# 시스템 프로그래밍

## 실습 과제(12)(malloc lab - explicit)

**01분반**

malloc lab
 - explicit free list

목차

(1) explicit 알고리즘 설명

implicit free list 의 단점을 보완한 방법이다.
가용 블록들을 명시적 자료구조로 구성한다. 각 가용블록 내에 pred(predecessor)와 succ(successor) 포인터를 포함하는 이중 연결 가용 리스트를 구성한다.
이중 연결 리스트를 사용하면 first fit 할당 시간을 전체 블록 수에 비례하는 것에서 가용 블록의 수에 비례하는 것으로 줄일 수 있다.

할당을 할 때는, 가용 리스트에서 적당한 맞춤의 블록을 찾은 후에 블록을 할당하고, 분할을 수행한다. 최소 블록의 크기는 헤더, 풋터, pred, succ를 포함해서 24바이트가 된다.

아래의 그림과 같은 free block format을 사용한다. prev(pred)와 next(succ)에 대한 포인터를 free block 내에 저장한다.

```
*
* free block
*  +--------+------+------+---------+--------+
*  | header | prev | next | payload | footer |
*  +--------+------+------+---------+--------+
*
```

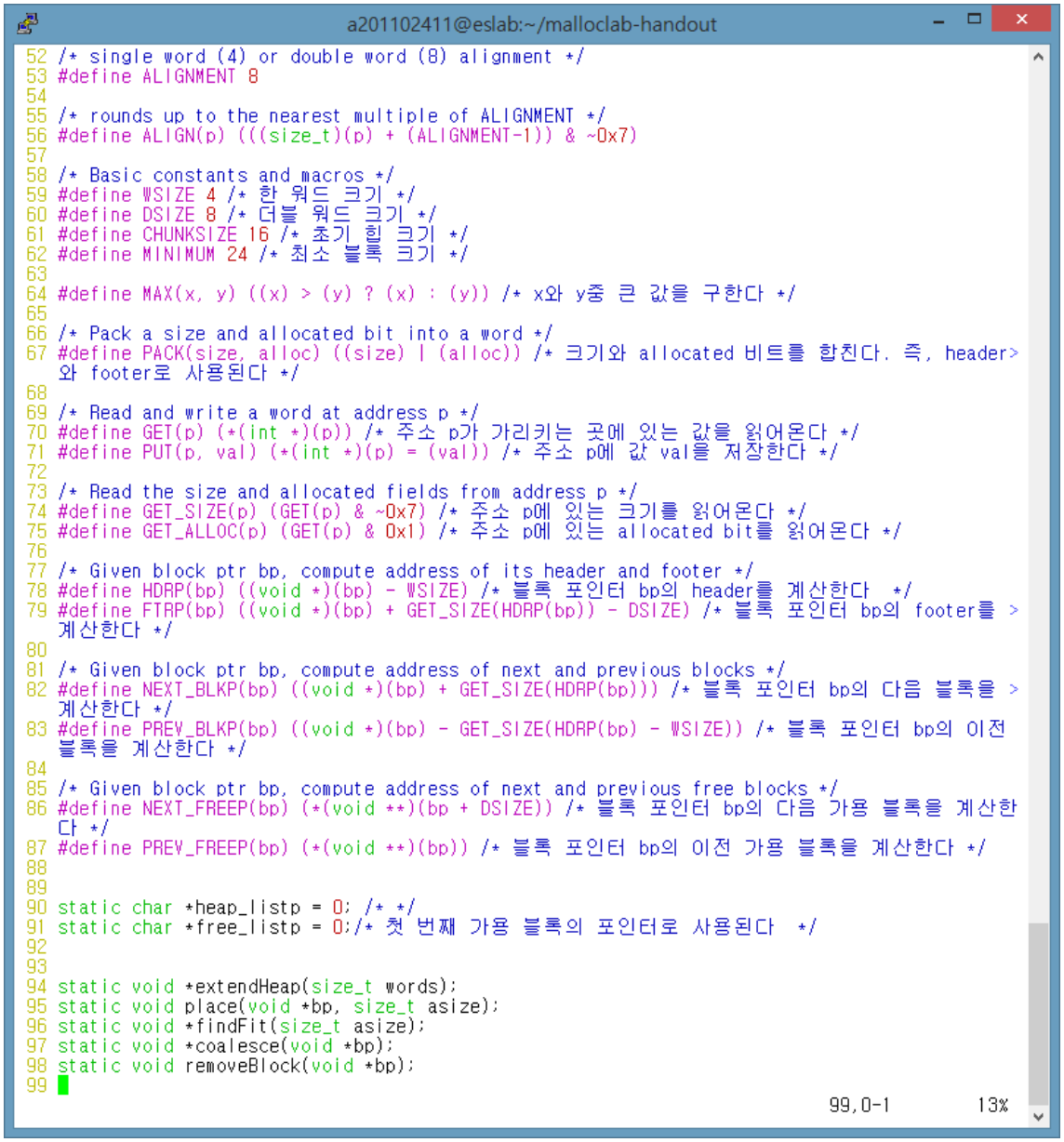
명시적 가용 리스트를 구현하기 위해 이중 연결 리스트를 사용하면 포인터가 인접한 블록을 가리킬 수도 있지만, 멀리 있는 블록들을 가리키면서 섞이는 현상이 발생한다.

