Multicore Programming Project 3

담당 교수: 박성용

이름: 한석기

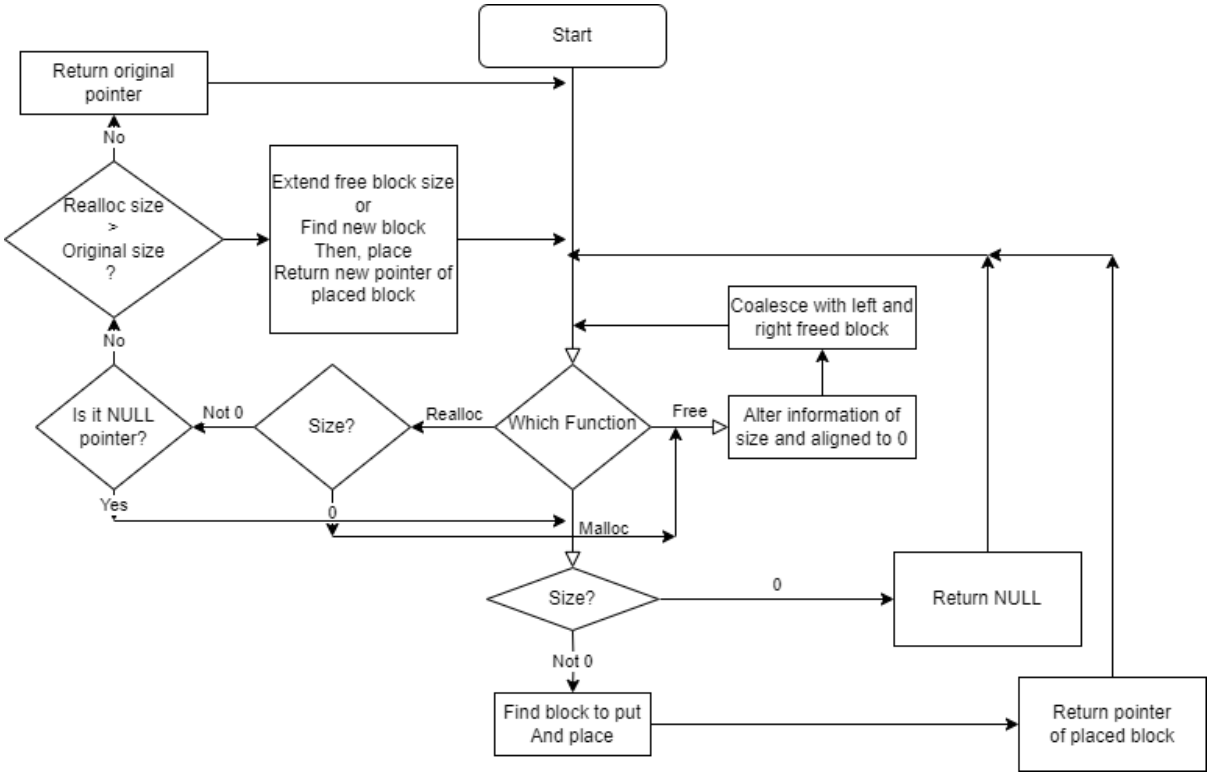
학번: 20211606

1. **개발 목표**

C언어에서 동적 메모리를 할당하는 과정을 구현한다. 사용자 정의 malloc, free, realloc 함수를 구현해보며 정확하고 효율적이고 빠른 실행을 창의적으로 설계하며 동적할당에 대한 이해를 높인다.

