2021.04.30.
시스템 프로그래밍

Malloclab : Report

사범대학 국어교육과
2017-14342 도양훈

1. 구현 과정

```
/*
 * Below is the structure of heap which I mean to implement.
 * It is special in that it maintains information of heap inside itself,
 * which means the segregated free list is saved in dynamic memory area.
 * Also it has a prologue block which consists of segregated free list,
 * but it does not have a epilogue block, and has HEAP_T macro instead.
 * HEAP_T always points end of heap by calculation with HEAP_SIZE.
 *
 * Main methods used are
 *   - segregated free list
 *   - explicit free list
 *   - immediate coalesce
 *
 *     31 30 29 28 27 26 25 24 ... 07 06 05 04 03 02 01 00
 *     ------------------prologue block-------------------
 * <= HEAP_H
 *  00 padding(0)
 *  01 [HDR] size of prologue block                    | 1
 * <= heap_h
 *  02 size of overall heap
 *  03 SEG(i) will be pointer of a block.
 *  04 SEG(i) will be pointer of a block.
 * ...
 *   N SEG(i) will be pointer of a block.
 * N+1 [FTR] size of prologue block                    | 1
 *     ------------------prologue block-------------------
 */
```

우선 힙의 구성을 임의로 변경하였다. 교과서에서는 prologue block과 epilogue block을 모두 소개하고 있지만 본 코드에서는 prologue block만 사용하였다. 초기 구현 의도는 epilogue block의 memory를 save하는 것이었는데, 아주 작은 공간이고, 또 이 때문에 coalesce 등에서 번거로운 heap area checking 과정을 여러 번 거쳐야 했기 때문에 결과적으로는 좋은 선택이라고 보기 어려웠던 것 같다.
또 한 가지 본 코드의 특징은 heap 내부에 segregated free list를 운영한다는 것이다. 연속되는 N개의 주소는 각각 자신의 segregated 범위의 block 하나를 point한다. 정확히는 그 중 가장 최근에 free된 block을 point한다. 이는 LIFO를 구현함과 동시에 malloc을 위한 find_fit의 시간을 획기적으로 줄인다. segregated free list의 경우 2의 k제곱의 size를 범위로 각각 갖는다. 이 때 최소 block의 크기 때문에 k = 0, k = 1에는 block이 존재할 수 없기 때문에 이를 조절하여 최적화를 조금 더 구현할 수 있다. 다만 본 코드에서는 코드의 간결함을 위해 구현하지 않았다.

```
/*********************************************************
 * original constants and macros
 *********************************************************/

/* segregated free list is devided in 2^k form */
#define MAX_SEGS 20
#define SEG(i) *((void**)(heap_h + ((unsigned int) (i + 1) * sizeof(void*))))

/* overall size of heap will be shown by footer of prologue block */
#define HEAP_SIZE GET(heap_h)
```

```
/* first address and last address of heap */
#define HEAP_H heap_h - DSIZE
#define HEAP_T heap_h - WSIZE + HEAP_SIZE

/* increase heap size */
#define HEAP_INCR(size) PUT(heap_h, PACK(size, 0))

/* get or set predecessor, successor in seglist */
#define SET_PRED(ptr, pred) PUT(((void**)(ptr)), pred)
#define GET_PRED(ptr) (*(void**)(ptr))
#define SET_SUCC(ptr, succ) PUT(((void**)(ptr + WSIZE)), succ)
#define GET_SUCC(ptr) (*(void**)(ptr + WSIZE))
```

20MB를 cover하기 위해 segregated free list의 개수는 20개로 설정하였고, SEG(i)라는 매크로를 통해서, array를 global/static하게 정의하지 않았지만 각 원소에 접근 가능하도록 설정하였다. 위의 도식에도 나와있지만 HEAP_H와 HEAP_T는 heap의 가장 위와 가장 아래를 가리킨다. mem_heap_hi()와 같은 함수를 사용할 수도 있었지만 좀 더 정확하고 직관적인 접근을 위해 주소에 직접 access하였다. prologue block에는 segregated free list와 이전에 heap 전체 size를 수동적으로 저장하도록 했기 때문에, 이를 이용하여 HEAP_SIZE를 구할 수 있다.

```
/************************************************************
 * global variables
 ************************************************************/

void* heap_h; /* this points first address in heap */
```

