Multicore Programming Project 3

담당 교수 : 박성용

이름 : 김도현

학번 : 20181256

1. 개발 목표

해당 프로젝트는 수업 시간에 배운 1 word가 4 byte, 8 byte alignment 가정 하에 heap 구조를 토대로 c언어의 dynamic memory allocator인 malloc 함수, free 함수, realloc 함수를 직접 구현해보는 것이다. 이때 무작정 짜는 것이 아니라 segment memory utilization, throughput 을 모두 고려하여 구현을 해야 높은 점수를 받을 수 있다. 방법은 implicit list using length, explicit list, segregated free list, blocks sorted by size 등이 있는데 필자는 segregated free list 로 구현해보고자 한다. 그리고 free block search 에서도 best fit, first fit, next fit 등이 있는데 best fit을 통해 memory utilization 을 조금 더 높이고자 하였다.

2. 개발 내용

A. 개발 세팅

여기서는 기존의 skeleton 코드에서 필자가 segregated free list 를 구현하기 위해 추가한 코드들을 정리 할 것이다.

```
/*my macro*/
#define PREV_BLK_SP_R(bp) (GET((char*)bp))
#define NEXT_BLK_SP_R(bp) (GET((char*)(bp)+WSIZE))
#define PREV_BLK_SP_W(bp, ptr) (PUT(bp, ptr))
#define NEXT_BLK_SP_W(bp, ptr) (PUT((char*)(bp)+WSIZE, ptr))
```

우선 매크로에 대한 내용이다. PREV_BLK_SP_R(Previous Block Segregated Pointer Read) 은 현재 free 블록을 기준으로 그 이전에 있는 free 블록을 읽어들이는 매크로이다. 자세한 구현 내용은 개발 내용에서 설명하도록 하겠다. NEXT_BLK_SP_R 은 현재 free 블록을 기준으로 그 다음에 있는 free 블록을 읽어들이는 매크로이다. PREV_BLK_SP_W 은 현재 free block에 있는 이전 free 블록을 가리키는 부분에 대해 새로운 이전 free block인 ptr을 적어주는 것이다. NEXT_BLK_SP_W 은 현재 free block에 있는 다음 free 블록을 가리키는 부분에 대해 새로운 다음 free block인 ptr을 적어주는 것이다.

```
static char* heap_listp;
size_t segregate_num=8;
static char** segregate_list_pointer;
```

다음은 block들을 연결하는 heap영역의 linked list의 첫번째 포인터라고 할 수 있는 heap_listp이다. Segregated_num 은 segregated free list 구현 시 꼭 필요한 크기 별로 free list를 따로 가지고 있어야 하기 때문에 필요한 전역 변수 이다. 여기서 필자는 8개로 잡아 각 block의 size에 따라 8개로 분류하여 관리하고자 하였다. segregated free list를 구현하기 위해 2중 포인터를 이용하였다.

B. 개발 내용

구현한 함수들을 하나 씩 설명하면서 개발 내용을 살펴보도록 하겠다.

- int mm_init(void)

  여기서는 가장 처음 heap 메모리를 우리의 m_heap으로 만들어 주는 과정을 하는 것이다. 초기화 함수라고 볼 수 있다. 우선 free block 들을 관리해주기 위해 필요한 segregate_list 공간을 mem_sbrk(segregate_num*WSIZE) 를 통해 할당 해준다. 여기에서 각 index 들을 통해 free block 관리해주는 공간이 생긴다. 그리고 초기에는 각 index 별 pointer들을 모두 NULL로 초기화 한다. 이후 기존의 코드와 똑같이 heap_listp 에 mem_sbrk(4*WSIZE)를 할당하여 수업 시간에 배운 block의 구조인 header, footer 가 있는 모양으로 만들면 된다. 여기서는 PUT 매크로를 통해 처음엔 alignment padding, 다음엔 Prologue header, prologue footer, Epilogue header에 PACK(0,1)를 size가 0인 넣어 m_heap의 마지막임을 알 수 있도록 해준다. 마지막으로 heap_listp에 word size*2, 즉 DSIZE 를 더해줘서 메모리 할당을 받을 준비를 끝낸다.

```
int mm_init(void)
{
    /* Create the initial empty heap */
    if ((segregate_list_pointer = mem_sbrk(segregate_num*WSIZE)) == (void *)-1)
    return -1;

    for(int i=0;i<segregate_num;i++){
    segregate_list_pointer[i]=NULL; //seglist 초기화
}
if ((heap_listp = mem_sbrk(4*WSIZE)) == (void *)-1)
    return -1;
    PUT(heap_listp, 0);                          /* Alignment padding */
    PUT(heap_listp + (1 * WSIZE), PACK(DSIZE, 1)); /* Prologue header */
    PUT(heap_listp + (2 * WSIZE), PACK(DSIZE, 1)); /* Prologue footer */
    PUT(heap_listp + (3 * WSIZE), PACK(0, 1));     /* Epilogue header */
    heap_listp += (2 * WSIZE);

    return 0;
}
```

