Project #7: 동적메모리 할당 및 빈공간 관리기법 구현

신정은, 조재호, 조한주
School of Software, Soongsil University

1. 구현 목표

동적 메모리 관리 기능은 태스크가 사용할 메모리를 미리 할당해 놓는 것이 아니라 필요한 시점에 할당받아 사용한 후 다시 반환하는 방식이다. 메모리를 특정 태스크에 고정하지 않으므로 한정된 메모리를 효율적으로 사용할 수 있다. 따라서 동적 메모리 할당과 해제 기능을 구현해 보고 동적으로 메모리를 처리할 때 발생하는 단편화 문제와 이를 효과적으로 줄일 수 있는 방법을 구현해 본다.

기본 구현을 해보았다면 빈 공간 관리 기법을 추가 구현해 본다. 기본 구현의 버디할당자를 통해 큰 메모리 chunk를 할당한 뒤, malloc과 free 명령어를 구현하여 메모리 할당자를 구현해 본다. 빈공간의 크기와 다음 빈공간의 주소를 저장하는 헤더 자료구조도 구현하여 모든 빈공간을 linked list로 연결한다.

2. 구현 내용

태스크가 고정적으로 메모리를 할당받는 것이 아니라 필요할 때 할당받는 방식을 동적 메모리 할당이라고 한다. 동적 메모리 관리 기법의 최대 관심사는 메모리 할당과 해제를 반복할 때 메모리를 조각나지 않게 관리하면서 최대한 신속하게 메모리를 할당하는 것이다.

2.1 메모리 단편화와 버디 블록 알고리즘

메모리 공간은 유한하고 태스크의 메모리 요청은 다양하기 때문에 요청이 들어올 때마다 무작정 할당해주면 자투리 메모리만 남아서 할당할 수 없게 될 수 있다. 이렇게 메모리가 작은 영역으로 조각난 상태를 메모리 단편화라고 한다. 그 중 외부 단편화는 남은 여유 공간을 합하면 요청한 메모리 크기를 만족하지만, 여유 공간이 연속적이지 않아서 메모리를 할당할 수 없는 상태를 말한다. 외부 단편화를 줄일 방법으로 버디 블록 알고리즘이 있다.

버디블록 알고리즘은 메모리 할당과 해제가 반복될 때마다 인접한 블록 그룹을 합하여 상위 블록으로 만들거나 상위 블록을 절반으로 나누어 하위 블록 그룹으로 분리한다. 버디 블록 알고리즘은 요청한 크기와 근접한 블록을 우선으로 검색하고, 메모리가 해제된 뒤에는 인접한 블록을 합하여 더 큰 블록으로 결합하기 때문에 문제를 효율적으로 처리할 수 있다.

버디블록 알고리즘은 블록의 크기가 최소 크기 블록의 2배, 4배, 8배,,2^n배로 고정되어 있다. 따라서 블록 단위보다 조금 더 큰 크기를 요청하면 사용하지 않는 메모리까지 할당받게 되어 내부단편화 문제가 일어나게 된다. 버디 블록 크기에 최대한 맞춰서 할당하거나 메모리를 풀 형태로 구성하여 최대한 버디 블록 단위로 할당받은 뒤 나누어 사용하는 것도 좋은 해결책이 될 수 있다.

2.2 버디 블록 알고리즘 구현

2.2.1 동적 메모리 할당을 위한 메모리 영역 지정

메모리를 동적으로 할당하고 해제하려면 전체 메모리 영역에서 어느 만큼을 동적 메모리로 사용할 지 정해야 한다. 지금까지 약 17MB영역 아래를 사요했으므로 동적 메모리 영역의 시작 어드레스를 17MB로 한다. 그리고 3GB범위 내의 물리 메모리의 끝까지를 동적 메모리 영역으로 설정한다.

2.2.2 버디 블록 알고리즘 세부 설계

버디 블록 알고리즘은 메모리를 할당하고 해제할 때마다 블록을 분할하고 결합하는 작업을 한다. 이러한 작업을 빠르게 수행할 수 있도록 블록 리스트를 유지하는 것이 중요하다. 하지만 리스트에 데이터가 많으면 메모리를 해제할 때 인접한 블록을 확인하는데 상당한 시간이 걸리므로 데이터 존재 유무만 확인하면 되므로 비트맵을 사용한다. 비트맵은 데이터당 1비트를 할당하여 0이나 1로 정보를 표시하는 방법이다. 리스트와 달리 인접한 블록에 바로 접근할 수 있다. 1바이트 내에서는 비트0부터 비트7까지 순서로 비트를 할당하며, 바이트 순서는 첫번째 바이트부터 마지막 바이트 순으로 할당하는 방식을 사용한다.

