

2014년 시스템 프로그래밍

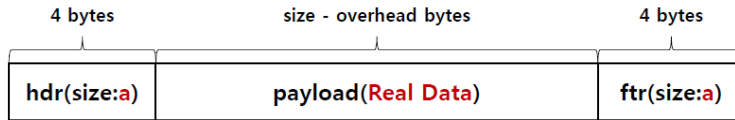
- HW 10 -

분 반	03
이 름	정원희
학 번	200802201
제출일	2014.12.10.

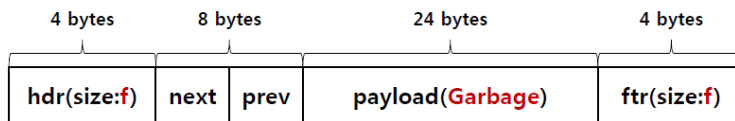
1. Explicit

가. 해결 포인트

▪ <Allocated block>



▪ <Free block>



위 그림은 explicit 에서 사용되는 블록의 잘 구조이다. Explicit 은 implicit 과 다르게 명시적으로 free block 들을 관리한다. Free 블록들을 링크드리스트로 엮기 위해 freeblock 내부에는 next 노드를 가리키는 next 필드와 이전 free 블록을 가리키는 prev 필드가 존재하게 된다. 알맞은 free 블록을 찾기 위해 거의 모든 블록을 다 찾았던 implicit 과 다르게 explicit 은 free 블록의 리스트만 순회하여 찾게 된다 이 때문에 더 빠르고 효율적으로 메모리를 관리 할 수 있게 된다.

explicit 에서 가장 중요한 처리는 free 블록들을 링크드리스트로 유지 하는 것이다. 새로운 블록이 free 되었을 때, free 블록이었던 블록을 할당할 때 링크드리스트를 수정하여야 한다. 새로운 블록이 FREE 되었을 때에는 해당 노드를 링크드리스트에 추가하여야 한다. LIFO 로 관리를 한다면 링크드리스트의 처음을 가리키고 있는 포인터를 새로운 블록을 가리키도록 수정하고 추가된 블록은 이전의 처음블록을 가리키도록 처리 하여야 한다. 또 삭제 되었을 때 처음블록인가, 가장 끝의 블록인가, 중간에 있는 블록이었는데가 따라 각 블록의 prev, next 필드를 적절하게 수정 하여야 한다.

이러한 링크드리스트 연산만 제대로 처리 한다면 explicit 할당기 구현은 비교적 간단하게 끝낼 수 있을 것이다.

나. 매크로 설명

A. #define HDRSIZE 4

Header 의 사이즈

B. #define FTRSIZE 4

Footer 의 사이즈

C. #define WSIZE 4

1 Word 사이즈

D. #define DSIZE 8

Double Word 사이즈

E. #define CHUNKSIZE (1<<12)

초기 가용블록과 힙확장을 위한 기본크기를 지정한다. 1 을 12 비트 left shift 한 값으로 지정한다. 1000 0000 0000 (4096) 이 된다.

F. #define OVERHEAD 8

header 와 footer 의 크기의 합을 나타낸다. 실제로 메모리블록을 할당할 때 사용자가 요청한 크기의 블록만 할당하는 것이 아니라 이전과 다음위치로 이동할 수 있도록 할당 사이즈와 할당정보를 저장하기 위한 필드가 더 필요하다 때문에 OVERHEAD 라 불릴 수 있다.

G. #define MAX(x,y) ((x)>(y) ? (x):(y))

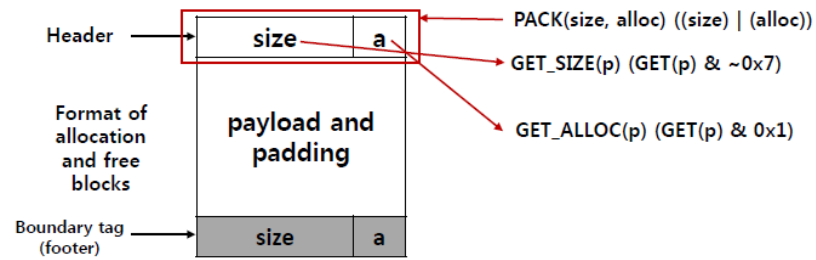
x 와 y 중 큰 값을 반환한다.

H. #define MIN(x,y) ((x)<(y)) ? (x):(y))

x 와 y 중 작은 값을 반환 한다.

I. #define PACK(size, alloc) ((unsigned) ((size)| (alloc)))

메모리 블록의 구조를 보면 위 그림과 같다. Header, payload, footer 로 이루어 지게 된다. header 는 현재 할당된 메모리의 크기, 할당됐는지 안됐는지를 나타내는 정보를 저장한다. payload 는 실제 사용자가 저장할 공간을 나타내고, footer 는 header 와 마찬가지로 정보를 지닌다.



J. #define GET(p) (*(unsigned *) (p))

p 가 참조하는 한 워드를 읽어서 리턴 한다

K. #define PUT(p,val) (*(unsigned*)(p)= (unsigned)(val))

p 가 가리키는 곳에 한 word 에 val 값을 저장한다.

L. #define GET8(p) (*(unsigned long *) (p))

p 가 참조하는 한 워드를 읽어서 리턴 한다. Unsigned long 은 32 비트 64 비트 시스템에서 동일한 동작을 하기 위해서 unsigned long 형으로 형변환을 한다. Unsigned 의 경우 32 비트 시스템과 64 비트 시스템 모두 4 바이트이지만 64 비트에서는 8 바이트 이다. 64 비트 시스템의 주소 범위를 unsinged 에서 표현 불가능하므로 unsigned long 으로 읽어 32 비트와 64 비트에서 동일하게 동작하도록 한다.

