MicroC/OS-II에서의 효율적인 메모리 관리에 관한 연구

전영식*, 허 신*
*한양대학교 컴퓨터공학과

e-mail: k345ys@osnn.hanyang.ac.kr

A Study for Effective Management of Memory to MicroC/OS-II

Young-Sik JEON*, Shin Heu*
*Dept of Computer Science Engineering, Han-Yang University

요 약

MicroC/OS-II에서는 연속된 메모리 공간으로 구성된 파티션에서 고정 크기의 메모리 블록을 할당할 수 있는 방법을 제공하며, 이 파티션은 사용 가능한 메모리 블록의 개수를 유지하고, 모두 같은 크기를 갖는 메모리 블록을 단일 연결 리스트의 형태로 관리 한다. 이런 형태의 메모리 관리 시스템은 메모리 단편화 현상이 잘 일어나지 않지만 이런 단순한 구조로 메모리 공간을 통합 관리, 블록을 할당하고 반환하는데 필요한 검사등을 효율적으로 수행할 수 없다. 본 논문에서는 MicroC/OS-II에서의 단편화문제를 해결하는 방법에 더 나아가 효율적으로 메모리를 통합하고 관리하는 방법에 대해 제안하고자 한다.

1. 서론

임베디드 리얼타임 시스템에서는 ANSI C 컴파일러가 제공하는 malloc()과 free()함수들을 사용하는 것이 메모리 할당과 해제에 따른 메모리 단편화 현상 때문에 바람직하지 않다.

이를 위해 MicroC/OS-II에서는 연속된 메모리 공간으로 구성된 파티션에서 고정 크기의 메모리 블록을 할당할 수 있는 방법을 제공하며, 이 파티션은 사용 가능한 메모리 블록의 개수를 유지하고, 모두 같은 크기를 갖는 메모리 블록을 단일 연결 리스트의 형태로 관리 한다. 이런 형태의 메모리 관리 시스템은 메모리 단편화 현상이 잘 일어나지 않지만 이런 단순한 구조로는 메모리 공간을 통합 관리, 블록을 할당하고 반환하는데 필요한 검사등을 효율적으로 수행할 수 없다.

본 논문에서는 MicroC/OS-II에서의 단편화문제를 해결하고, 더 나아가 효율적으로 메모리를 통합하고 관리하는 방법에 대해 제안하고자 한다.

2. 관련연구

2.1 리눅스 버디 시스템(Buddy System) 알고리즘

외부 단편화 문제를 해결하기 위한 알고리즘으로, 리눅스에서는 사용하지 않는 모든 페이지 프레임을 그룹별로 묶어서 블록 리스트 11개에 넣는다. 각 리스트는 연속된 페이지 프레임 1, 2, 4, ... , 256, 512, 1024개로 구성된 그룹을 담는다.

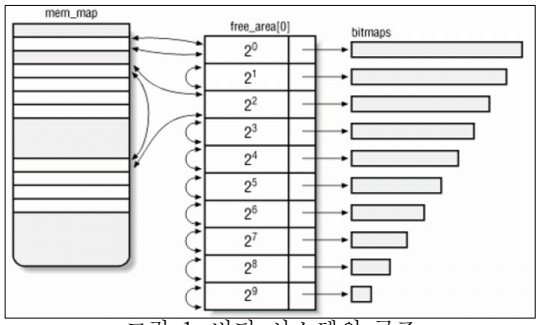

그림 1. 버디 시스템의 구조

(1) 알고리즘의 동작 예

연속된 페이지 프레임 128개로 구성된 그룹(즉 512KB)을 요청할 경우, 먼저 128-페이지 프레임 리스트에 여유 블록이 있는지 검사한다. 여유 블록이 없으면 다음으로 큰 블록, 즉 256-페이지 프레임 리스트에서 여유 블록을 찾는다. 여기에서 여유 블록 리스트를 발견하면 페이지 프레임 256개 중에서 128를 요청한 쪽으로 할당하고, 나머지 페이지 프레임 128개를 128-페이지 프레임 블록 리스트에 추가한다. 여유 256-페이지 블록이 없다면 다음으로 큰 블록 512-페이지 프레임으로부터 블록을 찾는다. 이

과정은 블록을 찾을때까지 1024-페이지 블록까지 반복되며 1024-페이지 프레임 블록 리스트가 비어 있다면 할당을 포기하고 에러를 발생 시킨다.

