Multicore Programming Project 3

Dynamic Memory Allocator

담당 교수 : 박성용

이름 : 이진용

학번 : 20191630

1. 개발 목표

- 해당 프로젝트에서 구현할 내용을 간략히 서술.

이번 프로젝트에서는 malloc과 같은 dynamic memory allocator을 구현한다. 구현할 함수는 malloc, free, realloc으로 기존의 malloc에서 작동하는 것과 동일하게 기능하도록 구현한다.

학습한 implicit allocator에서 확장하여 explicit allocator을 구현하는 것을 목적으로 한다. 또한 realloc을 구현할 때 불필요한 연산을 줄여 throughput을 올리고 최적화를 통해 memory utilization을 올리도록 구현한다.

구현한 allocator의 throughput과 utilization은 해당 내장된 mdriver을 통해 성능을 측정한다. 이때 총 11개의 tracefile을 통해 구현한 malloc과 free를 사용하여 테스트하게 된다.

2. 개발 범위 및 내용

A. 개발 범위

- 아래 항목을 구현했을 때의 결과를 간략히 서술

1. Implicit Allocator

먼저 바탕이 되는 기본 allocator를 구현한다. block의 header과 footer에 블럭의 사이즈를 정하여 다음 블럭과 이전 블럭의 위치를 계산할 수 있는 구조로 구현한다. 또한 header의 lsb에는 해당 블럭이 할당되었는지 여부를 저장한다.

메모리 할당 시에 heap의 head부터 비어있는 block을 확인해가며 충분히 할당할 수 있는 블럭을 linear search로 찾게 된다. 만약 충분한 block이 없다면 mem_sbrk()함수를 통해 전체 heap size을 늘리게 된다.

메모리의 할당을 해제할 때, 해당 블럭의 allocate bit을 초기화 시켜준다. 또한 양 옆에 free된 block이 있다면 해당 블럭과 합치는 과정을 진행한다.

2. Explicit Allocator

implicit allocator에서 explicit allocator으로 확장한다. implicit에서 메모리를 할당할 때, head부터 시작하여 next block으로 이동하며 해당 block이 할당이 되어 있는지 할당을 위한 사이즈가 충분한지를 확인한다. explicit 방식은 free된 block들을 linked list로 관리하여 2 WORD의 블럭에 각각의 prev, next 포인터를 저장하여 이전과 이후의 free block의 주소를 저장해준다. 때문에 새로운 메모리를 할당할 때, free된 list중에서 linear search를 통해 적당한 size의 block을 선택하여 할당하게 된다.

free된 블럭을 따로 관리하기 때문에 coalesce시에 추가적으로 linked list을 업데이트해주는 과정을 추가해준다. 마찬가지로 메모리 할당 시 allocate size보다 큰 block에 할당하는 경우 남은 블럭을 새로운 free block으로 추가하고 이를 list에다 넣어준다.

3. Realloc

malloc과 free가 구현이 완료 후 구현을 진행한다. realloc 구현에서 가장 큰 목표는 memory utilization을 올리는 것이다. 기존의 block에서 size를 늘리는 경우 해당 블럭을 free하고 새로운 블럭을 할당하고 기존의 payload를 복사하게 된다. 이렇게 구현하는 경우 조금의 사이즈를 차츰 늘려가는 case의 경우 매번 새로운 malloc과 free를 해야 하기에 utilization과 throughput 모두 떨어지는 결과를 갖는다.

이를 해결하기 위해 realloc 시 coalesce와 마찬가지로 양옆의 free된 블럭을 확인하고 만약 해당 prev, next 블럭의 사이즈만으로도 realloc 할당이 가능할 경우 free와 malloc없이 size를 조절한다.

추가적으로 heap의 tail에 가까운 즉 가장 마지막의 block 사이즈를 늘리게 될 경우 새로운 malloc을 하기보다 필요한 크기만큼 heap을 extend하여 해당 realloc을 할당해주게 구현한다.