M. #define PUT8(p,val) (*(unsigned long*)(p)= (unsigned long)(val))

p 가 가리키는 곳에 한 word 에 val 값을 저장한다.

N. #define GET_SIZE(p) (GET(p) & ~0x7)

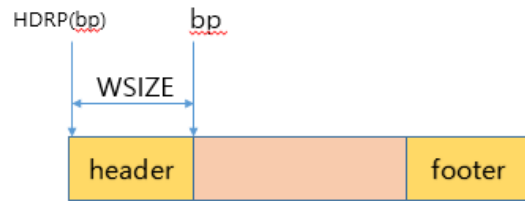
p 가 참조하고 있는 곳에 한 워드 값의 뒤에 3 비트를 버린다. 이는 header 와 footer 에서 size 값을 읽어 반환 하는 것과 같다.

O. #define GET_ALLOC(p) (GET(p)& 0x1)

p 가 참조하고 있는 곳의 한 워드 값을 읽어 하위 1 비트만 읽는다. 이는 header 와 footer 에서 할당비트를 읽는 것과 마찬가지로 효과를 지닌다.

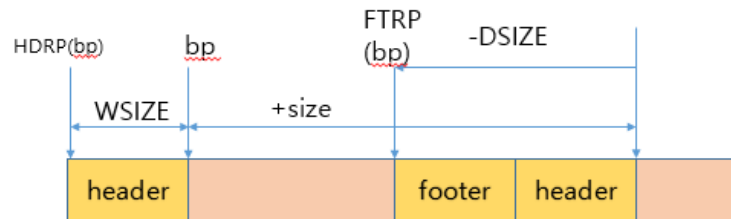
P. #define HDRP(bp) ((char*)(bp)-WSIZE)

주어진 포인터 bp 에서 header 주소를 반환한다.



Q. **#define FTRP(bp) ((char*)(bp)+ GET_SIZE(HDRP(bp))-DSIZE)**

주어진 포인터에서 footer 의 주소를 반환한다.



R. **#define NEXT_BLKBP(bp) ((char*)(bp)+ GET_SIZE(HDRP(bp)))**

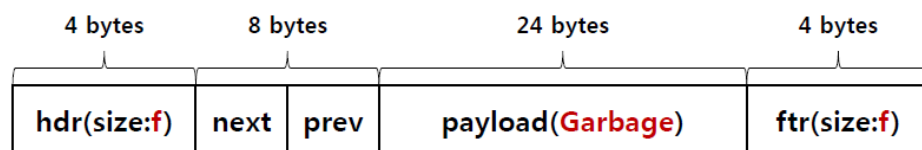
주어진 포인터 bp 를 이용하여 다음 블록의 주소를 계산한다

S. **#define PREV_BLKBP(bp) ((char*)(bp)-GET_SIZE((char*)(bp)-DSIZE))**

주어진 포인터를 이용하여 이전 블록의 주소를 계산한다.

T. **#define NEXT_FREEBP(bp) ((char*)(bp))**

주어진 포인터 bp 가 가리키는 FREE Block 의 NEXT FREE Block 을 가리키는 포인터가 저장되어있는 필드의 포인터를 반환한다.



U. **#define PREV_FREEBP(bp) ((char*)(bp)+WSIZE)**

주어진 포인터 bp 가 가리키는 FREE Block 의 PREV FREE Block 을 가리키는 포인터가 저장되어있는 필드의 포인터를 반환한다. WSIZE 를 더하여 접근가능하다.

V. **#define NEXT_FREE_BLKBP(bp) ((char*) GET8((char*)(bp)))**

포인터 bp 가 가리키고 있는 블록의 NEXT_FREE 가 가리키고 있는 필드의 내용을 반환한다. 즉 다음 FREE Block 의 주소를 반환한다.

W. #define PREV_FREE_BLKPB(bp) ((char*)GET8((char*)(bp)+WSIZE))

포인터 bp 가 가리키고 있는 블록의 PREV_FREE 가 가리키고 있는 필드의 내용을 반환한다. 즉 이전 FREE Block 의 주소를 반환한다.

다. 함수 설명

A. int mm_init(void)

```

c200802201@eslab:~/malloclab-handout
int mm_init(void) {
    char* h_ptr;
    if((h_ptr= mem_sbrk(DSIZE+4*HDRSIZE))==NULL)
        return -1;
    heap_start=h_ptr;

    PUT(h_ptr, NULL); //next
    PUT(h_ptr+ WSIZE, NULL); //prev
    PUT(h_ptr+DSIZE, 0);

    // prologue
    PUT(h_ptr+DSIZE+HDRSIZE, PACK(OVERHEAD,1)); // prologue hdr
    PUT(h_ptr+DSIZE+HDRSIZE+FTRSIZE, PACK(OVERHEAD,1)); // prologue ftr
    PUT(h_ptr+DSIZE+2*HDRSIZE +FTRSIZE,PACK(0,1)); //epilogue

    h_ptr+=DSIZE+DSIZE; //prologue hdr, ftr 사 이 에 위 치 .
    freeblkp=NULL;

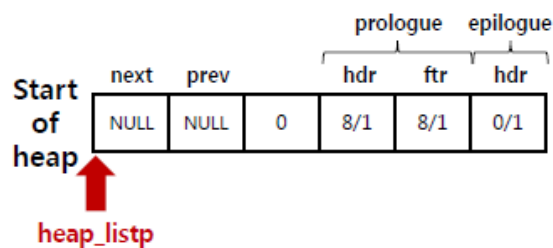
    //nextftrp=NULL;

    epilogue= h_ptr+HDRSIZE;
    if(extend_heap(80/WSIZE)==NULL)
        return -1;

    return 0;
}

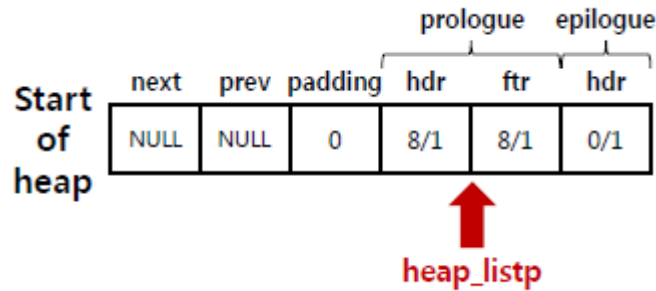
```

위 함수는 동적 할당을 위하여 힙을 초기화 하는 함수 이다.



