Lab Assignment 3: Malloc Lab

공과대학 컴퓨터공학부
2020-14378
윤교준 / Gyojun Youn
youngyojun@snu.ac.kr

구현

Specification

Segregated free list

충분히 좋은 space utilization을 위하여 segregated free list를 사용하였다. 다음과 같이 총 14개의 size classes를 채택하였다. 작은 크기의 size class는 그 분류를 세분화하여, memory loss를 줄이고자 하였다.

```
(0,16], (16,24], (24,32], (32,48], (48,64], (64,128], (128,256], ..., (8192,16384],
(16384,+inf)
```

각 free list는 freed blocks을 양방향으로 연결하는 linked list의 형태로 관리되며, root pointer는 첫 freed block pointer을 가리킨다. 선행 연구를 따라 LIFO policy를 채택하였다.

Best fit policy를 채택하였다. Request size 이상의 가장 작은 freed block을 선택하여 사용한다. 단, find fit 과정에서 너무 많은 시간을 소모하면 안되기에, 각 size class의 linked list에서 첫 64개의 freed blocks만 확인하도록 제한하였다.

Block structure

`blocksize` bytes의 allocatable space를 가진 block의 총 크기는 항상 `blocksize + 8` bytes다. Double-word alignment를 위하여, `blocksize`는 8의 배수인 양의 정수라야 한다.

Freed blocks와 allocated blocks 모두 앞에 8 bytes의 header를 가진다. 이 header에는 `blocksize`와 allocated/freed 정보가 담겨있다. Freed blocks는 예외적으로, header에 previous freed block pointer address를 가지며, next freed block pointer address의 정보가 있는 footer 또한 가진다.

Freed blocks

```
- Structure
    byte:      4     4         s-8          4     4
             +-----+-----+----------------+-----+-----+
             |  h  |  p  |  (free space)  |  f  |  n  |
             +-----+-----+----------------+-----+-----+
             ^     ^     ^                ^     ^     ^
    addr:  bp-8  bp-4   bp             bp+s-8 bp+s-4 bp+s

- Metadata
  - s  := allocatable bytes (aligned in double-word)
```

```
  - bp := block pointer address
  - h  := header metadata (== s)
  - f  := footer metadata (== s ^ FOOT_HSH)
  - p  := previous freed block address
  - n  := next freed block address
```

일반적으로, header의 `h`와 footer의 `f`의 값은 block size `s`로 같게 관리된다. 그러나, allocated block은 footer를 가지지 않기 때문에, user-accessible한 arbitrary memory value가 freed block의 것인지, 아니면 user의 우연한 혹은 의도적인 값인지 구분하는 과정이 필요하다. 이를 위하여, 실제 구현에서는 랜덤한 32 bits 값 `FOOT_HSH`를 이용하여 footer의 값을 xor-hash 하였다. 이는 구현 설명에서 다시 언급한다.

Allocated blocks

```
- Structure
    byte:      4       4                s
            +-----+----------+--------------------+
            |  h  | (unused) |  (allocated space) |
            +-----+----------+--------------------+
            ^     ^          ^                    ^
    addr:  bp-8  bp-4        bp                  bp+s

- Metadata
  - s  := allocatable bytes (aligned in double-word)
  - bp := block pointer address
  - h  := header metadata (== s | 0x1)
```

Allocated blocks는 footer가 없는 freed blocks와 구조가 유사하다. Header의 `h`에는 block size `s`와 `0x1`을 bitwise-or한 값이 적혀있다. Freed blocks의 previous freed block address `f`에 대응되는 위치에는 임의의 값이 들어있을 수 있다. Single-word alignment였다면 이 4 bytes는 allocated space에 포함되었을 것이다.

Macros

Compiler 기술의 발달로 현재에는 좋은 practice인가에 대한 논쟁이 있지만, 충분한 주의를 기울이면 C preprocessor는 짧은 구현과 빠른 성능에 도움이 된다.

