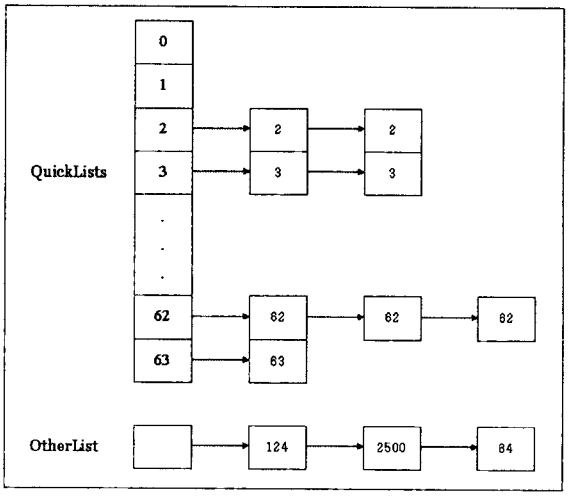
2. 연구 배경

2.1 기존의 동적 메모리 할당 알고리즘들

2.1.1 순차 적합(sequential fits)

고전적인 할당 알고리즘들은 메모리의 모든 프리 블록을 하나의 선형 리스트(single linear list)로 관리한다. 일반적으로, 순차 적합 알고리즘들은 경계 태그(boundary tag) 기술[5]을 이용하며, 합병을 빠르게 하기 위해 이중 연결 리스트(doubly linked list)를 이용한다. 그리고 이러한 프리 리스트들은 이미 잘 알려져 있는 최초-적합(first-fit), 다음-적합(next-fit), 최악-적합(worst-fit), 최적-적합(best-fit) 등의 메모리 할당 전략에 의해 메모리가 할당된다[3].

순차 적합 알고리즘들은 크기나 주소 순서에 관계없이 이중 연결 구조로 되어 있기 때문에 할당과 반환 시에 순차적으로 탐색해야 하므로 최악의 경우 실행 시간(WCET)을 예측할 수 없다는 문제점이 있다. 따라서 이 알고리즘들을 실시간 시스템에 바로 적용하는 데에는 문제가 있다.



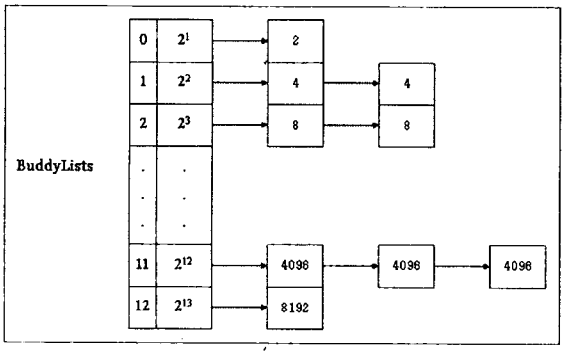
<그림 1> MaxQL=63인 경우의 빠른-적합

2.1.2 빠른 적합(quick-fit)

빠른-적합은 다중 프리 리스트와 단일 프리 리스트를 복합적으로 사용하는 알고리즘이다[3, 4]. 자주 요청되는 작은 크기들에 대해서 하나의 프리 리스트를 관리하고, 큰 크기의 블록들에 대해서는 크기에 상관없이 단일 프리 리스트로 관리하는 방식이다. 이 방식은 다양한 응용 프로그램들이 대부분 40워드 크기 이하의 메모리 할당을 요청한다는 연구 결과를 이용한 것이다[3, 4, 6]. 이러한 경우 빠른-적합 방식은 exact-fit 전략을 사용하기 때문에 높은 메모리 이용 효율을 가진다.

할당 요청이 들어온 메모리의 크기를 s라 할 때 $s \leq MaxQL$인 경우에 워드(word) 사이즈별로 관리되는 프리 리스트로부터 할당되므로, 리스트 안에서의 탐색이 없으므로 할당에 걸리는 시간이 매우 짧다. 또한 요청된 사이즈에 대해서 동일한 크기의 메모리를 할당해 주기 때문에 내부 조각율(internal fragmentation ratio)이 매우 적다.

하지만 큰 크기의 블록들에 대해서 단일 프리 리스트를 이용하기 때문에 탐색 시간이 길어져 할당에 걸리는 WCET를 예측할 수 없다는 단점이 있다.



<그림 2> 버디 시스템

2.1.3 이진 버디 시스템(binary buddy system)

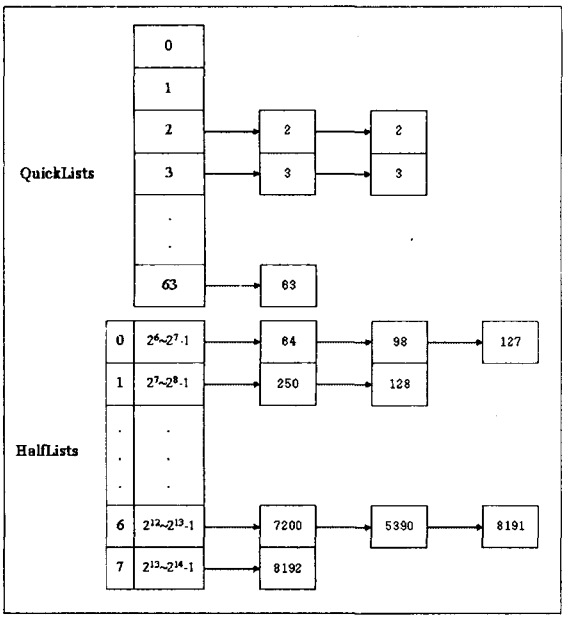
이진 버디 시스템은 제한적이지만 효과적인 분할과 합병을 제공하는 엄격한 분리 적합(strict segregated fit)의 한 종류이다[3, 6]. 이진 버디 시스템(binary buddy system)은 2의 지수 승의 크기를 가지는 프리 블록들을 리스트로 관리한다. 여기에서 하나의 메모리 블록은 버디(buddy)라는 두 개의 영역으로 나누어지며, 이 두 영역 중의 하나를 원하는 크기의 블록이 될 때까지 두 개의 버디들로 계속해서 분할하여 할당한다. 그리고 사용된 후 반환된 메모리 블록은 동일한 리스트 상에 존재하고 인접한 버디들끼리 합병될 수 있다.