위와 같은 초기 메모리를 구성하기 위해 HEAP 영역을 DSIZE+4*HDRSIZE 만큼 확장한다 즉 8+4*4 만큼 확장한다. 다음 초반의 사용하지 않는 영역을 세팅한다.

위 그림에서 next, prev , 0 에 해당하는 부분을 위 그림처럼 설정한다. 다음 prologue 블록과 epilogue 블록을 implicit 과 같이 설정한다.



설정을 다한 후 heap 블록의 시작을 알리는 heap_listp 를 프롤로그블록의 header와 footer사이에 위치 시킨다. 본 실습에서는 freeblock을 링크드리스트로 관리 하므로 freeblock 의 링크드리스트를 가리키는 freeblkp 라는 포인터 변수를 전역으로 설정 하였다. Heap_listp 로 사용할 수도 있지만 heap_listp 는 전체 블록의 리스트를 가리키는 의미로 사용하기 위하여 따로 freeblock 의 리스트를 가리키는 freeblkp 를 선언하였다. 이 변수는 아직 free block 이 존재하지 않으므로 NULL로 초기화 한다. Freeblkp가 NULL이라면 최초 할당이라는 의미를 가지게 된다.

다음 free 블록으로 사용될 초기 heap 을 확장한다. 이 때 CHUNKSIZE 만큼의 블록을 생성하지 않고 80 이라는 크기의 힙만 확장하였다. 이는 무작정 크게 힙을 확장하여 utilization 이 떨어지는 것을 막기 위하여 적절한 크기의 정수로 지정하였다. 8 의 배수인 80 을 넣었을 때 가장 효과가 큰 것으로 나타났다

B. void* extend_heap(size_t words)

implicit 에서 extend_heap 과 같다. extend_heap 은 요청한 워드 수 만큼의 힙영역을 확장한다. 이 함수는 힙을 초기화 할 때, malloc 호출시 적당한 사이즈를 찾을 수 없어 메모리를 확장할 수 없을 때 호출된다. 메모리 정렬을 유지하기 위해 요청된 싸이즈를 8 의 배수에 가깝게 보정한다. 확장된 메모리 만큼 힙 메모리를 확장하고 header와 footer 를 세팅한다. 다음 에필로그 해더를 세팅 한다. coalesce 함수를 호출하여 확장 이전의 블록과 통합을 한다.

```

c200802201@eslab:~/malloclab-handout
inline void *extend_heap(size_t words)
{
    unsigned *old_epilogue;
    char *bp;
    unsigned size;

    size=(words%2 ) ? (words+1)*WSIZE: words*WSIZE;

    if ((long) (bp=mem_sbrk(size))<0)
        return NULL;

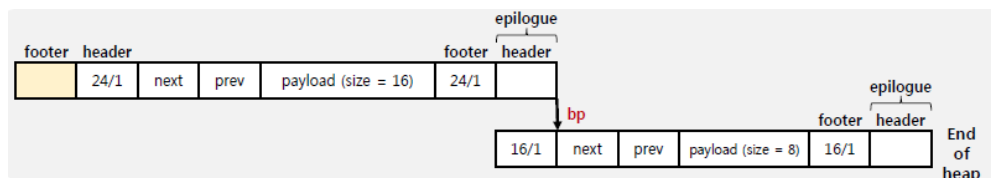
    old_epilogue= epilogue;
    epilogue= bp+size- HDRSIZE;

    PUT(HDRP(bp), PACK(size,0));
    PUT(FTRP(bp), PACK(size, 0));
    PUT(epilogue, PACK(0,1));

    return coalesce(bp);
}

```

Epilogue 블록은 새로 확장된 heap 블록의 header 로 이용된다.



C. void addFreeBlock(void* bp)

explicit 에서 링크드리스트 연산을 위하여 정의한 함수이다. addFreeBlock 함수는 새로운 Free block 을 링크드리스트에 추가하는 함수이다. 이때 LIFO 방식으로 추가 됨으로 링크드리스트의 처음노드에 추가시키게 된다. Free 블록 링크드리스트의 처음노드를 가리키고 있는 변수 freeblkp 는 NULL 로 초기화 되어있다. 때문에 NULL 인지 아닌지에 따라 다르게 동작하여야 한다. NULL 이라면 아무런 free 블록이 추가 되어있지 않은 상태이므로 인자로 전달된 bp 가 가리키고 있는 블록을 처음 노드로 추가 하여야 한다. 이 때 추가된 free 블록 밖에 없으므로 free 블록의 next, prev 필드는 NULL 을 가리키게 설정한다.

반면 이미 free 블록들이 링크드리스트에 추가가 되어있는 경우, 기존의 freeblkp 가 가리키고 있는 블록을 새로이 추가될 블록으로 바꿔 줘야 한다. 그리고 추가된 블록의 next 를 이전에 freeblkp 가 가리키고 있는 노드로 바꿔주고 prev 는 NULL 로 설정한다. 또한 freeblkp 가 가리키던 노드의 prev 필드를 새로 추가된 블록을 가리키도록 설정하여야 한다.

