제 19 회 한국정보처리학회 춘계학술발표대회 논문집 제 10 권 제 1 호 (2003. 5)

실시간 운영체제에서 메모리 관리 구현

안희중†, 박윤미†, 성영락‡, 이철훈†
†충남대학교 컴퓨터공학과, ‡국민대학교 전자공학부
e-mail : †{hjahn, ympark, chlee}@ce.cnu.ac.kr, ‡yeong@mail.kookmin.ac.kr

Implementation of Memory Management for Real-Time Operating Systems

Hee-Joong Ahn†, Yun-Mi Park†, Yeong Rak Seong‡, and Cheol-Hoon Lee†
†Dept. of Computer Engineering, Chung-Nam National University
‡School of Electrical Engineering, Kookmin University

요 약

실시간 운영체제는 효율성 및 공평성을 중시하는 다른 범용 운영체제와는 달리 논리적 정확성과 시간적 정확성을 보장하기 위해 스케줄링하는 운영체제를 의미한다. 임베디드 시스템에서 동작하는 실시간 운영체제는 제한된 환경에서 제한된 목적으로 사용되고 있다. 특히 메모리는 실시간 운영체제의 실행에 있어서 꼭 필요한 자원이므로 이에 대한 효과적인 관리가 필수적이라 할 수 있다. 본 논문은 동적 메모리 할당을 위한 힙 스토리지 매니저와 메모리 풀을 설계하고 구현한 내용을 기술한다.

1. 서론

임베디드 시스템이란 미리 정해진 특정 기능을 수행하기 위해 컴퓨터의 하드웨어와 소프트웨어가 조합된 전자 제어 시스템을 의미한다. 이러한 임베디드 시스템에 사용하는 운영체제가 실시간 운영체제(RTOS : Real-Time Operating System)이며 제한된 환경에서 제한된 목적을 수행할 수 있도록 주어진 자원의 효율적인 관리는 필수적이다. 데드라인에 따라서 실시간 운영체제는 주어진 데드라인 내에 수행해야만 하는 경성 실시간 운영체제(Hard Real-Time System)와 주어진 데드라인을 조금 어기더라도 그 결과를 인정하는 연성 실시간 운영체제(Soft Real-Time System)로 나뉘어진다. 실시간 운영체제도 다른 범용 운영체제와 같이 멀티태스킹, ITC(Inter-Task Communication), 예외처리등을 제공하지만 시간 결정성을 보장해야 한다는 점이 다르다. 시간 결정성은 어떠한 작업을 빨리 수행하는 것이 아니라 주어진 데드라인 내에 작업 수행이 가능해야 하며, 작업 수행 완료시점에 대한 예측이 가능함을 의미한다. 제한된 환경하에서 자원의 효율적인 관리는 필수이지만 그 중에서도 가장 중요한 자원은 프로그램 실행에 꼭 필요한 메모리이다. 태스크에 메모리를 할당하는 방법은 전역변수로 선언하는 정적 메모리 할당(Static Memory Allocation)과 실행 시간에 메모리를 할당하는 동적 메모리 할당(Dynamic Memory Allocation)이 있다. 임베디드 환경에서 실시간 운영체제의 실행 이미지는 롬(ROM)에 다운로드 하므로 가능한 작아야 한다. 그러나 전역변수를 사용하여 정적으로 메모리를 할당하게 되면 실행이미지가 증가하므로 필요한 메모리는 실행 시간에 동적으로 할당받아 사용한다.

본 논문에서는 실시간 운영체제 중에서 안정성 및 성능을 인정받은 iRTOS™을 연구대상으로 하였고, 타겟 보드는 삼성에서 제작한 ARM 920T 기반의 S3C2800 보드를 사용하였다. 동적 메모리 할당 방법으로 힙 스토리지 매니저를 사용하였고 힙 스토리지 매니저만을 사용하면 시간 결정성과 메모리 단편화 문제가 있어 이를 보완할 수 있는 메모리 풀을 설계하고 구현한 내용을 기술한다.