Size classes 종류의 수, footer hash value 등, 중요한 상수 값을 macro로 정의하였다.

```
#define NUM_CLASS (14)            /* number of size classes               */
#define MAX_SIZE  (0x3f3f3f30u)   /* maximum bytes allowed to allocate    */
#define FOOT_HSH  (0xcc0708f8u)   /* xor hash value for footer blocksize  */
```

이후, 다양한 함수에서 필요로 하는 기능을 macro로 구현하였다.

```
/* get root pointer of the corresponding class with block size s */
#define GET_ROOT(s)             (((void*)(listptr)) + (get_class(s) * WSIZE))

// 중략

/* the footer address where the block size is stored */
#define FOOT_SIZE_PTR(bp)       (((void*)(bp)) + READ_SIZE(bp) - DSIZE)
```

mm_init

```
/*
 * mm_init - Initialize the malloc package.
 */
#ifdef __MM_YGJ_DEBUG__
static int dbg_init(void)
#else   /* __MM_YGJ_DEBUG__ */
int mm_init(void)
#endif  /* __MM_YGJ_DEBUG__ */
{
  /* get NUM_CLASS words for segregated free list headers */
  if ((void*)(-1) == (listptr = mem_sbrk(NUM_CLASS * WSIZE))) {
    /* mem_sbrk error */
    return -1;
  }

  /* initialize header pointers as nullptr */
  for (int i = 0; i < NUM_CLASS; i++) {
    PUT(listptr + (i * WSIZE), 0);
  }

  /* set start/end heap pointer */
  heap_start = heap_end = listptr + (NUM_CLASS * WSIZE);

  /* successfully initialized */
  return 0;
}
```

필수적으로 작성되어야 하는 네 개의 dynamic storage allocator 함수 `mm_init`, `mm_malloc`, `mm_free`, `mm_realloc`는 선언부에 `ifdef` macro가 적용되었다. `-D__MM_YGJ_DEBUG__` compile option을 부여하면 디버깅을 할 수 있다. 기존의 `mm_*` 함수는 `dbg_*` 함수로 치환된다. 디버깅용 wrapper 함수가 함수 호출 과정을 track 한다.

`mm_init`에서는 segregated free list를 위한 14개의 roots of linked lists를 세팅한다.

mm_malloc

먼저, `size`가 0이거나 너무 큰 경우에 대한 처리를 한다.

```
/*
 * mm_malloc - Allocate a block by incrementing the brk pointer.
 *     Always allocate a block whose size is a multiple of the alignment.
 */
#ifdef __MM_YGJ_DEBUG__
static void* dbg_malloc(size_t size)
#else   /* __MM_YGJ_DEBUG__ */
void* mm_malloc(size_t size)
#endif  /* __MM_YGJ_DEBUG__ */
{
  /* trivial case */
  if (!size) return NULL;

  /* too large size */
  if (MAX_SIZE < size) {
    /* out of memory error */
    errno = ENOMEM;
    return NULL;
  }

  // 후략
}
```

Reasonable한 request block size `size`가 주어지면, 추후의 re-allocation을 대비하여 적당히 큰 값으로 바꾼다. `static uint enrich_size(uint bsize)`의 구현에는 몇 가지 실험을 통한 hard case-work가 포함되어 있다. `(0.86 * 2^k, 2^k]` 범위에 해당하는 block size는 `2^k`로 키울 때 space utilization이 좋았다.

```
  /* get enriched size for further re-allocations */
  size = enrich_size(size);
```

```
/*
 * enrich_size - Return more relaxable block size.
 *     It is guaranteed that the return value is aligned and at least bsize.
 */
static uint enrich_size(uint bsize)
{
  if (7048 <= bsize && bsize <= 8192) return 8192;
  if (3524 <= bsize && bsize <= 4096) return 4096;
  if (1760 <= bsize && bsize <= 2048) return 2048;
  if (880  <= bsize && bsize <= 1024) return 1024;
  if (440  <= bsize && bsize <= 512 ) return 512;
  if (224  <= bsize && bsize <= 256 ) return 256;
  return ALIGN(bsize);
}
```