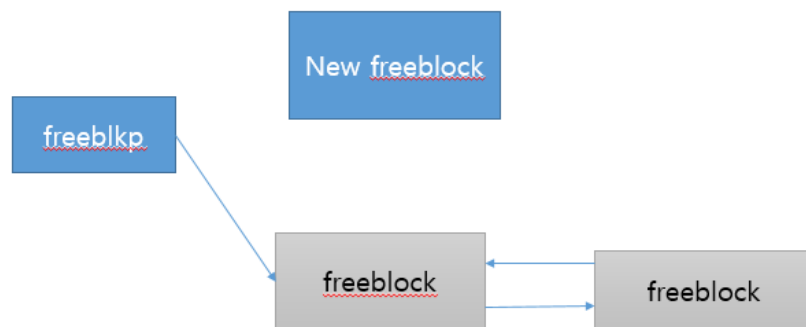

```
c200802201@eslab:~/malloclab-handout
static void addFreeBlock(void* bp)
{
    char *temp;

    if (freeblkp == NULL)
    {
        freeblkp = bp;
        nextfrip = freeblkp;
        PUT8 (NEXT_FREEP (freeblkp), NULL);

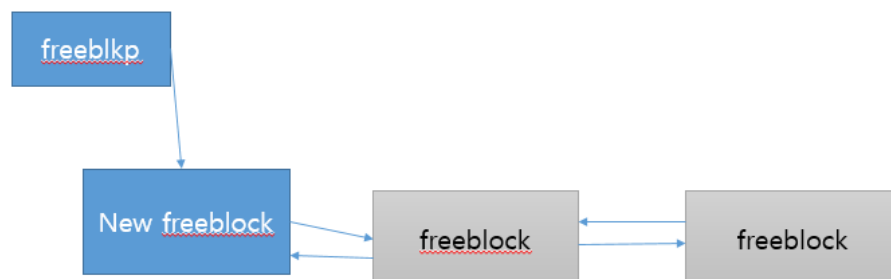
        PUT8 (PREV_FREEP (freeblkp), NULL);
    } else
    {
        PUT8 (NEXT_FREEP (bp), freeblkp);
        PUT8 (PREV_FREEP (bp), NULL);
        PUT8 (PREV_FREEP (freeblkp), bp);

        freeblkp = bp;
    }
}
```


그림으로 표현하면 아래와 같다.



추가된 후 아래와 같은 그림처럼 된다.



D. void deleteFreeBlock(void* bp)

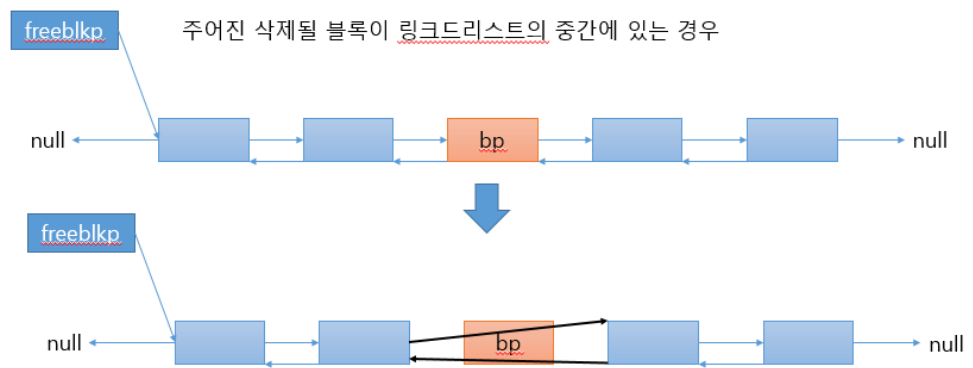


```
static void deleteFreeBlock(void* bp)
{
    if (NEXT_FREE_BLKP(bp) == NULL && PREV_FREE_BLKP(bp) == NULL) // 단독 .
    {
        freeblkp = NULL;
    }
    else if (NEXT_FREE_BLKP(bp) != NULL && PREV_FREE_BLKP(bp) != NULL) //중 간
    {
        PUT8(NEXT_FREEP(PREV_FREE_BLKP(bp)), NEXT_FREE_BLKP(bp));
        PUT8(PREV_FREEP(NEXT_FREE_BLKP(bp)), PREV_FREE_BLKP(bp));
    }
    else if (NEXT_FREE_BLKP(bp) == NULL && PREV_FREE_BLKP(bp) != NULL) //리 스토 의 끝 .
    {
        PUT8(NEXT_FREEP(PREV_FREE_BLKP(bp)), NULL);
    }
    else if (NEXT_FREE_BLKP(bp) != NULL && PREV_FREE_BLKP(bp) == NULL) //리 스토 의 처음
    {
        freeblkp = NEXT_FREE_BLKP(bp);
        PUT8(PREV_FREEP(NEXT_FREE_BLKP(bp)), NULL);
    }
}
```

deleteFreeBlock 함수 또한 링크드리스트 연산을 위한 함수이다. 위 함수는 free 블록 리스트에서 블록을 삭제할 때 삭제된 블록 이전과 이후 블록을 서로 연결시켜 주는 작업을 하는데 4 가지 조건에 따라 다르게 동작하게 된다.

첫번째, 인자로 주어진 리스트에서 해제될 블록이 링크드리스트에서 유일한 경우를 생각할 수 있다. 이때는 해당블록의 prev, next 필드가 NULL이다. 간단히 freeblkp를 NULL을 가리키게 함으로써 해결할 수 있다.