가용 리스트들을 관리하는 자료구조는 LIFO(last in first out) 자료구조를 사용한다. 반환하려는 블록을 가용 리스트의 처음에 삽입해서 관리한다.
초기 힙의 상태를 그림으로 나타내면 다음과 같다. 이는 최소 크기 블록으로 할당된 블록을 포함하도록 초기 힙을 구현한다. 그리고 이 초기 최소 크기 블록을 free_listp가 가리키도록 한다.
정적 전역 변수 free_listp 는 항상 가용 리스트의 첫 부분을 가리킨다.
이렇게 구현하면 이 초기화된 최소 크기 블록은 항상 가용 리스트의 제일 마지막 부분이 된다. 이는 할당된 블록으로 표현되어, 가용 리스트를 검색할 때 제일 마지막을 쉽게 찾을 수 있도록 한다.

```
*
* 초기 힙 상태
*
*   padding   hdr  prev next 8byte  ftr  epilogue
* +---------+------+---+---+------+------+-----+
* | padding | 24/1 | 0 | 0 |      | 24/1 | 0/1 |
* +---------+------+---+---+------+------+-----+
*
```

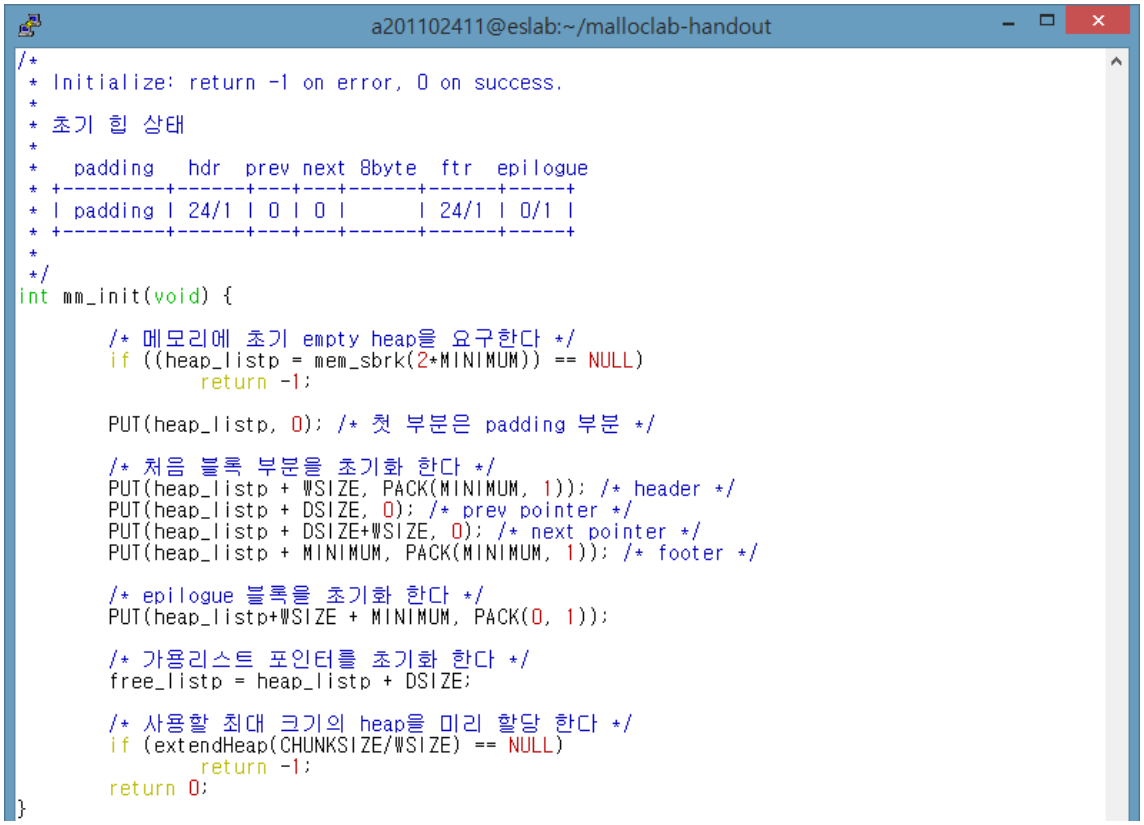
블록을 반환할 때는 인접한 블록들에서 가용 블록이 존재하는지 확인하고, 있다면 해당 블록과 연결(합병)을 해준다.

(2) explicit macro



explicit macro

(3) explicit source




```
/*
 * Initialize: return -1 on error, 0 on success.
 *
 * 초기 힙 상태
 *
 *   padding   hdr  prev next 8byte  ftr  epilogue
 * +---------+------+---+---+------+------+-----+
 * | padding | 24/1 | 0 | 0 |      | 24/1 | 0/1 |
 * +---------+------+---+---+------+------+-----+
 *
 */
int mm_init(void) {

        /* 메모리에 초기 empty heap을 요구한다 */
        if ((heap_listp = mem_sbrk(2*MINIMUM)) == NULL)
                return -1;

        PUT(heap_listp, 0); /* 첫 부분은 padding 부분 */

        /* 처음 블록 부분을 초기화 한다 */
        PUT(heap_listp + WSIZE, PACK(MINIMUM, 1)); /* header */
        PUT(heap_listp + DSIZE, 0); /* prev pointer */
        PUT(heap_listp + DSIZE+WSIZE, 0); /* next pointer */
        PUT(heap_listp + MINIMUM, PACK(MINIMUM, 1)); /* footer */

        /* epilogue 블록을 초기화 한다 */
        PUT(heap_listp+WSIZE + MINIMUM, PACK(0, 1));

        /* 가용리스트 포인터를 초기화 한다 */
        free_listp = heap_listp + DSIZE;

        /* 사용할 최대 크기의 heap을 미리 할당 한다 */
        if (extendHeap(CHUNKSIZE/WSIZE) == NULL)
                return -1;
        return 0;
}
```

int mm_init(void)

힙을 초기화하는 함수는 위와 같은 구조를 사용하고, 구현한다.