본 논문의 구성은 2 장에서는 실시간 운영체제의 전체적인 구성을, 3 장에서는 메모리 관리체제에 대한 설계 및 구현 내용을, 4 장에서는 테스트 환경 및 결과를 기술한다. 마지막으로 5 장에서는 결론 및 향후 연구과제를 기술한다.

2. 실시간 운영체제(Real-Time Operating System)

보통 임베디드 시스템은 한 개의 CPU 를 가지고 수행하므로 여러 개의 태스크가 동시에 수행할 수 있도록 멀티태스킹 환경을 지원해야 한다. 그리고 태스크 수행에 대한 시간 결정성을 보장하기 위하여 우선순위 기반의 선점형 스케줄러를 제공한다. iRTOS™은 태스크의 우선순위를 0 부터 255 까지 256 단계로 구분하고 있으며, 낮은 숫자가 높은 우선순위를 의미한다. 실행 준비된 태스크들 중에서 정해진 시간 내에 가장 높은 우선순위의 태스크를 찾기 위해 별도의 테이블과 리스트를 관리하며, 같은 우선순위의 태스크 사이에서는 선입선출(FIFO) 방법과 Round-Robin 방법을 지원한다. 그리고 멀티 태스킹 환경하에서 여러 태스크들 사이의 동기화와 통신을 위해 세마포(Semaphore), 메시지 큐(Message Queue) 등을 지원한다.

2.1 ITC(Inter-Task Communication)

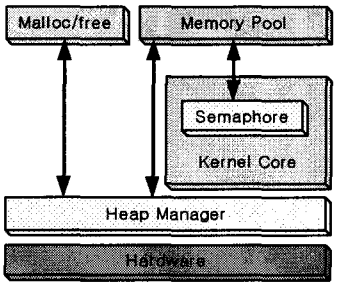
- 세마포(Semaphore)
  공유자원의 안전한 관리를 위한 상호배제(Mutual Exclusion)와 태스크간 동기화에 사용된다.
- 메시지 큐(Message Queue)
  특정 태스크나 ISR 에서 다른 태스크로 여러 개의 메시지를 전달할 때 사용된다.
- 메시지 메일박스(Message Mailbox)
  메시지 큐의 특수한 경우로 하나의 메시지를 빠르게 전송할 때 사용된다.
- 메시지 포트(Message Port)
  메시지 큐와 같이 여러 개의 메시지를 다른 태스크로 전달할 때 사용한다. 메시지 큐는 메시지를 복사해서 전달하지만 메시지 포트는 메시지의 포인터만을 전달한다.

2.2 메모리 관리 체계

동적 메모리 할당은 실시간 운영체제의 설계에서 임베디드 시스템에 적용되는 특수한 상황때문에 매우 중요하게 인식된다. 메모리는 시스템의 동작을 위해서 꼭 필요한 부분이기 때문에 실시간 운영체제의 여러 중요한 자원들 중에서도 가장 중요하다. 메모리를 관리하기 위한 방법으로는 임의 크기의 메모리를 할당하는 힙 스토리지 매니저와 고정된 크기의 메모리를 할당하는 메모리 풀이 있다. 그러나 힙(Heap)영역에서 동적 메모리 관리를위해 힙 스토리지 매니저를 사용하는 경우 두가지의 문제점이 발생한다. 첫째, 시간 결정성을 해치게 된다. 메모리를 할당하지 않은 영역은 프리 리스트(Free List) 자료구조를 통해서 관리되므로 메모리 할당 요구가 생기면, 프리 리스트에서 찾게 되는데 리스트가 많아질수록 그만큼 시간이 오래 걸리게 된다. 둘째, 메모리 단편화(Memory Fragmentation) 문제이다. 총 할당 가능한 메모리는 태스크가 요구한 메모리 크기보다 크지만 메모리가 비연속적으로 있기 때문에 메모리를 할당할 수 없는 상황이 발생한다. 위의 두가지 문제를 보완하기 위해 메모리 풀(Memory Pool)을 사용한다. 메모리 풀은 할당될 메모리의 크기를 추정하여 미리 힙에서 할당받아 자료구조를 유지하며, 고정된 크기의 버퍼를 할당하고 해제한다. 메모리 풀을 사용하면 필요한 메모리보다 더 많은 메모리를 할당하므로 힙 스토리지 매니저의 사용보다 메모리 낭비가 심하지만, 메모리 풀의 사용은 위의 두가지 문제 때문에 불가피하다. 그러나 임의 크기의 메모리 할당이 불가피한 경우가 있으므로 힙 스토리지 매니저와 메모리 풀 모두를 제공하고 있다.

