CSED211: Lab 12 (due Dec. 24)

손량(20220323)

Last compiled on: Sunday 24th December, 2023, 12:36

1 개요

C언어에서 동적 메모리 할당을 위해 사용하는 함수인 `malloc`을 구현하고, 효율적인 작동을 위해 메모리 사용과 성능을 최적화해 본다.

2 프로그램의 구현

2.1 Block and Free List Oranization

이 구현에서는 explicit free list에 기반하여 구현하였다. Free block은 첫 번째 word에 크기를 저장하고, 8 byte alignment를 활용하여 마지막 2비트에는 0 대신 `inuse`, `prev_inuse` 비트를 저장하였다. 이들은 각각 현재 블록이 할당 상태인지, 그리고 이전 블록이 할당 상태인지를 나타내는 비트이다. 두 번째, 세 번째 word에는 explicit free list에서 사용할 이전 블록과 다음 블록을 가리키는 포인터를 저장하였다. 마지막 word에는 다시 블록 크기를 저장해 footer 역할을 수행하도록 구현하였다. Allocated block의 경우 첫 번째 블록 이외에 나머지 공간은 필요 없기 때문에 payload로 사용하였고, `inuse` 비트를 적당히 세팅하였다.

힙 공간을 초기화할 때, prologue, epilogue 블록을 사용해 alignment를 맞추면서, coalescing 등에 있어 일종의 경계로 사용되도록 하였다. Free list는 LIFO order에 맞게 리스트에 맨 앞에서 삽입이 일어나도록 하였다.

2.2 `mm_check` – Heap consistency check

힙 메모리의 invariant를 검사하기 위한 목적으로 `mm_check` 함수를 다음과 같이 구현하였다.

```
int mm_check(void) {
  struct header *current;
  word_t *current_block;
  bool found;

  /* Check free list consistency. */
  for (current = freelist_head; current != NULL; current = current->next) {
    if (HEADER_INUSE(current->size_with_flags)) {
      fprintf(stderr, "%p: Free list contains an allocated block.\n",
          current);
      return 0;
    }
  }
```

```
current_block = (word_t *)mem_heap_lo() + 3;
while (current_block < (word_t *)mem_heap_hi() - 1) {
  /* Check if the block has nonzero size. */
  if (HEADER_SIZE(DEREF(current_block)) == 0) {
    fprintf(stderr, "%p: Block with size zero\n", current_block);
    return 0;
  }

  /* Check if header and footer size are consistent. */
  if (!HEADER_INUSE(DEREF(current_block)) &&
      HEADER_SIZE(DEREF(current_block)) != DEREF(FOOTER(current_block))) {
    fprintf(stderr,
            "%p: Header and footer size of free block is inconsistent.\n",
            current_block);
    return 0;
  }

  /* Check if coalescing is correct. */
  if (!HEADER_INUSE(DEREF(current_block)) &&
      !HEADER_INUSE(DEREF(NEXT_BLOCK(current_block)))) {
    fprintf(stderr, "%p: Two consecutive free blocks.\n", current_block);
    return 0;
  }

  /* Check if inuse and prev_inuse bits are consistent. */
  if (HEADER_INUSE(DEREF(current_block)) !=
      HEADER_PREVINUSE(DEREF(NEXT_BLOCK(current_block)))) {
    fprintf(stderr,
            "%p: The inuse bit of current block and prev_inuse bit of the "
            "next block is inconsistent.\n",
            current_block);
    return 0;
  }

  /* Check if the block is in the free list. */
  if (!HEADER_INUSE(DEREF(current_block))) {
    found = false;
    for (current = freelist_head; current != NULL; current =
        current->next) {
      if (current == (struct header *)current_block) {
        found = true;
        break;
      }
    }
    if (!found) {
      fprintf(stderr, "%p: Block not in free list.\n", current_block);
      return 0;
    }
  }
```

```
    current_block = NEXT_BLOCK(current_block);
  }

  return 1;
}
```

주석에서도 설명했듯, `mm_check`에서는 우선 free list를 순회하면서 free list의 모든 block이 free block인지 확인한다. 또한, 힙의 모든 block에 대해 다음을 확인하였다.

- Prologue, epilogue block 이외에 모든 block은 nonzero size를 가져야 한다.
- Free block의 경우 header와 footer가 같은 크기를 가져야 한다.
- Coalescing이 수행되므로 block이 연속되어 있어서는 안 된다.
- `inuse`, `prev_inuse` 비트는 모두 정확한 정보를 가지고 있어야 한다.
- Free block은 free list에 있어야 한다.