위와 같은 노력을 통해 global variable을 단 하나만 정의할 수 있었다. 이 변수 역시 mem_heap_hi로 대체할 수 있었지만, 코드 구현 의도와 조금 다르다고 생각하였고 다른 이유로는 시간이 부족하기도 하여 그처럼 구현하지는 아니하였다.

```
/************************************************************
 * functions for debug
 * be careful in use since they cause bad performance(throughput)
 ************************************************************/

static void print_heap()
{
   printf("\nPrinting overall heap structure...\n");
   for(void* ptr1 = heap_h; ptr1 < HEAP_T; ptr1 = NEXT_BLKP(ptr1))
   {
      for(void* ptr2 = HDRP(ptr1); ptr2 <= FTRP(ptr1); ptr2 += WSIZE)
      {
         if(ptr2 == HDRP(ptr1))
         {
            printf("[%x][HDR][size = %d][a = %d]\n", ptr2, GET_SIZE(ptr2), GET_ALLOC(ptr2));
         }
         else if(ptr2 == FTRP(ptr1))
         {
            printf("[%x][FTR][size = %d][a = %d]\n", ptr2, GET_SIZE(ptr2), GET_ALLOC(ptr2));
         }
      }
   }

   return;
}

static void print_seglist()
{
   printf("\nPrinting overall seglist structure...\n");
```

```
   for(int i = 0; i < MAX_SEGS; i++)
   {
      printf("[%d]\n", i);
      for(void* ptr = SEG(i); ptr != NULL; ptr = GET_PRED(ptr))
      {
         printf("\t[%x][%d]\n", ptr, GET_SIZE(HDRP(ptr)));
      }
   }

   return;
}

int mm_check(void)
{
   /* loop in seglist checks...
    *  - is every block in the free list marked as free?
    *  - are there any contiguous free blocks
    *    that somehow escaped coalescing?
    *  - do the pointers in the free list point to valid free blocks?
    */
   for(int i = 0; i < MAX_SEGS; i++)
   {
      for(void* ptr = SEG(i); ptr != NULL; ptr = GET_PRED(ptr))
      {
         if(GET_ALLOC(HDRP(ptr)) || GET_ALLOC(FTRP(ptr)))
         {
            printf("heap is not consistent in [seglist] area.\n");
            return 0;
         }
         if(!GET_ALLOC(HDRP(PREV_BLKP(ptr))) || HDRP(NEXT_BLKP(ptr)) < HEAP_T &&
!GET_ALLOC(HDRP(NEXT_BLKP(ptr))))
         {
            printf("heap is not consistent in [seglist] area.\n");
            return 0;
         }
      }
   }
   /* loop in heaplist checks...
    *  - do any allocated blocks overlap?
    *  - do the pointers in a heap block point to valid heap addresses?
    */
   for(void* ptr1 = heap_h; ptr1 < HEAP_T; ptr1 = NEXT_BLKP(ptr1))
   {
      if(FTRP(ptr1) > HDRP(NEXT_BLKP(ptr1)))
      {
         printf("heap is not consistent in [heaplist] area.\n");
         return 0;
      }
   }

   return 1;
}
```

총 세 개의 debug function을 구현하였다. 처음에는 이 같은 debug function 없이 무턱대고 implement를 시작하였는데, 문제가 발생했을 때 원인을 찾지 못하고 고민하는 시간만 길어져, 초반에 시간이 조금 걸리더라도 debug function을 탄탄하게 작성하고 시작하자고 생각하고 작성하였다. print_heap()은 heap의 관점에서 heap의 처음부터 끝까지 block의 상태를 보여준다. block 내부의 값도 출력하도록 할 수 있지만 그렇게 되면 오히려 출력 값이 너무 많아 heap의 구조를 알아보기 힘들어 이 같은 구현은 이후 삭제하였다. print_seglist는 segregated free list의 관점에서 free block을 보여준다. 한 개 이상의 free block이 linked list로 이어지는 경우 이들을 모두 출력한다. 마지막 check_heap의 경우, 사실은 위의 두 함수를 통해 heap의