이진 버디 시스템은 외부 단편화(external fragmentation)가 거의 발생하지 않는다는 장점을 가지고 있지만, 반면에 내부 조각율(internal fragmentation ratio)이 다른 알고리즘들에 비해 상대적으로 매우 높다는 단점이 있다[6, 7]. 이러한 이진 버디 시스템의 단점을 해결하기 위해 버디를 응용한 여러 가지 알고리즘들이 나오기는 했지만 관리 측면에서의 오버헤드(overhead) 때

문에 여전히 이진 버디 시스템이 가장 많이 사용되고 있다.

2.1.4 절반 적합(half-fit)

절반-적합(half-fit) 알고리즘은 실시간 시스템 환경에서 사용할 수 있도록 개발된 시간 복잡도가 O(1)로 한정되어진 다중 프리 리스트를 사용하는 알고리즘이다. 이 알고리즘에서 크기가 s인 메모리 블록들은 $2i \le s < 2i+1$인 프리 리스트가 관리한다. 그리고 크기가 s인 메모리 블록의 할당 요청에 대해서는 $i+1$을 인덱스로 하여 할당한다. 그러므로 그 인덱스에 해당하는 리스트를 탐색하여 적당한 메모리 블록을 할당하는 것이 아니라, 그 다음 리스트의 첫 번째 메모리 블록에서 요청된 크기만큼을 할당함으로써 탐색하는 시간적인 오버헤드가 제거된 할당 알고리즘이다.



<그림 3> *MaxQL*=63인 경우의 빠른-절반-적합

하지만 절반-적합은 외부 조각율과 이에 따른 할당 실패율이 다른 DSA 알고리즘들에 비해 상대적으로 높다는 단점이 있다. 절반-적합에서는 할당 가능한 메모리 블록이 존재하더라도 더 큰 메모리 블록을 할당하고 남는 부분은 프리 리스트로 돌려보내기 때문에, 시간이 지남에 따라 메모리가 단편화되어 이에 따른 외부 조각율이 증가하게 된다.

2.1.5 빠른 절반 적합(quick-half-fit : QHF)

빠른-절반-적합은 빠른-적합(quick-fit)과 절반-적합(half-fit)의 장점을 살린 방식이다[8]. 이 알고리즘은 작은 크기의 메모리 요청에 대해서는 빠른-적합을 이용하여 워드 크기별로 프리 블록 리스트를 관리하고, 큰 크기의 메모리 요청에 대해서는 절반-적합을 이용하여 2의 거듭제곱 크기별로 프리 리스트들을 관리한다. 다시 말해서, 빠른-절반-적합은 $s \le MaxQL$ 크기의 메모리 블록에 대해서는 빠른-적합 방식을 사용하며, $s > MaxQL$ 크기의 메모리 블록에 대해서는 절반-적합 방식을 이용하여 프리 리스트들을 관리한다.

QHF에서 중요하게 고려되어진 부분은 예측이 가능한 할당 시간을 가지도록 하는 것이다. 이때 할당 시간에 많은 영향을 미치는 것은 절반-적합 부분에서 인덱스 계산에 필요한 *log* 연산에 걸리는 시간이라 판단했다.. 그러므로 QHF에서는 이러한 인덱스 계산을 고정 시간에 수행하기 위해 8비트 단위로 *log* 값을 미리 계산해 놓은 테이블을 이용한다.

빠른-절반-적합은 실시간 시스템에 적용 가능한 DSA 알고리즘이다. 하지만 테이블 사용에 따른 부가적인 메모리 오버헤드가 존재하며, 절반-적합을 사용하기 때문에 외부 조각율이 상대적으로 높다는 단점이 있다.

2.2 메모리 단편화(memory fragmentation)

전통적으로 메모리 단편화는 내부 단편화(internal fragmentation)와 외부 단편화(external fragmentation)로 분류된다[3].

외부 단편화는 메모리의 프리 블록들이 존재하지만, 프로그램에 의해 실제로 요청된 크기의 프리 블록이 존재하지 않는 경우에 발생한다. 복잡

한 할당 알고리즘에서는 프리 블록들이 매우 작게 분할되기 때문에 발생하며, 반대로 간단한 할당 알고리즘에서는 큰 블록이 작은 블록으로 분할되지 않기 때문에 발생한다.