첫 번째 워드는 패딩으로 지정하였고, 그 다음 최소 블록 크기(24바이트)를 갖는 블록이 오도록 구현한다. 이 블록은 가용 리스트의 끝 부분으로 구현된다. 가용 블록들은 리스트의 앞 쪽에 계속 삽입되므로, 가용 리스트의 제일 마지막은 최소 블록 크기로 할당된 이 초기화된 블록이 온다.

정적 전역 변수 free_listp는 항상 가용 리스트의 앞부분(루트)을 가리키도록 한다. 이 포인터와 위의 초기 힙 블록 사이에 가용 블록들이 리스트로 연결된다.

힙의 마지막 부분에는 크기가 0인 할당 블록으로 에필로그를 구현한다.

a201102411@eslab:~/malloclab-handout

```
/*
 * malloc
 *  해당하는 크기의 블록을 할당한다
 *  정렬을 고려하여 블록의 크기를 계산한다
 */
void *malloc (size_t size) {

        size_t asize; /* 조정된 블록 크기를 위한 변수 */
        size_t extendsize; /* 맞는 fit이 없을 경우 힙을 늘리는 양 */
        char *bp;

        /* size가 올바르지 않을 때 예외처리 */
        /* size가 음수 값일 때는 무시한다 */
        if (size <= 0)
                return NULL;

        /* block의 크기 결정 */
        /* 오버헤드와 정렬을 위해 필요한 부분을 포함해서 크기를 결정한다 */
        asize = MAX(ALIGN(size) + DSIZE, MINIMUM);

        /* 결정한 크기에 알맞은 블록을 list에서 검색하여 해당 위치에 할당한다 */
        if ((bp = findFit(asize))) {
                place(bp, asize);
                return bp;
        }

        /* free list에서 적절한 블록을 찾지 못했으면 힙을 늘려서 할당한다 */
        extendsize = MAX(asize, CHUNKSIZE);

        if ((bp = extendHeap(extendsize/WSIZE)) == NULL)
                return NULL;

        place(bp, asize);
        return bp;
}
```

`void *malloc (size_t size)`

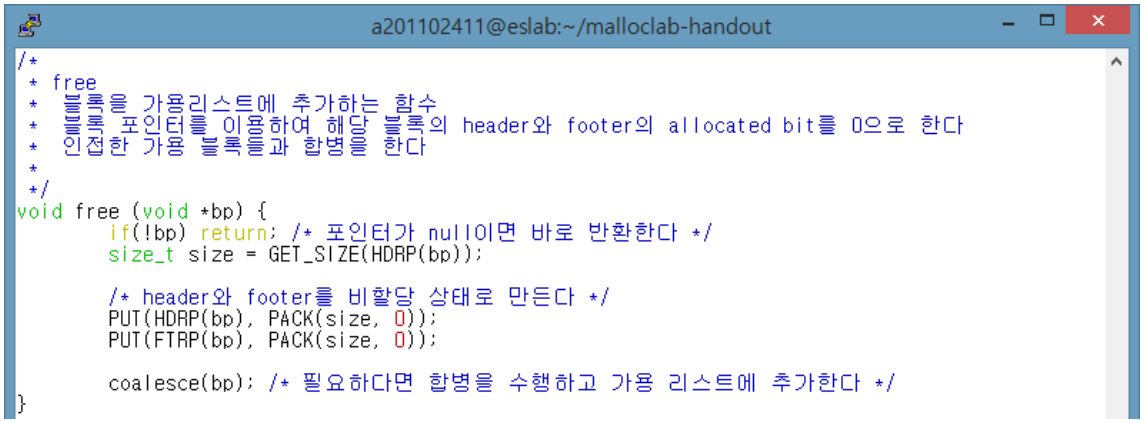
매개변수로 넘겨받는 크기의 블록을 할당하는 함수이다.

크기가 0보다 작거나 같으면 바르지 않은 값이 입력되었기 때문에, 예외처리를 해주는 조건문을 구현한다. 크기가 잘못된 값이 들어오면 null을 반환하도록 구현한다.

크기를 정렬에 맞춰 크기를 결정한다. 적어도 최소 블록 크기를 갖도록, 매크로로 구현한 MAX를 이용하여 크기를 결정한다.

크기를 결정한 후, findFit 함수를 이용하여 크기에 알맞은 블록을 검색하고, place 함수를 이용하여 할당한다.

