code/vm/malloc/mm.c

```
1  /* Basic constants and macros */
2  #define WSIZE       4       /* Word and header/footer size (bytes) */
3  #define DSIZE       8       /* Double word size (bytes) */
4  #define CHUNKSIZE  (1<<12)  /* Extend heap by this amount (bytes) */
5
6  #define MAX(x, y) ((x) > (y)? (x) : (y))
7
8  /* Pack a size and allocated bit into a word */
9  #define PACK(size, alloc)  ((size) | (alloc))
10
11 /* Read and write a word at address p */
12 #define GET(p)       (*(unsigned int *)(p))
13 #define PUT(p, val)  (*(unsigned int *)(p) = (val))
14
15 /* Read the size and allocated fields from address p */
16 #define GET_SIZE(p)  (GET(p) & ~0x7)
17 #define GET_ALLOC(p) (GET(p) & 0x1)
18
19 /* Given block ptr bp, compute address of its header and footer */
20 #define HDRP(bp)       ((char *)(bp) - WSIZE)
21 #define FTRP(bp)       ((char *)(bp) + GET_SIZE(HDRP(bp)) - DSIZE)
22
23 /* Given block ptr bp, compute address of next and previous blocks */
24 #define NEXT_BLKP(bp)  ((char *)(bp) + GET_SIZE(((char *)(bp) - WSIZE)))
25 #define PREV_BLKP(bp)  ((char *)(bp) - GET_SIZE(((char *)(bp) - DSIZE)))
```

code/vm/malloc/mm.c

Figure 9.43 **Basic constants and macros for manipulating the free list.**

Creating the Initial Free List

Before calling `mm_malloc` or `mm_free`, the application must initialize the heap by calling the `mm_init` function (Figure 9.44).

The `mm_init` function gets four words from the memory system and initializes them to create the empty free list (lines 4–10). It then calls the `extend_heap` function (Figure 9.45), which extends the heap by CHUNKSIZE bytes and creates the initial free block. At this point, the allocator is initialized and ready to accept allocate and free requests from the application.

The `extend_heap` function is invoked in two different circumstances: (1) when the heap is initialized and (2) when `mm_malloc` is unable to find a suitable fit. To maintain alignment, `extend_heap` rounds up the requested size to the nearest

code/vm/malloc/mm.c

```
1  int mm_init(void)
2  {
3      /* Create the initial empty heap */
4      if ((heap_listp = mem_sbrk(4*WSIZE)) == (void *)-1)
5          return -1;
6      PUT(heap_listp, 0);                          /* Alignment padding */
7      PUT(heap_listp + (1*WSIZE), PACK(DSIZE, 1)); /* Prologue header */
8      PUT(heap_listp + (2*WSIZE), PACK(DSIZE, 1)); /* Prologue footer */
9      PUT(heap_listp + (3*WSIZE), PACK(0, 1));     /* Epilogue header */
10     heap_listp += (2*WSIZE);
11
12     /* Extend the empty heap with a free block of CHUNKSIZE bytes */
13     if (extend_heap(CHUNKSIZE/WSIZE) == NULL)
14         return -1;
15     return 0;
16 }
```

code/vm/malloc/mm.c

Figure 9.44 `mm_init` **creates a heap with an initial free block.**

code/vm/malloc/mm.c

```
1  static void *extend_heap(size_t words)
2  {
3      char *bp;
4      size_t size;
5
6      /* Allocate an even number of words to maintain alignment */
7      size = (words % 2) ? (words+1) * WSIZE : words * WSIZE;
8      if ((long)(bp = mem_sbrk(size)) == -1)
9          return NULL;
10
11     /* Initialize free block header/footer and the epilogue header */
12     PUT(HDRP(bp), PACK(size, 0));         /* Free block header */
13     PUT(FTRP(bp), PACK(size, 0));         /* Free block footer */
14     PUT(HDRP(NEXT_BLKP(bp)), PACK(0, 1)); /* New epilogue header */
15
16     /* Coalesce if the previous block was free */
17     return coalesce(bp);
18 }
```

code/vm/malloc/mm.c

Figure 9.45 `extend_heap` **extends the heap with a new free block.**

multiple of 2 words (8 bytes) and then requests the additional heap space from the memory system (lines 7–9).

The remainder of the `extend_heap` function (lines 12–17) is somewhat subtle. The heap begins on a double-word aligned boundary, and every call to `extend_heap` returns a block whose size is an integral number of double words. Thus, every call to `mem_sbrk` returns a double-word aligned chunk of memory immediately following the header of the epilogue block. This header becomes the header of the new free block (line 12), and the last word of the chunk becomes the new epilogue block header (line 14). Finally, in the likely case that the previous heap was terminated by a free block, we call the `coalesce` function to merge the two free blocks and return the block pointer of the merged blocks (line 17).