비트맵 자료구조는 비트맵의 어드레스와 비트맵 내에 존재하는 데이터의 개수로 구성되어 있다. 전체가 비어있는 비트맵에서 블록을 찾느라 하는 시간낭비를 없애기 위해서이다. 버디블록 자료구조에는 할당된 블록이 속하는 블록리스트의 인덱스를 저장한다. 블록을 해제할 때 블록 리스트의 인덱스를 알아야 해당 블록 리스트에 삽입할 수 있기 때문이다. 또한 메모리의 크기에 따라 비트맵과 할당

된 메모리 크기를 저장하는 영역을 다르게 지정하기 위한 필드도 버디블록 자료구조에 저장한다. 할당된 메모리의 크기를 저장하는 영역과 비트맵 크기를 합하면 약 80KB가 된다. 따라서 동적 메모리로 사용하려고 할당된 영역의 앞부분을 비트맵과 할당된 크기를 저장하는 영역으로 사용하고, 나머지 영역을 실제 버디 블록이 존재하는 블록 풀로 사용한다.

2.2.3 동적 메모리 자료구조 초기화

비트맵과 할당된 크기를 저장하는 영역을 초기화한다. 이를 위해 각 영역이 사용하는 크기를 반환해주는 함수를 구현한다. 그리고 동적 메모리의 크기를 구하는 함수를 통해 동적 메모리 크기를 구한 후, 할당된 크기가 저장될 영역을 초기화한다.

비트맵 영역은 비트맵 자료구조와 실제 정보를 저장하는 비트맵으로 구성된다. 비트맵 자료구조의 시작 어드레스는 할당된 메모리의 블록 리스트 인덱스를 저장하는 영역의 바로 다음에 위치하므로, 동적 메모리 영역이 시작하는 어드레스에서 그만큼 크기를 더하면 된다. 어드레스 초기화가 끝나면 루프를 돌면서 비트맵과 비트맵의 시작 어드레스, 비트맵에 존재하는 데이터의 개수를 삽입하는 작업을 수행한다. 홀수가 되어 상위 블록으로 결합될 수 없는 자투리 블록은 블록 리스트의 가장 마지막에 연결해서 여분으로 사용할 수 있도록 한다.

2.2.4 메모리 할당 기능 구현

메모리 할당 과정은 요청된 메모리 크기를 버디 블록 크기로 정렬하고, 할당된 블록에 대한 정보를 저장하는 부수적인 부분과 실제로 버디 블록 리스트에서 블록을 할당하는 부분으로 나눌 수 있다. 그 전에 버디 블록 할당 함수를 호출하여 블록을 찾은 후 실제 메모리 어드레스로 변환하는 과정을 거쳐야 한다. 이 과정은 할당 받은 블록의 크기와 오프셋을 곱하여 블록 풀을 기준으로 하는 어드레스를 계산한 후, 다시 블록 풀의 시작 어드레스를 더해 실제 메모리 어드레스로 변환하는 것이다. 메모리 블록을 할당한 후 반드시 할당한 블록이 속한 블록 리스트의 인덱스를 저장하여야 한다. 블록 해제시 필요하기 때문이다. 그리고 kGetBuddyBlockSize()함수를 통해 요청한 메모리의 크기를 버디 블록 크기로 정렬한다. 이제 할당을 위한 준비가 끝났으니, kAllocationBuddyBlock()함수를 통해 버디 블록 리스트에서 존재하는 블록을 찾아 할당한 후 블록의 상태에 따라 비트맵을 설정하도록 한다.

2.2.5 메모리 해제 기능 구현

해제하는 과정은 할당하는 과정을 반대로 진행한다. 크게 어드레스로 버디 블록이 속한 블록 리스트의 인덱스와 블록 오프셋을 구하는 부분과 블록 리스트에 블록을 반환하는 부분으로 나눌 수 있다. 실제 메모리 어드레스에서 블록 풀의 시작 어드레스를 뺀 뒤 블록 풀을 기준으로 하는 어드레스를 계산하여 블록 리스트 인덱스를 구한다. 어드레스를 구했다면 이를 가장 작은 버디 블록의 크기로 나누어 할당된 블록 리스트의 인덱스를 저장하는 영역의 오프셋을 찾고, 블록 풀을 기준으로 하는 어드레스를 할당한 블록의 크기로 나누어 블록 오프셋을 계산한다.