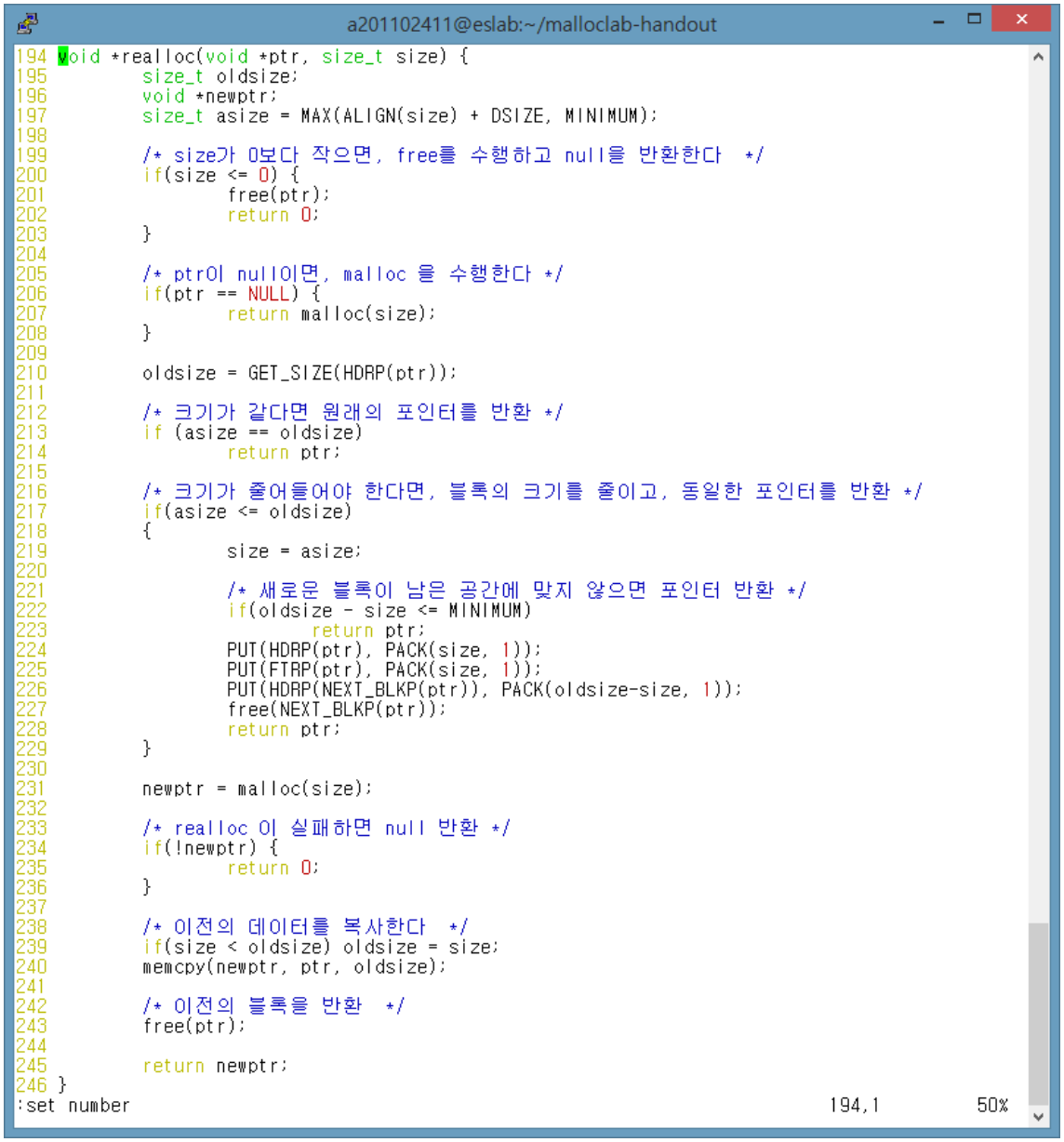
만약, 적절한 블록을 찾지 못하면, 메모리에 힙을 더 요구하는 함수인 extendHeap을 이용하여 힙을 더 할당 받고, 그곳에 블록을 할당한다.




void free (void *bp)

주어진 블록을 반환하는 함수이다.

매개변수로 전달받은 블록의 헤더와 풋터의 allocated bit를 0으로 설정하여 가용 블록으로 만든다. 그리고 난 뒤 coalesce 함수를 실행하여 인접한 가용 블록들이 있으면 연결하도록 구현한다.



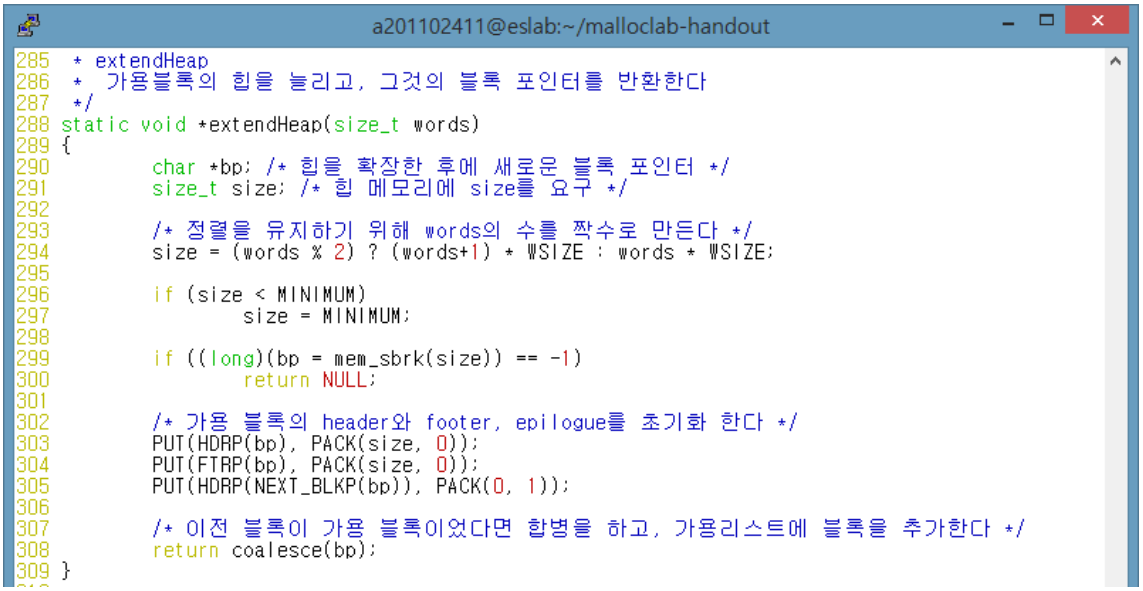

void *realloc (void *ptr, size_t size)

블록의 크기를 재할당하는 함수이다. 이는 mm-naive를 기초하여 구현한다.

size가 null이면 기본적으로 free를 수행하고, ptr이 null이면 malloc을 수행하도록 구현한다. 원래 크기와 같도록 재할당한다면, 원래 블록의 포인터 그대로를 반환하도록 한다.