(2) 페이지 프레임 블록의 해지

커널은 크기가 b인 이웃하는(버디) 여유 블록 쌍을 크기가 2b인 더 큰 블록으로 만들려고 한다. 다음의 조건을 충족하면 두 블록은 버디 블록이다.

- 두 블록의 크기가 똑같이 b이다
- 두 블록이 연속된 물리적인 주소에 위치한다.
- 첫 번째 블록의 첫 번째 페이지 프레임의 물리 주소는 2 x b x 4K의 배수이다.

이 알고리즘은 순환적이다. 커널은 해지된 블록을 합친 후에 b의 값을 두 배로 늘려 더 큰 블록을 만들려고 한다.

2.2 슬랩 할당자(Slab Allocator)의 개념

1994년 썬 마이크로 시스템즈(Sun Microsystems)가 솔라리스 2.4 운영체제용에서 처음 사용한 정책에서 유래한다. 이는 다음과 같은 전제에 바탕을 둔다.

- 메모리 영역을 일련의 자료 구조와 '생성자(Constructor)'와 소멸자(Destructor)'라는 메소드(Method), 즉 두 함수를 포함한 '객체(Object)'로 바라본다. 생성자는 메모리 영역을 초기화하고, 소멸자는 나머지에 대한 정리를 한다. 슬랩 할당자는 객체를 반복해서 초기화하지 않도록 할당하다가 해지한 객체를 폐기하지 않고 메모리에 그대로 저장한다. 새로운 객체를 요청하면 초기화를 다시 하지 않고 메모리에서 이런 객체를 가져올 수 있다.
- 커널 함수는 같은 유형의 메모리 영역을 반복해서 요청하는 경향이 있다. 프로세스가 종료하면 이런 테이블을 포함하던 메모리 영역을 재활용 할 수 있다.
- 메모리 영역에 대한 요청은 빈도에 따라 분류할 수 있다. 자주 발생할 것 같은 특정 크기에 대한 요청은 크기가 동일한 특수 목적의 객체 집단을 만들어 가장 효율적으로 처리함과 동시에 내부 단편화 문제도 피할 수 있다.

슬랩 할당자는 객체를 모아서 '캐시(Cache)'로 만든다. 각 캐시는 타입이 같은 객체의 '창고(Store)'이다. 예를 들어, 파일을 열면 해당 '열린 파일(Open File)' 객체를 저장하는 데 필요한 메모리 영역을 filp(File Pointer)라는 슬랩 할당자 캐시에서 가져온다.

캐시가 들어 있는 주 메모리 영역을 '슬랩(Slab)'으로 나눈다. 각 슬랩은 연속된 페이지 프레임의 하나 이상으로 구성되며, 할당한 객체와 여유 객체를 모두 포함한다.

2.3 MicroC/OS-II에서의 메모리 관리

MicroC/OS-II의 메모리 관리 기능을 활성화하려면 OS_CFG.H에 있는 상수를 설정해야 한다. OS_MEM_EN을 설정하면 메모리 관리에 관련된 어떤 서비스도 활성화하지 않는다

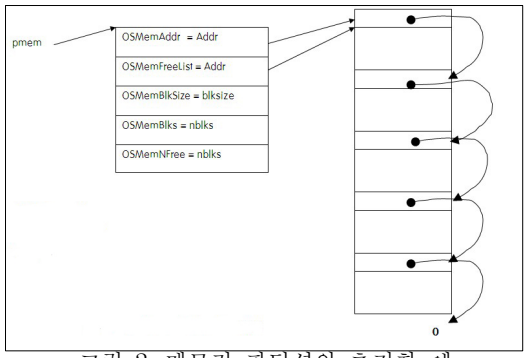

그림 2. 메모리 파티션의 초기화 예

(1) 메모리 컨트롤 블록

파티션의 정보를 유지 관리하기 위해 메모리 컨트롤 블록이라는 구조체를 사용한다. 각 파티션은 각자 자신의 메모리 컨트롤 블록을 갖는다.