4. Swap split place

할당할 블럭의 사이즈가 free block size보다 최소 블럭 사이즈보다 작게 된다면 해당 free block은 필요한 만큼만 할당하고 필요없는 부분은 free block으로 추가해주게 된다. 이때 단순히 free block의 앞부분에만 할당하게 된다면 일부 작은 크기의 블럭들이 free 되었을 때, 큰 블록들 사이에서 할당하기에 애매한 사이즈로 남게 되어 memory utilization이 떨어지게 된다.

이를 해결하기 위해 일정 크기 이상의 블럭들은 할당시 tail과 가깝게, 일정 크기 이하의 블럭은 head에 가깝게 할당한다. 이러면 작은 블럭들을 한편에 모아 free되었을 때, coalesce가 되기 유리하게 구현한다.

B. 구현

- **mm_init**

```
int mm_init(void)
{
    if ((heap_head = mem_sbrk(6 * WSIZE)) == (void *)-1)
    {
        return -1;
    }
    free_head = heap_head + WSIZE; /* Init head ptr */
    PUT(heap_head, 0);             /* Alignment padding */
    PUT(heap_head + WSIZE, 0);     /* free list head ptr*/
    PUT(heap_head + (2 * WSIZE), 0);
    PUT(heap_head + (3 * WSIZE), PACK(DSIZE, 1)); /* Prologue header */
    PUT(heap_head + (4 * WSIZE), PACK(DSIZE, 1)); /* Prologue footer */
    PUT(heap_head + (5 * WSIZE), PACK(0, 3));     /* Epilogue header */
    heap_head += (4 * WSIZE);

    /* Extend the empty heap with a free block of CHUNKSIZE bytes */
    if (extend_heap(CHUNKSIZE / WSIZE) == NULL)
    {
        return -1;
```

```
    }
    SET_TAG(HDRP(NEXT_BLOCK(heap_head)));
    INSERT_LIST(NEXT_BLOCK(heap_head), free_head);
    return 0;
}
```

구현한 malloc package를 사용하기전 초기화를 진행한다. 가장먼저 6byte의 heap size를 늘려 alignment padding, prologue, free list head, epilogue 순으로 할당해준다. 이후 chunksize만큼 heap을 늘려주고 첫 free block을 linked list로 insert해준다.

- **mm_malloc**

```
void *mm_malloc(size_t size)
{

    size_t alloc_size, extendsize;
    char *new;
    if (size == 0)
    {
        return NULL;
    }

    alloc_size = MAX(DSIZE * 2, (ALIGN(size) - size >= WSIZE) ?
ALIGN(size) : (ALIGN(size) + DSIZE));

    if ((new = find_fit_at_list(alloc_size)) != NULL)
    {
        return place(new, alloc_size);
    }
    extendsize = MAX(alloc_size, CHUNKSIZE);
    if (!GET_TAG(heap_tail))
    {

        extendsize -= GET_SIZE(heap_tail - WSIZE);
    }
    if ((new = extend_heap(extendsize / WSIZE)) == NULL)
    {
        return NULL;
    }

    return place(new, alloc_size);
}
```

구현한 malloc함수이다. 해당 사이즈만큼 allocate 요청이 들어오면 일단 header,

footer, alignment를 모두 고려한 실제 block allocate 사이즈를 계산한다. 이후 해당 사이즈를 할당 가능한지 find_fit_at_list함수를 통해 확인하고 가능하다면 해당 블럭의 위치를 return 받는다. 실제 할당을 위한 place함수를 call하고 종료하게 된다. 만약 할당할만한 free block이 없다면 epliogue block앞에 블럭이 비어있는지 확인 후 해당 블럭의 크기와 필요한 사이즈만큼을 추가적으로 extend heap함수를 통해 늘려준다. 이후 늘어난 위치에 해당 블럭을 할당해준다.

- **find_fit_at_list**

```
static inline void *find_fit_at_list(size_t size)
{
    char *cur = free_head;
    char *temp = NULL;

    while ((cur = SUCC(cur)) != free_head)
    {

        if (BLOCK_SIZE(cur) >= size)
        {
            if (temp == NULL)
            {
                temp = cur;
            }
            else
            {
                if (BLOCK_SIZE(cur) < BLOCK_SIZE(temp))
                {
                    temp = cur;
                }
            }
        }
    }
    if (temp == NULL)
        return NULL;
    ERASE(temp);
    return temp;
}
```

해당 함수는 malloc에서 call되어 free block중 적절한 block을 찾고 해당 위치를 반환해준다. 이때 best fit을 찾기 위해 어느정도 throughput을 포기하고 memory utilization을 올려준다. 할당할 사이즈보다 크고 free된 block중 가장 작은 것을 반환해준다. 반환하기 이전에 free list에서 해당 블럭은 delete시켜준다.