Best fit policy의 `find_fit`으로 적합한 freed block을 찾는다. 이 block의 space가 아주 충분히 크다면, 필요한 만큼만 잘라서 사용하고, unused space로 새로운 freed block을 만들어둔다. 이 과정은 `split_block` 함수가 처리한다.

```
  /* find the best fit for given block size */
  void *bp = find_fit(size);

  /* if found, then return */
  if (bp) {
    /* shrink if possible */
    if (split_block(bp, size)) return bp;

    /* cannot shrink */

    /* flag allocated in header */
    FLAG_ALLOC(HEAD_SIZE_PTR(bp));

    return bp;
  }
```

적합한 freed block이 없다면, heap 영역의 확장이 불가피하다. `extend_heap`으로 heap의 확장과 동시에 새로운 allocatable block을 생성하여 반환한다.

```
void* mm_malloc(size_t size)
{
  // 전략

  /* get a new freed block by extending the heap */
  bp = extend_heap(size);

  /* extending heap failed */
  if (NULL == bp) {
    /* out of memory error */
    errno = ENOMEM;
    return NULL;
  }

  return bp;
}
```

mm_free

주어진 allocated block을 freed block으로 바꾼 후, 인접한 freed blocks와 합치는 작업을 수행한 후, 적합한 free list에 삽입한다.

```
/*
 * mm_free - Freeing a block does nothing.
 */
#ifdef __MM_YGJ_DEBUG__
static void dbg_free(void *ptr)
#else   /* __MM_YGJ_DEBUG__ */
void mm_free(void *ptr)
#endif  /* __MM_YGJ_DEBUG__ */
{
  /* trivial case */
  if (NULL == ptr) return;

  /* valid (legally allocated) ptr is given */

  /* write freed info at the block */
  write_freed_info(ptr, READ_SIZE(ptr));

  /* coalesce about the block */
  ptr = coal(ptr);

  /* report new freed block */
  report_fb(ptr);
}
```

coal

교과서의 coalesce 구현은 두 개의 freed blocks을 합쳐주는 `merge_fb` 함수를 활용하면 깔끔하게 작성될 수 있다.

```
/*
 * coal - Coalesce adjacent free blocks,
 *    and return the block pointer of (merged) freed block.
 *    bp must be a freed-normalized valid block pointer.
 */
static void* coal(void *bp)
{
  /* get adjacent free blocks */
  void *lo_bp = get_lower_fb(bp);
  void *up_bp = get_upper_fb(bp);

  if (up_bp) {
    ll_pop(up_bp);
    merge_fb(bp, up_bp);      /* == bp    */
  }

  if (lo_bp) {
```

```
    ll_pop(lo_bp);
    bp = merge_fb(lo_bp, bp); /* == lo_bp */
  }

  return bp;
}
```

`get_lower_fb`와 `get_upper_fb`는 상하로 인접한 freed blocks을 반환한다. 여기서, lower adjacent block이 freed인지를 알 수 없기 때문에 `get_lower_fb`의 구현에 깊은 주의가 필요하다.

```
/*
 * get_lower_fb - Get block pointer of the lower freed block if exists.
 *    If not, then return nullptr.
 *    bp must be a valid block pointer.
 */
static void* get_lower_fb(void *bp)
{
  /* check the validity of the footer of lower freed block */
  if (bp < heap_start + QSIZE) return NULL;

  return valid_foot_bp(bp - QSIZE);
}
```

`static void* valid_foot_bp(void *ptr)`는 `ptr`가 valid한 freed block의 footer를 가리키고 있는지를 strict하게 검사한다. 이 과정에는 `FOOT_HSH` xor-hash 과정이 포함되어 있다.

mm_realloc

Trivial cases를 먼저 처리한다.

```
/*
 * mm_realloc - Implemented simply in terms of mm_malloc and mm_free.
 */
#ifdef __MM_YGJ_DEBUG__
static void* dbg_realloc(void *ptr, size_t size)
#else   /* __MM_YGJ_DEBUG__ */
void* mm_realloc(void *ptr, size_t size)
#endif  /* __MM_YGJ_DEBUG__ */
{
  /* equivalent to mm_malloc(size) */
  if (NULL == ptr) {
    return mm_malloc(size);
  }

  /* equivalent to mm_free(size) */
  if (!size) {
    mm_free(ptr);
    return NULL;
  }
```

```
  // 후략
}
```

