1 Writing a dynamic memory allocator package simulator

Solution The created simulator implements basic functions for a dynamic memory allocator (DMA). Three main function is based on standard DMA GNU Malloc library for C language:

- `void *malloc(size_t size);`
- `void *realloc(void *p, size_t size);`
- `void free(void *p);`

The implementation is based on the explicit free list structure with boundary tags for coalescing of the adjacent free objects in the heap. The source code of dynamic package allocator is shown on listing 1.

```
1  /*
2   * The simple memory allocator package implementing basic malloc function:
3   * malloc(size_t size),
4   * free(void *p),
5   * realloc(void *p, size_t size).
6   *
7   * It uses a memory alignment by 8 for the requested blocks.
8   * Structure of the block is an explicit list with block boundary headers for
9   * adjacent free block coalescing. Also each free block store two pointers to
        another free blocks
10  * thus reducing malloc function search time to O(free blocks), instead of
        O(all blocks)
11  * with implicit list structure.
12  */
13 #include <stdio.h>
14 #include <stdlib.h>
15 #include <stdint.h>
16 #include <assert.h>
17 #include <unistd.h>
18 #include <string.h>
19
20 #include "mm.h"
21 #include "memlib.h"
22
23 /* single word (4) or double word (8) alignment */
24 #define ALIGNMENT 8
25
26 /* rounds up to the nearest multiple of ALIGNMENT */
27 #define ALIGN(size) (((size) + (ALIGNMENT - 1)) & ~0x7)
28
29 #define SIZE_T_SIZE (ALIGN(sizeof(size_t)))
30 #define PTR_SIZE (ALIGN(sizeof(uintptr_t)))
31
32 #define CHUNKSIZE (1 << 12)
33
34 #define MAX(x, y) ((x) >= (y) ? (x) : (y))
35
36 #define GET(p) (*(size_t *)(p))
37 #define PUT(p, val) (*(size_t *)(p) = (val))
38
39 #define GETP(p) (*(void **)(p))
40 #define SETP(p, val) (*(void **)(p) = (val))
41
42 #define GET_SIZE(p) (GET(p) & ~0b111)
43 #define GET_ALLOC(p) (GET(p) & 0b01)
44
45 #define GET_PREV_ALLOC(p) ((GET(p) >> 1) & 0b01)
46 #define UNSET_PREV_ALLOC(p) (PUT((p), GET(p) & ~0b10))
47 #define SET_PREV_ALLOC(p) (PUT((p), GET(p) | 0b10))
48
49 #define HDRP(bp) ((char *)(bp)-SIZE_T_SIZE)
50 #define FTRP(bp) ((char *)(bp) + GET_SIZE(HDRP(bp)) - 2 * SIZE_T_SIZE)
51
52 #define PREV_FREEP(bp) ((char *)(bp))
53 #define NEXT_FREEP(bp) ((char *)(bp) + PTR_SIZE)
54
55 #define NEXT_BLKP(bp) ((char *)(bp) + GET_SIZE(HDRP(bp)))
56 #define PREV_BLKP(bp) ((char *)(bp)-GET_SIZE((char *)(bp)-2 * SIZE_T_SIZE))
```