(2) 파티션 생성 / OSMemCreate()

응용 프로그램에서 메모리 파티션을 사용하기 위해서는 먼저 OSMemCreate() 함수를 사용하여 생성해야 한다.

(3) 메모리 블록 할당 / OSMemGet()

OSMemGet() 함수를 사용하여 생성된 메모리 파티션으로부터 메모리 블록을 할당 받는다. 어떤 메모리 파티션으로부터 메모리 불록을 할당 받을 것인지는 OSMemCreate() 함수 호출 때 돌려 받았던 메모리 컨트롤 블록의 포인터에 따라 결정된다.

(4) 메모리 블록의 반환 / OSMemPut()

응용 프로그램에서 메모리 블록의 사용을 마치면 처음 할당 받은 파티션으로 해당 블록을 반환해야 하며 이때 사용되는 함수가 OSMemPut() 이다.

3. 설계

MicroC/OS-II에서는 메모리 관리 뿐 아닌 프로세스, 세마포어, 이벤트플래그등의 서비스를 사용하고 관리하기 위해 상수를 설정한다.

본 설계에서도 기존의 MicroC/OS-II에서 사용하는 메커니즘을 토대로 상수(OS_MEM_MAG)를 설정한다. 이 상수에 설정된 값에 따라 검사기능이 추가된 기존의 메모리 관리 방법을 사용할지, 버디시스템과 슬랩할당자를 이용한 메모리 관리 방법을 사용할지 선택하여 컴파일 할 수 있도록 설계 하였다.

(1) 메모리 반환시 검사를 수행

OS_MEM_MAG의 값이 0으로 설정된 경우 기존의 MicroC/OS-II에서 사용되는 파티션 관리 기법을 사용한다. 하지만 이 방식은 메모리 블록의 반환시 지정된 파티션으로부터 할당 받은 것인지 검사할 수 있는 기능이 결여되 있다.

따라서, OSMemChk() 함수를 제안하여 메모리 회수시 반환되는 블록을 검사하는 부분을 설계 하고자 한다.

```
INT8U OSMemPut(
    OS_MEM* pmem, void* pblk){
    ....
    pmem, pblk가 NULL값인지 검사
    OS_ENTER_CRITICAL();
    if(OSMemChk(pmem, pblk) ==
                    OS_NO_ERR){
        // pmem에 가득차 있는지 검사
        // 자유 메모리 블록 리스트에 삽입
        // 할당 가능한 블록 수 증가
    }
    OS_EXIT_CRITICAL();
    return (OS_NO_ERR);
}
```

그림 3. 수정된 OSMemPut() 함수

```
INT8U OSMemChk(
    OS_MEM* pmem, void* pblk){
    if(pmem으로부터 할당 받은 것이 아니면)
        return (OS_NOT_OWN);
    return (OS_NO_ERR);
}
```

그림 4. 추가된 OSMemChk() 함수

(2) 버디시스템 알고리즘과 슬랩할당자 개념의 적용

OS_MEM_MAG의 값을 1로 설정한 경우 기존의 MicroC/OS-II에서 사용되는 파티션 관리 기법을 배제하고 버디 시스템 알고리즘과 슬랩 할당자 개념을 적용할수 있도록 설계 하였다.

```
for(페이지 프레임 리스트의 순환 검색){
    if(해당 페이지 프레임(pmem)이 가용할 때)
        OSMemGet(pmem, err);
    return (OS_MEM_NO_FREE_BLKS);
}
```