3 메모리 관리체계 설계 및 구현

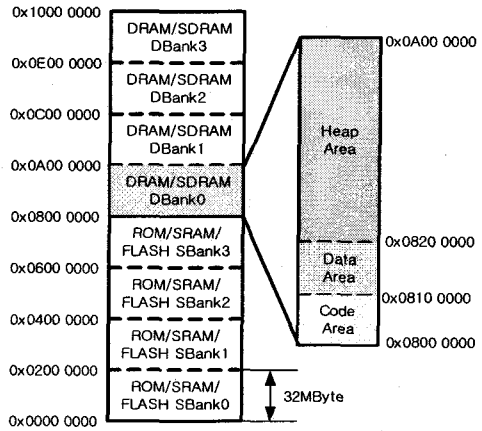
본 논문에서 구현한 메모리 관리체계는 [그림 1]과 같이 힙 스토리지 매니저(Heap Storage Manager)에서 임의의 메모리를 할당받을 수 있는 malloc()/free()와 고정 크기의 메모리를 할당받을 수 있는 메모리 풀을 제공하고 있다.



[그림 1] 메모리 관리체계 구성

3.1 힙 스토리지 매니저(Heap Storage Manager)

시스템의 메모리는 코드영역(Code Area), 데이터영역(Data Area)과 이를 제외한 힙영역으로 구분한다. 힙 영역은 힙 스토리지 매니저를 통해 관리한다.



[그림 2] S3C2800 Memory Map

[그림 2]는 본 논문에서 구현한 실시간 운영체제의 타겟보드인 S3C2800 의 메모리 맵(Memory Map)을 나타낸 것이다. 실행이미지는 DBank0 에 다운로드하여, 코드영역은 0x08000000 부터 1MByte 이고, 데이터 영역은 그 다음 1Mbyte 이고, 나머지가 힙영역이다.

```
typedef struct mk_heap_struct {
   UINT   h_Magic;
   int    h_Options;  /* FIFO, or Priority */
   UINT   h_StartofHeap;
   UINT   h_HeapSize;
   UINT   h_FreeSize;
   UINT   h_MinSize;
   char   h_pName[MK_NAME_MAX];
   struct mk_pending_list_struct h_PendingList;
   struct mk_memory_block_struct *h_pFreeMBlockList;
   struct mk_heap_struct *h_pNext;
   struct mk_heap_struct *h_pPrev;
} MK_HEAP;

STATUS MK_CreateHeapManager (MK_HEAP *pHeap,
char *pName, void *pAddr, UINT Length, UINT MinSize,
BOOLEAN Options)
```

[그림 3] Heap Control Block

[그림 3]은 힙 컨트롤 블록을 나타낸 것으로, 필드의 설명은 아래와 같다.

- h_Magic : 디버그 및 오류검사
- h_PendingList, h_Option : Pending 태스크 관리
- h_pFreeMBlockList : 프리 리스트 관리

힙 스토리지 매니저의 생성은 사용자에 의해 주어진 시작주소(pAddr)와 크기(Length)에 의해 구성된다. 따라서 사용자가 임의의 영역을 여러 개의 힙 스토리지 매니저로 구성할 수 있다.