2. **개발 범위 및 내용**

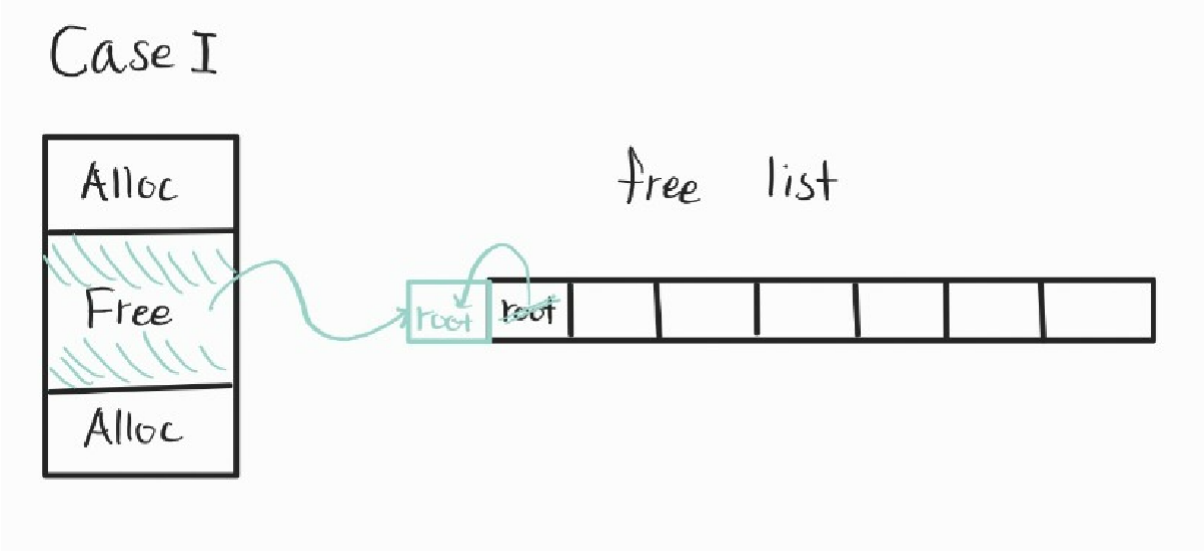
(1) Flow chart of mm.c program



Free, Realloc, Malloc의 총 세 가지 기능을 구현할 것이다. 해당 기능들에 대한 플로우 차트는 위 그림과 같다. Free, Malloc 등의 기능을 Realloc에서 재사용하는 흐름을 만들었고 Size, NULL Pointer 등, 여러가지 Exception Control 하는 부분도 추가했다. 위 플로우 차트처럼 구현하려면 기능을 함수로 만들어 여러 번 사용 가능하게 해야 한다.