- Void *mm_malloc(size_t size)

  이 함수는 malloc 함수와 같이 새로 할당 받을 메모리 사이즈를 인자로 받아 해당 block의 실제 값이 들어 있는 첫번째 주소를 포인터 인자로 반환 한다. Size가 0일때는 아무것도 할 게 없으므로 NULL 을 return 해준다. Size가 8 byte보다 작을 경우에는 최소 header, footer가 들어간 Block을 구현하기 위해서 필요한 사이즈인 2*DSIZE를 align size에 할당 한다. 아니라면 DSIZE*((size+2*DSIZE-1)/DSIZE) 공식을 통해 align size를 할당하면 된다. 사이즈를 알게 되었으면 해당 값을 heap에 넣어야 하는데 find_fit 함수를 통해서 segregated free list 들을 찾아보고 원하는 공간이 남아 있다면 place_seg 함수를 통해 해당 free block 을 free list에서 제거하고 block에 저장한다. 만약 heap에 충분한 공간이 없다면 extend_heap 함수를 통해서 heap 사이즈를 확장하고 이후 place_ext 함수에서 extend 된 공간에 mm_malloc 을 통해 할당 받고자 한 assize 를 넣어 주면 된다.

```
void *mm_malloc(size_t size)
{
    size_t asize;      /* Adjusted block size */
    size_t extendsize; /* Amount to extend heap if no fit */
    char* bp;

    /* Ignore spurious requests */
    if (size == 0)
    return NULL;

    /* Adjust block size to include overhead and alignment reqs. */
    if (size <= DSIZE)
    asize = 2 * DSIZE;
    else asize = DSIZE * ((size + (DSIZE)+(DSIZE - 1)) / DSIZE);

    /* Search the free list for a fit */
    if ((bp = find_fit(asize)) != NULL) {  //free list 찾고
        place_seg(bp, asize); //free list 제거 및 해당 block에 저장
        return bp;
    }

    /* No fit found. Get more memory and place the block */
    extendsize = MAX(asize, CHUNKSIZE);
    if ((bp = extend_heap(extendsize / WSIZE)) == NULL) //확장
        return NULL;
        place_ext(bp,asize); //extend해서 넣기
    return bp;
}
```

- Void* mm_realloc(void *ptr, size_t size)

여기서는 기존의 공간에서 추가적인 공간을 원할 때 새롭게 재할당 해주는 함수이다. Size가 0이라면 해당 ptr 을 더이상 사용하지 않는 다는 말이기 때문에 mm_free 시켜주고 NULL을 return 한다. 또한 ptr 자체가 NULL일 경우에는 mm_malloc 함수로 그냥 새롭게 공간을 할당해주면 된다.

위의 경우들이 아니라면 이제 현재의 공간, 추가적으로 필요한 공간에 대한 계산이 필요하다. 만약 newSize가 기존의 copySize보다 크다면 현재 block의 next block역시 Free block인지 check 해본다.

```
(!GET_ALLOC(HDRP(NEXT_BLKP(ptr)))&&copySize +  GET_SIZE(HDRP(NEXT_BLKP(ptr))) >= newSize)
```

그리고 copySize 와 next block Size를 더해서 newSize보다 크거나 같은지 확인해서 이에 맞다면 next block 을 segregated free list에서 seg_delete 함수를 통해 지운 다음 PUT 매크로를 통해 새롭게 기입하면 된다.

newSize가 기존의 copySize보다 작다면 그냥 기존의 ptr을 return 하면 된다.