Listing 1 (Cont.): .

```
57
58  #define PACK_HDR(size, prev_alloc, alloc) ((size) | (prev_alloc << 1) |
        (alloc))
59  #define PACK_FTR(size) (size)
60
61  static void *heap_listp = NULL;
62  static void *free_list_head = NULL;
63
64  static void *find_fit(size_t asize);
65  static void place(void *bp, size_t asize);
66  static void *extend_heap(size_t words);
67  static void *coalesce(void *bp);
68  static void add_to_free_list(void *bp);
69  static void remove_from_free_list(void *bp);
70
71  #ifdef DEBUG
72  static void mm_check_heap(char *caller_name);
73  static void check_block(void *bp, char *caller_name);
74  static void check_placed(void *placed, char *caller_name);
75  static void check_free(void *freed, char *caller_name);
76  #endif
77
78  /*
79   * mm_init - Initialize the malloc package.
80   */
81  int mm_init(void)
82  {
83      if ((heap_listp = mem_sbrk(3 * SIZE_T_SIZE)) == (void *)-1)
84      {
85          return -1;
86      }
87
88      PUT(heap_listp, PACK_HDR(2 * SIZE_T_SIZE, 1, 1));
89      heap_listp += SIZE_T_SIZE;
90      PUT(HDRP(NEXT_BLKP(heap_listp)), PACK_HDR(0, 1, 1));
91      free_list_head = NULL;
92  #ifdef DEBUG
93      mm_check_heap("mm_init");
94  #endif
95
96      return 0;
97  }
98
99  /*
100  * mm_malloc - Allocate a block by searching through the explicit free list,
101  requesting additional heap memory if no block was found.
102  */
103 void *mm_malloc(size_t size)
104 {
105     if (size == 0)
106     {
107         return NULL;
108     }
109
110     if (heap_listp == NULL)
111     {
112         mm_init();
113     }
114
115     int asize = MAX(ALIGN(size + SIZE_T_SIZE), 2 * PTR_SIZE + 2 *
            SIZE_T_SIZE);
116     void *bp;
117     if ((bp = find_fit(asize)) != NULL)
118     {
119         place(bp, asize);
120 #ifdef DEBUG
121         mm_check_heap("mm_malloc");
122 #endif
123         return bp;
124     }
125
126     size_t extend_size = MAX(asize, CHUNKSIZE);
127     if ((bp = extend_heap(extend_size)) == NULL)
```

Listing 1 (Cont.): .

```
128     {
129         return NULL;
130     }
131
132     place(bp, asize);
133 #ifdef DEBUG
134     mm_check_heap("mm_malloc");
135 #endif
136
137     return bp;
138 }
139
140 /*
141  * mm_free - Free a block by removing it from the explicit free list
142  and updating the boundary tags.
143  */
144 void mm_free(void *bp)
145 {
146     if (bp == NULL)
147     {
148         return;
149     }
150
151     size_t size = GET_SIZE(HDRP(bp));
152     PUT(HDRP(bp), PACK_HDR(size, GET_PREV_ALLOC(HDRP(bp)), 0));
153     PUT(FTRP(bp), PACK_FTR(size));
154     UNSET_PREV_ALLOC(HDRP(NEXT_BLKP(bp)));
155
156     coalesce(bp);
157 #ifdef DEBUG
158     mm_check_heap("mm_free");
159 #endif
160 }
161
162 /*
163  * mm_realloc - Keep the current block if the requested size is less or equal
164  * than the current block size. Allocate a new block in the heap otherwise.
165  * Th new block size is twice time bigger than the requested to facilitate
        further
166  * block size requsts for this object.
167  */
168 void *mm_realloc(void *bp, size_t size)
169 {
170     if (bp == NULL)
171     {
172         return mm_malloc(size);
173     }
174
175     if (size == 0)
176     {
177         mm_free(bp);
178
179         return NULL;
180     }
181
182     size_t asize = MAX(ALIGN(size + SIZE_T_SIZE), 2 * PTR_SIZE + 2 *
        SIZE_T_SIZE);
183     size_t csize = GET_SIZE(HDRP(bp));
184     if (asize > csize)
185     {
186         void *new_bp = mm_malloc(size * 2);
187         if (new_bp == NULL)
188         {
189             return NULL;
190         }
191
192         size_t copy_size = csize - SIZE_T_SIZE;
193         memcpy(new_bp, bp, copy_size);
194         mm_free(bp);
195 #ifdef DEBUG
196         mm_check_heap("mm_realloc");
197 #endif
198
```