3.1.1 힙 스토리지 매니저를 이용한 메모리 할당/해제

```
/* Free Memory Management */
typedef struct mk_memory_block_struct {
   UINT   mb_Magic;          /* Heap Pointer */
   UINT   mb_Size;
   struct mk_memory_block_struct *mb_pNext;
} MK_MBLOCK;

/* Allocation Memory Management */
typedef struct mk_heap_dummy_struct {
   struct mk_heap_struct *md_pHeap;   /* Heap Pointer */
   UINT   md_Size;            /* Allocated Memory Size */
} MK_MBLOCK_DUMMY;

/* Prototype malloc/free function */
STATUS MK_GetMemory(MK_HEAP *pHeap, void **pAddr,
UINT Size, long Ticks);
STATUS MK_FreeMemory(void *pAddr);
```

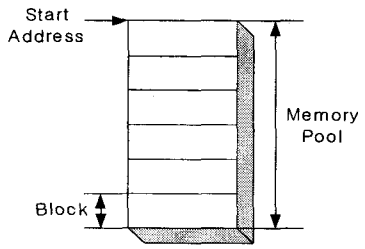
[그림 4] Memory malloc/free

동적 메모리의 할당은 MK_GetMemory() 함수를 이용하며 힙 스토리지 매니저와 할당받을 주소, 크기, 타임 틱을 매개변수로 사용한다. 타임 틱은 실시간 운영체제의 시간 결정성을 보장하기 위해 사용한다. 틱 값은 임의의 양수, MK_NO_WAIT(0), MK_WAIT_FOREVER(-1) 세 가지로 구분된다. 임의의 양수값을 주면 그 시간까지 메모리를 할당받던지 또는 못받던지에 대한 결과를 반납하며, MK_NO_WAIT 는 메모리를 요구할 당시의 결과를 반납하며, MK_WAIT_FOREVER 는 태스크가 메모리를 할당받을 때까지 기다리게 된다. 따라서 MK_WAIT_FOREVER 는 특수한 경우가 아니면 사용하지 않는다. 메모리의 할당은 프리 리스트에서 퍼스트-핏(First-Fit) 알고리즘에 따라 할당하며 남은 메모리는 프리 리스트에 연결한다. 할당된 메모리에 대해서는 어떤 힙에서 할당 받았는지에 대한 mk_pHeap(4Byte)과 할당된 크기를 나타내는 mk_Size (4Byte)있다. 할당된 메모리의 정보를 위한 8Byte 가 오버헤드가 될 수 있다. 그러나 타켓 보드와 같이 메모리를 뱅크(Bank) 단위로 나누어 사용하는 프로세서에서는 각 뱅크마다 힙 스토리지 매니저를 두어 여러 개의 힙으로 나누어 관리하면 메모리를 효율적으로 사용할 수 있다. 따라서 힙을 여러 개로 나누어 관리하므로 할당된 메모리가 어떤 힙인지에 대한 힙 포인터(md_Heap)는 필수적이다. 그리고 할당된 메모리의 크기(mk_Size)는 힙 스토리지 매니저 내에서 자료구조로 가지고 있으면 그 내용을 저장할 또다른 공간이 필요하고 구현도 복잡하므로, 이를 할당된 메모리내에 자료구조로 유지하고 있다. 그리고 하나의 힙 스토리지 매니저 내에 또다른 힙 스토리지 매니저를 두어(Heap Storage Manager Nesting), 네스팅된 힙 스토리지 매니저에서는 특정 태스크에 대해서만 메모리를 할당함으로써 중요한 태스크가 메모리를 할당받지 못하는 일이 없도록 할 수 있다.

