2002년 한국정보처리학회 춘계학술발표논문집 제 9 권 제 1 호

리눅스 실시간 시스템에서의 효율적인 동적 스토리지 할당 알고리즘

이영재, 추현승, 윤희용
성균관대학교 전기전자 및 컴퓨터공학부
E-mail : munchi@ece.skku.ac.kr

Efficient Dynamic Storage Allocation Algorithm for Linux Real-time System

Young Jae Lee, Hyunseung Choo, Hee Yong Youn
School of Electrical & Computer Engineering, Sungkyunkwan University

요 약

동적 메모리 할당 방식은 사전에 그 메모리의 크기를 결정할 수 없는 경우에 효과적인 프로그래밍 기술이다. 하지만 메모리 조각화 문제와 최악의 경우 실행 시간을 예측할 수 없는 단점 때문에 실시간 시스템에는 거의 적용되지 않고 있다. 본 연구에서는 리눅스 기반의 실시간 시스템을 위한 동적 메모리 할당 알고리즘인 QB(Quick-Buddy)를 제안한다. 제안된 알고리즘은 작은 크기의 메모리 요구에 대해서 워드 크기별로 프리 리스트를 관리하고, 큰 크기의 메모리 요구에 대해서는 이진 버디 시스템을 이용하여 관리한다. 이 알고리즘에서는 작은 크기의 메모리 요구에 대해 완전-적합(exist-fit) 전략을 사용하여 메모리 이용 효율을 증가시킨다. 또한 큰 크기의 메모리 요구에 대해서 버디 시스템을 적용함으로써 외부 조각화를 제거시키고 처리량(throughput)을 증가시킨다. 제안된 알고리즘의 성능을 확인하기 위해서 제안된 알고리즘과 이진 버디 시스템(binary buddy system), 빠른-적합(quick-fit)의 메모리 이용 효율성 및 메모리 조각화 율을 비교할 것이다.

1. 서론

전통적인 실시간 시스템에서는 최악의 경우에 실행 시간을 사전에 예측할 수 있고 메모리 조각화가 발생하지 않는 정적 할당(static allocation) 방법을 주로 사용하였다[1,2]. 이것은 메모리 블록의 크기를 고정시키고, 이를 각 태스크에게 사전에 할당 시키거나 시스템 초기화 시에 각 태스크가 사용할 메모리 블록을 미리 할당 받는 방법이다. 이에 비해 동적 메모리 할당(dynamic storage allocation : DSA) 방법은 사전에 그 크기를 결정할 수 없을 경우에 효과적인 프로그래밍 기술이다. 기존의 실시간 시스템에서 정적 할당 방법을 사용한 이유는, 대부분의 경성 실시간 시스템(hard real-time system) 개발자들이 동적 메모리 할당은 메모리 조각화 문제와 최악의 경우 실행시간(worst-case execution time : WCET)을 사전에 예측할 수 없기 때문에 실시간 시스템에 적합하지 않다는 인식을 가지고 있었기 때문이다. 이러한 이유 때문에 실시간 시스템을 위한 동적 메모리 할당 연구가 활성화되지 못하고 있다[1,2].

따라서 본 연구팀은 메모리 이용 효율성이 높고, WCET 을 예측할 수 있는 동적 메모리 할당 알고리즘을 제안한다. 본 논문에서는 작은 크기의 메모리 요구에 대해서 워드 크기별로 프리 리스트를 관리하고, 큰 크기의 메모리 요구에 대해서는 이진 버디 시스템을 이용하여 관리하는 QB(quick-buddy) 방식을 제안한다. 또한 가장 적절한 프리 리스트가 비어 있는 경우에 다음 이용 가능한 프리 리스트를 고정 시간 내에 찾는 알고리즘을 제안한다.

본 논문의 구성은 다음과 같다. 제 2 절에서는 기존에 연구되어 왔던 동적 메모리 할당 알고리즘에는 어떤 것들이 있으며, 실시간 시스템에 적용할 때의 문제점은 없는지 살펴본다. 제 3 절에서는 본 논문에서 제안하는 알고리즘을 설명한다.

2. 동적 메모리 할당 알고리즘들과 연구 배경

본 절에서는 기존의 동적 메모리 할당 알고리즘들을 알아보고, 실시간 시스템에 적용했을 때 생길 수 있는 문제점들을 논의한다.