먼저, 기존의 allocated block space를 활용하여 required block size를 만족하도록 shrink할 수 있는지 판단한다.

```
  /* original block size */
  uint bsize = READ_SIZE(ptr);

  /* enriched required size */
  uint rich_size = enrich_size(size);

  /* shrink if possible */
  if (split_block(ptr, rich_size)) return ptr

  /*
   * the block has enough space
   * but cannot shrink for a new freed block
   */
  if (size <= bsize) {
    /* nothing to do */
    return ptr;
  }
```

이제, 현재의 block space가 너무 작은 상황에 놓였다. 인접한 freed blocks와 merge하여 그 크기를 최대한 키워본다. 이 작업으로 충분하다면, 이 새로운 block으로 re-allocation을 재시도한다.

```
  /* try to extend the block up/down-wards */
  ptr = extend_block(ptr);

  /* re-compute block size */
  bsize = READ_SIZE(ptr);

  /*
   * if the block space becomes large enough,
   * then try to re-allocate again
   */
  if (size <= bsize) {
    return mm_realloc(ptr, size);
  }
```

Careful analysis를 통하여, infinity loop가 발생하지 않으며, 이 재귀 과정이 최대 한 번만 일어남을 증명할 수 있다.

이로도 부족하다면, 새로운 영역을 allocate한 후, 기존의 block을 free한다.

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

```
  /* allocate new block */
  void *bp = mm_malloc(size);

  /* malloc; out of memory error */
  if (NULL == bp) return NULL;

  /* copy data; bsize < size */
  memcpy(bp, ptr, bsize);

  /* free the old block */
  mm_free(ptr);

  return bp;
}
```

find_fit

Best fit policy를 유지하되, 각 linked list에 대하여 최대 64개의 freed blocks만 검사한다. Required block size 이상이면서, 그 중 최소 size의 freed block을 선택하여 반환한다.

```
/*
 * find_fit - Find appropriate freed block
 *    whose block size is at least bsize.
 *    The returned block will be removed from the linked list,
 *    and ready to be allocated.
 *    If not found, then return nullptr.
 */
static void* find_fit(uint bsize)
{
  /* root of the linked list */
  void *rt = GET_ROOT(bsize);

  /* best block pointer to return */
  void *ret_bp = NULL;

  /* block size of ret_bp; find the minimum */
  uint ret_bsize = (uint)(-1);

  /* traversal each linked list */
  while (rt < heap_start) {
    /* freed block pointer in the linked list rt */
    void *bp = (void*)GET(rt);

    /* maximum number of moves */
    uint left_cnt = 64;

    while (bp && left_cnt) {
      uint nw_bsize = READ_SIZE(bp);
```

```
      if (bsize <= nw_bsize && nw_bsize < ret_bsize) {
        ret_bp = bp;
        ret_bsize = nw_bsize;
      }

      bp = READ_NEXT(bp);
      left_cnt -= 1;
    }

    /* move to the next linked list */
    rt += WSIZE;
  }

  /* if not found, then return nullptr */
  if (NULL == ret_bp) return NULL;

  /* remove the found block pointer from the linked list */
  ll_pop(ret_bp);

  /*
   * remove footer metadata;
   * it must be done in order to prevent memory hack
   */
  remove_footer(ret_bp);

  return ret_bp;
}
```

mm_check

Heap 영역의 모든 blocks을 순회하면서 모든 가정이 성립하는지 strict하게 검사한다.

- `listptr`과 `heap_start`, `heap_end`의 global variables가 `mem_heap_lo()`, `mem_heap_hi()`의 정보와 일치하는가.
- 단일 block의 구조가 specification과 일치하는가.
- 두 개의 인접한 freed blocks가 존재하지 않는가.
- Freed blocks의 previous / next freed block pointer address가 정확한가.
- 모든 freed blocks가 대응되는 linked list 안에 들어있는가.
- 모든 blocks의 영역이 서로 겹치지 않고 unused space가 없도록 완전하게 붙어있는가.