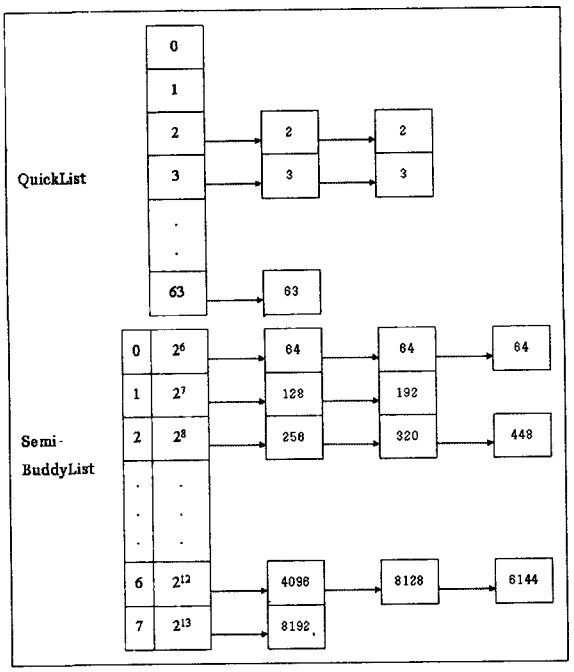
내부 단편화는 충분히 큰 프리 블록을 할당하였지만, 요구된 크기보다 큰 메모리를 할당하여 남는 부분이 생길 때 발생한다. 몇몇 간단한 할당 알고리즘에서는 할당의 편의성 때문에 내부 단편화가 일어나게 된다.

상대적으로 작은 메모리를 사용하는 실시간 시스템이나 임베디드 시스템의 경우, 메모리를 매우 효율적으로 사용하는 것이 요구된다. 따라서 동적 메모리 할당 알고리즘을 사용하는 경우에는 이러한 단편화 문제를 크게 염두에 두고 설계하여야 한다.

3. 제안하는 알고리즘

본 논문에서 제안하고자 하는 QSB(Quick Semi-buddy) 알고리즘은 작은 크기의 프리 메모리 블록에 대해서는 워드 크기 별로 프리 리스트를 관리하고, 큰 크기의 프리 메모리 블록에 대해서는 실시간 시스템에 적합하도록 변형된 이진 버디 시스템으로 관리한다. 다시 말해서, $MinQL \le s \le MaxQL$ 구간 내의 크기를 가지는 프리 메모리 블록 s는 워드 크기 별로 QuickList라는 리스트에 의해서 관리되며, 또한 $s > MaxQL$의 크기를 가지는 프리 메모리 블록 s는 BuddyList라는 리스트에 의해서 관리되게 된다.

QSB 알고리즘은 빠른-적합 방식과 이진 버디 시스템의 장점들을 이용하여 실시간 시스템에 적합하도록 설계된 할당 알고리즘이다. 이 알고리즘은 작은 크기의 메모리 블록에 대해서는 빠른-적합 방식을 이용하여 고정 시간 내에 메모리 할당을 제공하며, 큰 크기의 메모리 블록에 대해서는 이진 버디 시스템을 적용함으로써 외부 조각율을 감소시켰다.



<그림 4> *MaxQL*=63인 경우의 Quick Semi-Buddy

3.1 QuickLists와 비트맵(bitmap)

작은 프리 메모리 블록을 관리하는 QuickLists와 큰 프리 메모리 블록을 관리하는 Buddy-Lists는 <그림 5>와 같이 표현된다. 즉, 각각의 워드 크기별로 프리 리스트들이 존재하며, 이들은 하나의 벡터에 의해서 관리된다. 또한 이 벡터의 인덱스인 *Q_index*는 프리 리스트의 워드 크기와 같다.

$MinQL \le s \le MaxQL$ 구간 내의 크기를 가지는 프리 메모리 블록 s가 요청되면, Quick-Lists에서 s를 인덱스로 가지는 프리 리스트를 검사한다. 하지만 s를 인덱스로 가지는 프리 리스트 상에 프리 블록이 존재하지 않는다면 $s+1$에서 찾아야 하고, 또다시 $s+1$을 인덱스로 가지는 프리 리스트에도 프리 블록이 존재하지 않는다면 순차적으로 프리 블록이 있는지를 검사해야 한다. 이렇게 되면 최악의 경우 리스트 전체를 탐색해야 하며, 따라서 실행 시간은 $O(MaxQL-MinQL+1)$이 된다.

Second_bitmap

0	1
1	0
2	1
3	0
4	1
5	1
6	0
7	0

First_bitmap

	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	0	0
1	0	0	0	0	0	0	0	0
2	0	0	1	1	0	0	1	0
3	0	0	0	0	0	0	0	0
4	1	0	0	0	1	0	0	0
5	0	0	0	1	1	1	0	0
6	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0

<그림 5> bitmap들

이러한 탐색 비용을 줄이기 위해서, 버디 시스템에서는 각 리스트가 비어 있는지 아닌지에 대한 정보를 저장하는 비트맵을 사용한다. 하지만 *Q_index*의 개수가 많다면 탐색 시간 역시 길어지게 된다. 그러므로 본 논문에서는 이러한 탐색 시간을 더욱 줄이기 위해 참고문헌[8]에서 이용한 두 개의 비트맵을 사용한다.

따라서 이러한 두 개의 비트맵(bitmap)을 이용하여 비어있지 않은 리스트를 탐색한다. 첫 번째 비트맵(First_bitmap)은 그 비트맵이 가리키는 프리 리스트에 프리 블록이 존재하는지를 나타내며, 비트맵의 개수는 *Q_index*의 개수와 같다. 두 번째 비트맵(Second_bitmap)은 첫 번째 비트맵에서 8개의 엔트리(entry)를 하나의 비트맵 엔트리로 표현하여, 8개의 프리 리스트들 안에 프리 블록이 존재하는 지를 나타낸다. 즉, First_bitmap에서 8개의 비트가 모두 0이면 Second_bitmap의 이에 해당하는 비트는 0이고, 그렇지 않은 경우에는 1이 된다(그림 5).