2.1 기존의 동적 메모리 할당 알고리즘들

2.2.1 순차 적합(sequential fits)

몇몇 고전적인 할당 알고리즘들은 메모리의 모든 프리 블록을 하나의 선형 리스트(single linear list)로 관리한다. 일반적으로, 순차 적합 알고리즘들은 경계 태그(boundary tag) 기술[4]을 이용하며, 합병을 빠르게 하기 위해 이중 연결 리스트(doubly linked list)를 이용한다. 그리고 이러한 프리 리스트들은 이미 잘 알려져 있는 최초-적합(fisrt-fit), 다음-적합(next-fit), 최악-적합(worst-fit), 최적-적합(best-fit) 등의 메모리 할당 전략에 의해 메모리가 할당된다[3].

순차 적합 알고리즘들은 크기 순서에 관계없이 이중 연결 구조로 되어 있기 때문에 순차적으로 탐색해야 하는 문제점이 있다.

2.2.2 버디 시스템(buddy system)

버디 시스템은 제한적이지만 효과적인 분할과 합병을 제공하는 엄격한 분리 적합(strict segregated fit)의 한 종류이다[3,5]. 이진 버디 시스템(binary buddy system)은 2 의 지수 승의 크기를 가지는 프리 블록들을 리스트로 관리한다. 여기에서 하나의 메모리 블록은 버디(buddy)라는 두개의 영역으로 나누어지며, 이 영역들은 또다시 충분히 작은 크기의 블록이 될 때까지 두개의 버디들로 계속해서 분할된다. 해제되어 반환된 메모리 블록들은 동일한 리스트 상에 존재하고 인접한 버디들끼리 합병될 수 있다.

이진 버디 시스템은 외부 조각화(external fragmentation)가 발생하지 않는다는 장점을 가지고 있지만, 반면에 내부 조각화(internal fragmentation) 율이 상대적으로 매우 높다[5,6]는 단점이 있다

2.2.3 빠른-적합(quick-fit)

빠른-적합은 다중 프리 리스트와 단일 프리 리스트를 복합적으로 사용하는 알고리즘이다[3, 7]. 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 40 워드 크기 이하의 메모리 할당을 요청한다는 연구 결과를 이용한 것이다[3,5,7]. 이러한 경우 빠른-적합 방식은 높은 메모리 이용 효율을 가진다.

하지만 큰 크기의 블록들에 대해서 단일 프리 리스트를 이용하기 때문에 탐색 시간이 길어지는 단점이 있다.

2.2.4 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 요구를 위해서는 빠른-적합을 이용하여 워드 크기별로 프리 블록 리스트를 관리하고, 큰 크기의 메모리 요구를 위해서는 절반-적합을 이용하여 2 의 거듭제곱 크기별로 프리 블록 리스트를 관리한다. 이렇게 함으로써 알고리즘 복잡도(complexity)를 O(1)로 유지하고, WCET 를 쉽게 예측할 수 있도록 한다.

2.2 합병 전략들

동적으로 메모리를 할당하는 경우에 하나의 메모리 블록을 요청이 들어온 크기로 메모리를 분할하고, 요청된 메모리를 사용자에게 넘겨주고 남은 부분은 다시 프리 리스트로 되돌려주는 일이 발생한다. 그리고 사용되고 다시 프리 상태가 된 메모리 블록을 프리 리스트로 넘겨주는 일이 발생하게 된다. 이런 경우 서로 인접한 블록들 끼리 합병하는 작업이 필요하게 된다.

합병 전략에는 크게 즉시 합병(immediate coalescing)과 지연 합병(deferred coalescing)의 두 가지가 있다. 즉시 합병은 사용되던 메모리 블록이 반환되었을 때, 인접한 프리 블록이 있는지 검사하여 있다면 바로 합병하는 방법이다. 이에 비해 지연 합병은 반환되었을 때 바로 합병하지 않고, 더 이상 메모리 할당 요구를 만족시키지 못하는 경우에 합병하는 방법이다.

3. QB(Quick-Buddy) 알고리즘

본 논문에서 제안하고자 하는 QB 알고리즘은 앞에서 잠깐 설명한 것과 같이, 작은 크기의 프리 메모리 블록에 대해서는 워드 크기 별로 프리 리스트를 관리하고, 큰 크기의 프리 메모리 블록에 대해서는 버디 시스템으로 관리한다. 다시 말해서, MinQL=2, MaxQL=63 라고 하면, $MinQL \le s \le MaxQL$ 구간 내의 크기를 가지는 프리 메모리 블록 s 는 워드 크기 별로 QuickList 라는 리스트로 관리되며, 또한 $s \ge MaxQL$ 의 크기를 가지는 프리 메모리 블록 s 는 BuddyList 라는 리스트로 관리되게 된다.