2.3 `mm_init` – Initialize Allocator

`mm_init`의 코드는 다음과 같다.

```
int mm_init(void) {
  word_t *prologue_block, *init_block;
  struct header *init_block_header;

  prologue_block = mem_sbrk(4 * WORDSIZE);
  if (prologue_block == (void *)-1)
    return -1;

  prologue_block[0] = 0;
  prologue_block[1] = PACK_SIZE(8, 1, 0);
  prologue_block[2] = 8;
  prologue_block[3] = PACK_SIZE(0, 1, 1);

  init_block = expand_heap(CHUNKSIZE / WORDSIZE);
  if (init_block == NULL)
    return -1;

  init_block_header = (struct header *)init_block;
  freelist_head = NULL;
  freelist_tail = NULL;
  list_insert(init_block_header);
  return 0;
}
```

Prologue block과 epilogue block을 alignment에 맞게 할당하고, `expand_heap` 함수를 사용하여 `CHUNKSIZE` 만큼 heap의 크기를 늘려 주었다.

2.4 mm_malloc – Allocate new space

mm_malloc의 코드는 다음과 같다.

```
void *mm_malloc(size_t size) {
  size = handle_exceptional_size(size);

  int newsize = ALIGN(size + WORDSIZE);
  word_t *block = find_best_fit(newsize), *new_block, *next_block;

  if (size == 0)
    return NULL;

  if (block == NULL) {
    block = expand_heap(MAX(newsize, CHUNKSIZE) / WORDSIZE);
    if (block == NULL)
      return NULL;
    block = coalesce_block(block);
    list_insert((struct header *)block);
    block = find_best_fit(newsize);
  }

  list_remove((struct header *)block);
  if (should_split(block, newsize)) {
    new_block = split_block(block, newsize);
    HEADER_PREVINUSE_SET(DEREF(new_block));
    new_block = coalesce_block(new_block);
    list_insert((struct header *)new_block);
  } else {
    next_block = NEXT_BLOCK(block);
    HEADER_PREVINUSE_SET(DEREF(next_block));
  }
  HEADER_INUSE_SET(DEREF(block));

  return (void *)(block + 1);
}
```

새로운 블록을 best-fit 전략에 맞게 find_best_fit 함수를 사용하여 가져왔고, 만약 맞는 block이 없는 경우에는 expand_heap 함수를 사용하여 mem_sbrk로 메모리를 더 받아오도록 하였다. should_split 함수를 사용하여 만약 가져온 블록이 필요한 크기보다 충분히 큰 경우에는 internal fragmentation을 줄이기 위하여 필요한 부분만 잘라내도록 하있다. 잘라내고 남을 블록은 coalescing이 가능한 경우에는 이를 수행하도록 하였다.

2.5 mm_free – Deallocate

mm_free의 코드는 다음과 같다.

```
void mm_free(void *ptr) {
  word_t *block = (word_t *)ptr - 1, *next_block;

  HEADER_INUSE_CLEAR(DEREF(block));
```

```
  *FOOTER(block) = HEADER_SIZE(DEREF(block));

  next_block = NEXT_BLOCK(block);
  HEADER_PREVINUSE_CLEAR(DEREF(next_block));

  block = coalesce_block(block);
  list_insert((struct header *)block);
}
```

주어진 블록의 헤더의 `inuse` 비트를 적당히 설정하고, footer를 복구한 다음, 다음 블록의 `prev_inuse` 비트를 설정하였다. Free의 경우에도 coalescing을 시도한 후 free list에 LIFO order에 맞게 삽입하였다.