블록 크기를 줄여야 한다면, 블록 크기를 줄이고, 동일한 블록 포인터를 반환한다. 만약에 줄이는 차이가 최소 블록 크기보다 작다면, 해당 블록을 그대로 반환한다. 최소 블록 크기보다 큰 차이로 블록을 줄인다면, 블록을 나누도록 구현한다.

새롭게 malloc을 이용하여 해당하는 크기의 블록을 할당하고, memcpy를 이용하여 데이터를 복사한다. 그리고 이전에 존재하던 블록은 free를 이용하여 반환한다.

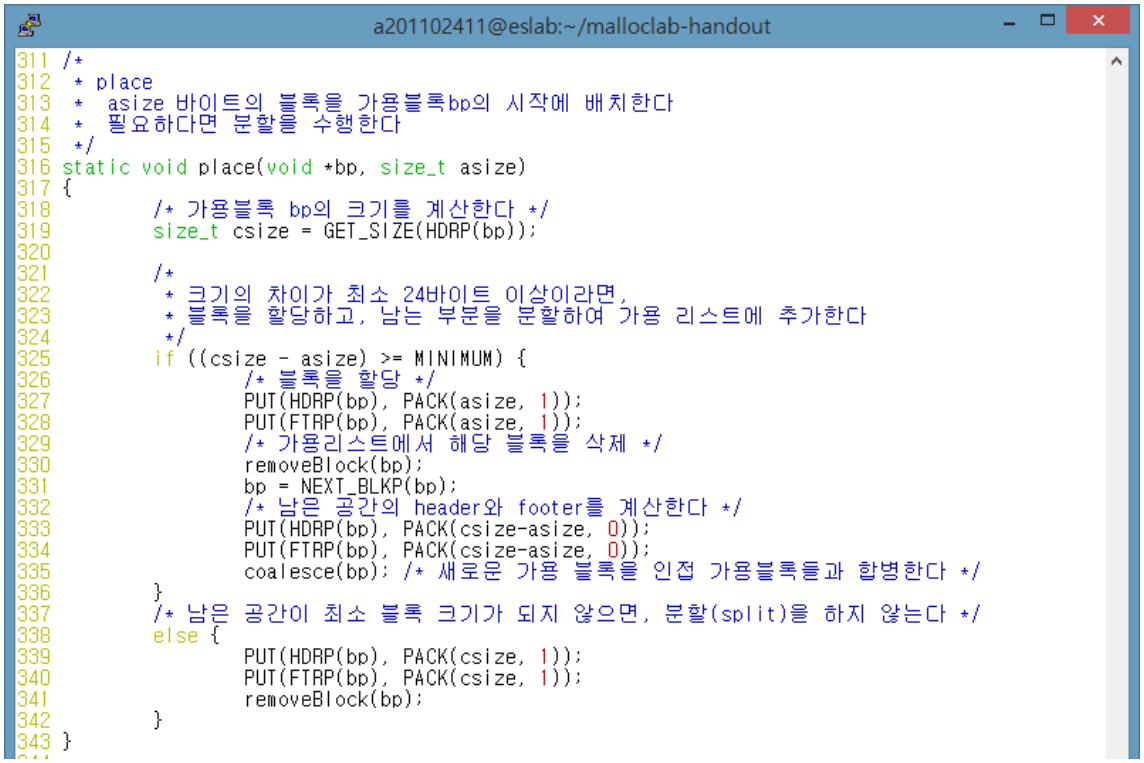



static void *extendHeap (size_t words)

메모리에 추가로 힙 공간을 요구하는 함수이다.

먼저 정렬을 유지하고 위해 요구되는 힙 크기를 짝수로 만들어준다. 그리고 sbrk를 이용하여 메모리에 힙 공간을 요구한다. 새롭게 할당된 공간을 가용 공간으로 만들어 준다. 이전에 존재하던 에필로그 부분이 새롭게 할당된 가용공간의 헤더 부분이 된다. 그리고 새롭게 생긴 가용 공간에 에필로그를 새로 정해준다.

리턴은 coalesce를 호출하면서 반환된다. 이는 새롭게 요구받은 가용 공간 이전 블록이 가용 블록이었다면 합병을 해준다. 그리고 가용 리스트에 추가한다.




```
311 /*
312  * place
313  *  asize 바이트의 블록을 가용블록bp의 시작에 배치한다
314  *  필요하다면 분할을 수행한다
315  */
316 static void place(void *bp, size_t asize)
317 {
318         /* 가용블록 bp의 크기를 계산한다 */
319         size_t csize = GET_SIZE(HDRP(bp));
320
321         /*
322          * 크기의 차이가 최소 24바이트 이상이라면,
323          * 블록을 할당하고, 남는 부분을 분할하여 가용 리스트에 추가한다
324          */
325         if ((csize - asize) >= MINIMUM) {
326                 /* 블록을 할당 */
327                 PUT(HDRP(bp), PACK(asize, 1));
328                 PUT(FTRP(bp), PACK(asize, 1));
329                 /* 가용리스트에서 해당 블록을 삭제 */
330                 removeBlock(bp);
331                 bp = NEXT_BLKP(bp);
332                 /* 남은 공간의 header와 footer를 계산한다 */
333                 PUT(HDRP(bp), PACK(csize-asize, 0));
334                 PUT(FTRP(bp), PACK(csize-asize, 0));
335                 coalesce(bp); /* 새로운 가용 블록을 인접 가용블록들과 합병한다 */
336         }
337         /* 남은 공간이 최소 블록 크기가 되지 않으면, 분할(split)을 하지 않는다 */
338         else {
339                 PUT(HDRP(bp), PACK(csize, 1));
340                 PUT(FTRP(bp), PACK(csize, 1));
341                 removeBlock(bp);
342         }
343 }
```