메모리의 해제는 MK_FreeMemory() 함수를 이용하며 메모리 주소를 매개변수로 사용한다. 메모리 해제시는 해제할 메모리와 인접한 곳에 프리 메모리가 존재하면 병합(Merge)하여 프리 리스트에 추가한다. 프리 메모리를 위해 사용한 12Byte 의 자료구조에 대한 오버헤드는 메모리를 할당한 것이 아니므로 고려사항이 될 수 없다.

3.2 메모리 풀(Memory Pool)

힙 스토리지 매니저를 통해 임의 크기의 메모리를 할당/해제함으로써 메모리 단편화가 생길 수 있으며, 프리 리스트를 통해 관리하므로 프리 리스트가 많으면 많아질수록 적당한 크기의 메모리를 찾는데 많은 시간이 소요되므로 시간결정성을 해치게 된다. 이를 보완하기 위해 고정크기의 버퍼를 할당/해제하는 메모리 풀을 제공한다.



[그림 5] Memory Pool

메모리 풀은 힙 스토리지 매니저에서 스케줄링을 시작하기 전에 미리 블록수 × 버퍼 크기만큼의 메모리를 할당받아 프리버퍼 리스트를 구성한다.

[그림 5]에서 메모리 풀의 생성은 힙에서 미리 할당받은 시작주소(pAddr)과 버퍼의 개수(Count), 버퍼의 크기(Size)로 구성된다. 메모리 풀의 구성은 할당받은 메모리를 주어진 고정 크기의 버퍼단위로 나누며, 각

버퍼는 세마포의 자원으로 의미를 부여하고 버퍼의 개수는 세마포의 카운트 값이 된다. 그리고 여러 개의 메시지를 전달하는 메시지 큐에서 메모리 풀을 사용하면 효과적이다. 고정된 크기의 메시지를 갖는 메시지 큐에서는 메시지를 저장할 버퍼를 메모리 풀에서 할당받고 해제하여 시간 결정성 보장 및 메모리 단편화를 줄일 수 있다.

```
/* Free Memory Management */
typedef struct mk_memory_block_struct {
   UINT   mb_Magic;          /* Heap Pointer */
   UINT   mb_Size;
   struct mk_memory_block_struct *mb_pNext;
} MK_MBLOCK;

/* Allocation Memory Management */
typedef struct mk_heap_dummy_struct {
   struct mk_heap_struct *md_pHeap;   /* Heap Pointer */
   UINT   md_Size;            /* Allocated Memory Size */
} MK_MBLOCK_DUMMY;

/* Prototype malloc/free function */
STATUS MK_CreateBufferPool
(MK_POOL *pPool, char *pName, void *pAddr, int Count,
UINT Size, BOOLEAN Options)
STATUS MK_GetMemory(MK_HEAP *pHeap, void **pAddr,
                    UINT Size, long Ticks);
STATUS MK_FreeMemory(void *pAddr);
```

[그림 6] Memory Pool Control Block

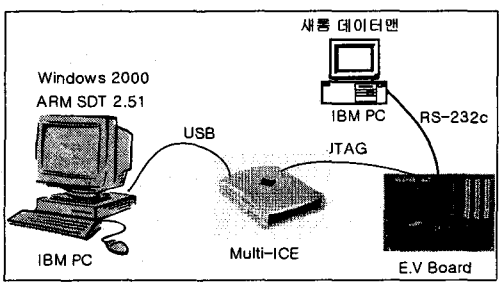
3.2.1 메모리 풀을 이용한 버퍼 할당/해제

메모리 풀에서 MK_GetBuf()를 이용하여 버퍼를 할당하며 메모리 풀, 할당받을 주소, 틱을 매개변수로 사용한다. 메모리 할당은 세마포 메커니즘에 따라 버퍼를 할당하고 세마포 카운트 값을 1 감소한다. 힙 스토리지 매니저와 마찬가지로 틱은 시간 결정성을 보장하기 위해 사용한다. 할당된 버퍼는 4Byte 의 자료구조로 자신이 속한 메모리 풀을 가리키며, 버퍼 해제 시에 자신의 메모리 풀을 찾아 해제할 수 있다. 그러나 4Byte 의 자료구조도 오버헤드가 될 수 있지만 힙 스토리지 매니저와 마찬가지로 메모리 풀 내에서 관리하는 것보다는 할당된 메모리에서 자료구조를 유지하는게 효과적이다. 그리고 프리 버퍼는 메모리 풀에 리스트로 연결되어 있어 리스트의 맨 앞에서 버퍼를 할당한다.