Listing 1 (Cont.): .

```
199             return new_bp;
200         }
201
202         return bp;
203     }
204
205     static void *find_fit(size_t asize)
206     {
207         void *bp;
208         for (bp = free_list_head; bp != NULL; bp = GETP(NEXT_FREEP(bp)))
209         {
210             if (asize <= GET_SIZE(HDRP(bp)))
211             {
212                 return bp;
213             }
214         }
215
216         return NULL;
217     }
218
219     static void place(void *bp, size_t asize)
220     {
221         size_t csize = GET_SIZE(HDRP(bp));
222         if (csize >= asize + 2 * SIZE_T_SIZE + 2 * PTR_SIZE)
223         {
224             PUT(HDRP(bp), PACK_HDR(asize, 1, 1));
225             remove_from_free_list(bp);
226             bp = NEXT_BLKP(bp);
227             PUT(HDRP(bp), PACK_HDR(csize - asize, 1, 0));
228             PUT(FTRP(bp), PACK_FTR(csize - asize));
229             add_to_free_list(bp);
230         }
231         else
232         {
233             PUT(HDRP(bp), PACK_HDR(csize, 1, 1));
234             SET_PREV_ALLOC(HDRP(NEXT_BLKP(bp)));
235             remove_from_free_list(bp);
236         }
237     }
238
239     static void *extend_heap(size_t size)
240     {
241         size_t asize = ALIGN(size);
242         void *bp;
243         if ((bp = mem_sbrk(asize)) == (void *)-1)
244         {
245             return NULL;
246         }
247
248         PUT(HDRP(bp), PACK_HDR(asize, GET_PREV_ALLOC(HDRP(bp)), 0));
249         PUT(FTRP(bp), PACK_FTR(asize));
250         PUT(HDRP(NEXT_BLKP(bp)), PACK_HDR(0, 0, 1));
251
252         return coalesce(bp);
253     }
254
255     static void add_to_free_list(void *bp)
256     {
257         SETP(PREV_FREEP(bp), NULL);
258         SETP(NEXT_FREEP(bp), free_list_head);
259         if (free_list_head != NULL)
260         {
261             SETP(PREV_FREEP(free_list_head), bp);
262         }
263
264         free_list_head = bp;
265     #ifdef DEBUG
266         check_free(bp, "add_to_free_list");
267     #endif
268     }
269
270     static void remove_from_free_list(void *bp)
271     {
```

Listing 1 (Cont.): .

```
272     void *prev = GETP(PREV_FREEP(bp));
273     void *next = GETP(NEXT_FREEP(bp));
274     if (prev != NULL)
275     {
276         SETP(NEXT_FREEP(prev), next);
277     }
278     else
279     {
280         free_list_head = next;
281     }
282
283     if (next != NULL)
284     {
285         SETP(PREV_FREEP(next), prev);
286     }
287 #ifdef DEBUG
288     check_placed(bp, "remove_from_free_list");
289 #endif
290 }
291
292 static void *coalesce(void *bp)
293 {
294     size_t prev_alloc = GET_PREV_ALLOC(HDRP(bp));
295     size_t next_alloc = GET_ALLOC(HDRP(NEXT_BLKP(bp)));
296     size_t size = GET_SIZE(HDRP(bp));
297     if (prev_alloc && !next_alloc)
298     {
299         remove_from_free_list(NEXT_BLKP(bp));
300         size += GET_SIZE(HDRP(NEXT_BLKP(bp)));
301         PUT(HDRP(bp), PACK_HDR(size, 1, 0));
302         PUT(FTRP(bp), PACK_FTR(size));
303     }
304     else if (!prev_alloc && next_alloc)
305     {
306         remove_from_free_list(PREV_BLKP(bp));
307         size += GET_SIZE(HDRP(PREV_BLKP(bp)));
308         PUT(FTRP(bp), PACK_FTR(size));
309         PUT(HDRP(PREV_BLKP(bp)), PACK_HDR(size, 1, 0));
310         bp = PREV_BLKP(bp);
311     }
312     else if (!prev_alloc && !next_alloc)
313     {
314         remove_from_free_list(PREV_BLKP(bp));
315         remove_from_free_list(NEXT_BLKP(bp));
316         size += GET_SIZE(HDRP(PREV_BLKP(bp))) +
317                 GET_SIZE(HDRP(NEXT_BLKP(bp)));
318         PUT(HDRP(PREV_BLKP(bp)), PACK_HDR(size, 1, 0));
319         PUT(FTRP(NEXT_BLKP(bp)), PACK_FTR(size));
320
321         bp = PREV_BLKP(bp);
322     }
323
324     add_to_free_list(bp);
325
326     return bp;
327 }
328
329 #ifdef DEBUG
330 static void mm_check_heap(char *caller_name)
331 {
332     char *bp = heap_listp;
333     if ((GET_SIZE(HDRP(heap_listp)) != 2 * SIZE_T_SIZE) ||
        !GET_ALLOC(HDRP(heap_listp)))
334     {
335         printf("Error %s: Bad prologue header\n", caller_name);
336     }
337
338     for (bp = heap_listp; GET_SIZE(HDRP(bp)) > 0; bp = NEXT_BLKP(bp))
339     {
340         check_block(bp, caller_name);
341     }
342
343     if ((GET_SIZE(HDRP(bp)) != 0) || !(GET_ALLOC(HDRP(bp))))
```

