예측 가능한 실행 시간을 가진 동적 메모리 할당 알고리즘

정 성 무[†] · 유 해 영[††] · 심 재 홍[†††] · 김 하 진[††††] · 최 경 희[††††] · 정 기 현[†††††]

요 약

본 연구에서는 실시간 시스템용 동적 메모리 할당 알고리즘 *QHF(quick-half-fit)*를 제안한다. 제안된 알고리즘은 작은 크기의 메모리 요구를 위해서 워드 크기별로 프리 블록 리스트(free block list)를 관리하고, 큰 크기의 메모리 요구를 위해서는 2의 거듭제곱 크기별로 프리 블록 리스트를 관리한다. 이는 작은 크기의 메모리 요구에 대해 완전-적합(exact-fit) 전략을 사용함으로써 높은 메모리 사용 효율성을 제공한다. 또한 알고리즘 복잡도가 $O(1)$이면서 최악의 경우 실행 시간(worst case execution time WCET)을 보다 쉽게 예측 가능하게 한다.

제안된 알고리즘의 실용성을 확인하기 위해 알고리즘 복잡도가 역시 $O(1)$인 절반-적합(half-fit) 알고리즘, 이진 버디 시스템(binary buddy system)과의 메모리 사용 효율성을 비교하였다. 실험 결과 제안된 알고리즘은 메모리 크기에 상관없이 일정한 WCET을 보장하면서 조각율(fragmentation ratio), 메모리 할당 실패율 등에서 다른 알고리즘에 비해 보다 우수하다는 것을 확인하였다.

A Dynamic Storage Allocation Algorithm with Predictable Execution Time

Sung-Moo Jung[†] · Hae-Young Yoo[††] · Jae-Hong Shim[†††] · Ha-Jine Kimn[††††] · Kyung-Hee Choi[††††] · Gi-Hyun Jung[†††††]

ABSTRACT

This paper proposes a dynamic storage allocation algorithm, *QHF(quick-half-fit)* for real-time systems. The proposed algorithm manages a free block list per each word size for memory requests of small size, and a free block list per each power of 2 size for memory requests of large size. This algorithm uses the exact-fit policy for small size requests and provides high memory utilization. The proposed algorithm also has the time complexity $O(1)$ and enables us to easily estimate the worst case execution time (WCET).

In order to confirm efficiency of the proposed algorithm, we compare the memory utilization of proposed algorithm with that of half-fit and binary buddy system that have also time complexity $O(1)$. The simulation result shows that the proposed algorithm guarantees the constant WCET regardless of the system memory size and provides lower fragmentation ratio and allocation failure ratio than other two algorithms.

† 정 회 원 : 이주대학교 대학원 컴퓨터공학과
†† 정 회 원 : 단국대학교 전산통계학과 교수
††† 준 회 원 : 이주대학교 대학원 컴퓨터공학과
†††† 정 회 원 : 아주대학교 정보 및 컴퓨터공학부 교수
††††† 정 회 원 : 아주대학교 전기전자공학부 교수

논문접수 2000년 5월 17일, 심사완료 2000년 6월 28일

1. 서 론

동적 메모리 할당(dynamic storage allocation : DSA)은 사전에 그 크기를 결정할 수 없는 객체를 효과적으로 관리하기 위해 사용하는 유용한 프로그래밍 기술이다. DSA는 또한 데스크니 프로세스보다 생명주기가 훨씬 짧은 객체를 동적으로 관리함으로써 제한된 자원인 메모리의 사용 효율성을 증가시키기 위해 사용된다.

메모리는 많은 실시간 시스템에서 전체 시스템 가격을 결정하는 중요한 H/W 자원 중의 하나다. 시스템 가격을 줄이기 위해서는 이 같은 제한된 자원을 효과적으로 사용할 수 있는 기술이 반드시 필요하다. 따라서 메모리를 동적으로 재 할당할 수 있다면, 이는 H/W 가격을 고정시킨 채 시스템 성능을 개선할 수 있는 좋은 방법이 될 것이다.

실시간 시스템에서 시간 제약(time constraints)을 어기지 않고 동적 메모리 할당을 효과적으로 지원한다는 것은 매우 어려운 일이다. 메모리를 동적으로 사용하기 위해서는 사용자가 반환한 모든 메모리 영역을 미래의 재할당 요구에 이용할 수 있도록 해야 한다. 이를 위해서는 사용되지 않는 메모리 조각(fragmentation)이 제거되어야 하며 적어도 최소한으로 유지되어야 한다. 뿐만 아니라, 테스크의 최악의 경우 실행 시간(worst-case execution time : WCET)을 사전에 예측할 수 있어야 한다. 이를 위해선 실시간 시스템용 DSA 알고리즘 역시 WCET을 손쉽게 예측 할 수 있어야 한다[1, 3].

대부분의 경성(hard) 실시간 시스템 설계자 또는 개발자는 DSA가 실시간 시스템에서는 부적합하다는 인식을 가지고 있다. 이는 시스템의 실행이 지속되면서 이용 가능한 메모리 공간이 점점 조각화(fragmented)되어, 충분한 메모리가 있음에도 불구하고 궁극적으로는 더 이상의 메모리 할당을 할 수 없는 사태가 발생할 수 있다고 판단하기 때문이다. 또 다른 이유는 메모리 영역을 할당하거나 반환하는데 소요되는 최악의 경우 실행 시간을 사전에 예측할 수 없다는데 있다. 이런 이유로 실시간 시스템용 동적 메모리 할당 연구가 활성화 되지 못하고 있다[1-3].

그러나 본 연구팀은 실시간 시스템에서 동적 메모리 할당은 중요한 시스템 구성 요소 중의 하나라 판단하고, 메모리 사용 효율성이 높고 예측 가능한 실행 시간을 가진 DSA 알고리즘을 제안하고자 한다. 본 연구에서는 DSA의 첫번째 문제점인 조각화로 인한 메모리 고갈 문제는 오류 복구를 위한 예외 처리자(exception handler)를 통으로써 해결할 수 있다고 가정하고[4, 5], 두 번째 문제점인 DSA 알고리즘의 시간 복잡도 및 WCET에 초점을 두기로 한다.

일반적으로 WCET은 두 가지 측면, 즉 알고리즘 측면과 구현 측면으로 고려되어 질 수 있다. 알고리즘 측면에서의 WCET은 최악의 경우 매개 변수에 의한 그 알고리즘의 반복횟수를 나타내는 시간 복잡도이다. 구현 측면에서의 WCET은 알고리즘이 실제 구현된 시스템에서의 최악의 경우 실행 시간이다. 알고리즘 측면에서 시간 요구를 만족하는 알고리즘이 실제 시스템에서 그 알고리즘의 장점을 최대한 충족시킬 수 있도록 구현될 수 없는 경우도 존재한다. 따라서 본 논문에서는 알고리즘 측면과 구현 측면 등 두 가지 측면 모두의 WCET을 고려하기로 한다.

본 논문에서 제안하는 DSA 알고리즘은 작은 크기의 메모리 요구를 위해서 워드 크기별로 프리 블록 리스트(*free block list*)를 관리하고, 큰 크기의 메모리 요구를 위해서는 2의 거듭제곱 크기별로 프리 블록 리스트를 관리한다. 그러나 제안된 알고리즘은 프리 블록 리스트의 수가 상대적으로 증가되면서, 요구된 메모리 블록 크기에 가장 적절한 프리 리스트를 고정 시간 내에 찾는 알고리즘과, 찾은 프리 리스트가 비어 있을 경우(empty list) 다음 이용 가능한 프리 리스트를 고정 시간 내에 찾는 알고리즘을 필요로 한다. 따라서 본 연구에서는 이러한 알고리즘도 함께 제시하여 많은 프리 리스트를 관리하면서도 쉽게 WCET을 예측할 수 있게 하였다.

본 논문의 구성은 다음과 같다. 2절에서는 기존에 연구되었던 DSA 알고리즘을 분석하고, 실시간 시스템으로의 적용상의 문제점을 논의한다. 3절에서는 실시간 시스템용으로 사용 가능한 새로운 DSA 알고리즘을 제안하고, 시간 복잡도 및 최악의 경우 실행 시간 예측 가능성을 분석한다. 4절에서는 제안된 알고리즘과 기존의 알고리즘을 시뮬레이션 해보고 그 결과를 분석한다. 그리고 마지막 5절에서 향후 연구 계획에 대해 논의한다.

2. 연구 배경

본 절에서는 동적 메모리 할당 알고리즘을 설계하는

데 있어 결정해야 할 중요한 설계 요소들을 논의하고, 본 연구에서 제안하는 DSA 알고리즘과 관련이 있는 기존 DSA 알고리즘들을 분석해 본다. 또한 이들 알고리즘을 실시간 시스템 환경에 적용하는데 있어서의 문제점을 지적함으로써, 본 연구에서 해결해야 할 요소들을 검토해 보고자 한다.

2.1 동적 메모리 할당을 위한 고려 사항

컴퓨터의 출현 이후 지금까지 중요한 시스템 자원의 하나인 메모리를 효과적으로 관리하기 위한 동적 메모리 할당(DSA)은 수 많은 연구자들에 의해 연구되어 왔다[8]. 동적 메모리 할당자(allocator)는 경우에 따라 하나의 커다란 연속된 메모리 블록을 여러 개의 작은 블록들로 분할(*splitting*)하며, 사용자의 메모리 힐당 요구를 만족하기 위해 이들 중 하나를 넘겨 준다 그리고 남은 블록들은 이후의 할당 요구를 충족시키기 위해 계속해서 작은 블록들로 핀리한다. 사용 후 반환(deallocation)된 프리 블록들은 서로 인접한 블록끼리 합병(*coalescing*) 작업을 해야 하는데, 이는 두개의 작은 블록으로는 만족할 수 없는 보다 큰 메모리 요구에 대비하기 위해서다. 이러한 합병 전략으로는 즉시 합병(*immediate coalescing*)과 지연 합병(*deferred coalescing*)이 있다. 즉시 합병은 사용된 메모리가 반환되는 순간 곧 바로 이웃한 프리 블록이 있는지 조사하고, 만약 존재한다면 그 즉시 합병 작업을 수행하는 전략이다. 반면, 지연 합병은 연속된 프리 블록돌을 메모리 반환 시에 합병하지 않고 이후 더 이상의 메모리 할당 요구를 충족시키지 못하는 순간에 합병 작업을 수행하는 전략이다.