- **place**

```
static void inline *place(char *bp, size_t asize)
{
    size_t size = GET_SIZE(HDRP(bp));
    size_t next_size;
    char *next;

    SET_ALLOC(HDRP(bp));
    if (size != asize && size - asize >= 2 * DSIZE)
    {
        next_size = size - asize;

        if (asize < 12 * DSIZE || next_size < 4 * DSIZE)
        {
            // alloc current block
            SET_SIZE(HDRP(bp), asize);
            next = NEXT_BLOCK(bp);
            PUT(HDRP(next), PACK(next_size, 0));
            PUT(FTRP(next), PACK(next_size, 0));
        }
        else
        {
            // alloc next to next block
            next = bp;
            SET_SIZE(HDRP(next), next_size);
            PUT(FTRP(next), PACK(next_size, 0));
            // swap bp and next pointer
            bp = NEXT_BLOCK(next);
            SET_ALLOC(HDRP(bp));
            SET_SIZE(HDRP(bp), asize);
        }
        SET_TAG(HDRP(NEXT_BLOCK(bp)));
        next = coalesce(next);
        SET_TAG(HDRP(next));
        CLR_ALLOC(HDRP(next));
        CLR_TAG(HDRP(NEXT_BLOCK(next)));

        INSERT_LIST(next, free_head);
    }
    SET_TAG(HDRP(NEXT_BLOCK(bp)));
    return bp;
}
```

해당 함수는 malloc에서 call되며 parameter로 전달된 free block pointer에 해당

크기만큼을 할당해준다. 이때, 할당할 사이즈와 block size의 크기가 최소 블럭 사이즈 이상 차이난다면, 해당 블럭은 split하여 추가적인 free list에 추가해준다.

또한 A에서 설명한 바와 같이 split을 하는 경우에 특정 블럭 사이즈보다 큰 블럭을 할당한다면 split할 블럭과 swap하여 우측에 할당하게 구현한다. split된 free block은 coalesce를 진행하고 free list에 insert 해준다.

- **extend_heap**

```
static inline 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 */
    SET_SIZE(HDRP(bp), size); /* Free block header */
    CLR_ALLOC(HDRP(bp));
    PUT(FTRP(bp), PACK(size, 0));          /* Free block footer */
    PUT(HDRP(NEXT_BLOCK(bp)), PACK(0, 1)); /* New epilogue header */
    heap_tail = HDRP(NEXT_BLOCK(bp));

    /* Coalesce if the previous block was free */

    return coalesce(bp);
}
```

해당 함수는 heap을 늘려주는 기능을 수행한다. 요청한 사이즈를 alignment을 고려하여 heap size를 늘려준다. 이후 기존의 epliogue block을 초기화하고 heap의 맨뒤에 다시 할당해준다. 이후 늘어난 heap의 free block과 이전 block의 coalesce를 진행하고 해당 free block의 위치를 반환해준다.

- **coalesce**

```
static inline void *coalesce(void *bp)
{

    size_t prev_alloc = GET_TAG(HDRP(bp));
    size_t next_alloc = GET_ALLOC(HDRP(NEXT_BLOCK(bp)));
    size_t size = GET_SIZE(HDRP(bp));

    if (prev_alloc && next_alloc)
    { /* Case 1 */
    }
    else if (prev_alloc && !next_alloc)
    { /* Case 2 */
        ERASE(NEXT_BLOCK(bp));
        size += GET_SIZE(HDRP(NEXT_BLOCK(bp)));
        SET_SIZE(HDRP(bp), size);
        PUT(FTRP(bp), PACK(size, 0));
    }
    else if (!prev_alloc && next_alloc)
    { /* Case 3 */
        ERASE(PREV_BLOCK(bp));
        size += GET_SIZE(HDRP(PREV_BLOCK(bp)));
        PUT(FTRP(bp), PACK(size, 0));
        SET_SIZE(HDRP(PREV_BLOCK(bp)), size);
        bp = PREV_BLOCK(bp);
    }
    else
    { /* Case 4 */
        ERASE(PREV_BLOCK(bp));
        ERASE(NEXT_BLOCK(bp));
        size += GET_SIZE(HDRP(PREV_BLOCK(bp))) +
GET_SIZE(HDRP(NEXT_BLOCK(bp)));
        SET_SIZE(HDRP(PREV_BLOCK(bp)), size);
        PUT(FTRP(NEXT_BLOCK(bp)), PACK(size, 0));
        bp = PREV_BLOCK(bp);
    }

    return bp;
}
```

