실시간 운영체제 iRTOS™ 를 위한 메모리 관리 체계 설계 및 구현

박희상[0] 안희중 김용희 이철훈
충남대학교 컴퓨터공학과
(hspark[0], hjahn, yhkim)@pplab.ce.cnu.ac.kr, chlee@ce.cnu.ac.kr

Design and Implementation of Memory Management Facility for Real-Time Operating System, iRTOS™

Hi-Sang Park[0], Hee-Jung Ahn, Yong-Hee Kim, Cheol-Hun Lee
Dept. of Computer Engineering, Chungnam National Univ.

요 약

실시간 운영 체제(Real-Time Operating System)는 특정 태스크가 정해진 시간 안에 수행될 수 있도록 시간 결정성(Determinism)을 보장하는 운영 체제이다. 실시간 운영체제는 멀티태스킹(Multitasking) 및 ITC(InterTask Communication 혹은 IPC, InterProcess Communication)을 제공한다는 점에서는 일반 운영체제인 Unix™, Linux™, Windows™ 등과 같지만, 시간 결정성을 보장한다는 점에서 일반 운영체제와 다르다. 또한 실시간 운영 체제를 포함한 임베디드 시스템(Embedded System)은 일반적으로 디지털 카메라, 디지털 TV, DVD 등에서 수행되므로 실행 이미지(Image) 크기가 작아야 한다. 본 논문에서는 실시간 운영체제의 실행 이미지를 줄이면서 시간 결정성을 보장할 수 있도록 메모리 관리 체계를 설계하고 구현한 내용을 설명한다.

1. 서론

실시간 운영체제는 시간 결정성(Determinism)과 작은 크기의 실행이미지(Image)를 특징으로 하며, 초기에는 군사용 제어, 산업 기기 제어, 원자력 발전 기기 시스템 분야와 같은 특수한 목적으로 사용되어 왔으나 1990 년대 중반 이후에는 네트워크 장비, 정보 가전 등 여러 분야에서 폭 넓게 사용되고 있다. 그러나 국내에서 사용되는 대부분의 실시간 운영체제는 국외에 비싼 사용료(royalty)를 지불하고 있다. 국내에서도 실시간 운영 체제의 고 부가 가치를 인식하고 몇몇 연구소를 중심으로 실시간 운영체제의 국산화에 노력하고 있으나 아직 상용화에 이르지는 못하고 있다.

실시간 운영체제는 멀티태스킹(Multitasking)과 ITC(InterTask Communication)를 근간으로 한다. Unix™, Linux™, Windows™ 등의 일반 운영체제도 멀티태스킹과 ITC(혹은 IPC, InterProcess Communication)를 제공하지만, 실시간 운영체제는 이들 운영체제와 달리 시간 결정성(Determinism)을 보장하도록 설계되어야 한다. 또한 대부분의 실시간 운영체제는 임베디스 시스템(Embedded System) 환경에서 수행되므로 실행 이미지(Image)크기가 작을수록 유리하다. 임베디드 시스템에서 실행 이미지 크기를 줄이기 위해서는 전역변수를 사용하는 대신 동적 메모리(Dynamic Memory)를 사용해야 한다. 또한 실시간 운영체제는 동적 메모리를 할당(Allocation)하고 해제(Free)하는데 시간 결정성이 보장되도록 설계되어야 한다. 본 논문에서 구현된 메모리 관리 체계는 임의의 크기를 할당할 수도 있고, 메모리 풀(Memory Pool)을 사용하여 고정된 크기의 메모리를 할당할 수도 있도록 설계되었다.

본 논문에서는 2 장에서 관련 연구를, 3 장에서 iRTOS의 전체적인 구성을, 4 장에서 메모리 관리 체계 설계 및 구현을, 5 장에서 테스트 환경 및 결과를, 6 장에서 결론 및 향후 과제를 기술한다.