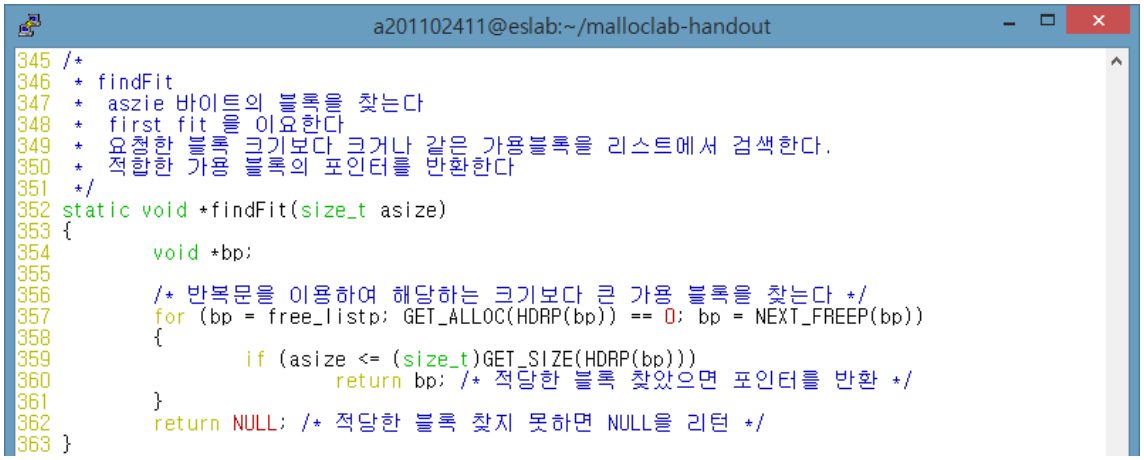
`static void place (void *bp, size_t asize)`

매개 변수로 넘겨받는 블록에 해당하는 크기의 블록을 배치하는 함수이다. 그리고 필요하다면 분할까지 수행하도록 구현한다.

가용 블록의 크기와 배치하려는 블록의 크기의 차이가 최소 블록 크기(24 바이트) 이상이라면, 블록을 할당하고 분할을 수행하도록 한다.

해당하는 크기의 블록을 할당하고, 이 블록을 가용 리스트에서 제거한다. 그리고 남는 공간을 가용 블록으로 만들어 주기 위해 헤더와 풋터를 설정한다. 그리고 coalesce를 호출한다.

만약 할당하고 남는 부분이 최소 블록 크기가 되지 않으면 분할을 하지 않고, 블록을 할당하고, 할당 받은 블록을 가용리스트에서 제거하는 것만 구현한다.



static void *findFit(size_t asize)

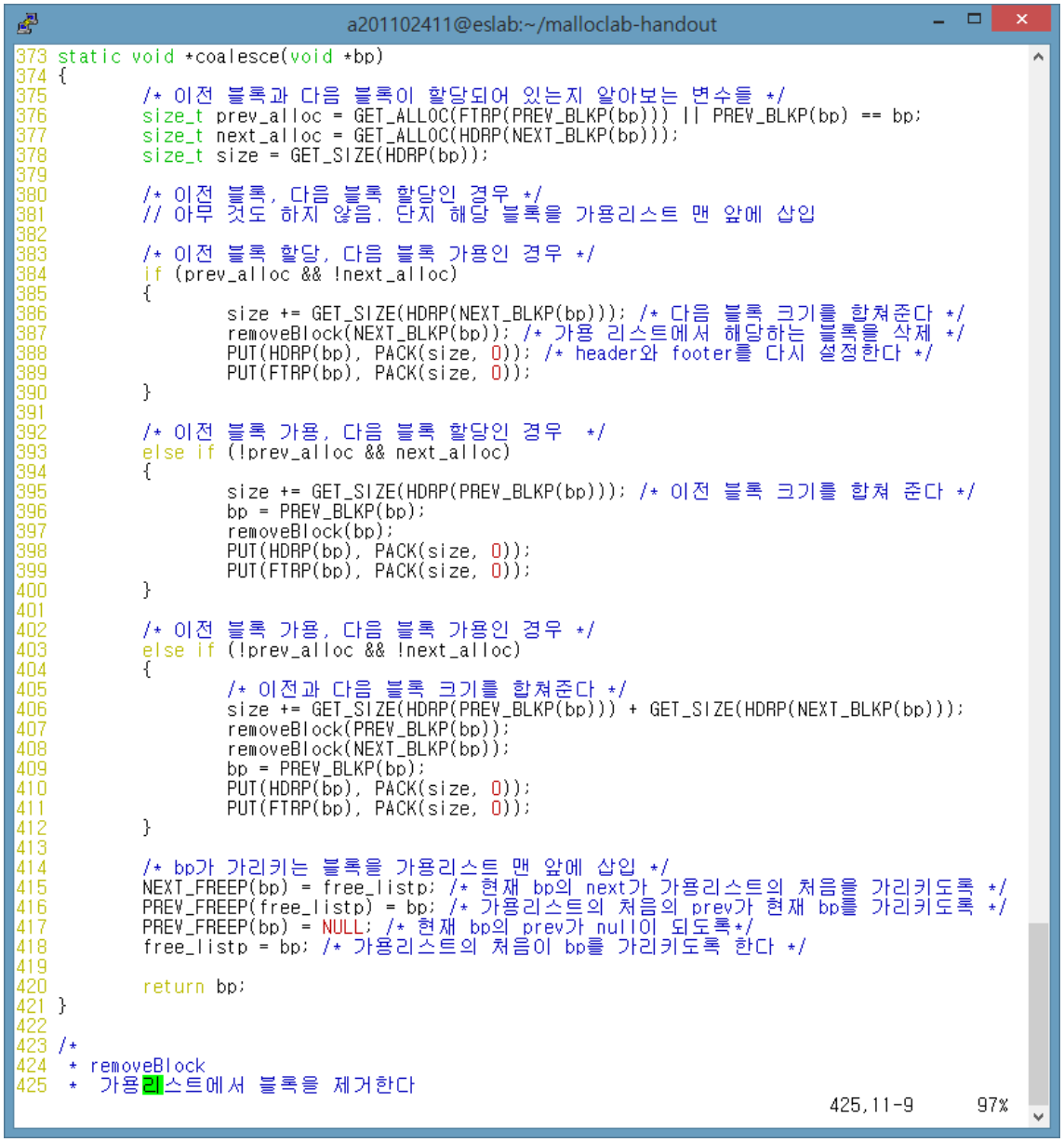
first fit을 수행하도록 구현한다.