여기에서 *MaxQL*을 어떻게 설정하느냐에 따라 그 성능이 달라질 수 있다. 본 논문에서는 *MaxQL*=63으로 가정한다. 이는 참고문헌 3, 4, 5에서 설명되어진 것처럼 대부분의 응용 프로그램들이 대부분 40 워드 크기 이하의 메모리 할당을 요청한다는 연구 결과를 이용한 것으로, 이러한 작은 메모리 요청을 빠르게 처리해 주기 위한 것이다.

QuickLists로부터 새로운 프리 블록에 대한 할당 요청이 들어온 경우, 요청된 프리 블록의 크기를 s, Second_bitmap의 한 비트가 관리하는 First_bitmap의 한 비트열의 비트 수를 $b1$, Second_bitmap의 비트 수를 $b2$라 하면 할당되는 과정은 다음과 같다.

(1) 요청된 블록 사이즈를 가지는 리스트가 비어있는지 First_bitmap을 이용하여 알아낸다.
(2) (1)에서 리스트가 비어있지 않다면, 그 리스트의 첫 번째 프리 블록을 할당한다.
(3) (1)에서 리스트가 비어있다면, Second_bitmap의 마지막 비트부터 s / $b1$ 번째 비트까지 내림차순으로 각 비트의 값이 1인지를 검사한다.
(4) (3)에서 비트의 값이 1인 비트를 찾았다면, 이 비트에 해당하는 First_bitmap의 비트 열에서 오름차순으로 각 비트의 값이 1인지를 검사한다.
(5) (4)에서 비트의 값이 1인 비트를 찾았다면, 이 비트에 해당하는 QuickLists의 리스트에서 프리 블록을 할당한다.
(6) (4)에서 비트의 값이 1인 비트를 찾지 못했다면, BuddyLists의 첫 번째 프리 리스트로부터 한 블록을 할당받아 요청된 크기인 s 만큼을 사용자에게 할당하고 남은 부분은 QuickLists로 반환한다.

이와 같은 할당 방식을 사용하면, 요청된 크기의 블록이 존재하지 않을 때 최초-적합 방식을 이용함으로써 탐색에 걸리는 시간을 단축할 수 있다. 또한 할당을 위해 비트맵을 검사하는데 걸리는 시간은 Second_bitmap의 길이와 Second_bitmap의 비트열의 길이에 의존하므로, WCET는 $O(b2\text{-}1+b1)$이 된다.

QSB에서의 합병은 기본적으로 경계 태그 방식을 이용하였다. QuickLists 내부에서의 합병 작업은 사용된 후 반환된 블록과 인접한 프리 블

록 사이에서 발생한다. 그러므로 최대 3개의 프리 블록들이 합병작업을 수행할 수 있다. 그리고 합병되어 그 크기가 BuddyLists의 첫 번째 프리 리스트가 관리하는 크기와 같다면, BuddyLists의 첫 번째 프리 리스트에 추가된다.

3.2 BuddyLists로부터의 메모리 할당

BuddyLists는 *MaxQL*보다 큰 메모리 요청을 처리하기 위해서 프리 블록들을 $(MaxQL+1)\times n$의 크기 별로 관리하는 벡터이다. 앞의 <그림 5>는 *MaxQL*=63인 경우 BuddyLists가 어떤 구조를 가지는 지를 보여준다.

BuddyLists에 존재하는 프리 블록들의 구조는 기존의 이진 버디 시스템의 프리 블록들의 구조와는 조금 다르다. 기존의 버디 시스템은 최악의 경우에 할당을 위해 버디 리스트의 리스트 개수만큼의 분할이 발생할 수 있다. 따라서 총 메모리의 크기가 커지면 커질수록 리스트의 수도 늘어나게 되며, 리스트의 수가 늘어나면 할당을 위한 분할 횟수가 증가하게 되므로 결과적으로 할당에 걸리는 WCET를 예측할 수 없게 된다.

그러므로 이러한 분할 과정을 단순화시키기 위해서 본 논문에서는 이진 버디 시스템을 변형한 semi-buddy를 제안한다. Semi-buddy 프리 리스트에서는 $(MaxQL+1)\times n$의 크기를 가지는 프리 블록들을 관리한다. 즉, 연속적으로 분할되는 과정을 제거하기 위해서 $2n$ 크기의 블록들만이 프리 리스트에 존재하게 하는 것이 아니라, *MaxQL*이 63인 경우 프리 리스트에는 64의 배수 크기의 프리 블록들이 존재하게 하였다. 예를 들어 설명하면, 64의 크기를 가지는 메모리 블록에 대한 할당 요청이 들어왔으며 256 이상의 프리 블록들만이 존재하는 경우, 256에서 64 만큼을 할당해 주고, 남은 192 크기의 프리 블록은 semi-buddy 프리 리스트로 돌려보내도록 하는 것이다. 이렇게 함으로써 연속적으로 분할되는 것을 방지할 수 있으며, 따라서 할당에 걸리는 시간을 예측 가능하도록 하였다.

*MaxQL*보다 큰 메모리 요청에 대한 할당 방법은 이진 버디 시스템과 같다. $s>MaxQL$인 s 크기의 메모리 할당 요청이 들어오는 경우, $log(s) - log(MaxQL+1)$을 인덱스로 하여 메모리를 할당해 준다. 하지만 앞 절에서 설명한 것처럼 QuickLists에서 $MinQL \le s \le MaxQL$에 존재하는 크기 s의 요청에 대해서 할당해 줄 프리 블록이 존재하지 않는다면, *B_index*=0인 프리 리스트의 첫 번째 프리 블록을 할당해 주게 된다.