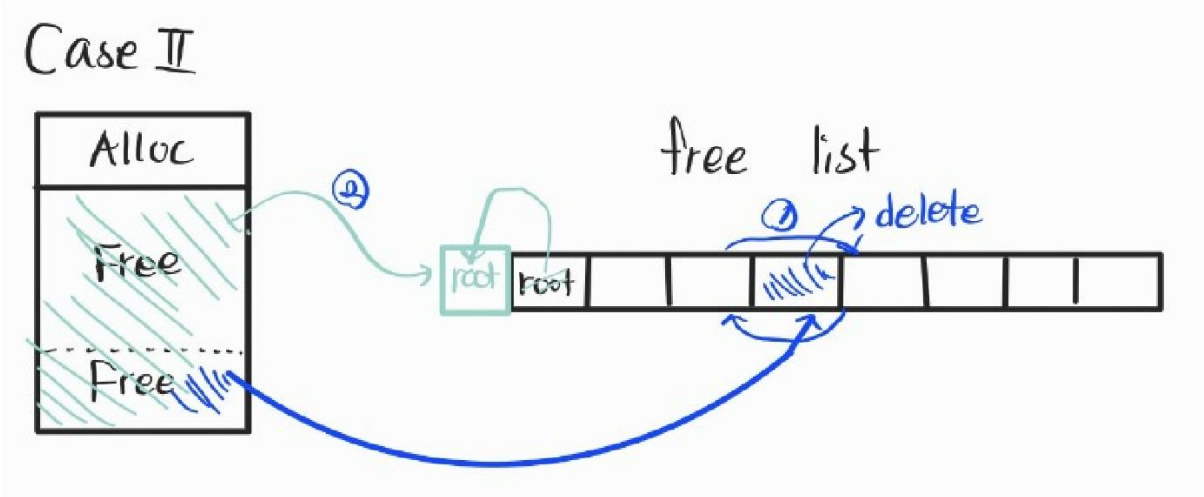
(2) Coalesce하는 부분 그림 구조

- Case 1



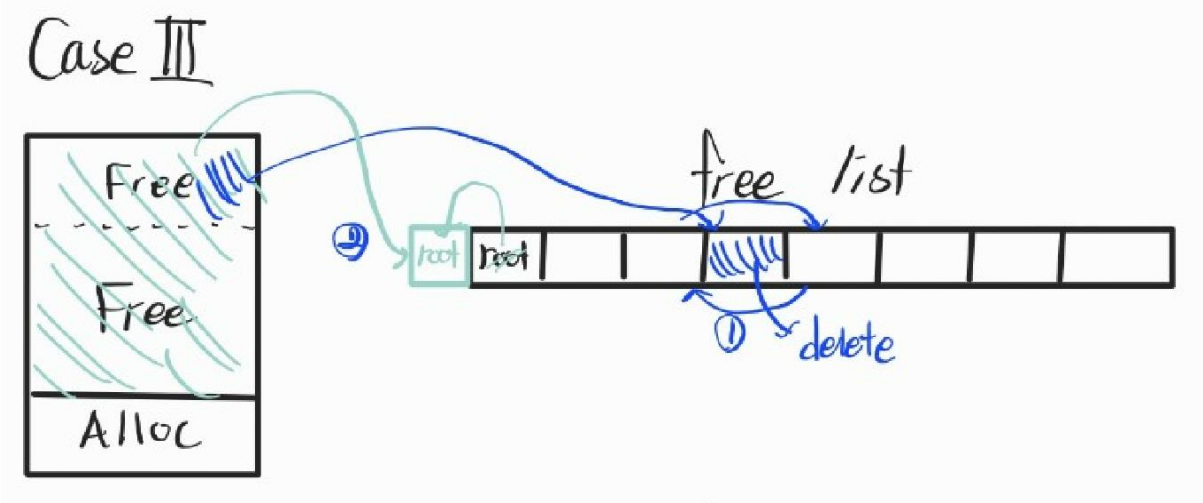
현재 Free가 된 곳의 양 옆 Block이 모두 Allocated일 경우다. 이럴 경우 그저 free list 앞에 추가하고 root를 바꿔주면 된다.

- Case 2



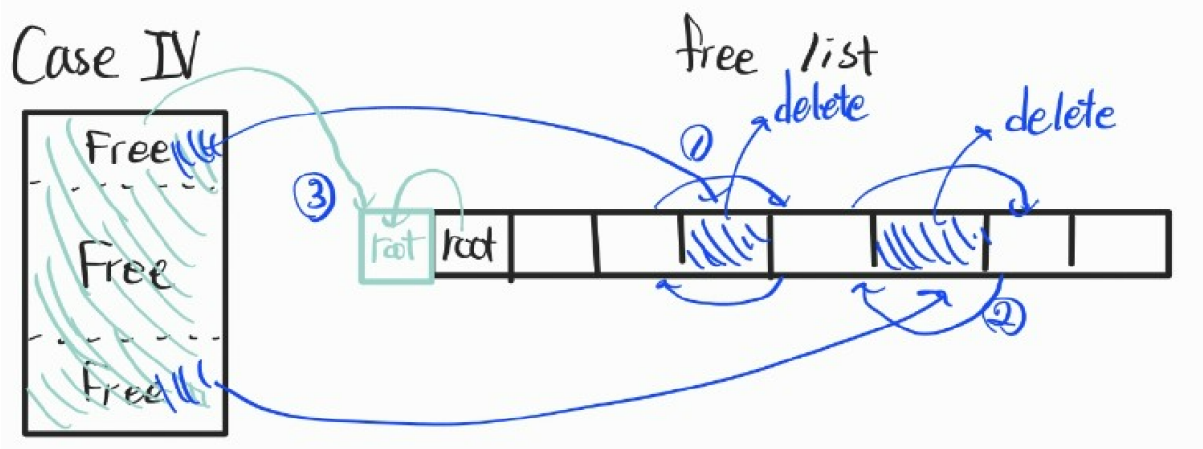
현재 Free가 된 곳의 뒤 Block이 Free된 블록이고 앞에 블록이 Allocated인 경우다. 이 경우, 뒤 블록에 해당하는 free list 요소를 제거하고 현재 Free된 곳과 합쳐 free list 앞에 추가하고 root를 바꿔준다.