consistency를 확인할 수 있지만, 이를 자동화하기 위해 (그리고 과제의 spec을 만족하기 위해) 작성하였다. heap의 관점, seglist의 관점에서 loop를 돌며 inconsistent한 area가 있을 경우 오류 메시지를 printf하고, 0을 return한다.

```
/*
 * coalesce - it manages out of boundary issue
 *            with comparing block pointer and HEAP_H, HEAP_T
 *            although comparison with HEAP_H is completely useless.
 */
static void* coalesce(void* ptr)
{
   size_t prev_alloc;
   size_t next_alloc;
   size_t size = GET_SIZE(HDRP(ptr));

   if(PREV_BLKP(ptr) < HEAP_H) prev_alloc = 1;
   else prev_alloc = GET_ALLOC(FTRP(PREV_BLKP(ptr)));
   if(NEXT_BLKP(ptr) > HEAP_T) next_alloc = 1;
   else next_alloc = GET_ALLOC(HDRP(NEXT_BLKP(ptr)));
   if(prev_alloc && next_alloc)
   {
      /* A - F - A */
      return ptr;
   }
   else if(prev_alloc && !next_alloc)
   {
      /* A - F - F */
      delete_free(ptr);
      delete_free(NEXT_BLKP(ptr));
      size += GET_SIZE(HDRP(NEXT_BLKP(ptr)));
      PUT(HDRP(ptr), PACK(size, 0));
      PUT(FTRP(ptr), PACK(size, 0));
      add_free(ptr);
      return ptr;
   }
   else if(!prev_alloc && next_alloc)
   {
      /* F - F - A */
      delete_free(PREV_BLKP(ptr));
      delete_free(ptr);
      size += GET_SIZE(HDRP(PREV_BLKP(ptr)));
      PUT(HDRP(PREV_BLKP(ptr)), PACK(size, 0));
      PUT(FTRP(PREV_BLKP(ptr)), PACK(size, 0));
      add_free(PREV_BLKP(ptr));
      return PREV_BLKP(ptr);
   }
   else
   {
      /* F - F - F */
      delete_free(PREV_BLKP(ptr));
      delete_free(ptr);
      delete_free(NEXT_BLKP(ptr));
      size += GET_SIZE(HDRP(PREV_BLKP(ptr))) + GET_SIZE(HDRP(NEXT_BLKP(ptr)));
      PUT(HDRP(PREV_BLKP(ptr)), PACK(size, 0));
      PUT(FTRP(PREV_BLKP(ptr)), PACK(size, 0));
      add_free(PREV_BLKP(ptr));
      return PREV_BLKP(ptr);
   }
}
```