2. 관련 연구

동적 메모리 관리 체계는 실시간 운영체제 및 임베디드 시스템 설계에 있어서 매우 중요한 특징으로 인식되어, " Dynamic Memory" 혹은 " Storage Allocation" 이라는 주제로 여러 논문에서 다루고 있다. 현재까지 연구된 내용을 종합하면 다음과 같다.

잘못 설계된 동적 메모리 관리 체계는 2 가지 사항에서 문제를 야기할 수 있다. 첫째로, 힙(heap)에서 메모리를 할당할 경우 동적 메모리 할당 시간이 오래 걸리고 시간 결정성을 저해하게 된다. 적당한 크기의 메모리 블록(Memory Block)을 찾기 위해서는 메모리 블록 프리 리스트(Free List)를 찾아야 하는데 이때 걸리는 시간은 시간 결정성을 심각하게 저해하기 때문이다. 둘째로, 힙(Heap)의 단편화(External Fragmentation)때문에 메모리 블록을 제대로 할당 할 수 없는 문제가 발생할 수 있다. 이를 해결하기 위해서는 필요한 메모리 크기를 미리 추정하여 최대 크기의 고정된 메모리 풀(Memory Pool)로부터 메모리를 할당하는 것으로 해결 할 수는 있다. 메모리 풀 사용으로 인하여 메모리 낭비가 심하지만 메모리 풀의 사용은 불가피하다[3][4].

위와 같은 이유로 기존의 상용 실시간 운영 체제들은 고정된 크기의 메모리 풀을 만들고, 풀로부터 메모리 블록을 할당 받도록 하고 있다. 그렇지만, 몇몇 실시간 운영체제는, 단편화 문제 등이 있기는 하지만 동적 메모리의 편리성 때문에 힙(Heap)에서 임의의 크기 메모리를 할당 받을 수도

있도록 하고 있다.

3. iRTOS™

iRTOS™는 실시간 운영체제의 핵심이라고 할 수 있는 멀티태스킹(MultiTasking) 및 ITC 환경을 제공한다. 태스크(Task)는 독립적으로 수행되는 실행 프로그램을 의미하며, 멀티태스킹 환경은 여러 개의 태스크가 동시에 수행될 수 있는 스케줄링 정책을 제공함을 의미한다. ITC 환경은 태스크와 태스크의 공동 작업(Cooperation)을 위한 동기화(Synchronization) 및 통신(Communication)을 위하여 세마포어, 메시지 메일박스, 메시지 큐등을 지원함을 의미한다[1].

3.1 멀티태스킹

iRTOS™는 서로 다른 우선 순위의 경우, 우선 순위가 높은 태스크를 선점(Preemption)하여 실행하고, 동일한 우선 순위의 태스크들은 타임 슬라이스(Time Slice) 동안 차례로 수행되는 라운드 로빈 방식을 따른다[1].

3.2 ITC(InterTask Communications)

iRTOS™는 ITC로서 세마포어, 메시지 메일박스, 메시지 큐를 지원한다[1].

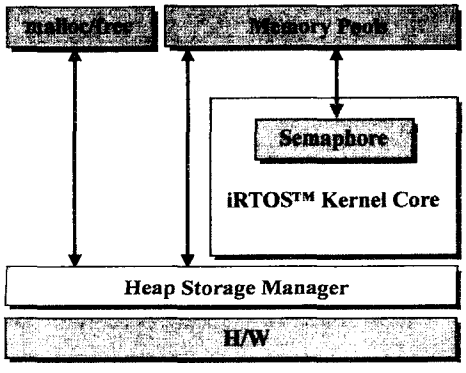
- 세마포어(Semaphore)
  세마포어는 안전한 공유 자원 관리를 위해 사용될 뿐만 아니라, 동기화(Synchronization) 및 상호 배제(Mutual Exclusion)를 위해서도 사용된다.

- 메시지 큐(Message Queue)
  메시지 큐는 특정 태스크나 ISR에서 다른 태스크로 여러 개의 메시지를 전송할 수 있다.

- 메시지 메일박스(Message Mailbox)
  메시지 메일박스는 메시지 큐(Binary Message Queue)의 특수한 경우로 이해할 수 있으며, 빠른 처리 속도를 위해 별도로 구현되었다.