Listing 1 (Cont.): .

```
344     {
345         printf("Error %s: Bad epilogue header\n", caller_name);
346     }
347 }
348
349 static void check_free(void *freed, char *caller_name)
350 {
351     int free_found = 0;
352     char *bp;
353     for (bp = free_list_head; bp != NULL; bp = GETP(NEXT_FREEP(bp)))
354     {
355         if (!GET_ALLOC(HDRP(bp)))
356         {
357             if (bp == freed)
358             {
359                 free_found += 1;
360             }
361         }
362         else
363         {
364             printf("Error %s: allocated in free list\n", caller_name);
365         }
366     }
367
368     if (free_found != 1)
369     {
370         printf("Error %s: freed block not added to free list\n", caller_name);
371     }
372 }
373
374 static void check_placed(void *placed, char *caller_name)
375 {
376     int placed_found = 0;
377     char *bp;
378     for (bp = free_list_head; bp != NULL; bp = GETP(NEXT_FREEP(bp)))
379     {
380         if (!GET_ALLOC(HDRP(bp)))
381         {
382             if (bp == placed)
383             {
384                 placed_found += 1;
385             }
386         }
387         else
388         {
389             printf("Error %s: allocated in free list\n", caller_name);
390         }
391     }
392
393     if (placed_found != 0)
394     {
395         printf("Error %s: placed block in free list\n", caller_name);
396     }
397 }
398
399 static void check_block(void *bp, char *caller_name)
400 {
401     if ((size_t)bp % 8)
402     {
403         printf("Error %s: wrong double word aligned\n", caller_name);
404     }
405
406     if (!GET_ALLOC(HDRP(bp)))
407     {
408         if (GET_SIZE(HDRP(bp)) != GET_SIZE(FTRP(bp)))
409         {
410             printf("Error %s: header does not match footer\n", caller_name);
411         }
412
413         void *prev = GETP(PREV_FREEP(bp));
414         void *next = GETP(NEXT_FREEP(bp));
415         if (prev != NULL && GET_ALLOC(HDRP(prev)))
416         {
```

Listing 1 (Cont.): .

```
417
418             printf("Error %s: prev points to allocated\n", caller_name);
419         }
420
421         if (next != NULL && GET_ALLOC(HDRP(next)))
422         {
423             printf("Error %s: next points to allocated\n", caller_name);
424         }
425     }
426 }
427 #endif
```

Listing 1: .

Results The program was run against the evaluator which computes memory utilization and package throughout. The values are displayed as percentage values of the reference GNU Malloc package. For the most part of the test traces the simulator shows more than 80% memory utilization (util column) and good throughout (Kops column). As explicit free list is not the optimal structure for DMA it is unable to perform well across all kind of traces. Memory utilization lows up to 50% as well as throughput drops by an order of magnitude for traces with binary allocating patterns. And up to 39% for traces with realloc() calls. But overall it has Perf index equal to 80 out of 100. The screenshot of the evaluator run is presented on figure 1.

```
Results for mm malloc:
trace  valid  util     ops      secs  Kops
 0       yes   89%    5694  0.000228 24988
 1       yes   92%    5848  0.000141 41484
 2       yes   94%    6648  0.000216 30803
 3       yes   96%    5380  0.000179 30053
 4       yes   99%   14400  0.000165 87448
 5       yes   87%    4800  0.000273 17581
 6       yes   85%    4800  0.000275 17436
 7       yes   55%   12000  0.002585  4642
 8       yes   50%   24000  0.002661  9018
 9       yes   39%   14401  0.000158 91029
10       yes   43%   14401  0.000115125085
Total          75%  112372  0.006997 16061

Perf index = 45 (util) + 34 (thru) = 80/100
```

Figure 1: Benchmark results