두번째, 주어진 삭제될 블록이 링크드리스트의 중간에 있는 경우를 생각할 수 있다. 조건은 prev 필드와 next 필드가 모두 이전과 다음블록을 가리키고 있는 상태이다. 아래 그림과 같이 bp의 앞뒤 블록들을 서로 가리키도록 바꿔주는 연산을 수행한다. 이렇게 됨으로써 bp는 리스트에서 삭제된 상태가 된다.



세번째, 리스트의 끝인 경우를 생각 할 수 있다. 조건은 bp의 next 필드가 NULL, prev 필드가 NULL 이 아닌 경우이다. 아래 그림과 같이 bp의 이전 블록의 next 필드를 null로 돌려 놓음으로 해결 할 수 있다.



네번째, 리스트의 처음인 경우이다. 조건은 next 필드가 NULL 이 아니고, prev 필드가 NULL 일 때 이다. 아래 그림과 같이 freeblkp를 bp의 next 블록을



가리키게 바꾸고 next 블록의 prev 필드를 NULL 로 바꿔 처음 노드를 삭제 할 수 있다.

E. void* coalesce(void* bp)

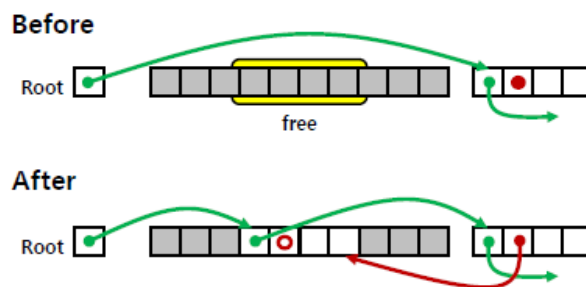


```
static void* coalesce(void* bp)
{
    size_t prev_alloc= GET_ALLOC(FTRP(PREV_BLKP(bp)));
    size_t next_alloc=GET_ALLOC(HDRP(NEXT_BLKP(bp)));
    size_t size= GET_SIZE(HDRP(bp));
    char *tempp;
    if(prev_alloc && next_alloc)
    {
        addFreeBlock(bp);
    }
    else if(prev_alloc && !next_alloc)
    {
        size+= GET_SIZE(HDRP(NEXT_BLKP(bp)));
        deleteFreeBlock(NEXT_BLKP(bp));
        PUT(HDRP(bp),PACK(size,0));
        PUT(FTRP(bp), PACK(size,0));
        addFreeBlock(bp);
    }
    else if(!prev_alloc && next_alloc){
        size+=GET_SIZE(HDRP(PREV_BLKP(bp)));
        PUT(FTRP(bp),PACK(size,0));
        PUT(HDRP(PREV_BLKP(bp)),PACK(size,0));
        deleteFreeBlock(PREV_BLKP(bp));
        addFreeBlock(PREV_BLKP(bp));
        bp=freeblkp;
    }
    else
    {
        size+=GET_SIZE(HDRP(PREV_BLKP(bp)))+GET_SIZE(FTRP(NEXT_BLKP(bp)));
        PUT(HDRP(PREV_BLKP(bp)),PACK(size,0));
        PUT(FTRP(NEXT_BLKP(bp)),PACK(size,0));
        deleteFreeBlock(NEXT_BLKP(bp));
        deleteFreeBlock(PREV_BLKP(bp));
        bp=PREV_BLKP(bp);
        addFreeBlock(bp);
        bp=freeblkp;
    }
    return bp;
}
```

coalesce 함수는 implicit 에서와 같이 메모리 할당 해제시 할당 해제된 블록들을 통합해 주는 작업을 한다. Implicit 에서와 동일한 기능을 한다. 하지만 통합된

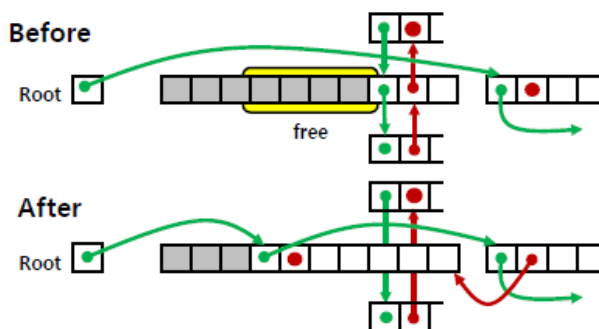
블록은 새로운 free 블록으로 간주하고 free 블록 리스트에 새로이 추가 하는 작업이 추가 되었다. 기본적인 원칙은 앞이나 뒤에 주어진 블록과 통합되 하나가 된다면 앞이나 뒤의 free 블록을 리스트에서 제거하고 통합된 블록을 리스트의 맨 앞에 추가 시키는 것이다. 이 작업에서 앞에서 정의한 deleteFreeBlock, addFreeBlock 함수를 사용한다. 단순히 링크드리스트 연산만 추가 되었을 뿐이다. 경우에 따른 처리는 다음과 같다.

첫번째, 주어진 할당 해제된 블록의 앞뒤 블록이 할당해제 상태가 아닌 경우이다.



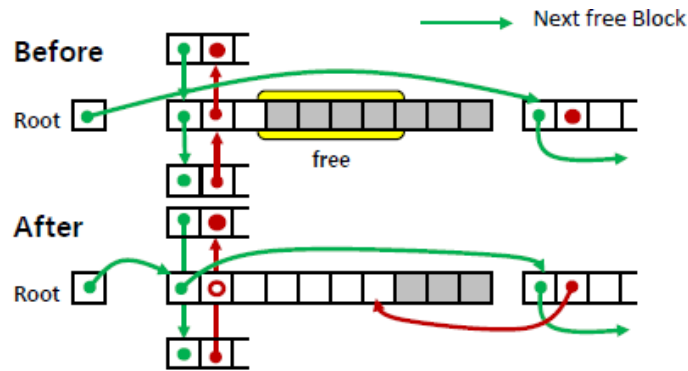
통합될 블록이 없는 상태이므로 온전한 free block 이다. LIFO 방식으로 리스트를 유지 하고 있으므로 이 블록을 addFreeBlock 함수를 이용하여 링크드리스트의 처음으로 추가시킨다.