위의 두 경우 둘 다 아닌 경우, 즉 newSize가 copySize보다는 크지만 NextBlock의 Size를 더해도 더 작은 경우, 혹은 nextBlock이 free Block 이 아닌 경우, 새롭게 newptr에 mm_malloc(size)를 할당 한다. 이후 memcpy를 통해 해당 정보들을 넣어주고 mm_free(ptr)로 기존의 block을 free 시키면 된다.

```
void *mm_realloc(void *ptr, size_t size)
{
    if(size==0){
        mm_free(ptr);
        return NULL;
    }
    if(ptr==NULL)
    return mm_malloc(size);
    size_t copySize=GET_SIZE(HDRP(ptr));
    size_t newSize= DSIZE * ((size + (DSIZE)+(DSIZE - 1)) / DSIZE);

    if(newSize>copySize){ //다음 free block 공간에 넣을 수 있다


        if (!GET_ALLOC(HDRP(NEXT_BLKP(ptr)))&&copySize +  GET_SIZE(HDRP(NEXT_BLKP(ptr))) >= newSize)
        {
            void*  nextptr = NEXT_BLKP(ptr);
            seg_delete(nextptr);

             PUT(HDRP(ptr), PACK(copySize +  GET_SIZE(HDRP(nextptr)), 1));
             PUT(FTRP(ptr), PACK(copySize +  GET_SIZE(HDRP(nextptr)), 1));
              return ptr;
        }
    }
    else{
        return ptr;
    }
    void*  newptr = mm_malloc(size);
    if (newptr == NULL) //새로운 malloc 실패시 null return
      return NULL;
    if (size < copySize)
      copySize = size;
    memcpy(newptr, ptr, copySize);
    mm_free(ptr);
    return newptr;
}
```

- Void mm_free(void *bp)

해당 함수는 block을 free 시키는 함수이다. Bp 기준으로 header, footer에 alloc을 0으로 할당하여 free block을 바꾸고 coalesce 함수를 통해 segregate

free list 에서 병합되도록 해준다.

- Static Void* coalesce(void *bp)

  Free block 앞 뒤를 모두 체크해서 병합을 해주는 함수이다. 우선 case가 4가지가 있다.

  1번 케이스는 앞 뒤 모두 free block이 아닐 경우이다. 이때는 아무것도 하지 않는다.

  2번 케이스는 뒤 즉 next block이 free block일 경우이다. 이때 size를 next block의 크기까지 더해준 다음 seg_delete 함수를 통해 next block을 segregate free list에서 지워준다. 이후 block을 PUT 매크로를 통해 병합한다.

  3번 케이스는 앞 즉 prev block이 free block일 경우이다. 이때 size를 prev block의 크기까지 더해준 다음 seg_delete 함수를 통해 prev block을 segregate free list에서 지워준다. 이후 block을 PUT 매크로를 통해 병합한다.

  4번 케이스는 앞, 뒤 즉 prev block, next block 이 free block일 경우이다. 이때 size를 prev block의 크기, next block의 크기까지 더해준 다음 seg_delete 함수를 통해 prev block, next block 을 segregate free list에서 지워준다. 이후 block을 PUT 매크로를 통해 병합한다.

  최종적으로 수정된 bp를 seg_insert 함수를 통해 segregate free list에 해당 block만 넣어주면 된다.

```
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));

    if (prev_alloc && next_alloc) {        //앞 뒤로 있을때     /* Case 1 */
    }

    else if (prev_alloc && !next_alloc) {   //뒤에가 free   /* Case 2 */
        size += GET_SIZE(HDRP(NEXT_BLKP(bp)));
        seg_delete(NEXT_BLKP(bp));
        PUT(HDRP(bp), PACK(size, 0));
        PUT(FTRP(bp), PACK(size, 0));

    }
    else if (!prev_alloc && next_alloc) {    //앞에가 free   /* Case 3 */
        size += GET_SIZE(HDRP(PREV_BLKP(bp)));
       seg_delete(PREV_BLKP(bp));
        PUT(FTRP(bp), PACK(size, 0));
        PUT(HDRP(PREV_BLKP(bp)), PACK(size, 0));
        bp = PREV_BLKP(bp);
    }
    else {                                  /* Case 4 */
        size += GET_SIZE(HDRP(PREV_BLKP(bp))) + GET_SIZE(FTRP(NEXT_BLKP(bp)));
       seg_delete(PREV_BLKP(bp));
       seg_delete(NEXT_BLKP(bp));
        PUT(HDRP(PREV_BLKP(bp)), PACK(size, 0));
        PUT(FTRP(NEXT_BLKP(bp)), PACK(size, 0));
        bp = PREV_BLKP(bp);
    }

    seg_insert(bp);
    return bp;
}
```