할당 요청을 만족하는 크기와 같은 크기의 프리 메모리가 관리되는 리스트가 비어 있다면, 요청된 크기보다 크고 비어있지 않은 리스트를 탐색해야 한다. 하지만 총 메모리의 크기가 커지면 커질수록 프리 리스트의 수도 증가되어야 하므로, 탐색 시간이 점점 오래 걸리게 되어, 할당하는데 걸리는 시간을 예측할 수 없게 된다. 그러므로 탐색 시간을 예측 가능하게 하기 위해 QuickLists에서와 마찬가지로 두 개의 비트맵을 이용하여 프리 리스트의 존재 유무를 판단하고, 적절한 블록을 할당한다.

결과적으로, 본 논문에서 제안하는 알고리즘은 빠른-적합과 이진 버디 시스템을 합성하여 동적 메모리를 효율적으로 할당하도록 하였다. 하지만 이러한 두 가지 구조를 이용하고 적절하게 유지하기 위해서는 이진 버디 시스템 부분에서의 정확한 크기의 블록 할당과 블록의 유효성 검사가 필요하다. 그러므로 주소들이 *MaxQL*+1 크기의 배수가 되었을 때 이를 감지하고, 버디 시스템 부분에서 이를 정확하게 합병하고 유지하는 것이 매우 중요하다.

4. 성능 평가

이 장에서는 본 논문에서는 제안하는 알고리즘인 QSB와 기존의 DSA 알고리즘들, 즉 이진 버디 시스템과 QHF와의 성능을 비교하였다. 비교를 위해 전체 메모리 용량을 제한하여 인위적으로 생성된 메모리 할당 및 반환 패턴을 사용하였으며[8, 9], 이를 통해 메모리 사용 효율성과 조각율, WCET, 그리고 메모리 할당 실패율 등을 측정하였다.

4.1 시뮬레이션 방법

조각율(fragmentation ratio)은 DSA 알고리즘이 적용되는 응용 프로그램, 라이브러리, 운영체제와의 상관관계에 따라 다양하게 정의될 수 있다[6, 8]. 그러므로 본 논문에서는 실시간 시스템의 특성에 가장 적합한 참고문헌 [4]의 방법에 따라 실험하였다. 따라서 내부 조각율 $IF = A / R$ 로 정의한다. 여기에서 A는 할당한 총 메모리 량이고, R은 사용자가 요청한 총 메모리 량이다. 또한 외부 조각율은 할당이 실패할 때마다 측정되며, 이는 $EF = M / A$ 로 정의한다. 여기에서 M은 실험에 사용된 총 메모리 량이고, A는 할당이 실패할 때까지 할당된 총 메모리 량이다. 실험에서 여러 번의 할당 실패가 발생할 수 있으므로, 모든 외부 조각율의 평균으로 실험의 외부 조각율을 계산한다. 총 조각율(TF: total fragmentation ratio)은 내부 조각율과 외부 조각율로 인해 발생하는 메모리 손실로서, 다음과 같이 정의한다.

$$TF = IF \times EF = (A/R) \times (M/A) = M/R$$

따라서 총 조각율은 총 메모리 용량과 사용자가 요청한 순수한 메모리 량과의 상대적인 비율로 나타난다.

할당 실패율(AF: allocation failure ratio) $AF = FN / RN$ 로 정의한다. 여기에서 RN은 사용자가 요청한 총 할당 횟수이고, FN은 할당이 실패한 총 횟수를 나타낸다.

기존의 연구에 따르면 메모리 할당 시뮬레이션은 요구된 할당 크기, 할당된 블록의 시스템 내의 존속 기간, 그리고 할당 요구가 들어오는 시간 간격 등의 세 가지 분포에 상당한 영향을 받는다[8]. 본 논문에서는 참고문헌 [8, 9]에서와 같이 요구된 할당 크기 분포에 지수 분포와 균등 분포의 두 가지 확률 분포를 사용한다. 크기 분포의 평균은 8, 10, 12, 14, 16, 32, 64, 128, 256, 512, 1024, 2048 워드로 변화시켰다. 그러나 이때 사용되는 총 메모리 용량은 항상 32K 워드로 고정시켰다. 할당된 메모리 블록의 시스템 내의 존속 기간 분포는 균등 분포를 따르며, 5에서 15 사이의 시간 단위(time unit) 범위 내에 존속하며 평균은 10 시간 단위이다. 메모리 요구 도착 시간 간격 분포는 지수 분포를 따른다. 따라서 시뮬레이션은 M/G/∞ 큐잉(queueing) 모델을 따르며, 시스템 내에 존속하는 할당된 메모리 블록의 개수는 평균값 λ/μ 를 가지는 포아송(Poisson) 분포를 따른다. 여기에서 $1/\lambda$는 요구 도착 시간 간격 분포의 평균이며, $1/\mu$는 할당된 블록의 존속 기간 분포의 평균이다.

실험에 사용된 총 메모리 용량(32K)은 항상 고정되어 있으므로, 평균 할당 크기가 결정되면 시스템 내에 존속하는 할당된 메모리 블록의 평균 개수 λ/μ를 결정할 수 있다. 평균 존속 기간 $1/\mu$ 또한 고정된 값이므로 평균 요구 도착 시간 간격 $1/\lambda$를 결정할 수 있다.