합병 작업은 일반적으로 주소 순서(address-ordered) 프리 리스트를 이용하는 방식과 경계 테그(*boundary tags*)를 이용하는 방식이 있다. 주소 순서 프리 리스트 방식은 주로 최초-적합(first-fit), 최적-적합(best-fit) 알고리즘에서 프리 블록을 주소 순서로 관리할 때 사용할 수 있는 방식이다. 그러나 프리 리스트가 n개의 프리 블록을 가지고 있을 경우, $O(n)$번을 탐색 해야 하는 단점이 있다. 이는 실시간 시스템에서 WCET을 예측할 수 없는 결과를 초래하며 회피 되는 방식이다. 대안으로 경계 태그를 사용할 수 있다[13]. 전통적으로 각 메모리 블록의 첫번째 워드는 그 메모리 블록의 크기를 저장하는데 사용된다. 이 크기 값이 양수일 경우 프리 블록을 의미하고, 음수일 경우 이미 사용자에게 할당되어 사용 중인 메모리 블록임을 의미한다. 그러나 합병 작업을 위해 경계 테그를 시용할 경우 블록의 마지막 워드를 별도로 필요로 한다. 이 워드는 첫번째 워드와 같이 그 블록이 프라인지 아닌지를 구분하기 위해 사용된다.

프리 블록들을 분할하고 합병하는 과정에서 필연적으로 나타나는 문제가 메모리 조각(*fragmentation*) 현상이다[9]. 메모리 조직 현상이 심화되면 비록 사용되지 않는 메모리 블록들의 크기의 총합이 사용자가 요청한 메모리 크기보다 더 클지라도 이를 충족시켜줄 수 없는 사태가 발생할 수 있다. 조각은 다시 내부 조각(*internal fragmentation*)과 외부 조각(*external fragmentation*)으로 나눌 수 있다. DSA 알고리즘은 사용할 수 없는 이러한 메모리 조각을 제거하거나 적어도 최소한으로 유지하여 메모리 사용 효율성을 극대화하도록 해야 한다

2.2 기존 동적 메모리 할당 알고리즘

기존의 모든 DSA 알고리즘들은 프리 메모리 블록들을 관리하기 위해 프리 리스트 또는 트리 구조를 사용하였다 단일 프리 리스트(*single free list*)는 하나의 프리 리스트만을 가지고 전체 프리 블록들을 관리하는 방식이다. 이 경우 다양한 크기의 프리 블록들이 크기 순서와 상관없이 이중 연결 (또는 트리) 구조로 하나의 프리 리스트에서 관리된다 따라서 요구된 크기의 메모리를 할당하기 위해서는 이 보다 크거나 같은 메모리 블록을 찾기 위해 불가피하게 프리 리스트를 순서적으로 탐색해야 한다. 이들 프리 블록들은 일반적으로 FIFO, LIFO, 또는 주소 순서(address order : AO)에 의해 프리 리스트상에 정렬되어지며, 최초-적합(*first-fit*), 다음-적합(*next-fit*), 최적-적합(*best-fit*) 등의 널리 알려진 메모리 힐당 전략에 의해 사용자에게 할당되어진다[13, 14]. 프리 메모리 블록들을 관리하기 위한 또 다른 방식은 여러 개의 프리 리스트를 사용하는 다중 프리 리스트(*multiple free lists*)이다. 여기서 각 프리 리스트에는 사전에 정의된 등일 크기 또는 비슷한 크기의 블록들을 함께 관리한다. 각 프리 리스트는 이 리스트 내에 포함할 수 있는 프리 블록들의 크기에 대한 일정 범위를 가진다. 일반적으로 많이 사용되는 크기 범위 체계로는 2의 거듭제곱 크기(즉, 4, 8, 16, 32 워드 등등)를 사용한다.

버디 시스템(*buddy system*)은 메모리 블록의 분할

과 합병 작업을 제한적이지만 효과적으로 지원하는 다중 프리 리스트의 변형된 알고리즘이다[15, 16]. 이 알고리즘에서 하나의 메모리는 개념적으로 버디(buddies)라고 불리는 커다란 두개의 영역으로 나누어진다. 이들 영역들은 다시 충분히 작은 크기의 블록이 될 때까지 반복하여 보다 작은 두개의 버디들로 양분된다. 반환된 메모리 블록의 합병은 반드시 계층적 분할 구조의 동일한 계층의 프리 블록인 버디와만 가능하다. 메모리 블록의 할당 및 합병의 이러한 제약은 블록의 유일한 버디를 항상 간단한 주소 계산에 의해 찾을 수 있도록 해주는 장점이 있다. 이진 버디(*binary buddy*) 시스템은 가장 간단하면서 널리 알려진 버디 시스템이다[13, 15]. 여기서 모든 버디의 크기는 항상 2의 거듭제곱이며, 각 버디는 다시 두개의 똑 같은 크기의 버디들로 양분될 수 있다.

빠른-적합(*quick-fit*)은 다중 프리 리스트와 단일 프리 리스트가 복합된 알고리즘이다[10, 11]. 자주 요청되는 여러 작은 블록 크기, 즉 $MinQL \le s \le MaxQL$ 크기 구간 내에 존재하는 각각의 블록 크기 s에 대해 하나의 프리 리스트(이를 *quick* 리스트라 함)가 존재한다. 여기서 *MinQL*은 할당 가능한 최소의 블록 크기이고, *MaxQL*는 quick 리스트에서 관리 되는 가장 큰 블록 크기를 의미한다. *MaxQL*보다 큰 블록들은 크기에 상관없이 하나의 단일 프리 리스트로 관리된다. 빠른-적합 알고리즘은 거의 대부분의 메모리 할당 및 반환이 quick 리스트에서 수행되기 때문에 높은 메모리 사용 효율성을 제공한다. 이러한 전략의 타당성은 다양한 응용 프로그램의 메모리 요구 크기를 조사한 결과 거의 대부분이 30-40 워드 크기 이하라는 최근의 여러 연구 결과가 이를 뒷받침하고 있다[6-12]. 이를 바탕으로 본 연구에서도 작은 크기의 프리 블록에 대해서는 빠른-적합 전략을 채택하기로 했다.

절반-적합(*half-fit*)은 실시간 시스템 환경에서 사용할 목적으로 개발된 시간 복잡도가 $O(1)$으로 한정(bounded)된 다중 프리 리스트를 이용하는 DSA이다[1, 17]. 크기 범위 체계로는 2의 거듭제곱 크기들을 사용한다. 블록의 크기 s가 $2^i \le s < 2^{i+1}$인 프리 블록들은 i를 색인으로 사용하는 하나의 프리 리스트에서 관리된다. 따라서 $[2^{i-1}+1, 2^i]$ 크기 영역 내의 메모리 요청이 있을 경우 프리 리스트 i에서 프리 블록이 할당되고 반환된다. 만약 프리 리스트 i가 비어 있다면, i보다 크면서 비어 있지 않은 가장 작은 프리 리스트에서 하나의 블록을 빼낸다. 이 블록을 분할하여 요청된 크기만큼의 메모리를 할당하고, 나머지는 해당 크기의 프리 리스트에 다시 삽입한다. 반환된 블록은 즉시 합병 방식에 의해 합병되며, 합병된 프리 블록은 기존의 프리 리스트에서 삭제되고 해당 크기의 프리 리스트로 삽입된다.

2.3 실시간 시스템으로의 적용 가능성

실시간 시스템에서의 DSA 알고리즘은 사용할 수 없는 메모리 조각을 최소화하면서 WCET을 손쉽게 예측 할 수 있어야 한다. 그리고 알고리즘 측면의 시간 복잡도를 $O(1)$으로 유지하면서 실제 구현에 있어서도 시스템 환경의 변화와는 무관하게 일정한 WCET을 보장할 수 있어야 한다. 이러한 관점에서 볼 때, 기존의 DSA 알고리즘을 실시간 시스템에 적용하기에는 다음과 같은 문제점을 가진다.

기존의 다목적용 동적 메모리 할당 연구들은 DSA 알고리즘이 평균적으로 얼마나 빨리 그리고 얼마나 효율적으로 메모리 블록들을 할당할 수 있느냐에 초점을 두었다. 단일 프리 리스트에서는 메모리 효율성을 동일하게 유지하면서 (즉, 기존 메모리 할당 전략을 고수하면서) 메모리 할당을 최대한 빠른 시간 내에 달성할 수 있도록 평균 리스트 탐색 시간을 최소화하는 방법에 초점을 맞추어 진행되었다. 그 결과 리스트 탐색 부하를 줄이기 위해 프리 리스트를 트리 구조로 관리하는 구현 방법[18-20]과 메모리 캐싱을 이용하는 구현 방법[21-23] 등이 제안되었다. 그러나 여전히 최악의 경우 시간 복잡도가 $O(n)$[19], $O(log_2\ n)$[18], 또는 $O(log_2\ w)$[20]이므로 실시간 시스템에 적용하기에는 WCET을 예측하기가 쉽지 않다. 여기서 n은 메모리내의 프리 블록의 수이고, w는 할당된 가장 큰 메모리 블록의 크기이다.