구현은 대략 200줄이다.

```
/*
 * mm_check - Check every validity and assumptions.
 *    If succeeded, then return 0.
 *    If failed, then assertion failed error occurs.
 */
int mm_check(void)
{
```

```
#ifdef __MM_YGJ_DEBUG__

  /* check the validity of global variables */

  // 중략

  /*
   * check all blocks in heap area
   */

  void *cur_ptr = heap_start; /* current block start pointer */
  uint is_prev_freed = 0;     /* is the previous block freed */

  while (cur_ptr < heap_end) {
    /* get block pointer */
    void *bp = cur_ptr + DSIZE;

    // 중략

    cur_ptr = bp + bsize;
  }

  fprintf(stderr, "[mm_check] end loop blocks\n");
  fprintf(stderr, "[mm_check] total; cnt_alloc=%u, cnt_freed=%u\n", cnt_alloc,
cnt_freed);

  assert(cur_ptr == heap_end);  /* loop ends well */

  /*
   * traversal along the linked lists
   */

  while (root_ptr < heap_start) {
    uint cnt_cur = 0; /* number of blocks in the current linked list */
    void *bp = (void*)GET(root_ptr); /* freed block pointer */

    fprintf(stderr, "[mm_check] current root %p of id=%u\n", root_ptr, class_id);

    /* check that it meets with flag_con */
    assert((NULL != bp) == ((flag_con >> class_id) & 0x1));

    while (bp) {
      // 중략

      bp = READ_NEXT(bp);
    }

    fprintf(stderr, "[mm_check] linked list %p of id=%u has %u freed blocks\n", root_ptr,
class_id, cnt_cur);

    // 중략
  }
```

```
  fprintf(stderr, "[mm_check] end loop linked lists\n");

  assert(root_ptr == heap_start); /* loop ends well */

  assert(cnt_freed == cnt_llfb);  /* every freed block is in the linked lists */

  fprintf(stderr, "[mm_check] done\n");

#endif  /* __MM_YGJ_DEBUG__ */

  return 0;
}
```

어려웠던 점

방대한 크기의 heap 영역 위에서 포인터를 가지고 기능을 구현해야 하기 때문에, 디버깅 과정에 있어 많은 노력이 필요했다. 특히, 기능 간의 연쇄적인 과정이 상당히 많기 때문에, 구현의 모든 부분이 정확하지 않다면, 어느 특정 부분에서 오류가 있는지 알아차리기 쉽지 않았다. 충분한 code review를 통해 명료하고 정확한 코드를 작성할 수 있었다.

기능을 직접 구현하기 전까지는 explicit과 segregated 중 어떤 방식이 throughput과 space utilization 측면에서 얼마나 득실이 있는지 가늠하기 어려웠다. Macro로 정의한 constant values를 조작하면서 다양한 policy와 scenario에 대한 실험을 수행하였다. 제출본의 결과는 그 실험의 결과로 얻어진 것이다.

병적인 allocation scenario에 대해서, space utilization이 효과적으로 좋을 것이라고 확신이 듦과 동시에 구현이 간결한 방법을 떠올릴 수 없었다. GNU libc의 구현체는 small-size과 large-size 간의 allocation policy를 구분하였으며, 충분한 practical study을 적용하여 상당히 복잡하였다. Memory 효율에 민감한 환경을 위한 여러 독자적인 구현체와 GNU libc의 것 모두 공통적으로 단일 page에 작은 크기의 blocks을 욱여넣도록 되어 있는데, 이것에 대한 insight를 가지지 못했다. 그 결과로 제출본에도 `mem_pagesize` 함수를 활용하지 못하였다.

새롭게 배운 점

Implicit과 explict, segregated, 혹은 find fit나 free list 관리 등의 policy 간의 utilization 효율 차이가 극명함을 직접 확인하였다. 특히, 이들을 어떻게 조합하냐에 따라서 시너지가 다르게 생겼고, allocation scenario에 따라 그 결과가 다르기 때문에, 신중하고 충분한 practical study가 필요함을 깨달았다.

실무에 사용되는 `malloc` 구현체는 `pagesize` 등 메모리 구조를 활용한 "과학적 접근", 그리고 'Make the common case fast'의 idea에 입각하여 자주 발생하는 allocation scenario에서 효율이 좋도록 하는 "ad-hoc한 접근"이 조화를 이루어 적용되어 있음을 알게 되었다.