최악의 경우 실행 시간(WCET)은 총 메모리 크기에 따라 다른 실행 시간을 가질 수 있다. 그러므로 WCET의 측정을 위해 총 메모리 크기를 64KB에서 256MB까지 변화시켜 가면서 측정하였다.

4.2 시뮬레이션 결과

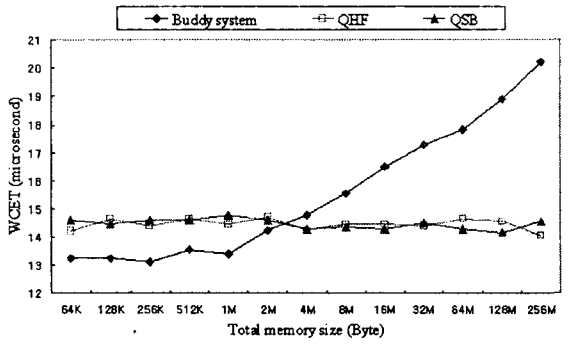
본 절에서는 앞에 기술한 큐잉 모델에 따라 각 크기별 메모리 할당 및 반환 패턴을 생성하여, QSB와 이진 버디 시스템 및 QHF의 메모리 할당 및 반환에 소요되는 WCET, 내부 조각율, 외부 조각율, 총 조각율 및 할당 실패율 등을 측정하였다.

실험은 Pentium III 800MHz 프로세서에 256MB 메모리를 장착한 컴퓨터에서 실행되었으며, 운영체제는 Linux 배포판 중의 하나인 WowLinux 7.1 Paran을 이용하였다.

<그림 6>은 총 메모리 용량에 따른 알고리즘들의 메모리 할당에 소요된 WCET를 보인 것이며, <그림 8>은 할당에 소요된 평균 시간을 보여준다. 여기에서는 총 메모리의 크기를 64KB에서 256MB로 변화시켰으며 요청되는 평균 할당 크기를 8워드로 고정시켜 측정하였다. 이는 할당 크기가 매우 작은 경우에 최악의 경우가 발생하

기 때문이다. 이 그림들은 1,000,000번의 할당과 반환을 수행하여 나온 결과로써 gettimeofday() 시스템 콜을 이용하여 측정하였다. 또한 테스트 프로그램의 코드 및 데이터 영역을 mlock() 함수를 통하여 locking 함으로써, 가상 기억 장치의 영향력은 배제되었다[8].

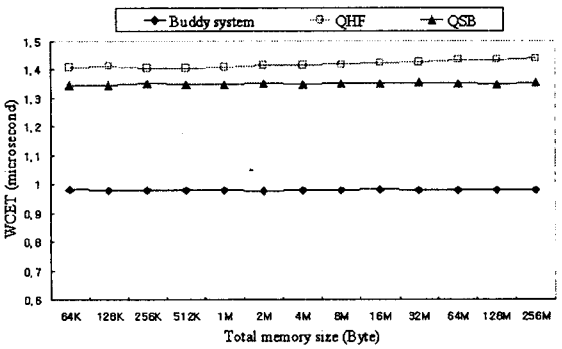
<그림 6>에서 QSB와 QHF는 총 메모리의 크기에 상관없이 일정한 WCET를 보인 반면에 이진 버디 시스템은 총 메모리 크기에 비례하여 WCET가 증가하는 것을 볼 수 있다. 이것은 이진 버디 시스템은 프리 리스트의 수가 증가할수록 분할 횟수가 증가하여 WCET가 증가하게 됨을 나타내는 것이다.



<그림 6> 메모리 할당에 소요된 WCET

<그림 7>에서는 평균 할당 시간을 보여주고 있다. 여기에서는 대부분의 내부 연산이 비트 연산으로 되어 있는 이진 버디 시스템이 가장 좋은 성능을 보여주고 있다. 하지만 실시간 시스템에서는 WCET가 예측 가능한 특징을 지녀야 하는데 이러한 점을 이진 버디 시스템은 만족시켜 주지 못하고 있다.

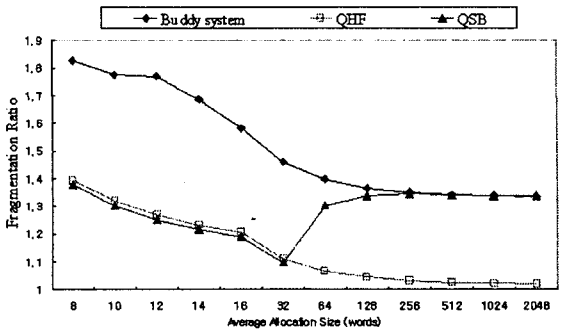
이 결과를 통해 확인할 수 있는 것은 WCET 측면에서 QHF보다 본문에서 제안한 알고리즘이 0.67% 정도의 성능 향상을 이루었다는 것이다. 또한 평균 할당 시간 측면에서 QHF와 비교하였을 때에는 5.1% 정도 성능이 향상되었다.