coalesce의 경우 교과서와 비슷하게 구현했다. 내 구현에서는 epilogue block이 없었기 때문에 HEAP_T와도 비교해 마치 뒤가 allocated block인 것처럼 간주하도록 하였다.

```
/*
 * find_seg - use >> operation to caculate 2^k size.
 *            segregated list can be optimized more if i starts from -1.
 */
static int find_seg(size_t size)
{
   int i;

   for(i = 0; size > 1 && i < 19; i++)
   {
      size = size >> 1;
   }

   return i;
}

/*
 * add_free - add block in free list,
 *            size and alloc bit in hdr, ftr should be changed already.
 */
static void add_free(void* ptr)
{
   int i = find_seg(GET_SIZE(HDRP(ptr)));

   /* BLOCK -> PRED  */
   /* SUCC  <- BLOCK */
   if(SEG(i) == NULL)
   {
      /* no block in this seglist */
      SET_PRED(ptr, NULL);
      SET_SUCC(ptr, NULL);
      SEG(i) = ptr;
   }
   else
   {
      /* yes block in this seglist */
      SET_PRED(ptr, SEG(i));
      SET_SUCC(ptr, NULL);
      SET_SUCC(SEG(i), ptr);
      SEG(i) = ptr;
   }
}

/*
 * add_free - deete block in free list,
 *            size and alloc bit in hdr, ftr should not be changed now.
 */
static void delete_free(void* ptr)
{
   int i = find_seg(GET_SIZE(HDRP(ptr)));

   /* BLOCK -> PRED  */
   /* SUCC  <- BLOCK */
   if(SEG(i) == ptr)
   {
      if(GET_PRED(ptr) == NULL)
      {
         /* [B] */
```

```
            SEG(i) = NULL;
            SET_PRED(ptr, NULL);
            SET_SUCC(ptr, NULL);
        }
        else
        {
            /* [B] - ... */
            SET_SUCC(GET_PRED(ptr), NULL);
            SEG(i) = GET_PRED(ptr);
            SET_PRED(ptr, NULL);
            SET_SUCC(ptr, NULL);
        }
    }
    else
    {
        if(GET_PRED(ptr) == NULL)
        {
            /* ... - [B] */
            SET_PRED(GET_SUCC(ptr), NULL);
            SET_PRED(ptr, NULL);
            SET_SUCC(ptr, NULL);
        }
        else
        {
            /* ... - [B] - ... */
            SET_SUCC(GET_PRED(ptr), GET_SUCC(ptr));
            SET_PRED(GET_SUCC(ptr), GET_PRED(ptr));
            SET_PRED(ptr, NULL);
            SET_SUCC(ptr, NULL);
        }
    }

    return;
}
```

add_free와 delete_free는 각각 segregated list의 pred와 succ를 조정하여 free block을 list에 추가하거나 list에서 삭제한다. LIFO 방식을 따르기 때문에 add의 경우 무조건 마지막에 삽입하는 routine만 구현하였다.

```
/*
 * place - splits the free block when there will be allocated block,
 *         but do not split if block left behind is too small.
 */
static void place(void* ptr, size_t asize)
{
    size_t old = GET_SIZE(HDRP(ptr));

    delete_free(ptr);
    if(old - asize < 2 * DSIZE)
    {
        PUT(HDRP(ptr), PACK(old, 1));
        PUT(FTRP(ptr), PACK(old, 1));
    }
    else
    {
        PUT(HDRP(ptr), PACK(asize, 1));
        PUT(FTRP(ptr), PACK(asize, 1));
        PUT(HDRP(NEXT_BLKP(ptr)), PACK(old - asize, 0));
        PUT(FTRP(NEXT_BLKP(ptr)), PACK(old - asize, 0));
        add_free(NEXT_BLKP(ptr));
    }
```

```
    return;
}
```

place는 결국 block을 split하는 역할을 한다. 이외에도 allocation bit를 조정한다. 당연히 남겨지는 free block이 너무 작을 경우 split하지 않고 이를 포함하여 allocate하도록 하는데, 이것이 utility 면에서는 크게 긍정적인 방법은 아닌 것 같다.

```
/*
 * find_fit - returns pointer to a block if there is proper one in seglist.
 */
static void* find_fit(size_t asize)
{
    for(int i = find_seg(asize); &SEG(i) != FTRP(heap_h); i++)
    {
        if(SEG(i) != NULL)
        {
            for(void* ptr = SEG(i); ptr != NULL; ptr = GET_PRED(ptr))
            {
                if(GET_SIZE(HDRP(ptr)) >= asize)
                {
                    return ptr;
                }
            }
        }
    }

    return NULL;
}
```