이는 가용 리스트의 처음부터 해당하는 크기보다 큰 가용블록을 찾을 수 있도록 반복문을 사용하여 구현한다.

가용리스트의 첫 부분을 가리키고 있는 free_listp를 이용한다.

반복문의 종료 조건은 블록의 allocated bit가 0이 아닐 때로 한다. 가용 리스트의 제일 마지막은 할당된 최소 크기의 블록이 있다.

반복문 안에 조건문을 이용하여 적당한 크기의 블록을 찾으면 해당 블록의 포인터를 반환하면서 종료하도록 구현한다.




```
373 static void *coalesce(void *bp)
374 {
375         /* 이전 블록과 다음 블록이 할당되어 있는지 알아보는 변수들 */
376         size_t prev_alloc = GET_ALLOC(FTRP(PREV_BLKP(bp))) || PREV_BLKP(bp) == bp;
377         size_t next_alloc = GET_ALLOC(HDRP(NEXT_BLKP(bp)));
378         size_t size = GET_SIZE(HDRP(bp));
379
380         /* 이전 블록, 다음 블록 할당인 경우 */
381         // 아무 것도 하지 않음. 단지 해당 블록을 가용리스트 맨 앞에 삽입
382
383         /* 이전 블록 할당, 다음 블록 가용인 경우 */
384         if (prev_alloc && !next_alloc)
385         {
386                 size += GET_SIZE(HDRP(NEXT_BLKP(bp))); /* 다음 블록 크기를 합쳐준다 */
387                 removeBlock(NEXT_BLKP(bp)); /* 가용 리스트에서 해당하는 블록을 삭제 */
388                 PUT(HDRP(bp), PACK(size, 0)); /* header와 footer를 다시 설정한다 */
389                 PUT(FTRP(bp), PACK(size, 0));
390         }
391
392         /* 이전 블록 가용, 다음 블록 할당인 경우  */
393         else if (!prev_alloc && next_alloc)
394         {
395                 size += GET_SIZE(HDRP(PREV_BLKP(bp))); /* 이전 블록 크기를 합쳐 준다 */
396                 bp = PREV_BLKP(bp);
397                 removeBlock(bp);
398                 PUT(HDRP(bp), PACK(size, 0));
399                 PUT(FTRP(bp), PACK(size, 0));
400         }
401
402         /* 이전 블록 가용, 다음 블록 가용인 경우 */
403         else if (!prev_alloc && !next_alloc)
404         {
405                 /* 이전과 다음 블록 크기를 합쳐준다 */
406                 size += GET_SIZE(HDRP(PREV_BLKP(bp))) + GET_SIZE(HDRP(NEXT_BLKP(bp)));
407                 removeBlock(PREV_BLKP(bp));
408                 removeBlock(NEXT_BLKP(bp));
409                 bp = PREV_BLKP(bp);
410                 PUT(HDRP(bp), PACK(size, 0));
411                 PUT(FTRP(bp), PACK(size, 0));
412         }
413
414         /* bp가 가리키는 블록을 가용리스트 맨 앞에 삽입 */
415         NEXT_FREEP(bp) = free_listp; /* 현재 bp의 next가 가용리스트의 처음을 가리키도록 */
416         PREV_FREEP(free_listp) = bp; /* 가용리스트의 처음의 prev가 현재 bp를 가리키도록 */
417         PREV_FREEP(bp) = NULL; /* 현재 bp의 prev가 null이 되도록*/
418         free_listp = bp; /* 가용리스트의 처음이 bp를 가리키도록 한다 */
419
420         return bp;
421 }
422
423 /*
424  * removeBlock
425  *  가용리스트에서 블록을 제거한다
```

static void *coalesce (void *bp)