두번째, 다음 블록이 할당 해제 상태, 이전 블록은 할당 상태인 경우이다.



이때 deleteFreeBlock(NEXT_BLK(bp)); 함수를 호출하여 free 블록 리스트에 포함되어 있는 다음 블록을 리스트에서 해제 시킨다. 다음 두 블록을 통합하여 addFreeBlock(bp);를 호출해 링크드리스트에 추가 시킨다.

세번째, 이전 블록이 할당상태, 다음 블록이 할당 해제 상태인 경우이다.



Free 상태인 이전 블록을 `deleteFreeBlock(PREV_BLK(b));`를 이용하여 freeblock 리스트에서 삭제한다. bp 블록과 이전 블록을 통합한 후 `addFreeBlock`을 이용하여 리스트의 앞에 추가 한다. 다음 bp를 freeblkp와 같게 한다.

마지막으로 이전과 다음 블록이 모두 free 상태인 경우, 이전블록, bp, 다음블록을 모두 통합하여야 한다. 이때 이전 블록과 다음블록을 freeblock 리스트에서 삭제한다. 그리고 세 블록을 통합하고 `addFreeBlock` 함수를 통하여 리스트에 추가 시킨다. 다음 bp를 freeblkp와 같게 한다.

이렇게 `deleteFreeBlock` 함수와 `addFreeBlock` 함수를 이용하여 링크드리스트 연산을 쉽게 할 수 있다.

F. void *malloc (size_t size)



```
void *malloc (size_t size)
{
    char* bp;
    unsigned asize;
    unsigned extendsize;

    if (size == 0)
        return NULL;

    if (size <= DSIZE)
        asize = 2 * DSIZE;
    else
        asize = DSIZE * ((size + (DSIZE) + (DSIZE - 1)) / DSIZE);

    if ((bp = find_fit (asize)) != NULL)
    {
        place (bp, asize);
        return bp;
    }

    extendsize = MAX (asize, 80);

    if ((bp = extend_heap (extendsize / WSIZE)) == NULL)
        return NULL;

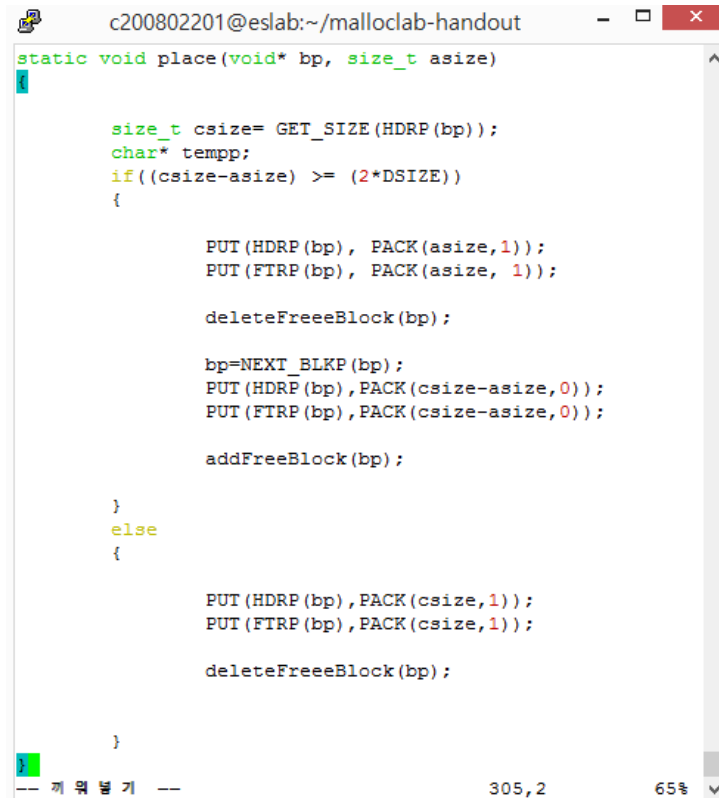
    place (bp, asize);

    return bp;
}
```

malloc 함수는 주어진 사이즈만큼 메모리 블록을 할당하는 함수이다. implicit 에서와 같다. 요청된 메모리는 header 와 footer 를 저장할 공간과 사용자가 요청한 사이즈의 공간을 포함하면서 Double Word 사이즈 정렬을 만족해야 한다. 즉 최소 블록사이즈는 16 바이트가 된다. malloc 함수에 요청된 사이즈가 DSIZE 보다 작다면 DSIZE 의 2 배하여 16 의 배수로 할당한다. 크다면 header 와 footer 를 합친 크기와 나머지를 8 의 배수로 할당하여 총 16 의 배수 사이즈만큼을 정한다. Find_fit 함수를 통해 정해진 사이즈에 알맞은 블록이 있는지 찾아 포인터를 리턴받는다. 이렇게 받은 블록은 원하는 사이즈보다 엄청 클 수도 있다. 때문에 place 함수로 원하는 사이즈 만큼 쪼개야 한다.