- Case 3



현재 Free가 된 곳의 앞 Block이 Free된 블록이고 뒤 Block은 Allocated인 경우다. 이 경우, 앞 블록에 해당하는 free list 요소를 제거하고 현재 Free된 곳과 합쳐 free list 앞에 추가하고 root를 바꿔준다.

- Case 4



현재 Free가 된 곳의 앞, 뒤 Block이 모두 Free된 블록인 경우다. 이 경우, 앞, 뒤 블록에 해당하는 free list 요소들을 제거하고 현재 Free된 곳과 합쳐 free list 앞에 추가하고 root를 바꿔준다.

- Coalesce시 유의사항

현재 Free가 된 곳이 아닌, 이전에 이미 Free가 된 곳에 해당하는 free list를 관리 시, delete를 하면 이전 것과 다음 것을 연결하는 과정이 필요하다. 이 때, 포인터를 적절히 활용해 진행하면 된다. 그리고 현재 Free가 된 곳과 합치는 과정에서 Size 정보를 업데이트해야 한다.

(3) 전역 변수

- static char *free_listp: free list를 관리하는 root pointer이다. freed block만 들어간다.

- static char *heap_listp: heap list를 관리하는 root pointer이다. freed block, allocated block 상관없이 모두 들어간다.

3. 구현 결과

(1) Macros

```
//Word and header/footer size (bytes)
#define WSIZE 4
//Double word size (bytes)
#define DSIZE 8
//Extend heap by this amount (bytes)
#define CHUNKSIZE (1 << 12)

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

//Pack a size and allocated bit into a word
#define PACK(size, alloc) ((size) | (alloc))

//Read and write a word at address p
#define GET(p) (*(unsigned int *)(p))
#define PUT(p, val) (*(unsigned int *)(p) = (val))

// Read the size and allocated fields from address p
// 8-byte alignment
#define GET_SIZE(p) (GET(p) & ~0x7)
// LSB is alloc/free bit
#define GET_ALLOC(p) (GET(p) & 0x1)

//Given block ptr bp, compute address of its header and footer
#define HDRP(bp) ((char *)(bp) - WSIZE)
#define FTRP(bp) ((char *)(bp) + GET_SIZE(HDRP(bp)) - DSIZE)

//Given block ptr bp, compute address of next and previous blocks
#define NEXT_BLKP(bp) ((char *)(bp) + GET_SIZE(((char *)(bp) - WSIZE)))
#define PREV_BLKP(bp) ((char *)(bp) - GET_SIZE(((char *)(bp) - DSIZE)))

//Given free list ptr p, compute address of next and previous freed block
#define NEXT_FP(p) (*(char **)(p + WSIZE))
#define PREV_FP(p) (*(char **)(p))
```

미리 정의해 놓은 매크로들은 위 그림과 같다. 해당하는 기능은 각 매크로 위에 주석으로 적었으며 적혀 있지 않은 MAX(x, y)는 의미 그대로 x, y 중 큰 값을 반환하는 매크로다.

(2) 강의자료 외 추가한 함수들

-delete_from_flist and add_to_flist

```
static void delete_from_flist(void *bp)
{
    // if bp is free_listp
    if(PREV_FP(bp) == NULL) {
        free_listp=NEXT_FP(bp);
        PREV_FP(free_listp) = NULL;
    }
    else {
        NEXT_FP(PREV_FP(bp)) = NEXT_FP(bp);
        PREV_FP(NEXT_FP(bp)) = PREV_FP(bp);
    }
}
static void add_to_flist(void *bp)
{
    // add to prev of free list head pointer
    NEXT_FP(bp) = free_listp;
    PREV_FP(free_listp) = bp;
    PREV_FP(bp)=NULL;
    // then, added free item is header
    free_listp = bp;
}
```

강의자료에는 implicit list로만 관리를 했기 때문에 explicit list로 관리하는 방법으로 만든 코드에 위와 같은 함수를 추가했다. 강의자료에 주어진 implicit list로만 관리하는 코드는 freed block과 allocated block을 같이 넣어준 heap_listp로 free된 곳을 탐색해 malloc과 realloc, free를 수행했는데 이것은 allocated된 것도 같이 탐색해야 하는 불필요한 시간이 소요된다. 하지만 explicit list로 관리하면 free list로 free된 곳만 탐색하면 된다.