4. 메모리 관리 체계 설계 및 구현

iRTOS™의 메모리 관리 체계는 [그림 1]과 같다.

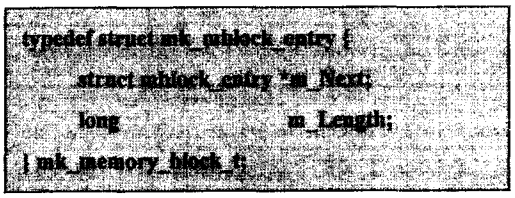


[그림 1] 메모리 관리 체계 구성

실시간 운영체제 iRTOS™는 시간 결정성 및 단편화에 방지를 위해서 고정 크기 메모리 블록을 위한 메모리 풀(Memory Pool)을 제공하고, 또한 malloc() 및 free()를 사용하여 임의의 크기 메모리 블록을 사용할 수 있도록 설계되었다.

4.1 Heap Storage Manager

시스템이 부팅(Booting)하게 되면 힙(Heap)영역을 초기화하고, 힙 스토리지 메니저(Heap Storage Manager)가 힙 영역의 메모리 관리를 전담하게 된다.

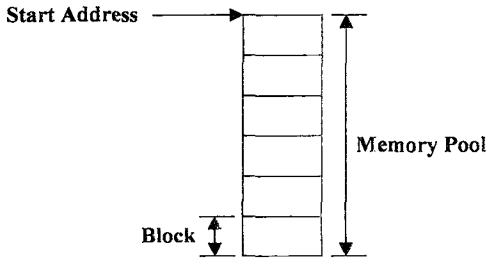


[그림 2] Heap Memory Control Block

힙의 사용 가능한 메모리 블록들은 [그림 2]의 자료구조를 사용하여 주소를 인덱스(Index)로 하여 정렬(sorting)된 리스트로 구성된다. [그림 2]의 자료구조의 메모리 블록은 연속된 메모리의 시작 주소 및 크기를 나타내며, 힙 메모리 요구가 발생하면 프리 리스트(Free List)에서 퍼스트 핏(First Fit) 알고리즘에 따라 힙 메모리를 할당하고 남은 영역은 다시 프리 리스트에 추가 된다. 반대로 메모리가 반납되었을 때는 인접한 메모리 블록이 존재하면 통합(Merge)하고 인접한 메모리 블록이 존재하지 않으면 프리 리스트에 추가한다. 또한, 힙 스토리지 매니저를 사용해 메모리를 반납할 경우 블록 시작 주소와 블록 크기를 매개 변수로 넘겨줘야 한다.

4.2 Memory Pools

힙 스토리지 메니저는 임의의 크기 메모리 블록을 할당하고 해제할 수 있기는 하지만, 2 장에서 언급하였듯이 실시간 운영체제에서 매우 중요한 시간 결정성(Determinism)을 보장하지 못하고 단편화(External Fragmentation)로 인한 시스템 폴트(System Fault)의 문제를 안고 있다. 이런 문제를 해결하기 위해서 iRTOS™는 [그림 3]과 같이 메모리 풀을 사용한 고정 크기의 메모리 블록 할당 및 해제를 지원한다.



[그림 3] Memory Pool

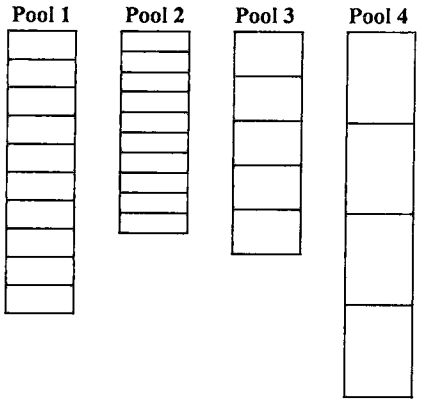
메모리 풀은 힙 스토리지 메니저로부터 블록크기×블록수의 메모리를 할당 받은 후 [그림 4]의 자료 구조를 사용하여 각각의 블록을 프리 버퍼 리스트(Free Buffer List)로 구성한다.

```
typedef struct mk_bufpool_entry {
    struct mk_bufpool_entry *bp_Next;
    long   bp_Size;     /* Block Size     */
    int    bp_Sem;      /* Semaphore ID */
} mk_buf_pool_t;
```