만약 알맞은 블록사이즈가 존재하지 않는다면 heap 영역을 확장해야 한다. 이때 얼마나 확장할 것 인지, asize,와 주어진 크기와 비교하여 큰 정수의 크기로 확장한다. 이때도 마찬가지로 무작정 큰 힙을 크게 확장하기보다 적당한 사이즈의 힙을 확장하는 것이 더 높은 throughput 나타낼 수 있다. 최소 블록사이즈 16 바이트의 배수를 대입하여 조사한 결과 80 을 넣었을 때부터 가장 좋은 결과가 나왔다.

G. void place(void* bp, size_t asize)



```
static void place(void* bp, size_t asize)
{
    size_t csize= GET_SIZE (HDRP (bp));
    char* temp;
    if ((csize-asize) >= (2*DSIZE))
    {
        PUT (HDRP (bp), PACK (asize,1));
        PUT (FTRP (bp), PACK (asize, 1));

        deleteFreeBlock (bp);

        bp=NEXT_BLK (bp);
        PUT (HDRP (bp), PACK (csize-asize,0));
        PUT (FTRP (bp), PACK (csize-asize,0));

        addFreeBlock (bp);
    }
    else
    {
        PUT (HDRP (bp), PACK (csize,1));
        PUT (FTRP (bp), PACK (csize,1));

        deleteFreeBlock (bp);
    }
}
```

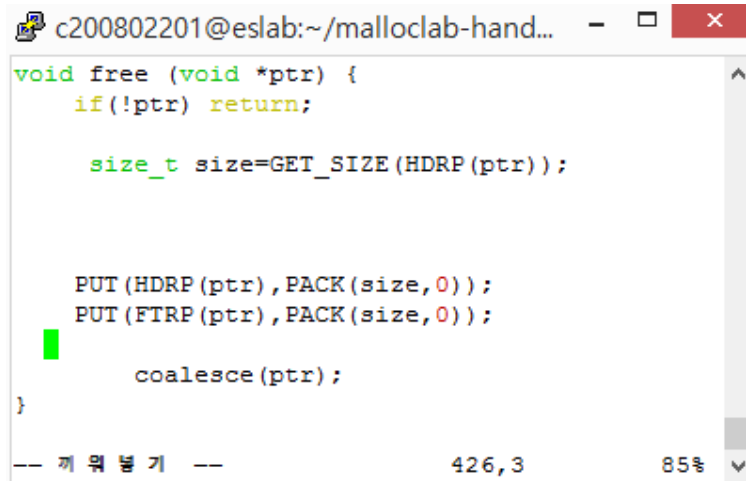
place 함수는 implicit 에서와 같은 기능을 하는 함수로 요청한 블록을 가용블록의 시작부분에 배치하여, 나머지 부분의 크기가 최소블록크기와 같거나 큰 경우 분할하는 기능을 수행한다. 기능에 대한 자세한 설명은 implicit 에서와 같음으로 생략한다.

implicit 에서와 다른 점은 링크드리스트 연산을 추가로 수행한다는 점이다. Bp 가 가리키고 있는 블록은 free 상태 블록이므로 이미 free 블록 링크드리스트에 추가가 되어있는 상태이다. 하지만 place 함수에서 알맞은 크기로 조정됨으로 free 블록의 일부가 잘려야 하는 상황이 된다. 현재 블록의 사이즈와 할당 받고자 하는 사이즈를 비교하여 이들의 차가 최소블록사이즈보다 같거나 크다면 할당된 부분을 제외한 free 블록만 다시 링크드리스트에 추가하는 작업을 해야한다. 일단 deleteFreeBlock 함수로 bp 가 가리키고 있는 블록을 링크드리스트에서 제거한다. 그리고 요청한 크기만큼 할당하고 남은 Free 블록을 addFreeBlock 함수를 통하여 리스트의 처음으로 추가한다.

만약 할당 현재 bp 가 가리키고 있는 블록의 사이즈와 할당을 요청한 블록사이즈의 차가 최소블록보다 작다면 bp 블록을 통째로 할당하고

deleteFreeBlock 함수를 이용하여 해당 블록을 리스트에서 삭제한다. 추가될 free 블록이 없으므로 addFreeBlock 은 호출하지 않는다.

H. void free (void *ptr)



```
void free (void *ptr) {
    if(!ptr) return;

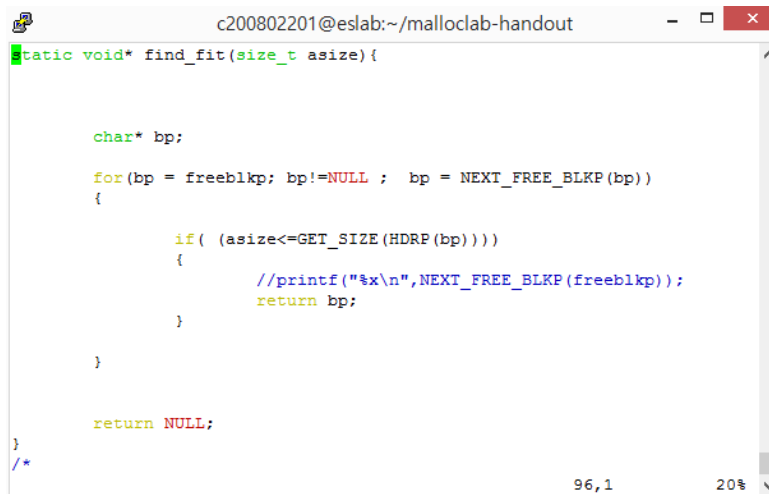
    size_t size=GET_SIZE (HDRP (ptr));

    PUT (HDRP (ptr), PACK (size, 0));
    PUT (FTRP (ptr), PACK (size, 0));

    coalesce (ptr);
}
```

free 함수는 메모리 블록을 할당 해제 상태로 만든다. 주어진 포인터가 가리키는 블록의 header 와 footer 의 할당 비트를 0 으로 돌려놓는다. 그리고 coalesce 함수를 호출하여 이전 블록과 다음블록의 상태에 따라 블록을 통합한다. implicit 에서 free 와 같다.