find_fit은 주어진 size를 확보할 수 있는 free block이 있는지 확인하는 것이다. 일반적으로 seglist는 각 범주 안에서 크기를 고정시킨다고 이해했는데, 본 코드에서는 free block을 큰 범위로 나누는 역할을 할 뿐, 크기를 고정시키지 않는다. 이 방법이 utility 면에서 긍정적이지 않다고 판단했기 때문이다. 그러다 보니 seglist에서 hit하였더라도 그 크기가 사실은 필요한 크기보다는 작은 경우가 발생한다. 따라서 완전히 constant time에 find_fit을 장담할 수 없다. 하지만 본 코드의 경우 20 개의 seglist가 있고, 주어진 size보다 현저히 작은 block에 대한 visit을 생략할 수 있어 시간적으로 유리함을 가져갈 수 있다.

```
/*
 * extend_heap - extend heap by asize,
 *               asize should be legal size.
 */
static void* extend_heap(size_t asize)
{
    void* ptr;

    if((ptr = mem_sbrk(asize)) == (void*) NULL) return (void*) -1;
    else
    {
        /* should set extended heap as a free block */
        PUT(HDRP(ptr), PACK(asize, 0));
        PUT(FTRP(ptr), PACK(asize, 0));
        add_free(ptr);
        /* add extended heap in overall heap size */
        HEAP_INCR(HEAP_SIZE + asize);
        /* coalesce after calculating size */
        ptr = coalesce(ptr);
    }

    return ptr;
}
```

heap을 extend하고 coalesce까지 수행하는 함수이다. 여기서 coalesce를 수행하지 않으면 malloc시 공간이 낭비되기 때문에 반드시 coalesce를 수행해주어야 한다. 구현 초기에는 immediate coalesce가 아닌 지연된 coalesce method를 사용하고 싶었으나, 이 방법의 장점이 잘 이해되지 않고, throughput 보다는 utility 면에서의 성능 향상이 더 필요하다고 판단하여 사용하지 않았다.

```
/*
 * mm_init - initialize the malloc package and seglist.
 */
int mm_init(void)
{
   size_t asize = ALIGN(MAX_SEGS * WSIZE + 2 * WSIZE + SIZE_T_SIZE);

   if((heap_h = mem_sbrk(asize)) == (void*)-1) return -1;
   else
   {
      PUT(heap_h, 0);
      PUT(HDRP(heap_h), PACK(asize, 1));
      PUT(FTRP(heap_h), PACK(asize, 1));
      HEAP_INCR(asize);
   }
   for(int i = 0; i < MAX_SEGS; i++)
   {
      SEG(i) = NULL;
   }

    return 0;
}
```

구현 단계에서 prologue block을 초기화하지 않았더니, 여러 번 코드를 수행하는 과정에서 이전의 data가 남아 코드의 동작을 방해하는 일이 생겼다. heap을 alloc하고 free하지 않을 경우 발생하는 문제를 간접적으로 겪은 셈이다. 따라서 매 init마다 이를 초기화하도록 하였다.

```
/*
 * mm_malloc - Allocate a block by incrementing the brk pointer.
 *     Always allocate a block whose size is a multiple of the alignment.
 */
void* mm_malloc(size_t size)
{
    int asize;
    void* ptr;
    size_t extendsize;

    if(size == 0) return NULL;
    asize = ALIGN(size + SIZE_T_SIZE);
    if((ptr = find_fit(asize)) == NULL || ptr >= HEAP_T)
    {
       if(!GET_ALLOC(HEAP_T - WSIZE))
       {
          extendsize = asize - GET_SIZE(HEAP_T - WSIZE);
       }
       else
       {
          extendsize = asize;
       }
       if((ptr = extend_heap(extendsize)) == (void*)-1) return NULL;
    }
    place(ptr, asize);
    //printf("\n[malloc][%x][%d = %x]\n", ptr, asize, asize);
    //print_heap();
```

```
    //print_seglist();
    //if(!mm_check()) printf("There was an inconsistency in heap, please check\n");

    return ptr;
}

/*
 * mm_free - Freeing a block does nothing.
 */
void mm_free(void *ptr)
{
   PUT(HDRP(ptr), PACK(GET_SIZE(HDRP(ptr)), 0));
   PUT(FTRP(ptr), PACK(GET_SIZE(HDRP(ptr)), 0));
   add_free(ptr);
   coalesce(ptr);
   //printf("\n[ free ][%x][%d = %x]\n\n", ptr, GET_SIZE(HDRP(ptr)), GET_SIZE(HDRP(ptr)));
    //print_heap();
    //print_seglist();
    //if(!mm_check()) printf("There was an inconsistency in heap, please check\n");

   return;
}
```

아래의 주석을 해제함으로 debug function을 불러올 수 있다. 각각 helper routine을 활용해 구성하였기 때문에 간결하게 구현할 수 있었다.