이진 버디 시스템의 경우 프리 리스트의 개수가 사전에 특정한 상수(한 워드의 비트 수)로 고정되므로, 시간 복잡도는 궁극적으로 $O(1)$이다[1]. 그러나 문제는 다른 DSA알고리즘과 비교 해 볼 때 내부 조각율이 상대적으로 매우 높다(25%)는 것이다[13, 16, 9]. 따라서 보다 많은 프리 리스트를 둠으로써 이러한 문제를 해결하기 위해 피보나치(fibonacci) 버디[24], 가중치(weighted) 버디[25], 이중(double) 버디[26] 등이 제안되었으나, 여전히 상대적으로 높은 조각율을 보였다.

빠른-적합은 *MaxQL*보다 큰 블록들은 크기에 상관없이 하나의 단일 프리 리스트로 관리되는 문제점을

가지고 있다. 이를 해결하기 위해 하나의 단일 프리 리스트를 다시 여러 개의 멀티 프리 리스트를 이용하여 개선하려는 노력이 참고 문헌 [6, 7] 등에 의해 시도되었다. 그러나 최악의 경우 여전히 단일 프리 리스트와 동일한 시간 복잡도를 가지고 또한 지연 합병 방식을 사용하므로, 실시간 시스템 환경에 수정 없이 바로 사용하기에는 곤란한 문제점을 가지고 있다.

절반-적합 알고리즘[1]에서는 요청된 메모리 크기 s가 주어질 경우, 색인 i를 $\lfloor log_2 s \rfloor$를 이용해 구한다. 저자는 이를 CPU의 비트 탐색 명령어에 의해 구현할 수 있으므로, 시간 복잡도가 진정한 $O(1)$이고 WCET을 고정된 상수로 유지할 수 있다고 주장한다. 그러나 비트 탐색 명령어는 모든 CPU에서 제공되는 명령어는 아니며, 범용적으로 이용되기 위해서 S/W적으로 구현되어야 할 필요가 있다. 시간 복잡도는 최악의 경우 시스템 워드의 비트 수에 의존하므로 여전히 $O(1)$을 유지할 수 있다. 그러나 이 알고리즘은 작은 크기의 메모리 블록의 요구가 많은 경우 상대적으로 높은 조각율을 보임을 본 연구의 실험을 통해 확인 할 수 있다.

실시간 시스템에서 DSA를 위해 고려해야 할 또 다른 사항은 어떠한 합병 전략을 택하느냐는 것이다. 지연 합병은 생명 주기가 짧은 작은 크기의 객체가 반복해서 사용될 경우 불필요한 합병과 분할 작업을 방지해 준다는 측면에서 일반 시스템에서는 널리 사용된다. 그러나 비 주기적으로 발생하는 특성으로 인해 WCET을 측정하기가 어려운 관계로 실시간 시스템에서는 사용하기 곤란하다. 따라서 즉시 합병 전략이 보다 적합하다고 판단된다. 이중 연결 구조를 가진 프리 리스트(double linked list)와 경계 태그를 이용한 합병 구현 전략은 고정된 상수 시간에 합병 작업을 수행할 수 있으므로, 실시간 환경에 사용하기 적합한 기술이다. 그러나 경계 태그를 위한 별도의 두 워드(프리 블록의 맨 앞과 맨 뒤)를 필요로 하는 부하가 있다. 다행히 사용중인 블록의 경우 하나의 워드만으로도 경계 태그와 똑 같은 효과를 발휘하는 변형된 알고리즘이 개발되었고[18], 최근 들어 그 중요성이 인식되면서 널리 사용되고 있다[8, 9, 12].

따라서 본 논문에서는 작은 크기의 메모리 요구에 대해 exact-fit 전략을 사용함으로써 높은 메모리 사용 효율성을 제공하는 빠른-적합의 장점과, 큰 크기의 메모리 요구에도 고정된 WCET을 보장하는 절반-적합의 장점을 이용하는, 실시간 시스템용 *quick-half-fit(QHF)* 알고리즘을 다음 절에서 제안한다.

3. Quick-Half-Fit(QHF) 동적 메모리 할당 알고리즘

앞서 언급한 바와 같이 빠른-적합은 $MinQL \le s \le MaxQL$ 크기 구간 내에 존재하는 모든 메모리 블록들에 대해 각 크기 s(워드 단위)별로 별도의 quick 리스트를 관리한다. 이는 자주 요청되는 여러 작은 블록들에 대해서 크기별 quick 리스트를 두어 보다 완전-적합(exact-fit) 전략을 사용하기 위함이다. $MaxQL$보다 큰 블록들은 크기에 상관없이 하나의 단일 리스트로 관리하며, 이 리스트를 예외(*miscellaneous*) 리스트라 한다. 따라서 큰 블록의 메모리 요청 시는 여전히 단일 프리 리스트와 동일한 시간 복잡도인 $O(n)$을 가진다[10, 11].

따라서 본 절에서는 작은 블록에 대해서는 빠른-적합과 동일한 전략을 사용하고, 큰 블록에 대해서는 여러 개의 예외 리스트들을 사용하는 quick-half-fit(QHF)이라 불리는 새로운 동적 메모리 할당 알고리즘을 제안한다.

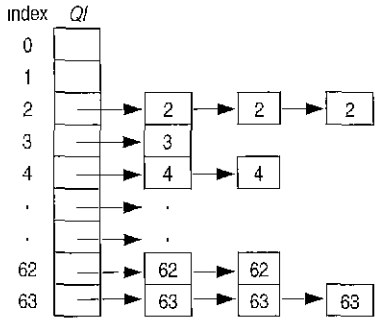
3.1 프리 블록(free blocks)의 관리

QHF는 빠른-적합과는 달리 하나의 예외 리스트를 가지는 것이 아니라, 크기 범위를 가지는 여러 개의 예외 리스트들을 가진다. (그림 1)은 $MinQL$이 2이고, $MaxQL$이 63이며, 크기 단위가 워드인 QHF의 프리 블록 관리 구조이다. QHF는 프리 리스트들을 관리하기 위해 하나의 긴 색인용 배열을 가진다. 이 색인용 배열은 다시 quick 리스트들을 위한 QI와 예외 리스트들을 위한 MI로 나눌 수 있다.

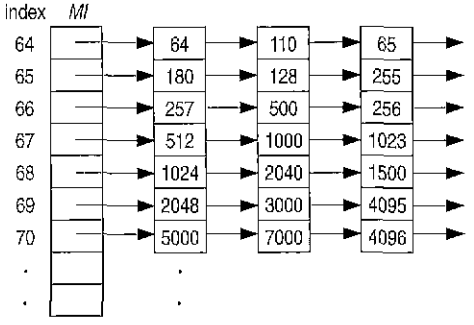
$MaxQL$보다 같거나 작은 소형 블록들은 배열 QI에 의해 포인팅(pointed) 되는 quick 리스트들에 연결되며, 따라서 크기가 s(워드 단위)인 블록은 s를 색인으로 사용하는 quick 리스트에 연결되어 관리된다. 하나의 quick 리스트는 항상 동일 크기의 블록만을 가진다[(그림 1(a))].

$MaxQL$보다 큰 대형 블록들은 배열 MI에 의해 포인팅 되는 예외 리스트들에 저장되며, 각 예외 리스트는 다양한 크기의 블록을 포함할 수 있지만 포함 가능한 블록의 크기 범위를 가진다. 크기 범위 체계로는 2의 거듭제곱을 사용한다. 즉, 크기 s(워드 단위)가 $2^i \le s < 2^{i+1}$인 프리 블록들은 $i+SMI$(그림에서는 $i+58$)를

색인으로 사용하는 예외 리스트 i에 관리된다[(그림 1(b))]. SMI는 가상적으로 MI가 시작되는 색인을 의미하며, $MaxQL - 1 - log_2(MaxQL+1)$값을 가진다. 만약 $MaxMem$이 시스템이 접근할 수 있는 가장 큰 메모리 주소 값(즉, 사용자가 요구할 수 있는 가장 큰 메모리 크기)이라 할 때, 요청 가능한 메모리 블록의 크기 s는 $MaxQL < s \leq MaxMem$ 범위의 값을 가질 수 있다 크기가 s인 블록이 연결되어야 할 예외 리스트의 색인을 위한 i 값은 $\lfloor log_2 s \rfloor$ 공식을 이용하여 계산할 수 있다. 이 경우 한 워드 당 비트 수가 $wordlength$라 가정할 때, i는 $\lfloor log_2(MaxQL+1) \rfloor \leq i \leq (wordlength-1)$ 사이의 값을 가질 수 있으며, 이는 곧 최대로 가질 수 있는 예외 리스트 수가 시스템 워드의 비트 수임을 의미한다.



(a) quick 리스트들



(b) 예외 리스트들

$MinQL = 2$, $MaxQL = 63$, size unit : word

(그림 1) QHF의 프리 블록 관리 구조

일반적으로 예외 리스트 개수는 한 워드의 비트 수를 넘지 못하므로 사용하는 시스템에 의해 사전에 결정된다. 그러나 $MaxQL$에 대한 가장 적합한 값은 응용 프로그램의 메모리 요구 분포에 따라 달라진다 따라서 현재 수행되는 응용 프로그램의 메모리 요구 분포 변화에 따라 동적으로 $MaxQL$을 결정할 수 있다면, 가장 효율적으로 메모리를 관리할 수 있다 그러나 참고 문헌 [7]은 실험을 통하여 이러한 전략을 사용하여 얻어지는 이득보다는 요구 분포에 대한 동적인 정보 관리에 드는 부하가 더 크다고 밝혔다. 따라서 $MaxQL$은 시스템 구성 시 사전에 결정되어져 할 구성 요소(configuration factor)이다. 비록 실시간 시스템 응용 프로그램은 아닐지라도 다양한 범용 응용 프로그램의 메모리 요구 분포를 분석한 결과, 대부분의 메모리 요구 크기는 30-40개의 워드 이하라고 많은 연구에서 보고되고 있다[6-12]

반환한 메모리 블록이나, 합병한 결과 생성된 새로운 블록, 또는 분할하고 남은 블록들은 다시 프리 리스트에 삽입되어야 하며, 이때 사용하는 블록 삽입 전략은 다음과 같다.