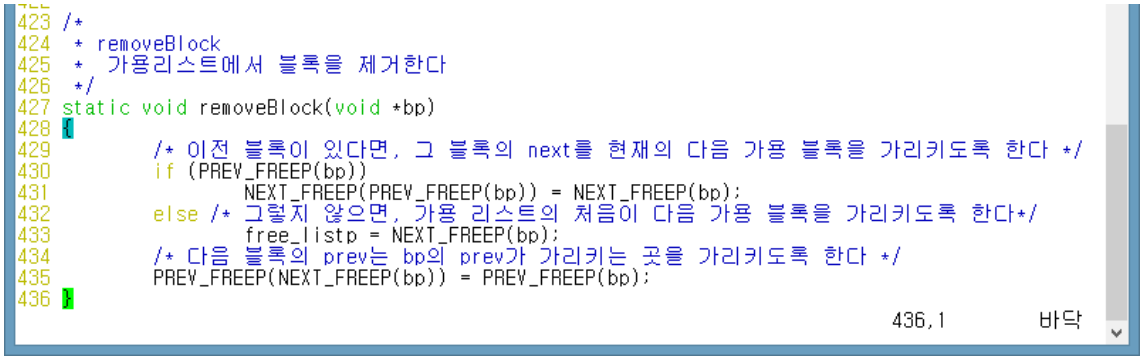
블록을 반환할 때, 해당하는 네 가지 case 중 하나를 수행하도록 구현한다. implicit을 구현할 때와 유사하다.

차이점이 있다면, 블록의 이전이나 다음에 가용 블록이 있다면, 이 가용 블록을 가용 리스트에서 제거한 후, 블록을 연결하고, 이 연결된 블록을 가용 리스트의 제일 처음에 삽입해야 한다는 것이다.

그리고 case1인 경우는 단순히 가용 리스트에 블록을 넣어준다.

따라서, 나머지 세 경우에 대해서 인접한 가용 블록들을 합치도록 구현하고, 마지막 부분에 합쳐진 가용 블록을 가용 리스트에 추가하도록 구현한다. 이렇게 구현해서 case1인 경우를 같이 구현되도록 한다.

또한 가용 블록을 가용 리스트에서 제거하는 함수인 removeBlock을 구현한다.



```
423 /*
424  * removeBlock
425  *  가용리스트에서 블록을 제거한다
426  */
427 static void removeBlock(void *bp)
428 {
429         /* 이전 블록이 있다면, 그 블록의 next를 현재의 다음 가용 블록을 가리키도록 한다 */
430         if (PREV_FREEP(bp))
431                 NEXT_FREEP(PREV_FREEP(bp)) = NEXT_FREEP(bp);
432         else /* 그렇지 않으면, 가용 리스트의 처음이 다음 가용 블록을 가리키도록 한다*/
433                 free_listp = NEXT_FREEP(bp);
434         /* 다음 블록의 prev는 bp의 prev가 가리키는 곳을 가리키도록 한다 */
435         PREV_FREEP(NEXT_FREEP(bp)) = PREV_FREEP(bp);
436 }
                                                              436,1         바닥
```

static void removeBlock (void *bp)

가용블록을 가용 리스트에서 제거하는 함수이다.
이는 단순히 연결된 리스트를 수정하여 구현할 수 있다.
만약 제거하려는 가용 블록의 앞부분에 블록이 존재한다면, 이 앞부분의 블록의 next가 제거하려는 다음 블록 다음에 있는 블록을 가리키도록 한다. 그리고 제거하려는 블록 다음에 있는 블록의 prev가 제거하려는 가용 블록 앞부분을 가리키도록 한다.
즉, 연결 구조를 바꾸어서 블록을 제거할 수 있다.
만약 제거하려는 가용 블록 앞부분에 블록이 존재하지 않다면, 이 블록이 가용리스트 제일 처음의 블록이라는 의미이므로, 가용 리스트의 제일 처음을 가리키는 free_listp가 제거하려는 가용 블록 다음의 블록을 가리키도록 한다.

(4) 결과 화면

```
[a201102411@eslab malloclab-handout]$ ./mdriver
Using default tracefiles in ./traces/
Measuring performance with a cycle counter.
Processor clock rate ~= 3392.3 MHz

Results for mm malloc:
   valid  util   ops    secs     Kops  trace
   yes     89%      10  0.000000 31645 ./traces/malloc.rep
   yes     92%      17  0.000000 34994 ./traces/malloc-free.rep
   yes    100%      15  0.000001 29346 ./traces/corners.rep
 * yes     84%    1494  0.000039 38015 ./traces/perl.rep
 * yes     84%     118  0.000003 36745 ./traces/hostname.rep
 * yes     89%   11913  0.000282 42259 ./traces/xterm.rep
 * yes     90%    5694  0.000259 21975 ./traces/amptjp-bal.rep
 * yes     92%    5848  0.000191 30562 ./traces/cccp-bal.rep
 * yes     95%    6648  0.000368 18052 ./traces/cp-decl-bal.rep
 * yes     96%    5380  0.000286 18793 ./traces/expr-bal.rep
 * yes     99%   14400  0.000303 47511 ./traces/coalescing-bal.rep
 * yes     87%    4800  0.000838  5729 ./traces/random-bal.rep
 * yes     55%    6000  0.005584  1074 ./traces/binary-bal.rep
10         87%   62295  0.008155  7639

Perf index = 56 (util) + 40 (thru) = 96/100
```

<결과 화면>

next fit으로 구현한 implicit 보다 조금 더 좋은 결과를 보여준다는 것을 알 수 있다.
best fit을 이용하면 가용 블록 리스트를 매번 탐색해야 하므로 성능이 하락한다.
따라서, first fit 으로 구현하였다. 또한, 가용 블록 리스트의 앞쪽에 반환되는 블록을 삽입하는 LIFO 방식으로 구현하였다. first fit은 가용 블록 리스트를 처음부터 탐색하므로 마지막에 반환된 블록부터 탐색하게 된다.