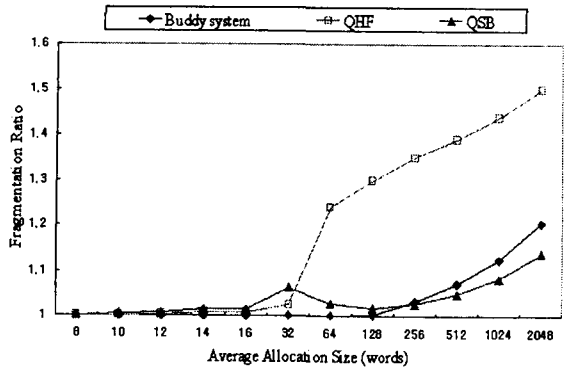
<그림 7> 메모리 할당에 소요된 평균 할당 시간

<그림 8>과 <그림 9>에서는 균등 분포에서의 내부 조각율과 외부 조각율을 비교하고 있다. <그림 8>에서 보면 QSB는 작은 크기의 메모리 할당에 대해서는 빠른-적합의 성격을 가지기 때문에 QHF와 비슷한 내부 조각율을 가지며, 할당 평균 크기가 64 이상이 되는 경우에는 버디 시스템의 결과를 따라가는 것을 확인할 수 있다. 그러므로 작은 크기의 메모리 할당에서는 조각율이 매우 적으며, 큰 크기의 메모리 할당에서는 조각율이 점점 커지게 된다.

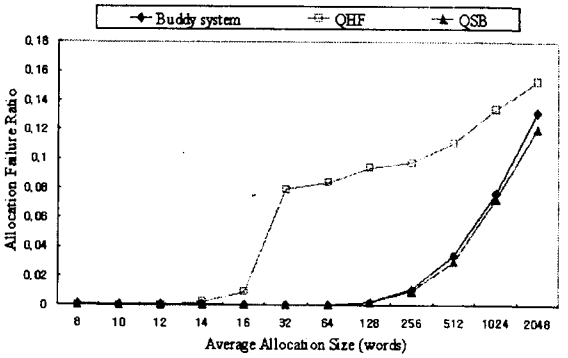
<그림 9>에서는 각각의 알고리즘들에 대한 외부 조각율을 보여주고 있다. 외부 조각율 면에서는 버디 시스템과 QSB가 매우 좋은 특성을 가지는 것을 확인할 수 있다. 또한 QHF의 경우는 큰 메모리 할당에 대해서 절반-적합을 이용하기 때문에 메모리 조각율이 커지는 것을 볼 수 있다.



<그림 8> 균등 분포에서의 내부 조각율

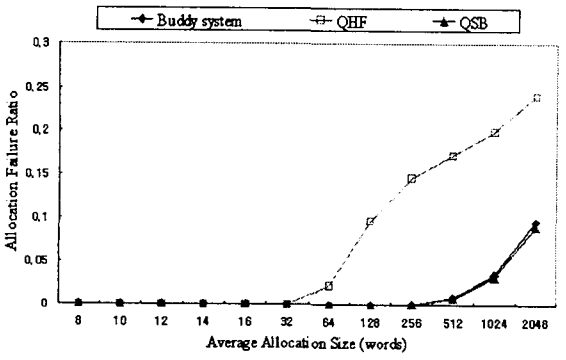


<그림 9> 균등 분포에서의 외부 조각율



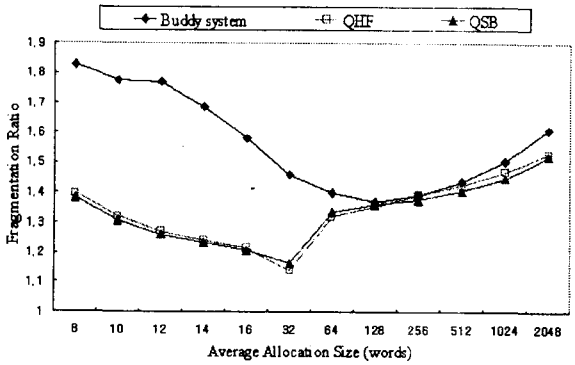
<그림 11> 지수 분포에서의 할당 실패율

<그림 10>과 <그림 11>에서는 할당 요청이 들어오는 평균 메모리 크기가 균등 분포와 지수 분포일 경우의 할당 실패율을 비교하고 있다. 두 그래프를 통해서 확인할 수 있듯이 버디 시스템과 QSB는 할당 실패율 면에서 매우 좋은 특성을 가지고 있음을 확인할 수 있다. 반면에 QHF는 요청되는 평균 메모리 크기가 커질수록 할당 실패율이 증가함을 알 수 있다. 이것은 QHF에서 큰 크기의 메모리 요청에 대해서 요청된 크기보다 큰 메모리를 할당하고 남은 부분을 다시 프리 리스트로 반환하는 과정이 빈번히 발생하기 때문에 메모리 단편화에 의해서 할당이 실패되는 것을 확인하였다.

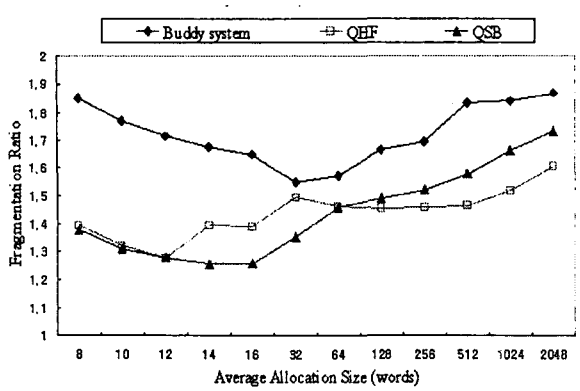


<그림 10> 균등 분포에서의 할당 실패율

마지막으로 <그림 12>와 <그림 13>은 균등 분포와 지수 분포에 대한 총 조각율, 즉 메모리 이용 효율성을 나타낸 것이다. 여기에서는 QHF와 QSB가 거의 유사한 메모리 이용 효율성을 가지는 반면에 버디 시스템은 메모리 이용 효율성이 좋지 않음을 확인할 수 있다. 또한 QHF와 QSB는 평균 할당 크기가 증가할수록 감소하다가 다시 증가하는 것을 볼 수 있는데, 이것은 평균 할당 크기가 작을수록 메모리 블록에 대한 블록 헤드의 비중이 커지기 때문이다.