1. 블록의 크기 s가 $MaxQL$보다 같거나 작은 경우, s를 색인으로 사용하는 quick 리스트의 첫 위치에 연결한다.
2. 블록의 크기 s가 $MaxQL$보다 큰 경우, $\lfloor log_2 s \rfloor + SMI$ 값을 색인으로 사용하는 예외 리스트의 첫 위치에 연결한다.

소형 블록인 경우 블록 크기 값 자체를 색인으로 사용하여 해딩 quick 리스트를 바로 찾을 수 있지만, 대형 블록의 경우 해당 예의 리스트의 색인을 구하기 위해선 log 계산을 해야 한다. 이를 위한 일고리즘을 비트 탐색 알고리즘이라 하며, 이에 대한 설명과 시간 분석은 3.4절과 3.6절에서 논의한다.

메모리 요청 시 할딩하고자 하는 프리 블록은 내부 조각화를 방지하기 위해 분할 가능한지 확인하고 필요한 경우 분할 작업을 해야 하는데, 이때 사용하는 블록 분할 전략은 다음과 같다

1 요청한 크기 r를 할당하기 위해 사용할 프리 블록의 크기가 s이고 min_block_size기 할당 가능한 최소의 블록 크기라고 할 때, $s \geq r + min_block_size$이면 과정 3을 수행한다.
2 블록 분할 없이 완료한다.
3 할당하고자 하는 프리 리스트의 마지막 블록을 해당 리스트로부터 제거한 후, 이 블록을 다시 r

크기와 $(s-r)$ 크기를 가진 두개의 블록으로 분할한다.

4. $(s-r)$ 크기를 가진 블록은 블록 삽입 전략에 따라 다시 적절한 프리 리스트에 삽입하고, 나머지 블록은 사용자에게 반환한다.

3.2 메모리 블록의 할당(allocation)

QHF는 프리 리스트 상의 적합한 크기의 블록을 찾기 위한 순차적 탐색을 제거하기 위해 특별한 메모리 할당 전략을 사용한다. s가 사용자가 요구한 메모리 크기라고 가정할 때, 블록 할당 전략은 다음과 같다

1. 만약 s가 $MaxQL$ 보다 같거나 작을 경우, s를 색인 idx의 값으로 설정한다.
   만약 s가 $MaxQL$ 보다 클 경우, $(i+SMI)$를 색인 idx의 값으로 설정한다. 예외 리스트 $(i-SMI)$는 $2^i \le sz < 2^{i+1}$ 사이의 다양한 크기를 가지는 프리 블록을 관리한다. 그러나 예외 리스트상에서의 적절한 크기의 블록을 가려내기 위한 별도의 탐색 작업을 하지 않기 위해, 예외 리스트 $(i+SMI)$는 $2^{i-1} < s \le 2^i$ 크기의 메모리 요청이 있을 경우에만 사용한다. 이 경우 리스트 내에서 별도의 순차적 탐색 작업 없이 그 리스트상의 어떤 블록이든 항상 메모리 요청을 만족할 수 있다. 따라서 i 값은 다음의 공식에 의거해서 비트 탐색 알고리즘을 적용하여 구한다.

$$i = \lfloor log_2(s-1) \rfloor + 1$$

2. idx에 의해 색인되는 리스트가 빈 리스트가 아니라면, 과정 4를 수행한다.
3. *Non-empty* 리스트 탐색 알고리즘을 이용해 idx 보다 큰 색인을 가지는 리스트들 중 프리 블록을 가진 최초의 non-empty 프리 리스트(quick 리스트 또는 예외 리스트)를 찾는다
4. 블록 분할 전략을 적용하고, 사용자에게 요청된 메모리를 할당한다.

블록 할당 전략 초기에 요청된 크기에 가장 적합하다고 판단하여 찾은 프리 리스트가 빈 리스트일 경우, 이보다 큰 블록을 가진 non-empty 프리 리스트들 중 가장 작은 크기의 블록을 가진 프리 리스트를 찾아야 한다. 이때 리스트가 빈 리스트인지 아닌지 판단하기 위한 순차적 리스트 검색을 지양하기 위해 QHF는 non-empty 리스트 탐색 알고리즘을 사용한다. 이 알고리즘에 대해서는 3.5절에서 논의한다.

3.3 메모리 블록의 반환(deallocation)

QHF는 실시간 특성을 고려하여 서로 인접한 프리 블록끼리의 합병은 즉시 합병 방법을 사용하며, 이는 블록 반환 시 수행된다. 블록 반환 전략은 다음과 같다.

1. 반환된 블록과 이웃한 프리 블록이 있는지 조사하고, 만약 존재하지 않다면 반환된 블록을 블록 삽입 전략에 따라 해당 프리 리스트에 삽입한다.
2. 인접한 프리 블록이 있다면 즉시 합병 작업을 수행하고, 합병된 새로운 블록은 블록 삽입 전략에 따라 해당 프리 리스트에 삽입한다.

즉시 합병을 위해 인접한 블록을 상수 시간 안에 찾고 또한 그 블록이 프리 블록인지 아닌지 판단하기 위해 QHF는 참고 문헌 [18]에서 제시된 개선된 경계 태그(boundary tags) 기술을 사용한다. 또한 합병될 프리 블록을 합병하기 전에 기존의 프리 리스트에서 탐색 없이 바로 삭제할 수 있게 이중 연결 구조를 사용하여 리스트를 관리하도록 한다.

3.4 비트 탐색(bit search) 알고리즘

앞서 언급한 바와 같이 소형 블록의 경우 블록 크기 값 자체를 색인으로 사용하여 해당 quick 리스트를 바로 찾을 수 있지만, 대형 블록의 경우 해당 예외 리스트의 색인을 구하기 위해선 log 계산을 해야 한다. 만약 시스템 CPU가 레지스터의 MSB(most significant bit)에서 LSB(least significant bit) 방향으로 처음으로 0이 아닌 1로 설정된 비트의 위치를 찾아 주는 비트 탐색(bit search) 명령어를 지원한다면, 한 명령어 주기 내에 $\lfloor log_2 s \rfloor$를 쉽게 구할 수 있다. 그러나 모든 CPU가 비트 탐색 명령어를 지원하는 것이 아니고 또한 이 명령어를 사용할 경우 H/W 의존적이 될 수 있으므로, QHF는 S/W적인 방법인 비트 탐색 알고리즘을 사용한다.

가장 쉬운 방법으로는 비트 이동(bit shift) 명령을 이용하여 바트 벡트(블록의 크기 값)의 매 비트 값을 비교하는 것이지만, QHF에서는 비트 벡트를 색인으로 사용하는 비트 위치 테이블을 이용한다. 예로 <표 1>은 비트 벡트가 4비트라고 가정할 때 이를 위한 비트 위치 테이블이다. 이 경우 만약 블록의 크기(비트 벡트

값)가 10이라면, 이 테이블을 참조하여 최초의 0이 아닌 비트 위치가 네 번째 비트임을 쉽게 찾을 수 있다. 그러나 이 방법은 비트 벡트의 비트 수가 커질 경우 비트 변환 테이블을 위한 과도한 메모리 오버헤드로 인해 사용할 수 없는 한계를 가진다.

〈표 1〉 가장 왼쪽 1의 위치를 찾는 비트 위치 테이블 (4 bits 기준)

색인(이진수)	비트 위치	색인(이진수)	비트 위치
0(0000)	0	8(1000)	4
1(0001)	1	9(1001)	4
2(0010)	2	10(1010)	4
3(0011)	2	11(1011)	4
4(0100)	3	12(1100)	4
5(0101)	3	13(1101)	4
6(0110)	3	14(1110)	4
7(0111)	3	15(1111)	4

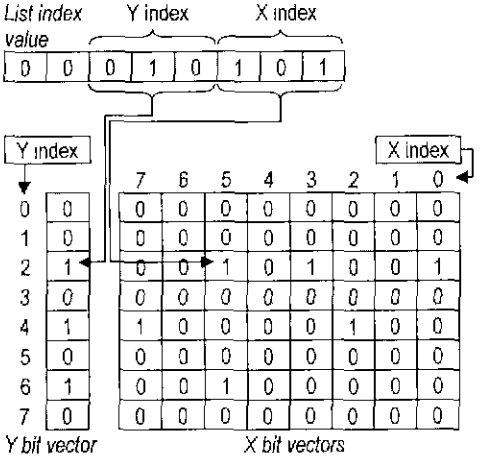
따라서 QHF에서는 8 비트 기준 비트 위치 테이블을 사용하고, 비트 벡트를 한 바이트씩 순서적으로 처리한다. 비트 벡트의 매 바이트별로 그 값이 0인지 아닌지 비교하고, 0인 경우 기준 위치(초기값 0) 값에 8을 더하고 비트 벡트의 다음 바이트를 비교한다. 만약 바이트 값이 0이 아닌 경우, 비트 위치 테이블에서 해당 비트 위치를 구한 다음 여기에 기준 위치 값을 더해서 비트 위치를 구한다. 한 워드의 크기가 32 비트(4 bytes)인 경우 최악의 경우(블록의 크기가 256보다 작은 경우) 3번의 비교가 필요하고 한번의 비트 위치 테이블 접근이 필요하다.