해당 함수에서는 free한 블럭의 앙옆 즉 next, prev block의 allocated되어있는지 확인하고 만약 free 되어있다면 해당 블럭과 현재 블럭을 merge해준다. 가능한 경우는 총 4개로 해당 case 모두 따로 처리해준다. 이때, merge된 블럭의 size를 다시 세팅해주고 free list에서 합쳐질 블럭들을 delete해준다.

- **free**

```
void mm_free(void *ptr)
{


    size_t block_size;
    block_size = GET_SIZE(HDRP(ptr));
    CLR_ALLOC(HDRP(ptr));
    SET_SIZE(FTRP(ptr), block_size);

    CLR_TAG(HDRP(NEXT_BLOCK(ptr)));

    ptr = coalesce(ptr);
    CLR_TAG(HDRP(NEXT_BLOCK(ptr)));

    INSERT_LIST(ptr, free_head);

}
```

해당 함수는 기존의 malloc package에 구현되어 있는 free와 동일한 기능을 수행한다. 할당된 블럭의 포인터를 입력으로 받아 해당 블럭의 allocate을 해제시켜주고 앙옆의 free된 블럭들과 coalesce를 진행한다. 이 후 free 된 블럭을 free list에 삽입하고 함수를 종료한다.

- **realloc**

```
void *mm_realloc(void *ptr, size_t size)
{

    void *oldptr = ptr;
    void *newptr;
    size_t copySize, alloc_size;
    size_t prev_size, next_size, sum_size;
    prev_size = 0;
    next_size = 0;
    copySize = BLOCK_SIZE(ptr) - DSIZE;

    if (ptr == NULL)
    {
```

```
        return mm_malloc(size);
    }
    if (size == 0)
    {
        mm_free(ptr);
        return NULL;
    }

    alloc_size = MAX(DSIZE * 2, (ALIGN(size) - size >= WSIZE) ?
ALIGN(size) : (ALIGN(size) + DSIZE));

    if (alloc_size <= BLOCK_SIZE(oldptr))
    {

        return ptr;
    }
    if (FTRP(oldptr) + WSIZE == heap_tail)
    {
        SET_TAG(HDRP(NEXT_BLOCK(oldptr)));
        if ((extend_heap((alloc_size - BLOCK_SIZE(oldptr)) / WSIZE)) ==
NULL)
        {
            return NULL;
        }
        SET_SIZE(HDRP(ptr), alloc_size);
        SET_SIZE(FTRP(ptr), alloc_size);
        SET_TAG(HDRP(NEXT_BLOCK(ptr)));
        return oldptr;
    }

    if (!GET_ALLOC(HDRP(NEXT_BLOCK(ptr))))
        next_size = BLOCK_SIZE(NEXT_BLOCK(ptr));
    if (!next_size && HDRP(NEXT_BLOCK(NEXT_BLOCK(oldptr))) == heap_tail)
    {

        if (alloc_size > next_size + BLOCK_SIZE(oldptr))
        {
            ERASE(NEXT_BLOCK(oldptr));
            if ((extend_heap((alloc_size - BLOCK_SIZE(oldptr) -
next_size) / WSIZE)) == NULL)
            {
                return NULL;
            }
            SET_SIZE(HDRP(ptr), alloc_size);
            SET_SIZE(FTRP(ptr), alloc_size);
            SET_TAG(HDRP(NEXT_BLOCK(ptr)));
            return oldptr;
```

```
        }
    }

    // TODO if append its size
    if (!GET_ALLOC(HDRP(PREV_BLOCK(ptr))))
        prev_size = BLOCK_SIZE(PREV_BLOCK(ptr));
    sum_size = BLOCK_SIZE(ptr) + prev_size + next_size;
    if (alloc_size <= sum_size)
    {
        CLR_ALLOC(HDRP(ptr));
        CLR_TAG(HDRP(NEXT_BLOCK(ptr)));

        newptr = coalesce(oldptr);
        memcpy(newptr, oldptr, copySize);

        if (sum_size >= alloc_size + 2 * DSIZE)
        {
            char *next;
            SET_SIZE(HDRP(newptr), alloc_size);
            SET_SIZE(FTRP(newptr), alloc_size);
            SET_ALLOC(HDRP(newptr));
            PUT(HDRP(NEXT_BLOCK(newptr)), PACK((sum_size - alloc_size),
0));
            PUT(FTRP(NEXT_BLOCK(newptr)), PACK((sum_size - alloc_size),
0));
            SET_TAG(HDRP(NEXT_BLOCK(newptr)));
            next = coalesce(NEXT_BLOCK(newptr));
            INSERT_LIST(next, free_head);
            CLR_TAG(HDRP(NEXT_BLOCK(next)));
        }
        else
        {
            SET_SIZE(HDRP(newptr), sum_size);
            SET_SIZE(FTRP(newptr), sum_size);
            SET_ALLOC(HDRP(newptr));
            SET_TAG(HDRP(NEXT_BLOCK(newptr)));
        }
    }
    else
    {

        newptr = mm_malloc(size);
        if (newptr == NULL)
            return NULL;

        memcpy(newptr, oldptr, copySize + WSIZE);
```

```
        mm_free(oldptr);
    }

    return newptr;
}
```