- Void seg_delete(void* bp)

  더이상 해당 block은 free block이 아니기 때문에 segregate free list에서 지우는 함수이다. 해당 free block의 next free block link를 확인해서 만약 있다면 해당 free block의 prev free block link 역시 확인해 본다. 있으면 next free block에 이제 prev block은 현재 free block이 아닌 현재 Block 기준 이전 block이라고 PREV_BLK_SP_W로 기입한다. Prev free block 이 없으면 그냥 NULL 을 넣어준다.

  해당 free block의 prev free block link를 확인해서 만약 있다면 해당 free block의 next free block link 역시 확인해 본다. 있으면 prev free block에 이제 next block은 현재 free block이 아닌 현재 Block 기준 next block이라고 NEXT_BLK_SP_W로 기입한다. next free block 이 없으면 그냥 NULL 을 넣어준다.

  이때 prev free block이 그냥 없다면 첫번째 segregate free list 라는 것이기 때문에 segregate list 도 수정을 해야 한다. 첫번째 가리키는 pointer를 segregate_list_pointer[seg_index(GET_SIZE(HDRP(bp)))]=NEXT_BLK_SP_R(bp) 이렇게 현재 block 기준 next block으로 바꿔준다.

```
void seg_delete(void* bp){

    if(NEXT_BLK_SP_R(bp)){
        if(!PREV_BLK_SP_R(bp)){
            PREV_BLK_SP_W(NEXT_BLK_SP_R(bp),NULL); //이전 포인터는 없다(첫번째)
        }
        else{
            PREV_BLK_SP_W(NEXT_BLK_SP_R(bp),PREV_BLK_SP_R(bp));
        }
    }

    if(PREV_BLK_SP_R(bp)){
        if(!NEXT_BLK_SP_R(bp)){
            NEXT_BLK_SP_W(PREV_BLK_SP_R(bp),NULL); //이후 포인터는 없다(마지막)
        }
        else{
            NEXT_BLK_SP_W(PREV_BLK_SP_R(bp),NEXT_BLK_SP_R(bp));
        }
    }
    else{
        segregate_list_pointer[seg_index(GET_SIZE(HDRP(bp)))]=NEXT_BLK_SP_R(bp); //다음 free b
    }


}
```

- Void seg_insert(void*bp)

  여기는 free block을 segregate free list에 넣어주는 함수이다. 항상 block을 넣을 때 list의 가장 앞에 넣어줄 것이기 때문에 PREV_BLK_SP_W(bp,NULL) 을 해서 가장 앞의 block 임을 알린다. 그리고 NEXT_BLK_SP_W 에는 기존의 segregate free list가 가리키고 있던 Ptr을 넣어준다. 그리고 이 ptr에도 이제 prev free block 이 생겼기 때문에 PREV_BLK_SP_W 로 bp를 넣어준다. 마지막으로 seg_delete 함수와 비슷하게 segregate_list_pointer 가 가리키는 ptr을 bp로 바꿔준다.

```
void seg_insert(void* bp){

    NEXT_BLK_SP_W(bp,segregate_list_pointer[seg_index(GET_SIZE(HDRP(bp)))]);

    if(segregate_list_pointer[seg_index(GET_SIZE(HDRP(bp)))]!=NULL){
        PREV_BLK_SP_W(segregate_list_pointer[seg_index(GET_SIZE(HDRP(bp)))],bp);
    }
      PREV_BLK_SP_W(bp,NULL);
    segregate_list_pointer[seg_index(GET_SIZE(HDRP(bp)))]=bp;
}
```

  -void seg_index(size_t A)

  이 함수는 size를 받아 segregate free list의 index를 결정해주는 함수이다. 2의 제곱근을 기준으로 범위를 나눴으며 현재 필자는 8개로 나눴다. 각 524288, 65536, 8192, 2048, 512, 128, 16, 그리고 그 외로 구성되어 있다.

```
int seg_index(size_t A){
    int index=0;
     if(A<=524288)
    index=7;
     if(A<=65536)
    index=6;
     if(A<=8192)
    index=5;
    if(A<=2048)
    index=4;
     if(A<=512)
    index=3;
     if(A<=128)
    index=2;
     if(A<=16)
    index=1;
    return index;
```