3.5 Non-empty 리스트 탐색 알고리즘

블록 할당 전략 초기에 요청된 크기에 가장 적합한 프리 리스트를 찾았으나 이 리스트가 빈 리스트일 경우, 이 보다 큰 프리 블록을 가진 non-empty 프리 리스트들 중 가장 작은 크기의 블록을 가진 프리 리스트를 찾아야 한다. 이를 위한 방법으로 리스트가 빈 리스트인지 아닌지 오름차순으로 순차적으로 리스트를 검색해 볼 수 있다. 그러나 이 방법은 리스트의 개수가 많아 질 경우, WCET이 커질 뿐 아니라 수행 시간 편차(jitter) 역시 커지는 문제가 있다. 또 다른 방법으로는 각 리스트별 하나의 대응 비트(bit)를 가지는 비트 벡트(bit vector)를 관리하면서 비트 탐색 알고리즘을 활용하는 방안을 들 수 있다. 이 방법은 이진 버디 시스템이나 절반-적합 알고리즘에서와 같이 리스트의 개수가 최고 *wordlength*인 경우에는 사용 가능하나, QHF와 같이 리스트 수 ($MaxQL + wordlength + 1$)가 많아 질 경우 WCET이 고정되지 않는 문제점이 있다.

따라서 QHF는 단계별 비트 벡트와 맵핑 테이블을 사용함으로써 고정된 WCET을 보장하는 non-empty 리스트 탐색 알고리즘을 사용한다. 이 알고리즘은 프리 블록을 가지는 새로운 리스트가 생성될 때마다 해당 리스트의 색인 값을 색인으로 사용하여 단계별 비트 벡트의 상응하는 비트를 1로 설정하고, 이후 그 리스트가 empty일 경우 상응하는 비트들을 다시 0으로 설정한다. 이후 설정된 단계별 비트 벡트를 색인으로 사용하여 맵핑 테이블에서 non-empty 리스트들 중 가장 작은 크기를 가지는 프리 리스트의 색인을 찾을 수 있다.

예를 들어, (그림 2)는 64개의 프리 리스트를 가지는 시스템에서 사용되는 두 단계 비트 벡트 구조이며, <표 2>는 이때 사용되는 맵핑 테이블이다. 그림에서 8비트로 구성된 하나의 Y 비트 벡트와 이 벡트의 각 비트별로 상응하는 8개의 X 비트 벡트들이 존재한다. 각 X 비트 벡트 역시 8비트로 구성된다. 프리 리스트의 색인 값은 0에서 63까지 가질 수 있으므로 6비트간 사용되며, 이중 상위 3비트는 Y 비트 벡트의 색인으로 사용되고, 하위 3비트는 X 비트 벡트의 색인으로 사용된다.



(그림 2) 두 단계 비트 벡트 구조

만약 그림에서와 같이 현재 리스트 색인 값으로 21을 가지는 프리 리스트가 있다면, 상위 3비트의 값은 2이고, 하위 3비트의 값은 5이다. 따라서 이 색인 21에 상응하는 Y 비트 벡트의 상응하는 비트는 3번째(2를 색인으로 사용하는) 비트이고, X 비트 벡트의 상응하

는 비트는 3번째(2를 색인으로 사용하는) X 비트 벡트의 6번째(5를 색인으로 사용하는) 비트이다. 새로운 non-empty 리스트가 생성될 때 해당 Y 비트 벡트와 X 비트 벡트의 상응하는 비트를 1로 설정하고, 기존의 non-empty 리스트가 empty 리스트로 될 때 해당 X 비트 벡트의 상응하는 비트를 다시 0으로 설정한다. Y 비트 맥토의 비트는 상응하는 X 비트 벡트의 값이 0일 경우(즉, X 비트 벡트의 모든 비트가 전부 0일 경우) 0으로 설정된다. 그림에서는 16, 19, 21, 34, 39, 그리고 53을 색인으로 가지는 프리 리스트가 현재 non-empty 리스트로 설정되어 있다.

〈표 2〉 가장 오른쪽 1의 비트 위치를 찾는 맵핑 테이블 (8 bits 기준)

색 인	비트 위치	색 인	비트 위치
0(00000000)	0	…	..
1(00000001)	0	84(01010100)	2
2(00000010)	1	…	…
3(00000011)	0	248(11111000)	3
4(00000100)	2	249(11111001)	0
5(00000101)	0	250(11111010)	1
6(00000110)	1	251(11111011)	0
7(00000111)	0	252(11111100)	2
…	…	253(11111101)	0
41(00101001)	0	254(11111110)	1
..	…	255(11111111)	0

문제는 이 중에서 가장 작은 크기의 프리 블록을 가지는 프리 리스트 16을 어떻게 찾아 내느냐 이다. 우선 현재의 Y 비트 벡트 값은 84(01010100)이므로, 이를 색인으로 사용하여 <표 2>의 맵핑 테이블에서 해당하는 맵핑 값 2를 찾는다. 이는 곧 가장 작은 크기를 가지는 프리 리스트의 색인 값에서 상위 3비트 값이 2라는 것을 의미한다. 또한 맵핑 값 2(즉, Y 비트 벡트의 색인 2)에 상응하는 X 비트 벡트의 값 41(00101001)을 색인으로 사용하여 맵핑 테이블에서 해당하는 맵핑 값 0을 얻는다. 이는 곧 가장 작은 크기를 가지는 프리 리스트의 색인 값에서 하위 3비트 값이 0이라는 것을 의미한다. 따라서 Y 비트 벡트 맵핑 값 2(010)와 X 비트 벡트 맵핑 값 0(000)을 이용하여 16(00.010 000)이라는 값을 얻을 수 있다.

이상에서 기술한 예제는 현재 설정된 프리 리스트들 중에서 가장 작은 크기의 프리 블록을 가지는 리스트를 찾는 방법이다. 그러나 QHF에서는 블록 할당 시 비트 탐색 알고리즘을 이용하여 요구된 크기에 가장 적합하다고 판단하여 찾은 프리 리스트가 빈 리스트일 경우, 이 보다 큰 프리 블록을 가진 프리 리스트들 중 가장 작은 크기의 블록을 가진 non-empty 프리 리스트를 찾는 경우이다. 이를 위해서는 상기 설명한 방법을 응용하여 X 비트 벡트를 매스킹(masking)한 후 맵핑 테이블을 참조하면 된다. 만약 64개의 리스트보다 많은 리스트를 관리해야 할 경우, 두 단계 비트 벡트 대신 세 단계 비트 벡트 구조를 사용하면 된다.

Non-empty 리스트 탐색 알고리즘은 맵핑 테이블과 단계별 비트 맥트를 관리해야 하는 부하를 가지지만, 항상 고정된 상수 시간 내(WCET 고정)에 원하는 non-empty 리스트를 찾아 주는 장점을 제공한다. 따라서 이 알고리즘은 non-empty 리스트를 찾는 유사한 전략을 사용하는 이진 버디 시스템이나 결합-적합 알고리즘 등에도 마찬가지로 적용할 수 있다

3.6 최악의 경우 실행 시간(WCET) 분석

일반적으로 실시간 시스템에 DSA를 적용하기 위해서는 시간 복잡도가 $O(1)$이어야 하고, WCET이 보다 쉽게 측정 가능해야 한다. 이진 버디 시스템의 경우 비록 시간 복잡도는 $O(1)$일지라도 WCET 계산이 쉽지 않다. 이유는 블록 분할 및 합병 시 매번 최악의 경우 워드의 비트 수 만큼 분할 및 합병을 수행해야 하며, 이 과정에서 각 프리 리스트에 블록의 삽입/삭제를 위한 메모리 전반에 걸친 광범위한 접근이 이루어지기 때문이다. 이는 곧 빈번한 캐시 및 TLB misses를 유발한다.

반면에 QHF는 합병할 블록이 최대 3개이고, 분할도 한번만 수행한다. 그리고 고정된 수행 시간을 보장하는 non-empty 리스트 탐색 알고리즘을 사용하므로, 이진 버디 시스템보다 쉽게 WCET을 측정할 수 있다. 따라서 QHF 알고리즘에서 WCET에 영향을 미치는 요소는 예외 리스트의 색인을 찾기 위한 $\lfloor \log_2(s-1) \rfloor$ 계산이다. 이는 비트 탐색 알고리즘을 이용하여 구하므로, 궁극적으로 QHF의 WCET은 비트 탐색 알고리즘에 의해 결정된다

비트 탐색 알고리즘은 8비트 기준 비트 위치 테이블을 사용하고, 블록 크기 값(비트 벡트)를 한 바이트씩 순서적으로 0인지 아닌지 체크하고, 0이 아닐 경우 최종적으로 비트 위치 테이블을 참조한다. 예를 들어, 블록 크기 값을 한 워드에 저장하고 워드의 크기가 32비트(4 bytes)라고 가정한다면, 최악의 경우(블록의 크

기가 256보다 작은 경우) 3번의 비교가 필요하고 한번의 비트 위치 테이블 접근이 필요하다. 따라서 이 알고리즘의 WCET은 한 워드의 길이에 의존하고, 이는 시스템 구성 시 사전에 결정되는 값이므로 비트 탐색 알고리즘의 시간 복잡도는 궁극적으로 $O(1)$이다. WCET은 블록의 크기가 256보다 작은 경우에 발생하므로 쉽게 측정이 가능하다.