<그림 12> 균등 분포에서의 메모리 이용 효율성



<그림 13> 지수 분포에서의 메모리 이용 효율성

이상의 결과를 통해 QSB는 할당에 소요되는 WCET가 예측 가능성을 가지며 할당 실패율이 좋은 알고리즘임을 확인하였다. 또한 메모리 이용 효율성 측면에서는 QHF와 비슷한 성능을 나타내고 있음을 확인하였다.

후기

본 논문에서는 기존의 동적 메모리 할당 알고리즘들을 기초로 하여 실시간 시스템에 적합한 메모리 할당 알고리즘을 제안한다. 제안하는 QSB(Quick Semi-Buddy) 알고리즘은 작은 크기의 프리 메모리 블록에 대해서는 워드 크기별로 프리 리스트를 관리하고, 큰 크기의 프리 메모리 블록에 대해서는 실시간 시스템에 적합하도록 변형된 이진 버디 시스템으로 관리한다.

제안된 알고리즘은 프리 리스트의 적절한 관리를 통해 메모리 이용 효율성을 높이면서, 실시간 시스템에 적합하도록 최악의 경우 실행 시간(worst-case execution time)을 예측할 수 있도록 하였으며, 할당 실패율 면에서의 우수성을 확인하였다.

하지만 QSB가 가지는 문제점인 *MaxQL* 보다 큰 크기의 메모리 요청에 대한 높은 내부 조각율 문제는 아직 해결되지 않았다. 이는 QSB에서 이용한 이진 버디 시스템이 가지고 있는 특징 중의 하나로써 일정한 크기, 즉 *MaxQL*+1의 배수 크기로 메모리를 할당하기 때문에 할당된 메모리의 일정 부분이 낭비되어 발생된다. 이러한 문제 때문에 상대적으로 큰 내부 조각율을 가지게 된다. QSB가 가지는 내부 단편화 문제는 앞으로 해결해야 할 과제로 남아있다.

기호설명

minQL: QuickList에서 관리되는 가장 작은 프리 블록의 크기
maxQL: QuickList에서 관리되는 가장 큰 프리 블록의 크기

참고문헌

[1] Kelvin D. Nilsen and Hong Gao, "The real-time behavior of dynamic memory management in C++", Proc. of Real-Time Technology and Application Symposium, pp.142-153, 1995.

[2] Ray Ford, "Concurrent algorithms for real-time memory management", IEEE Software, Vol.5, Issue 5, pp.10-23, 1988.

[3] Paul R. Wilson, Mark S. Johnstone, Michael Neely, and David Boles, "Dynamic storage allocation : a survey and critical review", International Workshop on Memory Management, 1995

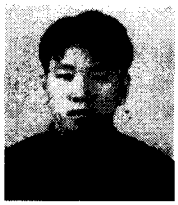
[4] A.K. Iyengar, "Scalability of dynamic storage allocation algorithms", In Frontiers '96 - The 6th Symposium on Frontiers of Massively Parallel Computing, IEEE Computer Society Press, pp. 223-232, 1996

[5] C.-T.D.Lo, W. Srisa-an and J.M. Chang, "Performance analyses on the generalised buddy system", Computers and Digital Techniques, IEE Proceedings-, Volume: 148 Issue:4-5, pp.167-175, July-Sept. 2001.

[6] Mark S. Johnstone and Paul R. Wilson, "The memory fragmentation problem : solved?", Proc. of Intn. Symposium on

Memory Management, pp.26-36, 1998
[7] J.L. Peterson and T.A. Norman, "Buddy systems", Communications of the ACM, Vol.20, No.6, pp.421-431, 1977
[8] 정성무, 유해영, 심재홍, 김하진, 최경희, 정기형, "예측 가능한 실행 시간을 가진 동적 메모리 할당 알고리즘", 「한국 정보처리학회 논문지」, 제7권, 제7호, Jul.2000, pp. 2204-2218
[9] Takeshi Ogasawara, "An algorithm with constant execution time for dynamic storage allocation", Proc. of 2nd Intn. Workshop on Real-Time Computing Systems and Applications, pp.21-25, 1995

저자소개



이영재
2001 대진대학교 통신공학 학사
2001.03~현재 성균관대학교 정보통신공학부 석사과정
관심분야: 운영체제

추현승
1988 성균관대학교 수학과 학사
1990 Univ. of Texas at Dallas, 컴퓨터공학 석사
1996 Univ. of Texas at Arlington, 컴퓨터공학 박사
1997 특허청 심사4국 컴퓨터 분야 심사관
1998.03~현재 성균관대학교 정보통신공학부 조교수

윤희용
1977 서울대학교 전기공학 학사
1979 서울대학교 전기공학 석사
1988 Univ. of Massachusetts, 컴퓨터공학 박사
1988.09~1991.05 Univ. of North Texas, Assistant Professor
1991.06~1993.05 Univ. of Texas at Arlington, Assistant Professor
1993.06~2000.08 Univ. of Texas at Arlington, Associate Professor
2000.09~현재 성균관대학교 정보통신공학부 교수