QB 알고리즘은 빠른-적합 방식과 이진 버디 시스템의 장점만을 살리기 위하여 제안한 것이다. 이 알고리즘은 참고문헌[8]의 QHF(quick- half fit) 알고리즘을 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 조각화를 제거하였다,

3.1 QuickList 와 비트맵

작은 프리 메모리 블록을 관리하는 QuickList 는 그림 1 과 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 s 가 요청되면, QuickList 에서 s 를 인덱스로 가지는 프리 리스트

를 검사한다. 하지만 s 를 인덱스로 가지는 프리 리스트에 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 실행 시간은 O(MaxQL+1)이 된다.

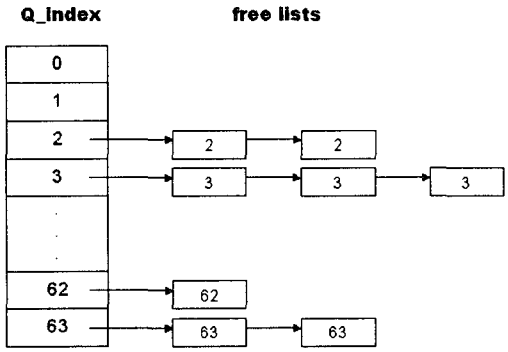

그림 1. MinQL=2, MaxQL=63 인 경우의 QuickList

이러한 탐색 비용을 줄이기 위해서, 절반-적합과 버디 시스템에서는 각 리스트가 비어 있는지 아닌지를 나타내는 비트맵 방식을 이용한다. 하지만 Q_index 의 개수가 많다면 탐색 시간 역시 길어지게 된다. 그러므로 본 논문에서는 이러한 탐색 시간을 더욱 줄이기 위해 하나의 비트맵을 더 추가한다.

비어있지 않은 리스트 탐색 알고리즘[8]은 두 개의 비트맵(bitmap)을 이용한다(그림 2). 첫 번째 비트맵(First_bitmap)은 그 비트맵이 가리키는 프리 리스트에 프리 블록이 존재하는 지를 나타내며, 비트맵의 개수 Q_index 의 개수와 같다. 두 번째 비트맵(Second_bitmap)은 첫 번째 비트맵에서 8 개의 엔트리(entry)를 하나의 비트맵 엔트리로 표현하여, 8 개의 프리 리스트들 안에 프리 블록이 존재하는 지를 나타낸다. 즉, First_bitmap 에서 8 개의 비트가 모두 0 이면 Second_bitmap 의 이에 해당하는 비트는 0 이고, 그렇지 않은 경우에는 1 이 된다.

QuickList 로부터 프리 블록을 요청하는 경우, 요청된 프리 블록의 크기는 s, Second_bitmap 의 한 비트가 관리하는 First_bitmap 의 비트 수를 b1, Second_bitmap 의 비트 수를 b2 라 하면 할당되는 과정은 다음과 같다.

(1) 요청된 블록 사이즈를 가지는 리스트가 비어있는지 First_bitmap 을 이용하여 알아낸다.
(2) (1)에서 리스트가 비어있지 않다면, 그 리스트의 첫 번째 프리 블록을 할당한다.
(3) (1)에서 리스트가 비어있지 않다면, Second_bitmap 의 마지막 비트부터 $\lceil s / b1 \rceil$번째 비트까지 내림차순으로 각 비트의 값이 1 인지를 검사한다.
(4) (3)에서 비트의 값이 1 인 비트를 찾았다면, 이 비트에 해당하는 First_bitmap 의 비트 열에 서 내림차순으로 각 비트의 값이 1 인지를 검사한다. 즉, 최악-적합(worst-fit)을 이용한다.
(5) (4)에서 비트의 값이 1 인 비트를 찾았다면, 이 비트에 해당하는 QuickList 의 리스트에서 프리 블록을 할당한다.
(6) (4)에서 비트의 값이 1 인 비트를 찾지 못했다면, BuddyList 의 첫 번째 프리 리스트로부터 한 블록을 할당받아서, 요청된 크기인 s 만큼을 사용자에게 할당하고, 남은 부분은 QuickList 로 반환한다.