위 코드는 그 과정에서 free list를 관리하는 코드다. 각각 free list에서 삭제하고 추가하는 함수이며 root인지 확인하는 과정, 각각의 previous freed block과 next freed block을 연결하는 과정이 들어있다. 그리고 add는 무조건 root의 앞에 추가하고 그것을 root로 지정한다.

-mm_realloc

```
void *mm_realloc(void *oldptr, size_t size)
{
    size_t oldsize, newsize;
    void *newptr;

    // Because size_t is unsigned integer type..
    // Type casting
    if((int)size < 0) return NULL;
    if(size == 0){
        mm_free(oldptr);
        return NULL;
    }

    // If current block is NULL, malloc
    if(oldptr == NULL) return mm_malloc(size);
    else{
        oldsize = GET_SIZE(HDRP(oldptr));
        newsize = size + (2 * WSIZE);
    }

    if(newsize > oldsize){
        size_t n_is_alloc = GET_ALLOC(HDRP(NEXT_BLKP(oldptr)));
        size_t n_blk_size = GET_SIZE(HDRP(NEXT_BLKP(oldptr)));
        size_t co_fsize = oldsize + n_blk_size;

        if(n_is_alloc == 0 && co_fsize >= newsize){
            delete_from_flist(NEXT_BLKP(oldptr));
            PUT(HDRP(oldptr), PACK(co_fsize, 1));
            PUT(FTRP(oldptr), PACK(co_fsize, 1));
            return oldptr;
        }
        else{
            newptr=mm_malloc(newsize);
            if(newptr == NULL) return NULL;

            place(newptr, newsize);
            memcpy(newptr, oldptr, oldsize);
            mm_free(oldptr);
            return newptr;
        }
    }
    else return oldptr;
}
```

강의자료에는 mm_realloc의 소스 코드가 나와있지 않았다. 따라서 oldptr, newptr을 통해 나눠서 구현했다. size는 size_t 자료형이며 이 자료형은 음수는 다루지 않기에 (int)로 type casting을 진행해, 음수면 NULL을 반환하는 조건을 넣었다. 그리고 size가 0이면 free하는 것과 같기에 구현해 놓은 free를 호출하고 NULL을 반환했다.

이전 allocated pointer가 NULL이면 처음 allocate를 realloc으로 진행하는 것이기에 mm_malloc을 호출하는 것과 같은 수행과정이라 구현해 놓은 mm_malloc을 호출한 것을 반환했다.

realloc으로 size는 줄일 수 없기에 새로운 size가 기존 size보다 작을 시, 그대로 기존 pointer를 반환했고 클 시, 다음 블록이 free된 것이고 합쳤을 때, size가 충분하면 header, footer 정보를 업데이트하고 해당 Pointer를 반환한다.

하지만, 다음 블록이 free된 것이지만 합쳤을 때, size가 그래도 부족하거나, 다음 블록이 free된 블록이 아니라면, free list에서 새로 찾아 할당하고 기존 것은 free해버린다. 그리고 새로 할당된 pointer를 return한다.