Freeing and Coalescing Blocks

An application frees a previously allocated block by calling the `mm_free` function (Figure 9.46), which frees the requested block (`bp`) and then merges adjacent free blocks using the boundary-tags coalescing technique described in Section 9.9.11.

The code in the `coalesce` helper function is a straightforward implementation of the four cases outlined in Figure 9.40. There is one somewhat subtle aspect. The free list format we have chosen—with its prologue and epilogue blocks that are always marked as allocated—allows us to ignore the potentially troublesome edge conditions where the requested block `bp` is at the beginning or end of the heap. Without these special blocks, the code would be messier, more error prone, and slower because we would have to check for these rare edge conditions on each and every free request.

Allocating Blocks

An application requests a block of `size` bytes of memory by calling the `mm_malloc` function (Figure 9.47). After checking for spurious requests, the allocator must adjust the requested block size to allow room for the header and the footer, and to satisfy the double-word alignment requirement. Lines 12–13 enforce the minimum block size of 16 bytes: 8 bytes to satisfy the alignment requirement and 8 more bytes for the overhead of the header and footer. For requests over 8 bytes (line 15), the general rule is to add in the overhead bytes and then round up to the nearest multiple of 8.

Once the allocator has adjusted the requested size, it searches the free list for a suitable free block (line 18). If there is a fit, then the allocator places the requested block and optionally splits the excess (line 19) and then returns the address of the newly allocated block.

If the allocator cannot find a fit, it extends the heap with a new free block (lines 24–26), places the requested block in the new free block, optionally splitting the block (line 27), and then returns a pointer to the newly allocated block.

code/vm/malloc/mm.c

```
1  void mm_free(void *bp)
2  {
3      size_t size = GET_SIZE(HDRP(bp));
4
5      PUT(HDRP(bp), PACK(size, 0));
6      PUT(FTRP(bp), PACK(size, 0));
7      coalesce(bp);
8  }
9
10 static void *coalesce(void *bp)
11 {
12     size_t prev_alloc = GET_ALLOC(FTRP(PREV_BLKP(bp)));
13     size_t next_alloc = GET_ALLOC(HDRP(NEXT_BLKP(bp)));
14     size_t size = GET_SIZE(HDRP(bp));
15
16     if (prev_alloc && next_alloc) {            /* Case 1 */
17         return bp;
18     }
19
20     else if (prev_alloc && !next_alloc) {      /* Case 2 */
21         size += GET_SIZE(HDRP(NEXT_BLKP(bp)));
22         PUT(HDRP(bp), PACK(size, 0));
23         PUT(FTRP(bp), PACK(size,0));
24     }
25
26     else if (!prev_alloc && next_alloc) {      /* Case 3 */
27         size += GET_SIZE(HDRP(PREV_BLKP(bp)));
28         PUT(FTRP(bp), PACK(size, 0));
29         PUT(HDRP(PREV_BLKP(bp)), PACK(size, 0));
30         bp = PREV_BLKP(bp);
31     }
32
33     else {                                     /* Case 4 */
34         size += GET_SIZE(HDRP(PREV_BLKP(bp))) +
35             GET_SIZE(FTRP(NEXT_BLKP(bp)));
36         PUT(HDRP(PREV_BLKP(bp)), PACK(size, 0));
37         PUT(FTRP(NEXT_BLKP(bp)), PACK(size, 0));
38         bp = PREV_BLKP(bp);
39     }
40     return bp;
41 }
```

code/vm/malloc/mm.c

Figure 9.46 `mm_free` **frees a block and uses boundary-tag coalescing to merge it with any adjacent free blocks in constant time.**

code/vm/malloc/mm.c

```
1  void *mm_malloc(size_t size)
2  {
3      size_t asize;      /* Adjusted block size */
4      size_t extendsize; /* Amount to extend heap if no fit */
5      char *bp;
6
7      /* Ignore spurious requests */
8      if (size == 0)
9          return NULL;
10
11     /* Adjust block size to include overhead and alignment reqs. */
12     if (size <= DSIZE)
13         asize = 2*DSIZE;
14     else
15         asize = DSIZE * ((size + (DSIZE) + (DSIZE-1)) / DSIZE);
16
17     /* Search the free list for a fit */
18     if ((bp = find_fit(asize)) != NULL) {
19         place(bp, asize);
20         return bp;
21     }
22
23     /* No fit found. Get more memory and place the block */
24     extendsize = MAX(asize,CHUNKSIZE);
25     if ((bp = extend_heap(extendsize/WSIZE)) == NULL)
26         return NULL;
27     place(bp, asize);
28     return bp;
29 }
```

code/vm/malloc/mm.c

Figure 9.47 `mm_malloc` **allocates a block from the free list.**

Practice Problem 9.8 (solution page 884)

Implement a `find_fit` function for the simple allocator described in Section 9.9.12.

```
static void *find_fit(size_t asize)
```

Your solution should perform a first-fit search of the implicit free list.

Practice Problem 9.9 (solution page 884)

Implement a `place` function for the example allocator.