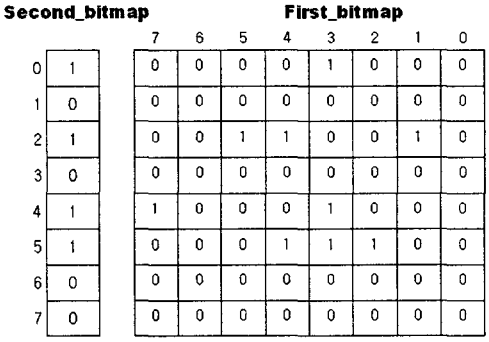

그림 2. Bitmap 들

이와 같은 할당 방식을 사용하면, 요청된 크기의 블록이 존재하지 않을 때 최악-적합 방식을 이용함으로써 외부 조각화를 최소화할 수 있다. 또한 할당하기 위해 비트맵을 검사하는데 걸리는 시간은 Second_bitmap 의 길이와 Second_bitmap 에서 찾은 비트에 의해 참조되는 First_bitmap 의 비트열의 길이에 의존하므로, WCET 는 O(b2-1+b1)이 된다.

QuickList 내부에서의 합병 작업은 사용되다가 반환된 블록과 인접한 프리 블록 사이에서 발생한다. 그러므로 최대 3 개의 프리 블록들이 합병작업을 수행할 수 있다. 그리고 합병되어 그 크기가 BuddyList 의 첫 번째 프리 리스트가 관리하는 크기와 같다면, BuddyList 의 첫 번째 프리 리스트에 추가된다.

3.2 BuddyList

BuddyList 는 MaxQL 보다 큰 메모리 요청을 처리하기 위해서 프리 블록들을 2^n 의 크기 별로 관리하는 벡터이다. 그림 3 은 MaxQL=63 일 경우 BuddyList 가 어떤 구조를 가지는 지를 보여준다.

MaxQL 보다 큰 메모리 요청에 대한 할당 방법은 일반적인 이진 버디 시스템과 같다. s≥ MaxQL 인 s 크기의 메모리 할당 요청이 들어오면, $\lfloor \log(s) - \log(MaxQL+1) \rfloor$ 을 인덱스로 하여 메모리를 할당해 준다. 하지만 앞 절에서 설명되어진 것처럼 QuickList 에서 MinQL≤ s≤ MaxQL 에 존재하는 크기 s 의 요청에 대해서 할당해 줄 프리 블록이 존재하지 않는다면, B_index=0 인 프리 리스트의 첫 번째 프리 블록을 할당해 준다.

그림에서 bitmaps 는 리눅스에서 구현되어 있는 것과 같이 병합 작업을 빠르게 하기 위해 존재한다. B_index 에 의해서 참조되는 프리 리스트 2 개의 할당 상태를 bitmaps 의 한 비트로 표현한다. 즉, 연속된 2 개의 프리 리스트가 모두 비어 있거나 모두 할당되어 있다면 이에 해당하는 bitmaps 의 비트는 "0"이라는 값을 가지며, 둘 중 하나만 할당되어 있다면 "1"의 값을 가진다. BuddyList 의 리스트 길이가 길어지면 할당하는데 많은 시간이 걸릴 수 있다. 이런 경우에는 할

당을 빠르게 하기 위해, 위에서 설명한 bitmaps 이외에 프리 리스트가 비어 있는지를 나타내는 비트맵을 추가할 수 있다.

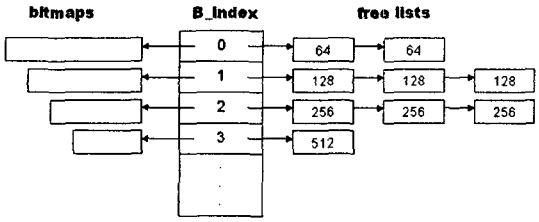

그림 3. MaxQL=63 인 경우의 BuddyList

결국 할당하는 데 걸리는 WCET 는 BuddyList 의 리스트 길이에 의존하므로, BuddyList 의 길이를 BL 라 하면 WCET 는 O(BL) 이 된다.