메모리 풀에서 MK_FreeBuf()를 이용하여 버퍼를 해제하며 해제할 메모리 주소를 매개변수로 사용한다. 버퍼의 할당과 마찬가지로 세마포 메커니즘에 따라 버퍼를 반납하고 세마포 카운트 값을 1 증가한다. 그리고 반납된 버퍼의 앞 4Byte 를 이용하여 버퍼가 속한 메모리 풀을 찾아 메모리 풀의 프리 리스트에 연결하고 해제한다.

지금까지 언급한 힙 스토리지 매니저와 메모리 풀은 동적으로 메모리를 관리하고 있다. 따라서 시스템 초기화시에 태스크에 필요한 태스크 컨트롤 블록과 태스크 스택, ITC 를 위한 메시지 큐와 메시지 메일박스, 메시지 포트 더 나아가 네트워크 처리등을 위와 같은 동적 메모리 관리체계를 이용하여 할당하면 태스크나 기타 다른 모듈들이 많아져도 실시간 운영체제의 실행이미지 크기가 변하지 않는 효과를 볼 수 있으며 실행이미지는 ROM 에 다운로드 되므로 시스템 설계에 있어서도 비용을 절약할 수 있는 효과가 있다.

4 테스트 환경 및 결과



[그림 7] 테스트 환경

본 논문에서 구현한 실시간 운영체제는 [그림 7]과 같이 32bit Addressing 을 하는 삼성에서 제작한 ARM920T 기반의 S3C2800 Evaluation Board 에서 구현하였다. 컴파일러는 ARM 사의 SDT2.51 을 사용하였다. TCB (Task Control Block)와 태스크 스택을 힙을 통해 할당받아 실행이미지는 태스크 수와 상관없이 23KByte 정도로 일정하다.

5 결론 및 향후 연구 과제

본 논문에서는 실시간 운영체제의 핵심이라 할 수 있는 동적 메모리 관리체계인 힙 스토리지 매니저 및 메모리 풀을 설계하고 구현한 내용을 기술하였다. 힙 스토리지 매니저로 동적 메모리를 할당/해제 할 수 있고, 힙 스토리지 매니저만 사용할 경우 생길 수 있는 시간 결정성 문제와 메모리 단편화 현상을 보완하기 위해 메모리 풀을 구현하였다.

향후 연구과제로는 스탠다드 라이브러리(Standard Library)에서 제공하는 함수인 calloc(), realloc(), valloc() 등에 관한 부분으로써 앞으로 더욱 연구되어야 할 것이다.

참고문헌

[1] Embedded System & RTOS, http://www.inestech.com.
[2] Brett Hammond, "*Software Quality Vs. Dynamic Memory Allocation*", Embedded System Programming, June 1995
[3] Timothy V. Fossum, "*OPERAING SYSTEM DESIGN VOL..1 : THE XINU APPROACH*", 1998.
[4] Jean J. Labrosse, "*uC/OS The Real-Time Kernel, R&D Publications*",1995.
[5] WindRiver, "*VxWorks Progammer's Guide*", 1997.
[6] Accelerated Technology, *Nucleus PLUS Internals Manual*, 1996.
[7] IEEE Std 1003.1b, *Portable Operating System*, 1993.
[8] 박희상, 안희중, 김용희, 이철훈, "Design and Implementation of Memory Management Facility for Real-Time Operating System, iRTOS™", 한국정보과학회, Vol. 29, No. 1, pp.58-60, 2002.04.