I. void* find_fit(size_t asize)



```
static void* find_fit(size_t asize){

    char* bp;

    for(bp = freeblkp; bp!=NULL ; bp = NEXT_FREE_BLKp(bp))
    {

        if( (asize<=GET_SIZE (HDRP (bp))) )
        {
            //printf("%x\n",NEXT_FREE_BLKp(freeblkp));
            return bp;
        }

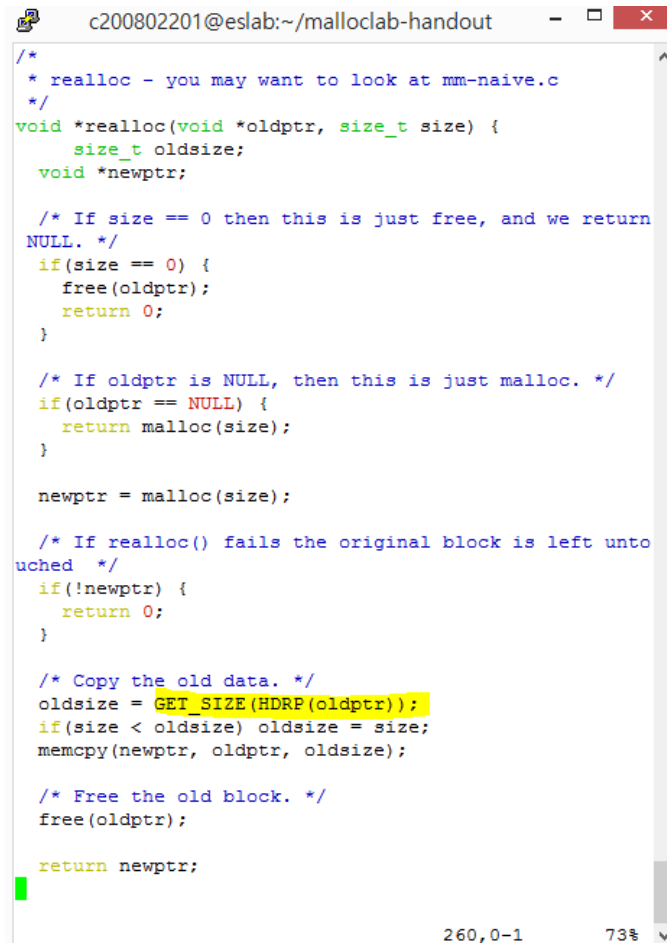
    }

    return NULL;
}
/*
```

Find_fit 은 freeblock 리스트를 순회하여 가장 알맞은 블록을 찾는 함수이다. Implicit 과 다르게 할당이 해제된 블록만 순회함으로 더 빠르게 순회가 가능하다. 전체적인 알고리즘은 firstfit 과 같은 알고리즘이다. 다른점은 freeblock 만 순회

한다는 것이다. 최근에 할당이 해제된 블록이 freeblock 리스트의 처음에 존재하여 먼저 검색하게 된다.

J. `void *realloc(void *oldptr, size_t size)`



```
/*
 * realloc - you may want to look at mm-naive.c
 */
void *realloc(void *oldptr, size_t size) {
    size_t oldsize;
    void *newptr;

    /* If size == 0 then this is just free, and we return
    NULL. */
    if (size == 0) {
        free(oldptr);
        return 0;
    }

    /* If oldptr is NULL, then this is just malloc. */
    if (oldptr == NULL) {
        return malloc(size);
    }

    newptr = malloc(size);

    /* If realloc() fails the original block is left untouched */
    if (!newptr) {
        return 0;
    }

    /* Copy the old data. */
    oldsize = GET_SIZE(HDRP(oldptr));
    if (size < oldsize) oldsize = size;
    memcpy(newptr, oldptr, oldsize);

    /* Free the old block. */
    free(oldptr);

    return newptr;
}
```

Implicit 과 같은 함수이므로 자세한 설명은 생략한다.

라. 실행결과

```
c200802201@eslab:~/malloclab-handout
[c200802201@eslab malloclab-handout]$ ./mdriver
Using default tracefiles in ./traces/
Measuring performance with a cycle counter.
Processor clock rate ~= 3392.3 MHz

Results for mm malloc:
  valid  util   ops    secs    Kops  trace
  yes    88%    10    0.000000  27921 ./traces/malloc.rep
  yes    95%    17    0.000000  45231 ./traces/malloc-free.rep
  yes    100%   15    0.000001  29028 ./traces/corners.rep
* yes    86%   1494    0.000035  42489 ./traces/perl.rep
* yes    86%    118    0.000002  66761 ./traces/hostname.rep
* yes    91%   11913   0.000184  64768 ./traces/xterm.rep
* yes    90%    5694   0.000195  29219 ./traces/amptjp-bal.rep
* yes    92%    5848   0.000183  31965 ./traces/cccp-bal.rep
* yes    95%    6648   0.000268  24800 ./traces/cp-decl-bal.rep
* yes    96%    5380   0.000210  25567 ./traces/expr-bal.rep
* yes    99%   14400   0.000265  54370 ./traces/coalescing-bal.rep
* yes    87%    4800   0.000598   8029 ./traces/random-bal.rep
* yes    55%    6000   0.004701   1276 ./traces/binary-bal.rep
10      88%   62295   0.006641   9381

Perf index = 57 (util) + 40 (thru) = 97/100
[c200802201@eslab malloclab-handout]$
```

실행결과 utilization 57 , throughput 40 점의 점수가 나온다.