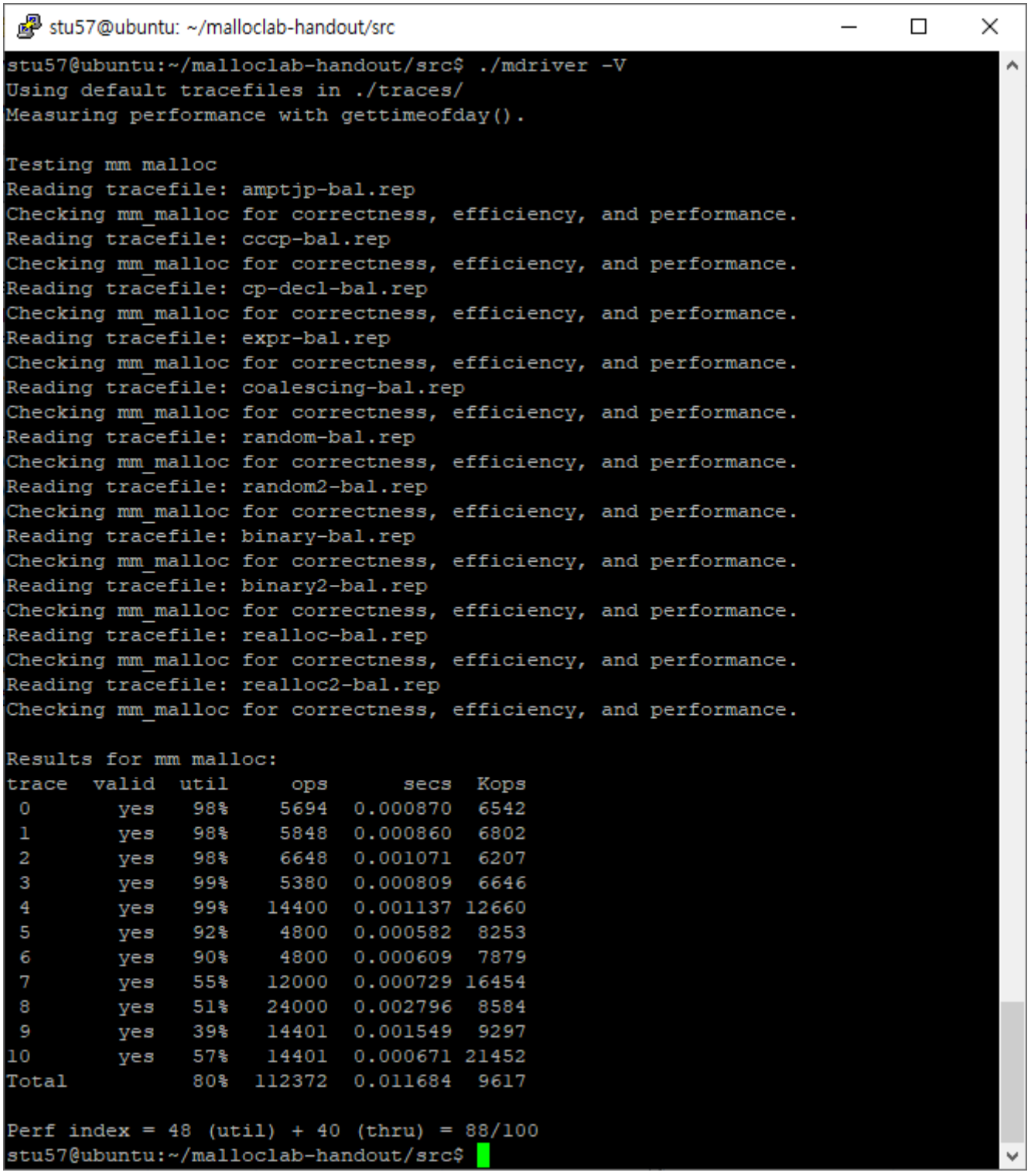
```
/*
 * mm_realloc - Implemented in cases.
 */
void* mm_realloc(void *ptr, size_t size)
{
   void* oldptr = ptr;
   void* newptr = ptr;
    size_t asize = ALIGN(size + SIZE_T_SIZE);

    if(ptr == NULL)
    {
       newptr = mm_malloc(size);
       return newptr;
    }
    if(size == 0)
    {
       mm_free(ptr);
       return NULL;
    }
    if(asize <= GET_SIZE(HDRP(oldptr))) /* block will remain as it was */
    {
       return oldptr;
    }
    if(NEXT_BLKP(oldptr) < HEAP_T && !GET_ALLOC(HDRP(NEXT_BLKP(oldptr))) &&
GET_SIZE(HDRP(NEXT_BLKP(oldptr))) + GET_SIZE(HDRP(oldptr)) > size)
    {
       place(NEXT_BLKP(oldptr), asize - GET_SIZE(HDRP(oldptr)));
       PUT(HDRP(oldptr), PACK(asize, 1));
       PUT(FTRP(oldptr), PACK(asize, 1));
    }
    else
    {
       newptr = mm_malloc(size);
       memcpy(newptr, oldptr, GET_SIZE(HDRP(oldptr)));
       mm_free(oldptr);
```

```
    }
    return newptr;
}
```