4. 성능 평가 결과

- Implicit list

```
[20211606]::NAME: Seokgi Han, Email Address: hski0930@sogang.ac.kr
Using default tracefiles in ./tracefiles/
Measuring performance with gettimeofday().

Testing mm malloc
Reading tracefile: amptjp-bal.rep
Checking mm_malloc for correctness, efficiency, and performance.
Reading tracefile: cccp-bal.rep
Checking mm_malloc for correctness, efficiency, and performance.
Reading tracefile: cp-decl-bal.rep
Checking mm_malloc for correctness, efficiency, and performance.
Reading tracefile: expr-bal.rep
Checking mm_malloc for correctness, efficiency, and performance.
Reading tracefile: coalescing-bal.rep
Checking mm_malloc for correctness, efficiency, and performance.
Reading tracefile: random-bal.rep
Checking mm_malloc for correctness, efficiency, and performance.
Reading tracefile: random2-bal.rep
Checking mm_malloc for correctness, efficiency, and performance.
Reading tracefile: binary-bal.rep
Checking mm_malloc for correctness, efficiency, and performance.
Reading tracefile: binary2-bal.rep
Checking mm_malloc for correctness, efficiency, and performance.
Reading tracefile: realloc-bal.rep
Checking mm_malloc for correctness, efficiency, and performance.
Reading tracefile: realloc2-bal.rep
Checking mm_malloc for correctness, efficiency, and performance.

Results for mm malloc:
trace  valid  util     ops      secs  Kops
 0       yes   99%    5694  0.020734   275
 1       yes   99%    5848  0.017251   339
 2       yes   99%    6648  0.030733   216
 3       yes  100%    5380  0.023132   233
 4       yes   66%   14400  0.000546 26383
 5       yes   92%    4800  0.014261   337
 6       yes   92%    4800  0.018631   258
 7       yes   55%   12000  0.350235    34
 8       yes   51%   24000  0.606616    40
 9       yes   42%   14401  0.000428 33663
10       yes   86%   14401  0.000195 74041
Total          80%  112372  1.082761   104

Perf index = 48 (util) + 7 (thru) = 55/100
```

- Explicit list

```
[20211606]::NAME: Seokgi Han, Email Address: hski0930@sogang.ac.kr
Using default tracefiles in ./tracefiles/
Measuring performance with gettimeofday().

Testing mm malloc
Reading tracefile: amptjp-bal.rep
Checking mm_malloc for correctness, efficiency, and performance.
Reading tracefile: cccp-bal.rep
Checking mm_malloc for correctness, efficiency, and performance.
Reading tracefile: cp-decl-bal.rep
Checking mm_malloc for correctness, efficiency, and performance.
Reading tracefile: expr-bal.rep
Checking mm_malloc for correctness, efficiency, and performance.
Reading tracefile: coalescing-bal.rep
Checking mm_malloc for correctness, efficiency, and performance.
Reading tracefile: random-bal.rep
Checking mm_malloc for correctness, efficiency, and performance.
Reading tracefile: random2-bal.rep
Checking mm_malloc for correctness, efficiency, and performance.
Reading tracefile: binary-bal.rep
Checking mm_malloc for correctness, efficiency, and performance.
Reading tracefile: binary2-bal.rep
Checking mm_malloc for correctness, efficiency, and performance.
Reading tracefile: realloc-bal.rep
Checking mm_malloc for correctness, efficiency, and performance.
Reading tracefile: realloc2-bal.rep
Checking mm_malloc for correctness, efficiency, and performance.

Results for mm malloc:
trace  valid  util     ops      secs  Kops
 0       yes   89%    5694  0.000337 16896
 1       yes   92%    5848  0.000251 23280
 2       yes   94%    6648  0.000456 14573
 3       yes   96%    5380  0.000354 15215
 4       yes   66%   14400  0.001598  9012
 5       yes   88%    4800  0.000630  7618
 6       yes   85%    4800  0.002281  2105
 7       yes   55%   12000  0.008077  1486
 8       yes   51%   24000  0.008617  2785
 9       yes   97%   14401  0.000305 47201
10       yes   38%   14401  0.000334 43091
Total          77%  112372  0.023239  4836

Perf index = 46 (util) + 40 (thru) = 86/100
```

- Explanation

강의자료에 주어진 implicit list를 이용한 방법에 realloc 코드를 추가해, 성능평가를 진행하고 explicit list를 이용한 방법으로 성능평가를 진행했다.

utilization 부분에서는 implicit list가 더 높았고 그 비율은 2/100에 달한다. 하지만 throughput 부분에서는 explicit list가 압도적으로 높았다. 무려 그 비율이 33/100이다. 따라서 전체적인 Performance 측면에서 implicit list는 55/100의 Performance를 보여줬고 explicit list는 86/100의 performance를 보여줬다.

두 코드 모두 first fit 방법에 해당하는 탐색으로 진행한 동일한 환경에서의 결과다. 그랬는데도 차이가 이렇게 벌어졌고 free list를 생성 및 관리하는 것을 추가하는 것만으로 throughput을 높일 수 있다는 점을 확인할 수 있었다.