4. 시뮬레이션

QHF와 절반-적합 그리고 이진 버디 시스템의 메모리 사용 효율성을 비교하기 위해, 본 연구팀은 전체 메모리 용량을 제한한 채 인위적으로 생성된 메모리 할당 및 반환 패턴을 사용하여[1, 11, 27], WCET 및 조각율, 그리고 메모리 할당 실패율 등을 측정하는 몇 가지 실험을 실시하였다. 본 절에서는 실험에 사용된 시뮬레이션 방법을 기술하고, 이 방법에 의해 실시된 실험 결과를 제시한 후 이를 분석한다.

4.1 시뮬레이션 방법

실시간 시스템용 DSA 알고리즘을 비교하기 위해 본 연구에서 사용하는 평가 기준은 메모리 할당 및 반환에 소요되는 WCET, 내부 조각율, 외부 조각율, 총 조각율, 할당 실패율 등이다. WCET은 알고리즘이 실제 구현된 시스템에서의 그 알고리즘의 최악의 경우 실행 시간이다. 이는 알고리즘 측면의 시간 복잡도 $O(1)$을 만족하는 알고리즘이 실제 구현된 시스템에서 구현 측면의 WCET을 일정하게 유지하고, 이를 쉽게 측정할 수 있느냐를 판단하기 위함이다.

조각율(fragmentation ratio)은 DSA 알고리즘이 적용되는 응용 프로그램, 라이브러리, 운영체제와의 상관관계에 따라 다양하게 정의될 수 있다[9]. 본 실험에서는 실시간 시스템의 특성에 가장 적합한 참고 문헌 [7]의 방법을 따르기로 했다. 따라서 내부 조각율(internal fragmentation ratio)은 $IF = A / R$로 정의한다. 여기서 A는 DSA 알고리즘이 할당한 총 메모리 양이고, R은 사용자가 요청한 총 메모리 양이다. 또한 외부 조각율(external fragmentation ratio)은 할당이 실패할 때마다 측정되며, $EF = M / A$로 정의한다 여기서 M은 실험에 사용된 총 메모리 용량(사전에 고정)이고, A는 할당이 실패할 때까지 할당된 총 메모리 양이다. 각 실험에서는 여러 번의 할당 실패가 발생할 수 있으며, 그 때마다 외부 조각율을 계산하여 이를 평균한 값이 그 실험에서의 외부 조각율이다 총 조각율(total fragmentation ratio)은 내부 조각율과 내부 조각율로 인해 발생하는 메모리 손실로서, 다음과 같이 정의한다.

$$TF = IF * EF = (A / R) * (M / A) = M / R$$

따라서 총 조각율은 총 메모리 용량과 사용자가 요청한 순수한 메모리 양과의 상대적인 비율로 나타난다. 이는 곧 DSA 알고리즘이 사용자가 요청한 메모리 양보다도 메모리 관리를 위해 얼마 만큼의 메모리 용량을 더 필요로 하는가를 의미하며, DSA 알고리즘의 메모리 사용 효율성을 나타내는 지표이기도 하다

할당 실패율(allocation failure ratio)은 $AF = FN / RN$로 정의한다. 여기서 RN은 사용자가 요청한 총 할당 회수이고, FN은 할당에 실패한 총 회수이다.

기존의 연구에 따르면 메모리 할당 시뮬레이션은 요구된 할당 크기(sizes of allocation requests), 할당된 블록의 시스템 내의 존속 기간(lifetimes), 그리고 할당 요구가 들어오는 시간 간격(arrival intervals) 등 세가지 분포에 상당한 영향을 받는다[27]. 실제 응용 프로그램의 메모리 요구 패턴과 가능한 유사한 패턴을 반영하는 실험 결과를 도출하기 위해서, 이러한 분포를 신중하게 선택하는 것이 중요하다

본 연구에서는 참고 문헌 [1]과 같이 요구된 할당 크기 분포를 위해 지수 분포와 균등 분포 등 두 가지 확률 분포를 사용한다. 크기 분포의 평균은 8, 10, 12, 14, 16, 32, 64, 128, 256, 512, 1024, 2048 워드로 다양하게 변화 시켰다. 그러나 이때 사용되는 총 메모리 용량은 항상 32K 워드로 고정 시켰다. 할당된 메모리 블록의 시스템 내의 존속 기간 분포는 균등 분포를 따르며, 5에서 15 사이의 시간 단위(time units) 범위 내에서 존속하며 평균은 10시간 단위이다. 메모리 요구 도착 시간 간격 분포는 지수 분포를 따른다. 따라서 시뮬레이션은 $M/G/\infty$ 큐잉(queuing) 모델을 따르며, 시스템 내에 존속하는 할당된 메모리 블록의 개수는 평균 값 λ/μ를 가지는 포아송(Poisson) 분포를 따른다. 여기서 $1/\lambda$는 요구 도착 시간 간격 분포의 평균 값이며, $1/\mu$는 할당된 블록의 존속 기간 분포의 평균 값이다

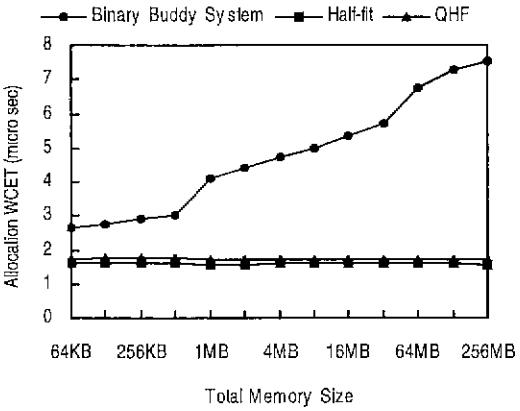
실험에 사용되는 총 메모리 용량(32K)은 항상 고정되어 있으므로, 평균 할당 크기가 결정되면 시스템 내에 존속하는 할당된 메모리 블록의 평균 개수 λ/μ 값

(총 메모리 용량/평균 할당 크기)을 결정할 수 있다. 평균 존속 기간 $1/\mu$(10) 또한 고정된 값이므로 평균 요구 도착 시간 간격 값 $1/\lambda$를 결정할 수 있다.

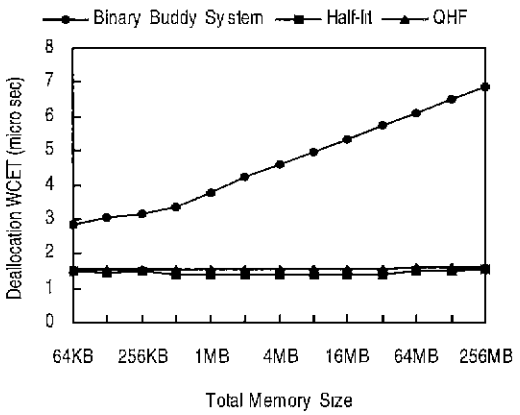
4.2 시뮬레이션 결과

앞서 기술한 큐잉 모델에 따라 각 크기별 메모리 할당/반환 패턴을 생성한 후, QHF, 절반-적합, 이진 버디 시스템의 메모리 할당 및 반환에 소요되는 최장 실행 시간(WCET), 내부 조각율, 외부 조각율, 총 조각율, 할당 실패율 등을 측정하였다. 각 DSA 알고리즘이 사용하는 메모리 블록의 헤드는 모두 동일한 구조를 가진다. 프리 블록 헤드는 프리 블록을 관리하기 위해 필요한 최소한의 데이터 구조로서 이중 연결 포인터(pointers for double linked list)와 두 개의 경계 태그(두개의 블록 크기 값)로 구성된다. 프리 블록 헤드의 크기는 할당 가능한 최소 블록 크기(*min_block_size*)를 결정한다 실험에서는 4개의 워드(16 bytes)가 프리 블록 헤드로 사용된다. 그러나 본 연구에서는 개선된 경계 태그 기술[18]을 사용하므로, 할당된 블록 헤드의 구조는 프리 블록 헤드와는 달리 이중 연결 포인터와 경계 태그 중 하나만(하나의 블록 크기 값) 사용한다. 할당된 블록의 주소는 이중 실수 값을 고려하여 항상 8 bytes 배수 주소 값으로 조정 시킨다. 본 연구에서는 주소 조정과 블록 헤드를 위해 별도로 더 할당된 메모리 공간을 내부 조각율 계산에 모두 포함시켰다. 또한 QHF가 사용하는 *MinQL*은 63 더블 워드(8 bytes)이며, 총 프리 리스트 개수는 87개이다. 빠른-적합과 이진 버디 시스템이 관리하는 프리 리스트 개수는 32개이다

(그림 3)은 총 메모리 용량별 세 DSA 알고리즘의 메모리 할당에 소요된 최장 실행 시간을 보인 것이고, (그림 4)는 할당된 메모리 블록의 반환에 소요된 최장 실행 시간을 보인 것이다. 실험은 UltraSPARC 2(200MHz) 프로세서를 탑재한 SUN UltraSPARC 워크스테이션에서 실행되었으며, 그림은 1,000,000을 수행하여 측정한 (*gettimeofday()* 시스템 함수 이용) 최장 실행 시간들의 평균 값을 보인 것이다 이는 메모리 할당/반환에 소요된 WCET을 측정하기 위한 일환으로 실시된 실험으로 실제 케시가 미치는 영향을 반영되지 않았다. 테스트 프로그램의 코드 및 데이터 영역을 *mlock()* 함수를 통해 locking 함으로써, 가상 기억 장치의 영향력은 배제되었다. 이 실험은 메모리 할당/반환의 진정한 WCET을 측정하기 보다는 비교하고자 하는 세 알고리즘이 메모리 크기에 상관없이 동일한 최장 실행 시간을 보이는가를 측정하기 위함이다