이제 블록을 해제한 뒤 인접한 블록을 결합하여 상위 블록으로 만든다. 반환된 블록의 오프셋이 짝수라면 홀수 위치가 인접한 블록이 되고, 홀수라면 짝수가 인접한 블록이 된다. 확인하고 인접한 블록이 존재하면 두 블록을 결합하여 상위 블록으로 만든다. dynamicmeminfo커맨드를 추가하여 동적 메모리 영역이 시작하는 어드레스와 해당 영역의 전체 크기, 비트맵과 할당된 블록 크기를 저장하는데 사용한 영역, 사용된 메모리 영역을 표시한다. 그리고 testseqalloc커맨드를 구현하여 동적메모리 할당과 해제를 테스트한다. 가장 작은 블록부터 가장 큰 블록까지 차례로 할당하고 검사한 후 해제하는 테스트이다. 또 testranalloc커맨드를 통해 태스크 100개를 동시에 생성하여 각 태스크가 임의의 메모리를 할당하고 해제하는 테스트를 수행한다. 멀티 태스킹 환경에서 동기화 문제가 발생하는지, 메모리 영역의 할당과 해제가 정상적으로 처리되는지 확인하기 위함이다. 모든 태스크가 종료되었을 때 동적 메모리 영역의 정보를 확인하여 이상 유무를 검사할 수 있다.

2.3 빈 공간 관리 기법 구현

빈 공간 관리 기법을 추가 구현해 본다. 먼저 빈 공간의 정보를 저장하기 위한 자료구조를 만든다. 빈 공간의 정보에는 빈공간의 크기와 다음 빈공간의 주소가 있어야 한다.

```
typedef struct __node_t
{
    unsigned int size;
    struct __node_t *next;
}nodet_t;
```

이제 실제로 malloc과 free를 구현해 본다. testinit명령어를 통해 3G 메모리 chunk를 할당한 한다. 할당하는 방법은 버디할당자를 통해 할당받는다. 다음은 3G메모리 chunk를 버디할당해주는 kMemoryInit() 함수이다.

```
void* kMemoryInit(QWORD qwAlignedSize)
{

    QWORD qwRelativeAddress;
    void *root;
    nodet_t *head;

    int iSizeArrayOffset;
    int iIndexOfBlockList;
    QWORD as;
    long ioffset;

    as = kGetBuddyBlockSize(qwAlignedSize);
    ioffset = kAllocationBuddyBlock(as);

    iIndexOfBlockList = kGetBlockListIndexOfMatchSize( qwAlignedSize );

    // 블록 크기를 저장하는 영역에 실제로 할당된 버디 블록이 속한 블럭 리스트의
    // 인덱스를 저장
    // 메모리를 해제할 때 블록 리스트의 인덱스를 사용
    qwRelativeAddress = as * ioffset;
    iSizeArrayOffset = qwRelativeAddress / DYNAMICMEMORY_MIN_SIZE;
    gs_stDynamicMemory.pbAllocatedBlockListIndex[ iSizeArrayOffset ] =
        ( BYTE ) iIndexOfBlockList;
    gs_stDynamicMemory.qwUsedSize += qwAlignedSize;
    root = ( void* ) ( as * ioffset + gs_stDynamicMemory.qwStartAddress );


    head = root;
    head->size = (unsigned int)(0xC0000000 - sizeof(nodet_t));
    head->next = NULL;


}
```

큰 메모리 chunk를 할당해 놓았으면, 빈 공간을 할당하는 malloc 명령어를 구현한다. 빈 공간 리스트에서 할당하려는 메모리+헤더 크기(nodet_t크기)보다 크거나 같은 빈 공간을 찾아 메모리를 할당한다. 빈 공간 리스트의 첫 번째 노드에서 메모리를 할당하는 경우와, 빈 공간 리스트의 중간 노드에서 메모리를 할당하는 경우, 빈 공간 리스트의 마지막 노드에서 할당하는 경우가 모두 만족하도록 구현한다. 할당해주는 메모리의 주소값을 리턴한다.

메모리를 할당하는 명령어를 구현하였으니 메모리를 해제하는 free 명령어를 구현한다. free 명령어의 파라미터로 해제할 메모리의 주소값을 넘겨준다. 받은 주소값에 헤더 크기를 뺀 후 head가 가르키도록 한다. 그 후 head-> next가 빈 공간 리스트의 첫번째 노드를 가르키도록 한다. 빈 공간 리스트를 가르키고 있는 포인터가 해제할 메모리를 가리키도록 하면 메모리 해제가 완료된다.

개발 시 경험한 문제점 및 해결점

빈 공간 리스트에서 메모리 할당시 다양한 케이스

빈 공간 리스트에서 메모리 할당시 다양한 케이스가 있어서 어려웠다. 할당 할 chunk에 연결된 리스트가 있는 경우와 없는 경우가 대표적인 케이스다. 최대한 모든 케이스에서 메모리 할당이 제대로 이루어지게 하다 보니까 어려운 점이 있었다.

Linked list의 연속적인 연결의 어려움

빈 공간 리스트의 중간에서 메모리 할당을 받는다면 앞의 노드와 뒤의 노드가 연결되도록 해야만 했다. 이러한 부분들을 해결하다 보니 많은 if else문장이 들어가게 되었다.

3. 조원별 기여도

신정은: 33.3%
조재호: 33.3%
조한주: 33.3%