4. 성능평가

할당하는데 필요한 WECT 는 QuickList 와 BuddyList 에서 나누어 생각할 수 있다. 3.1 절과 3.2 절에서 설명된 바와 같이 QuickList 의 WCET 는 O(b1-1+b2)이며, BuddyList 의 WCET 는 O(BL)이다. 하지만 최악의 경우 QuickList 가 BuddyList 에게 프리 블록을 요청할 수 있으므로, 전체적인 WCET 는 O(b1-1+b2+BL)이 된다. 하지만 비트 탐색 명령어를 지원하는 CPU 가 사용되는 경우에는 O(1)에 수렴하게 된다. 하드웨어적인 지원이 가능하다면 본 논문에서 제안한 알고리즘을 통해서 빠른 할당 알고리즘을 실현할 수 있다.

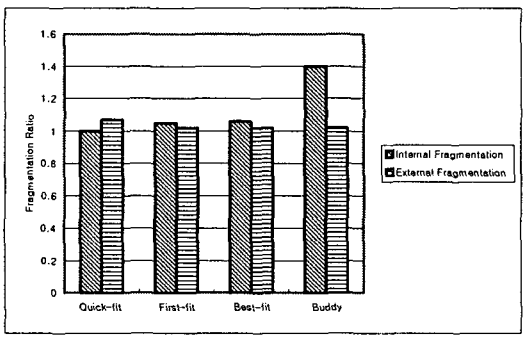

그림 4. 알고리즘들의 조각화 율 비교

참고문헌 [7, 8]의 실험 결과를 인용하면, 내부 조각화 문제에서는 빠른-적합이 가장 좋은 성능을 나타내고 있으며 외부 조각화 문제와 처리 효율(throughput) 면에서는 버디 시스템이 가장 좋은 성능을 보여주고 있다(그림 4 참조). 시간적인 문제로 인하여 기초적인 시뮬레이션 결과만을 확인할 수 있었는데, 이 결과로 보면 버디 시스템의 외부 조각화 문제가 많이 개선된 것을 볼 수 있었다. 아직 완전히 끝나지는 않아서 결과를 단정 지을 수는 없지만, 이 두 가지 알고리즘을 복합해서 만든 QB 알고리즘을 통해서 성능 향상을 이룰 수 있게 되었다. 하지만 여전히 내부 단편화는 개선되어야 할 문제이다.

또한 본문에서 제안한 QB 알고리즘은 외부 조각화가 발생하지 않는 장점을 가지고 있다. QuickList 에서도 최악-적합(worst-fit)을 사용함으로써 외부 조각화를 최소화 하였으며, BuddyList 에서도 버디 시스템을 사용하여 외부 조각화가 발생하지 않도록 하였다.

5. 결론

본 논문에서는 메모리의 외부 조각화와 WCET 를 쉽게 예측 가능하게 함으로써 실시간 시스템에 동적 메모리 할당 방식을 적용한 Quick-Buddy 알고리즘을 제안한다. 이 알고리즘은 MinQL≤ s≤ MaxQL 구간 내의 크기를 가지는 프리 메모리 블록 s 는 워드 크기별로 QuickList 라는 리스트로 관리되며, 또한 s≥ MaxQL 의 크기를 가지는 프리 메모리 블록 s 는 BuddyList 라는 리스트로 관리되게 된다. 이로써 작은 메모리 블록을 관리하기 위해서 사용되는 빠른-적합 방식을 이용하여 빠른 시간 안에 정확한 크기의 메모리를 할당 할 수 있다. 그리고 큰 메모리 블록의 할당을 위해서 사용된 버디 시스템은 외부 조각화를 방지할 수 있는 장점이 있다.

하지만 버디 시스템을 사용함으로써 다른 알고리즘들에 비해 상대적으로 큰 내부 조각화가 발생하게 된다. 내부 조각화 문제는 훗날의 해결 과제로 남아있다. 지금도 계속 연구 중에 있으므로 향후에는 이 문제를 해결할 수 있도록 하겠다.

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", Internation Workshop on Memory Management, 1995

[4] 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.

[5] Mark S. Johnstone and Paul R. Wilson, "The memory fragmentation problem : solved?", Proc. of Intn. Symposium on Memory Management, pp. 26-36, 1998

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

[7] A.K. Iyengar, "Scalability of dynamic storage allocation algorithms", Proc. of 5th IEEE Symposium on Parallel and Distributed Processing, pp.82-91, 1993

[8] 정성무, 유해영, 심재홍, 김하진, 최경희, 정기현, " 예측 가능한 실행 시간을 가진 동적 메모리 할당 알고리즘" , 한국 정보처리학회 논문지, 제 7 권, 제 7 호, pp.2204-2218, Jul. 2000