[그림 4] Memory Pool Control Block

이때, 메모리 풀의 각각의 블록은 세마포어 리소스(Resource)가 되고 블록수는 세마포어 리소스 카운트(Resource Count)가 된다. 따라서 프리 버퍼 리스트(Free Buffer List)가 널(NULL)인 경우에 버퍼를 요구하는 태스크는 세마포어 메커니즘(mechanism)에 따라 펜딩(Pending)되고 다른 태스크가 버퍼를 반납하였을 때 깨어나서 버퍼를 할당받을 수 있게 된다.

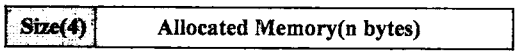
또한 [그림 5]와 같이 서로 다른 블록 크기의 메모리 풀을 여러 개를 생성하여 사용할 수도 있다.



[그림 5] Multiple Memory Pools

4.3 malloc / free 함수

힙 스토리지 메모리의 경우, 메모리를 반납할 경우에도 반납할 메모리의 크기를 기억하고 있어야 한다. 메모리를 할당받은 후 정확한 크기를 기억하는 것은 번거롭다. 이를 해결하기 위해 malloc 함수는 [그림 6]과 같이 (할당될 메모리 크기 + 4 바이트)의 메모리를 할당 받은 후 내부적으로 할당된 메모리 크기를 저장한 후 메모리를 반납할 때 사용하게 된다.



[그림 6] malloc()/Free() Memory 구조

5. 테스트 환경 및 결과

본 논문의 구현은 IBM 호환 PC를 기반으로 구현하였으며, 80x86 의 Large Model을 기반으로 하였다. 80x86 Large Model은 32 bit 주소 공간(Address Space)를 가진다. 컴파일러는 Microsoft Visual Studio™ 6.0을 사용하였다. 또, 본 논문에서는 실행 이미지의 크기를 줄이기 위해 TCB(Task Control Block)등의 내부 자료구조에 필요한 메모리를 힙 스토리지 메니저를 통해 할당 받고, 태스크 생성시 필요한 스택을 전역변수가 아닌 메모리 풀을 사용하였다. 이렇게 함으로써 태스크가 증가해도, 롬(ROM)에 저장되어야 하는 실행 이미지 크기는 21Kbyte로서 고정되었다.

6. 결론 및 향후 연구 과제

본 논문에서는 실시간 운영체제인 iRTOS™의 메모리 관리 체제를 개선하여 시간 결정성을 보장함은 물론 롬(ROM)에 저장되는 실행 이미지(Image)의 크기를 태스크의 수에 관계없이 일정하도록 했다. 32bit CPU의 경우 32 bit 주소 공간을 가지므로 하나의 힙 스토리지 메니저만으로 충분하지만, 16 bit 혹은 22bit CPU의 경우 세그먼트(Segment 혹은 bank)를 사용하기 때문에 하나 이상의 힙 스토리지 메니저가 존재하거나 또는 맵핑(mapping)을 사용해야 되는데 이 부분은 앞으로 계속 연구되어야 하겠다.

6. 참고문헌

[1] http://www.inestech.com
[2] D. Comer " Operation System Design VOL 1 : THE XINU APPROACH" , 1988.
[3] David Stepner, Nagarajan Rajan, David Hui, " Embedded Application Design Using a Real-Time OS" , Design Automation Conference, 36th, pp.151-156, 1999
[4] David Lafreniere, " An Efficient Dynamic Storage Allocator" , Embedded Systems Programming, Sept. 1998.
[5] Jean, J. Labrosse, " μ C/OS The Real-Time Kernel" , R&D Publications, 1992.
[6] IEEE Std 1003.1b, " Portable Operating System" , 1993.