-static void* extend_heap(size_t words)

이 함수는 기존의 책에서 제공하던 함수와 차이가 없다. 기존의 heap 공간이 부족하기 때문에 mem_sbrk 함수를 통해 새롭게 heap 공간을 할당해주는 함수이다.

```
static void* extend_heap(size_t words)
{
    char* bp;
    size_t size;

    /* Allocate an even number of words to maintain alignment */
    size = (words % 2) ? (words + 1) * WSIZE : words * WSIZE;
    if ((long)(bp = mem_sbrk(size)) == -1)
        return NULL;

    /* Initialize free block header/footer and the epilogue header */
    PUT(HDRP(bp), PACK(size, 0));         /* Free block header */
    PUT(FTRP(bp), PACK(size, 0));         /* Free block footer */
    PUT(HDRP(NEXT_BLKP(bp)), PACK(0, 1)); /* New epilogue header */
    return bp;
}
```

- static void place_ext(void * bp, size_t asize)

Free Block에 메모리를 할당 할 때 해당 메모리의 internal fragment가 생기는지 아닌지 확인해서 split 해주어 memory utilization을 높이는 함수이다. 여기서는 extend_heap을 통해 받은 bp 이다. 현재의 GET_SIZE(HDRP(bp)) 를 확인한 다음 csize-asize>=2*DSIZE 인지 확인하다. 만약 맞다면 split을 진행한다. 그리고 뒤에 남은 free block은 seg_insert 해서 segregate free list에 넣어준다. 그렇지 않다면 그냥 현재 bp에 값을 넣어준다.

```
static void place_ext(void* bp, size_t asize)
{ //extended 해서 넣을 때
    size_t csize = GET_SIZE(HDRP(bp));
    if ((csize - asize) >= (2 * DSIZE)) {
        PUT(HDRP(bp), PACK(asize, 1));
        PUT(FTRP(bp), PACK(asize, 1));
        //split
        bp = NEXT_BLKP(bp);
        PUT(HDRP(bp), PACK(csize - asize, 0));
        PUT(FTRP(bp), PACK(csize - asize, 0));
        seg_insert(bp);
    }
    else {
        PUT(HDRP(bp), PACK(csize, 1));
        PUT(FTRP(bp), PACK(csize, 1));

    }

}
```

- static void place_seg(void * bp, size_t asize)

  Free Block에 메모리를 할당 할 때 해당 메모리의 internal fragment가 생기는지 아닌지 확인해서 split 해주어 memory utilization을 높이는 함수이다. 여기서는 segregate free list 를 통해 여유 공간이 있는 free block을 받은 bp 이다.

  우선 seg_delete 함수를 통해 해당 free block을 free 해제 시켜준다. 현재의 GET_SIZE(HDRP(bp)) 를 확인 한 다음 csize-asize>=2*DSIZE 인지 확인하다. 만약 맞다면 split을 진행한다. 그리고 뒤에 남은 free block은 seg_insert 해서 segregate free list에 넣어준다. 그렇지 않다면 그냥 현재 bp에 값을 넣어준다.

```
static void place_seg(void* bp, size_t asize)
{
    size_t csize = GET_SIZE(HDRP(bp));

    //seglist에서 가져오기
        seg_delete(bp);
        if ((csize - asize) >= (2 * DSIZE)) {
        PUT(HDRP(bp), PACK(asize, 1));
        PUT(FTRP(bp), PACK(asize, 1));
        //split해서 seglist에 넣기
        bp = NEXT_BLKP(bp);
        PUT(HDRP(bp), PACK(csize - asize, 0));
        PUT(FTRP(bp), PACK(csize - asize, 0));
        seg_insert(bp);
    }
    else {
        PUT(HDRP(bp), PACK(csize, 1));
        PUT(FTRP(bp), PACK(csize, 1));
    }
}
```

- Static void* find_fit(size_t asize)