realloc의 경우 발생 가능한 각 case에 대해 if 구문으로 접근하였는데, 이보다 세련된 방법이 있었을 것 같다. 구현의 막바지에서 너무 직관적이지 않게 코드를 작성한 것 같다. 특히 마지막 if의 조건의 경우, 이를 변수화하는 등 길이를 줄일 수 있는 개선점이 여럿 있을 수 있다. 앞으로도 코딩이 끝나갈 때에도 그런만큼 코드를 빠르게 작성하기만 하는 것이 아니라 직관적이고 세련된 모습을 유지할 수 있도록 노력해야겠다.

2. 실행 결과



실행 결과는 위와 같았다. 기본적인 malloc, free의 경우 utility를 98%에서 99% 정도까지 최적화하였다. 하지만 7번부터 10번 tracefile에 대한 utility가 많이 떨어지는 모습을 보였다.

7번과 8번은 binary라고 하여 작은 block과 큰 block을 연달아 malloc한 뒤 큰 블록만 free하여 heap을 중간 중간 구멍이 난 형태로 만든다. 이후 작은 block과 큰 block을 합친 크기의 block을 malloc하는데, heap의 중간 중간 구멍에는 이 block이 들어가지 못하므로 heap을 extend하는 수밖에 없다. 이것이 반복되면 앞의 많은 구멍들만큼 heap이 낭비되게 된다. 이에 대한 최적화를 고민해보았지만, 이미 malloc된 block의 위치를 옮길 수 없으면 사이의 구멍을 조절하기 어렵다고 생각했다.
9번과 10번의 tracefile 역시 기본 idea는 7, 8번과 유사하게 번갈아가며 free하여 구멍을 내고, 이 구멍보다 큰 realloc 명령을 수행함으로써 heap 전체로 보면 공간이 많이 남음에도 불구하고 새로운 extend_heap 명령을 수행하도록 하는 것이다. allocated block을 한 곳으로 모아놓고 freed block을 한 곳으로 모은다면 훨씬 효율적인 공간 운용이 가능할텐데 아쉬웠다.
한 가지 아이디어로 떠올랐지만 구현하지 못한 것은 가상 가상 주소를 부여하는 것이다. malloc된 block에 가상 가상 주소를 부여한 뒤, 가상 가상 주소를 실제 가상 주소로 interpret하여 heap memory에 접근하도록 한다. 이러면 allocated block이 같은 곳에 머물러야할 이유가 사라진다. 따라서 block이 free되는 즉시 그 뒤의 allocated block을 앞으로 당길 수 있다. 물론 가상 주소 interpreter가 overhead로서 남겠지만, utility 면에서 효용성이 있을 것 같다. 다만 memory 접근을 반드시 allocator의 interpreter를 통하도록 하게 되는데, 이는 코드의 자율성 면에서는 큰 단점이지만 동시에 아래에서 지적하고자 하는 heap area의 보안 문제에 대해서도 좋은 해결책이 될 것 같다.