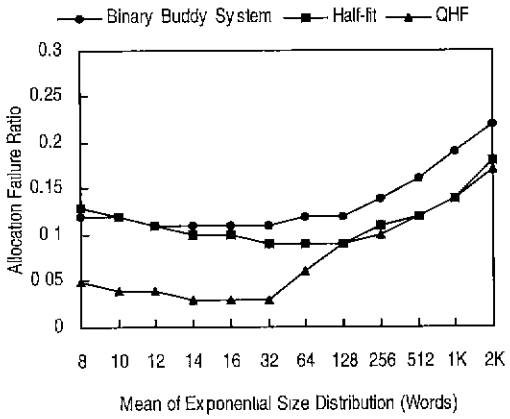
(그림 3) 메모리 할당에 소요된 최장 실행 시간



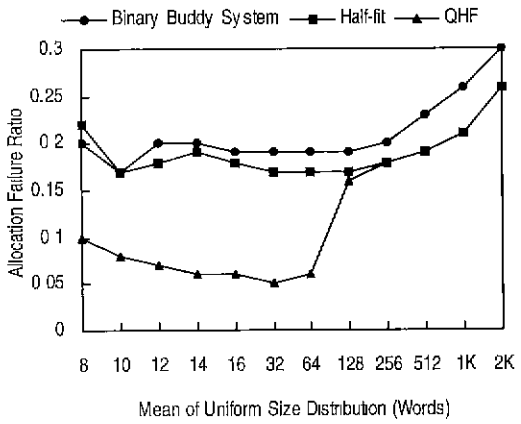
(그림 4) 메모리 반환에 소요된 최장 실행 시간

두 그림에서 QHF와 절반-적합은 총 이용 가능한 메모리의 용량에 상관없이 동일한 최장 실행 시간을 보인 반면 이진 버디 시스템은 메모리 용량이 증가함에 따라 최장 실행 시간도 함께 증가함을 볼 수 있다 이유는 이진 버디 시스템에서 블록 분할 및 합병 시 매번 최악의 경우 32번(워드의 비트 수)의 분할 및 합병을 수행해야 하며, 이 과정에서 각 프리 리스트에 블록의 삽입/삭제를 위해 메모리 전반에 걸친 광범위한 접근이 이루어지기 때문이다 이는 또한 알고리즘 측면의 시간 복잡도가 비록 $O(1)$일지라도 알고리즘이 실제 구현된 시스템에서 구현 측면의 WCET은 일정하게 유지되지 않는다는 것을 보이고 있다. 그림에서

QHF의 최장 실행 시간이 절반-적합보다 상대적으로 약간 높게 나타나는 것은 관리해야 하는 프리 리스트의 수가 절반-적합보다는 QHF가 상대적으로 많기(3배) 때문인 것으로 판단된다.

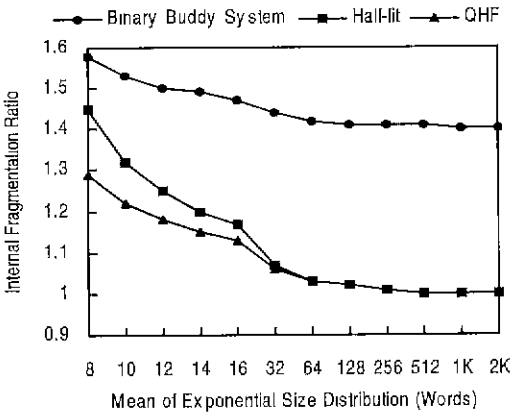


(그림 5) 지수 분포 때의 할당 실패율

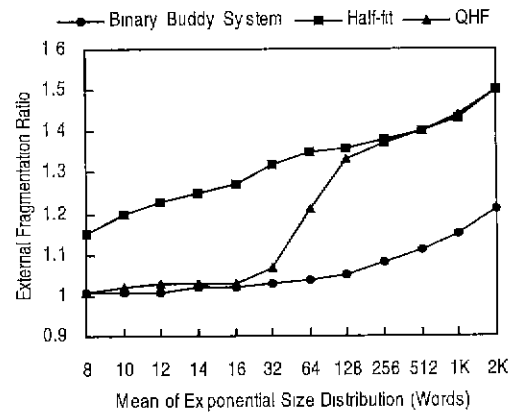


(그림 6) 균등 분포 때의 할당 실패율

(그림 5)는 할당 크기 분포가 지수 분포를 가질 때, (그림 6)은 균등 분포를 가질 때, 각 DSA 알고리즘의 평균 할당 크기별 할당 실패율을 보인 것이다. 두 그림 모두 할당 크기 분포에 상관없이 유사한 할당 실패율을 보임을 확인할 수 있다 Hal-fit과 이진 버디 시스템의 할당 실패율은 참고 문헌 [1]의 실험 결과와 유사한 수치를 보인다. QHF는 전 구간에 걸쳐 이진 버디 시스템보다 상대적으로 낮은 실패율을 보인다. 반면, 절반-적합은 평균 크기가 커질수록 QHF와 유사한 실패율을 보이고, 평균 크기가 작을수록 이진 버디 시스템과 비슷한 실패율을 보인다. 이 실험에서 QHF는 할당 크기가 작을 경우 다른 두 알고리즘에 비해 상당히 낮은 할당 실패율을 보임을 확인할 수 있다.



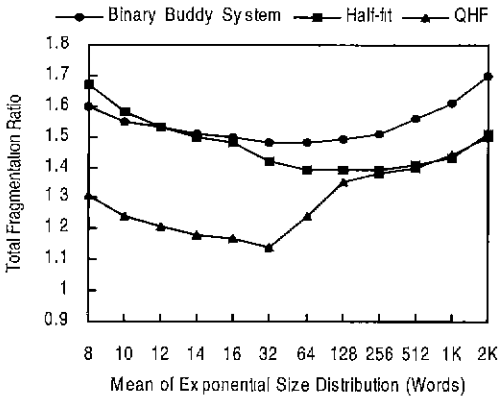
(그림 7) 내부 조각율(지수 분포)



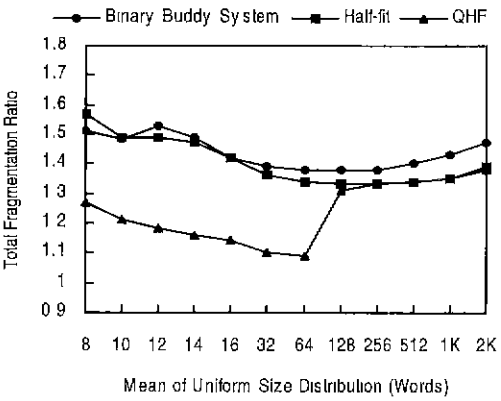
(그림 8) 외부 조각율 (지수 분포)

(그림 7), (그림 8), (그림 9)는 각각 평균 할당 크기별 각 DSA 알고리즘의 내부 조각율, 외부 조각율, 총 조각율을 보인 것이다. 각 그림에서 할당 크기 분포는 지수 분포를 가진다. 이미 예측되는 바와 같이 이진 버디 시스템의 내부 조각율은 QHF와 절반-적합에 비해 상대적으로 높지만, 외부 조각율은 상대적으로 낮다. QHF는 절반-적합에 비해 평균 크기가 커질수록 거의 같은 내/외부 조각율을 보이지만, 평균 크기가 작아질수록 절반-적합 보다 낮은 내/외부 조각율을 보인다. 이는 작은 크기의 메모리 요구에 대해 exact-fit 전략을 사용하고 큰 크기의 메모리 요구에 대해서는 절반-적합 전략을 사용하는 QHF의 특성으로 인한 결과이

다 이러한 현상은 총 조각율 (그림 9)에서 뚜렷이 확인할 수 있다. (그림 9)에서 QHF는 전 구간에 걸쳐 이진 버디 시스템보다 상대적으로 낮은 총 조각율을 보인다 반면, 절반-적합은 평균 크기가 커질수록 QHF와 유사한 조각율을 보이고, 평균 크기가 작을수록 이진 버디 시스템과 비슷한 조각율을 보인다. 할당 크기 분포가 균등 분포를 가지는 경우에도 유사한 패턴을 보임을 (그림 10)에서 확인할 수 있다. (그림 9)와 (그림 10)의 총 조각율은 (그림 5)와 (그림 6)의 할당 실패율과 비슷한 양상을 보인다.



(그림 9) 지수 분포 때의 총 조각율



(그림 10) 균등 분포 때의 총 조각율

할당 실패율과 조각율에서 특이한 사항은 평균 크기가 증가하면서 비율이 감소하다가 일정 크기를 지나면서 다시 증가한다는 사실이다. 이러한 현상은 평균 할당 크기가 작을수록 메모리 블록 관리를 위해 사용하는 블록 헤드 구조체의 부하가 상대적으로 증가한 것으로 판단된다 이러한 블록 헤드 구조체로 인한 부하는 참고 문헌 [9]에서 이미 지적된 사항이기도 하다.

이상의 실험을 종합해 볼 때, QHF는 평균 메모리 요구 크기가 30워드 이상일 경우에는 기존의 절반-적합과 동일한 메모리 관리 효율성을 보인다. 그러나 30워드 이하일 경우엔 조각화로 인한 메모리 손실을 QHF 보다 25-30% 정도 더 줄일 수 있음을 확인할 수 있다. 이는 다양한 응용 프로그램의 메모리 요구 분포가 거의 대부분 30-40 워드 범위 내라는 연구 결과 [6-12]를 고려해 볼 때, QHF가 평균 크기가 작을수록 절반-적합이나 이진 버디 시스템보다 뛰어난 메모리 관리 효율성을 보인다는 것은 의미 있는 연구 결과라 판단된다

5. 결 론

본 논문에서 메모리 사용 효율성이 높고 알고리즘 복잡도가 $O(1)$이면서 예측 가능한 실행 시간을 가진 실시간 시스템용 동적 메모리 할당 알고리즘 QHF를 제안했다. 제안된 알고리즘은 작은 크기의 메모리 요구를 위해서 워드 크기별로 프리 블록 리스트(free block list)를 관리하고, 큰 크기의 메모리 요구를 위해서는 2의 거듭제곱 크기별로 프리 블록 리스트를 관리한다. 이는 작은 크기의 메모리 요구에 대해 exact-fit 전략을 사용함으로써 높은 메모리 사용 효율성을 제공한다. 이러한 전략은 다양한 응용 프로그램의 메모리 요구 크기를 조사한 결과 거의 대부분이 30-40 워드 크기 이하라는 여러 연구 결과를 기반으로 하고 있다.