해당 함수도 마찬가지로 malloc package와 같은 기능을 하도록 구현한다. 다만 realloc의 경우 external fragment의 발생 가능성이 높아 memory utilization을 신경쓰며 여러 예외 케이스를 처리해준다.

첫번째로 기존에 할당된 블럭이 epilogue에 가장 가까운 경우이다. 이때 가능한 상황은 할당된 블럭의 next가 바로 epilogue인 경우와 다음이 쭉 free block인 경우인데 이 상황 모두 기존의 블럭에 추가적으로 필요한 size만큼 extend_heap을 하여 추가적인 free와 malloc없이 fit한 memory size만큼만 사용하게 구현한다.

두번째로 기존의 할당된 사이즈보다 큰 크기를 재할당할 때, 양옆에 free 블럭들이 있다면 기존 할당된 block size와 sum을 구해 해당 블럭들을 merge하여 처리해준다. place와 마찬가지로 merge된 free block에 만약 최소 블럭사이즈만큼 남게된다면 해당 블럭은 split하여 free list에 추가해준다.

위의 두개의 케이스 밖의 경우에는 malloc을 통해 새로운 사이즈의 크기를 할당하고 기존 payload를 복사한뒤 기존 블럭을 free해준다.

3. 구현 결과

- 성능평가

```
Results for mm malloc:
trace  valid  util     ops      secs  Kops
 0       yes   99%    5694  0.000359 15852
 1       yes  100%    5848  0.000345 16965
 2       yes  100%    6648  0.000464 14334
 3       yes  100%    5380  0.000262 20574
 4       yes   99%   14400  0.000241 59751
 5       yes   96%    4800  0.005193   924
 6       yes   95%    4800  0.005085   944
 7       yes   95%   12000  0.021312   563
 8       yes   88%   24000  0.020734  1157
 9       yes   99%   14401  0.000271 53121
10       yes   87%   14401  0.000231 62288
Total          96%  112372  0.054497  2062

Perf index = 58 (util) + 40 (thru) = 98/100
```

해당 결과를 확인해보면 throughput은 만점을 받아 추가적인 best fit을 찾기 위해 linear search time에 추가적인 시간을 줄이진 않았다. 다만 utilization을 향상시키기 위해 trace 파일의 여러가지 테스트케이스를 확인해보며 각각에 대한 예외를 감안하여 malloc package를 구현하였다.