  Segregate free list를 best fit으로 탐색하며 가장 최소한의 공간을 찾아가는 것

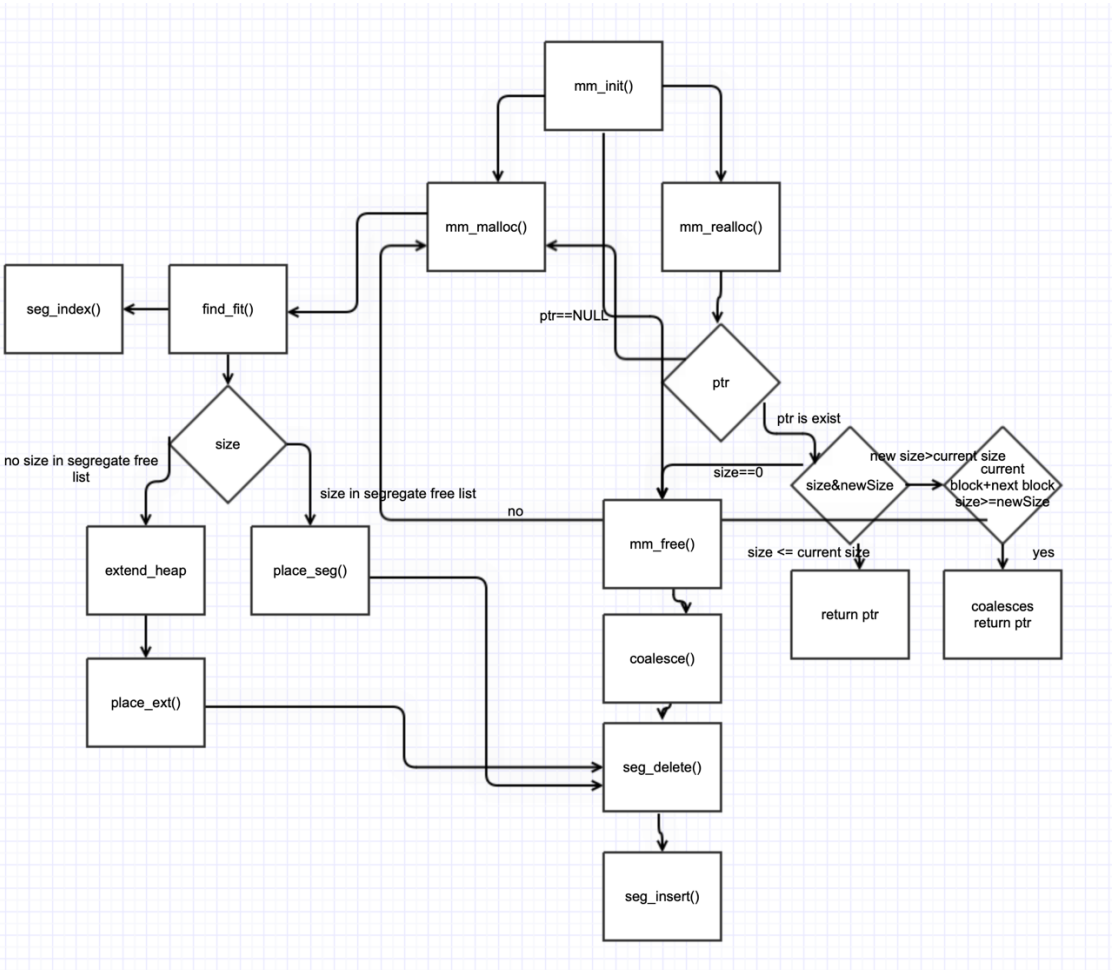
이다. Memory utilization을 최대화하고자 하였다.

우선 seg_index 함수를 통해 해당 size의 index를 받아온다. 그리고 반복문을 돌며 가장 최소한의 공간에 맞는 bp를 찾는다. 해당 index에 공간이 없다면 index++을 해서 최대 segregate_num까지 돌아본다. 이래도 만약 모든 공간에 bp가 없다면 NULL을 return 한다. 아니면 best_fit을 Return 한다.

```
static void* find_fit(size_t asize)
{
    void* bp;
    void* best_fit=NULL;
    size_t size;
    size_t least_size= 999999;
    int i=seg_index(asize);

    while(i<segregate_num){
        bp=segregate_list_pointer[i];
        i++;
        while (1) {
            if(!bp)
            break;
            size = GET_SIZE(HDRP(bp));
            if ((asize > size))
            {
                bp = NEXT_BLK_SP_R(bp);
                continue;}
            else{
                if (size < least_size) {
                    least_size = size;
                    best_fit = bp;
                }
                bp = NEXT_BLK_SP_R(bp);
            }
        }
        if (best_fit!=NULL) return best_fit;
    }
    return NULL;
}
```