또한 프리 블록 리스트의 수가 상대적으로 증가되면서 요구된 메모리 블록 크기에 가장 적절한 프리 리스트를 고정 시간 내에 찾는 비트 탐색 알고리즘과 찾은 프리 리스트가 비어 있을 경우 다음 이용 가능한 프리 리스트를 고정 시간 내에 찾는 non-empty 리스트 탐색 알고리즘도 제시하였다. 따라서 많은 프리 리스트를 관리하면서도 쉽게 WCET을 예측할 수 있게 하였다.

제안된 알고리즘의 실용성을 확인하기 위해 절반-적합 알고리즘, 이진 버디 시스템과의 메모리 사용 효율성을 비교하였다. 실험 결과 조각율, 그리고 할당 실패율 등에서 QHF가 절반-적합이나 이진 버디 시스템보다 우수하다는 것을 확인하였다. 특히 평균 메모리 요구 크기가 30워드 이하일 경우엔 조각화로 인한 메모리 손실을 QHF 보다 25-30% 정도 더 줄일 수 있었다.

그러나 본 연구에서 실시한 실험은 전체 메모리 용량을 제한한 채 인위적으로 생성된 메모리 할당/반환 패턴을 사용하였다. 이러한 시뮬레이션 방법은 사용하는 실제 응용 프로그램에 따라 상당히 다른 양상을 나

타낼 수 있다. 따라서 향후 실제 실시간 응용 프로그램의 메모리 할당/반환 패턴을 조사하고, 이를 기반으로 한 실험을 계속할 계획이다

참 고 문 헌

[1] 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.

[2] 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

[3] Ray Ford, "Concurrent Algorithms for Real-Time Memory Management," *IEEE Software*, Vol 5, Issue 5, pp.10-23, 1988

[4] Doug Smith, "Ada Quality and Style : Guidelines for Professional Programmers, Version 02.01.01," Technical Report SPC-91061-CMC, Software Productivity Consortium, Inc , 1992

[5] David B. Stewart, R. A Volpe, and Pradeep K. Khosla, "Design of Dynamically Reconfigurable Real-Time Software Using Port-Based Objects," *IEEE Transactions on Software Engineering*, Vol. 23, No.12, pp 759-776, 1997.

[6] Arun Iyengar, "Parallel Dynamic Storage Allocation Algorithms," *Proc. of 5th IEEE Symposium on Parallel and Distributed Processing*, pp.82-91, 1993.

[7] Arun Iyengar, "Scalability of Dynamic Storage Allocation Algorithms," *Proc. of 6th Symposium on the Frontiers of Massively Parallel Computing*, pp. 223-232, 1996.

[8] Paul R. Wilson, Mark S. Johnstone, Michael Neely, and David Boles, "Dynamic Storage Allocation : A Survey and Critical Review," *Proc. of Intn. Workshop on Memory Management*, pp 1-126, 1995

[9] Mark S. Johnstone and Paul R. Wilson, "The Memory Fragmentation Problem Solved?," *Proc. of Intn. Symposium on Memory Management*, pp 26-36, 1998.

[10] Charles B. Weinstock, "Dynamic Storage Allocation Techniques," Ph.D. thesis, Carnegie Mellon University, Pittsburgh, Pennsylvania, 1976.

[11] Charles B. Weinstock, "Quick Fit : An Efficient Algorithm for Heap Storage Allocation," *SIGPLAN Notices*, Vol.23, No.10, pp.141-148, 1988

[12] Doug Lea, *Implementation of malloc*, See also the short paper on the implementation of this allocator, Available at http : //g.oswego.edu.

[13] Donald E. Knuth, *The Art of Computer Programming. Volume1 : Fundamental Algorithms*, Addition-Wesley, Reading, Massachusetts, 1973.

[14] C. Bay, "A Comparison of Next-fit, First-fit, and Best-fit," *Communications of the ACM*, Vol 20, No. 3, pp.191-192, 1977.

[15] Kenneth C. Knowlton, "A Fast Storage Allocator," *Communications of the ACM*, Vol 8, No.10, pp 623-625, 1965

[16] J. L. Peterson and T. A. Norman, "Buddy Systems," *Communications of the ACM*, Vol.20, No.6, pp 421-431, 1977

[17] J. S. Fenton and D. W. Payne, "Dynamic Storage Allocation of Arbitrary Sized Segments," *Proc. of IFIPS*, pp.344-348, 1974.

[18] Thomas Standish, *Data Structure Techniques*, Addition-Wesley, Reading, Massachusetts, 1980

[19] C. J Stephon, "Fast Fits . New Method for Dynamic Storage Allocation," *Proc of the 9th Symposium on Operating Systems Principles*, pp.30-32, 1983. Also published as *Operating Systems Review*, Vol. 17, No.5, 1983.

[20] R. P. Brent, "Efficient Implementation of the First-Fit Strategy for Dynamic Storage Allocation," *ACM Transactions on Programming Languages and Systems*, Vol 11, No.3, pp 388-403, 1989.

[21] G. Bozman, "The Software Lookaside Buffer Reduces Search Overhead with Linked Lists," *Communications of the ACM*, Vol.27, No 3, pp 222-227, 1984

[22] R R. Oldehoeft and S. J. Allan, "Adaptive Exact-Fit Storage Management," *Communications of the ACM*, Vol.28, No.5, pp.506-511, 1985.

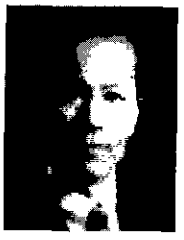
[23] D. Grunwald and B. Zorn, "CustoMalloc : Efficient Synthesized Memory Allocators." *Software Practice and Experience*, Vol 23, No.8, pp 851-869, 1993

[24] D S Hirschberg, "A Class of Dynamic Memory Allocation Algorithms," *Communications of the ACM*, Vol.16, No.10, pp.615-618, 1973.

[25] K. K. Shen and J. L. Peterson, "A Weighted Buddy Method for Dynamic Storage Allocation," *Communications of the ACM*, Vol.17, No 10, pp.558-562, 1974.

[26] David S. Wise, "The Double Buddy-System," Technical Report 79, Computer Science Department, In-

diana University, Bloomington, Indiana, 1978
[27] J. E. Shore, "Anomalous Behavior of the Fifty-percent Rule in Dynamic Memory Allocation," *Communications of the ACM*, Vol.20, No.11, pp. 812-820, 1977.



정 성 무

e-mail : smjung@keris.or.kr
1981년 충남대학교 전자공학과 졸업(학사)
1988년 한양대학교 전자공학과 졸업(석사)
2000년 아주대학교 컴퓨터공학과 박사과정 재학(수료)
1981년~1989년 잠실중, 경기공고 교사
1989년~1997년 한국교육개발원 부연구위원
1997년~현재 한국교육학술정보원 연구위원
관심분야 : Authoring Systems, 시스템 프로그램, 교육정보화 등

유 해 영

e-mail : yoohy@dankook.ac.kr
1979년 단국대학교 수학과 졸업(학사)
1981년 단국대학교 수학과 졸업(석사)
1994년~현재 아주대학교 컴퓨터공학과 박사과정 수료
1983년~현재 단국대학교 전산통계학과 교수
관심분야 : 소프트웨어 공학, 시스템 프로그램 등

심 재 홍

e-mail : jhshim@cesys.ajou.ac.kr
1987년 서울대학교 전산과학과 졸업(학사)
1989년 아주대학교 컴퓨터공학과 졸업(석사)
1989년~1994년 서울시스템 공학연구소
1995년~현재 아주대학교 컴퓨터공학과 박사과정(수료)
관심분야 : 운영 체제, 분산시스템, 실시간 및 멀티미디어시스템

김 하 진

e-mail : hjkimn@madang.ajou.ac.kr
1962년 서울대학교 수학과 졸업(학사)
1978년 프랑스 Grenoble 제1대학 응용수학과 졸업(석사)
1980년 프랑스 Saint-Etienne 대학 응용수학과 졸업(박사)
1974년~현재 아주대학교 정보 및 컴퓨터공학부 교수
관심분야 : 컴퓨터 그래픽스, 수치해석, 멀티미디어 Contents 등

최 경 희

e-mail : khchoi@madang.ajou.ac.kr
1976년 서울대학교 수학교육과 졸업(학사)
1979년 프랑스 그랑제꼴 Enseeiht 대학 졸업(석사)
1982년 프랑스 Paul Sabatier대학 정보공학부 졸업(박사)
1982년~현재 아주대학교 정보 및 컴퓨터공학부 교수
관심분야 : 운영 체제, 분산시스템, 실시간 및 멀티미디어시스템 등

정 기 현

e-mail : khchung@madang.ajou.ac.kr
1984년 서강대학교 전자공학과 졸업(학사)
1988년 미국 Illinois주립대 EECS 졸업(석사)
1990년 미국 Purdue대학 전기전자공학부 졸업(박사)
1991년~1992년 현대반도체 연구소
1993년~현재 아주대학교 전기전자공학부 교수
관심분야 : 컴퓨터구조, VLSI 설계, 멀티미디어 및 실시간 시스템 등