그림 5. 블록 할당 알고리즘

```
while(페이지 프레임 리스트의 순환){
    if(!OSMemIsBuddy(pmem, pblk))
        break;
    // 두 여유 블록의 결합
}

INT8U OSMemIsBuddy(
    OS_MEM* pmem, void* pblk){
    if(버디 블록의 조건을 만족하지 않는다면)
        return 0;
}
```

그림 6. 블록 해지 알고리즘

```
typedef struct{
    void* OSMemAddr;
    void* OSMemFreeList;
    void** OSMemSlab;  //Slab 객체와의 연결
    INT32U OSMemBlkSize;
    INT32U OSMemNBlks;
    INT32U OSMemNFree;
}
```

그림 7. 수정된 메모리 컨트롤 블록 구조체

이 방식은 메모리를 통합적으로 관리하고 효율적으로 운용할 수 있지만 기존의 파티션 관리 기법에 비해서 추가의 자료 공간이 필요하며 버디시스템에서 사용되는 보다 큰 블록으로 만들려는 순환적 알고리즘으로 계산시간이 증가할 수 있다는 단점이 존재한다.

4. 설계, 구현시 고려해야 할 프로세서 요구사항

- 재진입(Reentrancy)을 지원하는 코드를 생성할 수 있는 C 컴파일러
- C 언어에서 인터럽트 비활성화, 활성화 지원
- 인터럽트 지원 및 일정 주기로 발생하는 타이밍 인터럽트의 제공
- 프로세서 수준에서 지원하는 적정 크기의 하드웨어 스택 기능(수 KByte 정도 크기)
- 스택 포인터 또는 레지스터의 내용을 스택이나 메모리로 저장하고 가져올 수 있는 프로세서 명령어

5. 결론 및 향후 과제

본 논문에서는 MicroC/OS-II에서의 단편화문제를 해결하고, 더 나아가 효율적으로 메모리를 통합하고 관리하는 방법에 대해 제안하였다. 위에서 언급되었던 방법들을 실제로 구현함으로써 메모리 블록의 반환시 지정된 파티션으로 할당받은 것인지 검사할 수 있으며, 사용되고 있는 중이나 비어있는 메모리 공간들을 통합적으로 관리할 수 있으며, 부수적으로 메모리의 접근 시간을 감소시키는 효과도 가질 것이다. 또한, 메모리 요청과 반환에 대한 효율성도 증가할 것이다.

추후 더 나아가 설계시 부족했던 부분을 보완하고 실제로 MicroC/OS-II의 기능을 지원하는 프로세서에 포팅 해봄으로 실시간 운영체제로서의 실시간성을 분석해보고, 실제로 메모리가 얼마나 효율적으로 운용되었는지 분석이 필요하다.

참고문헌

[1] Beck, M., et. al., "Linux Kernel Programming 3rd ed.", Addison Wesley, 2002

[2] Allworth, Steve T. 1981. Introduction To Real-Time Software Design. New York: SpringerVerlag. ISBN 0-387-91175-8

[3] Gareau, Jean L. 1998. Embedded x86 Programming: Protected Mode. Embedded Systems Programming. April, p. 80-93

[4] Kernighan, Brian W. and Dennis M. Ritchie. 1988. The C Programming Language. 2nd edition. Englewood Cliffs, New Jersey: Prentice Hall. ISBN 0-13-110362-8

[5] Jean J. Labrosse, "MicroC/OS-II 실시간 커널 2판", 에이콘, 2005년

[6] Bonwick, J. "The Slab Allocator: An Object-Caching Kernel Memory Allocator", Proceedings of Summer 1994 USENIX Conference, pp. 87-98

[7] 한동훈, "리눅스 커널 프로그래밍", 한빛 미디어, 2007년. ISBN 978-89-7914-460-4