3. **Flow chart**



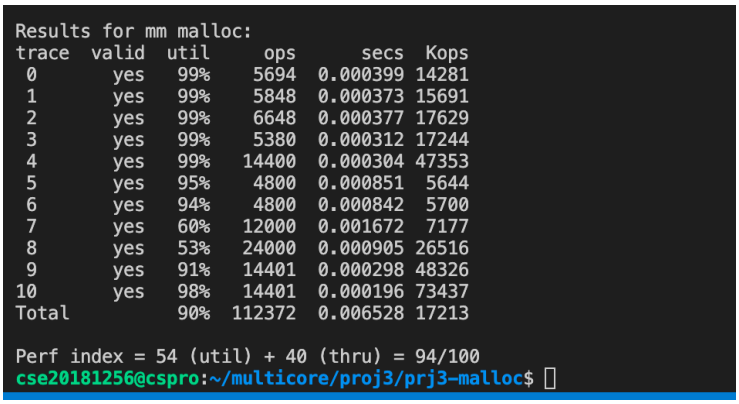
위에서 설명한 개발 내용을 flow chart로 그려 보았다.

다시 요약하자면 mm_init 함수를 통해 우리의 heap 공간을 할당해 놓는다. 이때 segregated free list 를 구현하기 위해 해당 공간 역시 할당해야 한다. 그리고 mm_malloc 혹은 mm_realloc 함수가 호출 될 것 이다. Mm_malloc 에서는 find fit 함수를 호출하여 segregate free list 를 탐색해 원하는 size에 맞는 block이 있는지 확인하여 있으면 place_seg() 함수를 호출해 해당 공간에 대한 split, segregated free list 관리를 해준다. 없다면 extend_heap 함수를 호출해 새로운 heap 공간을 만들어 준다. 여기서는 place_ext() 함수를 호출하여 free block split 등의 과정을 통해 segregated free list 관리를 해준다. Mm_realloc이 호출 되는 경우 기존의 block에 덮어쓰는 느낌이기 때문에 우선 new size 와

current size를 비교하여 현재 size가 더 크면 그냥 해당 ptr를 그대로 return 하면 된다. 이게 아니라면 current size, next block size를 더해 new size보다 큰지 확인하여 크다면 coalesces 해서 현재 ptr을 return 한다. Next block이 free block이 아니거나 더해서 더 작은 값이라면 새롭게 mm_malloc을 호출하여 새로운 공간을 할당한다.

Mm_free 함수를 호출할 때는 해당 block을 기준으로 앞, 뒤의 블록(prev, next)이 비어 있는지 확인하고 병합한 다음 적절하게 segregated free list를 관리해주면 된다.

4. 구현 결과



해당 결과는 terminal 에 ./mdriver -V 명령어를 쳐서 결과를 출력한 것이다. 일단 correctness 즉, valid는 모두 yes 가 나온 걸로 봐서 정상적으로 작동 하는 것으로 보인다. 그리고 performance의 경우 utilization 54 점, thru는 약 40점이 나오는 것을 확인 할 수 있다. 현재 memory utilization을 높이기 위해 best fit 으로 구현하고 있는데 만약 first fit으로 바꾸면 throughput이 조금 더 증가할 수 도 있을 것 같다. 하지만 이 경우 memory utilization이 낮아 질 수 도 있기 때문에 double sword 라 볼 수 있다.

또한 segregate_num = 8, segregate free list index 를 8 개로 잡았는데 이를 조금 더 세밀하게 mdriver 함수의 구현 원리를 파악하여 원하는 메모리 할당 사이즈를 정확하게 알고 나눈다면 memory utilization 측면에서 점수가 더 좋게 나올 것 같다.

하지만 필자의 경우 8, 12, 16 등등 나눠봤는데 유의미한 차이가 없는 것 같아 8 개로 나누었다.

처음에 코드에 대해 이해하기 위해 implicit list using length 를 활용하여 구현하였을 때는 점수가 약 55점 정도 나왔던 것 같다. 이해 후 segregate free list로 구현하니 최종적으로 총점 94 점을 도달 하였음을 알 수 있다. 이를 토대로 코드 자체는 훨씬 복잡하지만 segregate free list가 memory utilization, throughput 등 측면에서 모두 우수한 결과를 보임을 파악 가능했다.