3. 어려웠던 점

대부분의 연산이 포인터 연산이라는 점이 가장 어려웠다. 기존에 C 언어를 공부했었지만, 그때 포인터에 대해 기본을 충실히 다지지 않았다고 느껴졌다.
또한 segregated free list를 구현하려고 했는데 global/static array를 사용할 수 없다는 과제 spec 때문에 고민을 많이 했다. 이에 대해 고민하는 데에만 하루 정도를 소모한 것 같다. 크게 두 가지 option을 생각하게 되었다. 첫 번째는 본 코드에서 구현한 바와 같이 heap 안에 segregated free list를 두고 관리하는 방법이었고, 두 번째는 segregated free list를 구현하지 않고 순차적인 explicit free list를 구현하는 방법이었다. 첫 번째 방법은 segregated list를 사용하는 만큼 시간 면에서 유리하지만, heap을 고정적으로 차지하는 list가 작긴 해도 분명 있다는 것이 공간 면에서의 단점이었다. explicit list는 반대로 시간 면에서는 불리하지만 공간 면에서는 조금 유리하다고 할 수 있는 방법이었다.
크게 두 가지 측면에서 첫 번째 segregated free list를 선택했다. 첫 번째 이유는 segregated free list의 각 list를 explicit free list를 통해 구현하면 seglist 개수 * 1 word의, 크지 않은 공간만으로 구현할 수 있다는 것이었다. 두 번째 이유는 tracefile을 살펴보니 segregated free list를 위해 희생되는 공간에 비해 allocation되는 공간이 압도적으로 컸기 때문이다.
반대로, throughput은 어느 정도 quality로 구현할 자신이 있었던 데에 반해 utility는 최적화 방법이 많이 떠오르지 않아 두 번째 방법을 선택할지 고민하기도 했지만, 결국 첫 번째 방법으로 구현을 시작하였고, 그 뒤로는 그 방법이 마음에 들기도 하고 시간이 모자라기도 하다는 이유로 그 방법을 유지하게 되었다.
만약 실제 개발 상황이었다면 두 가지 방법을 모두 간단히 구현해보고 성능을 직접 비교하는 것도 좋았을 것 같다.

4. 놀라웠던 점 및 궁금했던 점

이 과제를 늦었지만 무사히 끝마칠 수라도 있었다는 점이 가장 놀라웠다. 일주일 정도 전부터 과제를 시작했지만 급하지 않다는 생각에 대충 임하게 되었는데, 시간만 낭비하고 아무런 성과를 거두지 못했다. 사흘을 남기고서야 교과서를 정독하며 dynamic memory allocation에 대해 처음부터 다시 공부했다. 이미 수업을 통해 접했던 내용이라 교과서를 읽으며 어떤 최적화 기법을 적용해볼지 자연스럽게 떠올릴 수 있었다.
더불어 궁금증이 생긴 부분도 있었다. 나 같은 경우 prologue block을 사용하고 여기에 segregated list를 저장했으며, 각 block들마다 header와 footer를 두었다. 본 코드에서는 매크로를 이용해 이들에 접근하고 있지만, 그 역시 다른 주소일 뿐이다. 따라서, malloc의 caller가 이 주소에 접근하게 될 가능성이 있다. 가령, 본 코드를 예로 들면, ptr = malloc(64)를 수행한 뒤, *(ptr - 4) 연산을 수행하면 이 block의 header에 접근할 수 있게 된다. header에 접근하게 되는 경우 block size 정보를 바꾸거나 allocation bit도 수정할 수 있다. 이때 dynamic memory allocation이 정상적으로 수행되지 않는 결과 역시 예측된다. 따라서 이에 대한 보안이 이루어질 수 있는지, 혹은 이 역시 caller의 책임으로 두고 굳이 구현하지 않는지 궁금했다.