2.6 `mm_realloc` – Reallocate

`mm_realloc`의 코드는 다음과 같다.

```
void *mm_realloc(void *ptr, size_t size) {
  int prev_inuse, prev_prev_inuse;
  word_t *block = PAYLOAD_HEADER(ptr), *next_block, *new_block, *prev_block;
  size_t oldsize = HEADER_SIZE(DEREF(block)), newsize = ALIGN(size +
      WORDSIZE),
         next_size, prev_size;
  void *newptr;

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

  prev_inuse = HEADER_PREVINUSE(DEREF(block));
  if (oldsize >= newsize) {
    if (should_split(block, newsize)) {
      next_block = NEXT_BLOCK(block);
      HEADER_PREVINUSE_CLEAR(DEREF(next_block));

      DEREF(block) = PACK_SIZE(newsize, 1, prev_inuse);
      DEREF_FROM_NTH(block, newsize) = PACK_SIZE(oldsize - newsize, 0, 1);
      DEREF_FROM_NTH(block, oldsize - WORDSIZE) = oldsize - newsize;
      new_block = NEXT_BLOCK(block);
      new_block = coalesce_block(new_block);
      list_insert((struct header *)new_block);
    }
    return ptr;
  } else {
    next_block = NEXT_BLOCK(block);
    next_size = HEADER_SIZE(DEREF(next_block));
    if (!HEADER_INUSE(DEREF(next_block)) && oldsize + next_size >= newsize) {
```

```
    list_remove((struct header *)next_block);
    DEREF(block) = PACK_SIZE(oldsize + next_size, 1, prev_inuse);
    if (should_split(block, newsize)) {
      DEREF(block) = PACK_SIZE(newsize, 1, prev_inuse);
      new_block = &DEREF_FROM_NTH(block, newsize);
      DEREF(new_block) = PACK_SIZE(oldsize + next_size - newsize, 0, 1);
      DEREF(FOOTER(new_block)) = oldsize + next_size - newsize;
      new_block = coalesce_block(new_block);
      list_insert((struct header *)new_block);
    } else
      HEADER_PREVINUSE_SET(DEREF(NEXT_BLOCK(block)));
    return ptr;
  } else if (!HEADER_PREVINUSE(DEREF(block)) &&
             oldsize + HEADER_SIZE(DEREF(PREV_BLOCK(block))) >= newsize) {
    prev_size = HEADER_SIZE(DEREF(PREV_BLOCK(block)));
    prev_block = PREV_BLOCK(block);
    prev_prev_inuse = HEADER_PREVINUSE(DEREF(prev_block));
    list_remove((struct header *)prev_block);
    DEREF(prev_block) = PACK_SIZE(prev_size + oldsize, 1, prev_prev_inuse);
    memmove(prev_block + 1, block + 1, oldsize - WORDSIZE);
    if (should_split(prev_block, newsize)) {
      DEREF(prev_block) = PACK_SIZE(newsize, 1, prev_prev_inuse);
      new_block = &DEREF_FROM_NTH(prev_block, newsize);
      DEREF(new_block) = PACK_SIZE(prev_size + oldsize - newsize, 0, 1);
      DEREF(FOOTER(new_block)) = prev_size + oldsize - newsize;
      new_block = coalesce_block(new_block);
      list_insert((struct header *)new_block);
      HEADER_PREVINUSE_CLEAR(DEREF(NEXT_BLOCK(new_block)));
    }
    return prev_block + 1;
  } else if (!HEADER_INUSE(DEREF(next_block)) &&
             !HEADER_PREVINUSE(DEREF(block)) &&
             oldsize + HEADER_SIZE(DEREF(PREV_BLOCK(block))) + next_size >=
                 newsize) {
    prev_size = HEADER_SIZE(DEREF(PREV_BLOCK(block)));
    prev_block = PREV_BLOCK(block);
    prev_prev_inuse = HEADER_PREVINUSE(DEREF(prev_block));
    list_remove((struct header *)prev_block);
    list_remove((struct header *)next_block);
    DEREF(prev_block) =
        PACK_SIZE(prev_size + oldsize + next_size, 1, prev_prev_inuse);
    memmove(prev_block + 1, block + 1, oldsize - WORDSIZE);
    if (should_split(prev_block, newsize)) {
      DEREF(prev_block) = PACK_SIZE(newsize, 1, prev_prev_inuse);
      new_block = &DEREF_FROM_NTH(prev_block, newsize);
      DEREF(new_block) =
          PACK_SIZE(prev_size + oldsize + next_size - newsize, 0, 1);
      DEREF(FOOTER(new_block)) = prev_size + oldsize + next_size - newsize;
      new_block = coalesce_block(new_block);
```

```
        list_insert((struct header *)new_block);
        HEADER_PREVINUSE_CLEAR(DEREF(NEXT_BLOCK(new_block)));
      } else
        HEADER_PREVINUSE_SET(DEREF(NEXT_BLOCK(prev_block)));
      return prev_block + 1;
    } else {
      newptr = mm_malloc(size);
      if (newptr == NULL)
        return NULL;
      memcpy(newptr, ptr, oldsize - WORDSIZE);
      mm_free(ptr);
      return newptr;
    }
  }
}
```

`mm_realloc`이 실행될 때, `ptr`이 `NULL`이거나 `size`가 0인 특별한 경우를 제외하면, 크기가 커지는 경우와 작아지는 경우를 처리해야 한다. 원래 크기보다 크기를 줄이는 경우에는 header와 footer를 적당히 세팅한 다음, 새롭게 생긴 free block에 대해 coalescing을 시도한다. 크기를 늘리는 경우에는 가능하다면 현재 block 양 옆의 free block과 현재 블록을 합치는 것을 시도하고, 불가능하다면 새로운 공간을 `mm_malloc`으로 할당받도록 하였다.

3 결론 및 제언

이로써 간단한 memory allocator를 구현할 수 있었고, 모든 테스트를 통과함을 알 수 있었다. 기회가 된다면 segregated freelist 등을 사용하여 구현한 memory allocator를 더 최적화 하는 것도 좋을 것이다.