Lab report #9

CSED 20170302 Kim Dae Hui

이번lab은 malloc lab이다. Malloc시 사용되는 할당과 free방법과 heap사용법을 충분히 익힌 후에 이를 이용한 코딩을 하면 된다. 우선적으로 알고 있는 것은 heap을 사용하기 위해서는 함수를 사용해야한다는 점이었다.

• You should not change any of the interfaces in mm.c.

• You should not invoke any memory-management related library calls or system calls. This excludes the use of malloc, calloc, free, realloc, sbrk, brk or any variants of these calls in your code.

• You are not allowed to define any global or static compound data structures such as arrays, structs, trees, or lists in your mm.c program. However, you are allowed to declare global scalar variables such as integers, floats, and pointers in mm.c.

• For consistency with the libc malloc package, which returns blocks aligned on 8-byte boundaries, your allocator must always return pointers that are aligned to 8-byte boundaries. The driver will enforce this requirement for you.

위 프로그래밍 룰을 지키면서 mm.c의 코드를 수정하면 된다.

|  |
| --- |
| //word사이즈랑 힙에 늘어난 청크싸이즈 지정  #define WSIZE sizeof(void \*)  #define DSIZE (2 \* WSIZE)  #define CHUNKSIZE (1 << 12)  #define MAX(x, y) ((x) > (y) ? (x) : (y))  //PACK으로 allocate된 지점과 size지정  #define PACK(size, alloc) ((size) | (alloc))  //get과 put으로 특정주소에 있는 값을 read/write가능  #define GET(p) (\*(uintptr\_t \*)(p))  #define PUT(p, val) (\*(uintptr\_t \*)(p) = (val))  //size와 allocate된 지점을 read  #define GET\_SIZE(p) (GET(p) & ~(DSIZE - 1))  #define GET\_ALLOC(p) (GET(p) & 0x1)  //adress상에서 block의 header와footer  #define HDRP(bp) ((void \*)(bp) - WSIZE)  #define FTRP(bp) ((void \*)(bp) + GET\_SIZE(HDRP(bp)) - DSIZE)  //next와 prev노드로 가는 포인터  #define NEXT\_BLK(bp) ((void \*)(bp) + GET\_SIZE(HDRP(bp)))  #define PREV\_BLK(bp) ((void \*)(bp) - GET\_SIZE((void \*)(bp) - DSIZE))  //next와 prev노드의 정보를 get하는 것  #define GET\_NEXT\_PTR(bp) (\*(char \*\*)(bp + WSIZE))  #define GET\_PREV\_PTR(bp) (\*(char \*\*)(bp))  //next와 prev노드를 재정의해주는 것  #define SET\_NEXT\_PTR(bp, qp) (GET\_NEXT\_PTR(bp) = qp)  #define SET\_PREV\_PTR(bp, qp) (GET\_PREV\_PTR(bp) = qp)  //globcl 포인터  static char \*heap\_listp = 0;  static char \*free\_list\_start = 0; |

우선 mm.c에서 사용하게 될 macro들을 정의한다. 대부분 블록에 관련한 것들과 블록 내부의 포인터를 지정하고 그 명칭을 알려준 것이다. global포인터로 heaplist와 freeliststart를 이용해서 노드를 리스트에 넣어주도록 할 수 있게 한다.

|  |
| --- |
| int mm\_init(void) {  //처음에 힙을 생성시킨다.  if ((heap\_listp = mem\_sbrk(8 \* WSIZE)) == NULL)  return -1;  //padding지정 후에 header와 footer를 설정  PUT(heap\_listp, 0);  PUT(heap\_listp + (1 \* WSIZE), PACK(DSIZE, 1));  PUT(heap\_listp + (2 \* WSIZE), PACK(DSIZE, 1));  PUT(heap\_listp + (3 \* WSIZE), PACK(0, 1));  free\_list\_start = heap\_listp + 2 \* WSIZE;  //heap에 최소의 크기로 요구하는만큼 extend  if (extend\_heap(4) == NULL) {  return -1;  }  return 0;  } |

Mm init함수에서는 힙영역에서 header와 footer등 필수적인 노드를 생성하고 이를 초기화시킨다. 힙에서 요구되는 최소크기로 지정함으로써 효율성을 올려준다.

|  |
| --- |
| void \*mm\_malloc(size\_t size)  {  size\_t asize;  size\_t extendsize;  void \*bp;  //예외처리  if (size == 0)  return (NULL);  //block size에 따라서 heap에 생성여부를 따짐  if (size <= DSIZE)  asize = 2 \* DSIZE;  else  asize = DSIZE \* ((size + DSIZE + (DSIZE - 1)) / DSIZE);  //heap에생성할공간 체크  if ((bp = find\_fit(asize)) != NULL) {  place(bp, asize);  return (bp);  }  //만약에 extend해야할 경우 heap에다가 생성  extendsize = MAX(asize, CHUNKSIZE);  if ((bp = extend\_heap(extendsize / WSIZE)) == NULL)  return (NULL);  place(bp, asize);  return (bp);  } |

Block size에 따라서 extend해줄지 여부를 검사하면서 malloc을 한다. 만약에 리스트에 충분한 공간이 없는 상태라면 heap으로 extend해주도록 하고 이니시하여 malloc을 한다.

|  |
| --- |
| void mm\_free(void \*bp) {  size\_t size;  //예외처리  if (bp == NULL)  return;  //사이즈만큼프리해주고 앞뒤정보 합침  size = GET\_SIZE(HDRP(bp));  PUT(HDRP(bp), PACK(size, 0));  PUT(FTRP(bp), PACK(size, 0));  coalesce(bp);  } |

free함수의 경우에는 size를 0으로 해주면서 앞뒤의 노드를 coalesce해주어 free한 후 블록단위로 리스트에 합류시킨다.

|  |
| --- |
| void \*mm\_realloc(void \*bp, size\_t size) {  if ((int)size < 0)  return NULL;  else if ((int)size == 0) {  mm\_free(bp);  return NULL;  }  else if (size > 0) {  size\_t original\_size = GET\_SIZE(HDRP(bp));  size\_t new\_size = size + 2 \* WSIZE; // 헤더와 푸터사이즈  //realloc시 기존 사이즈보다 더 작을 경우의 처리  if (new\_size <= original\_size) {  return bp;  }  //realloc시 기존사이즈보다 더 큰 경우의 처리  else {  size\_t next\_alloc = GET\_ALLOC(HDRP(NEXT\_BLK(bp)));  size\_t csize;  //만약에 블록이 비어있을 경우에는 그냥 합쳐주고 안되면은 place를 이용한 처리를 한다.  if (!next\_alloc && ((csize = original\_size + GET\_SIZE(HDRP(NEXT\_BLK(bp))))) >= new\_size) {  node\_remove(NEXT\_BLK(bp));  PUT(HDRP(bp), PACK(csize, 1));  PUT(FTRP(bp), PACK(csize, 1));  return bp;  }  else {  void \*new\_ptr = mm\_malloc(new\_size);  place(new\_ptr, new\_size);  memcpy(new\_ptr, bp, new\_size);  mm\_free(bp);  return new\_ptr;  }  }  }  else  return NULL;  } |

Realloc의 경우 기존의 사이즈와 realloc한 블록의 사이즈의 크기와 extend의 여부에 따라 나누어 처리해준다. 새로 alloc하는 경우의 header와 footer지정도 필수이다.

|  |
| --- |
| static void \*coalesce(void \*bp) {  //이전블록과 이후블록의 상태에 따라서 coalesce하도록 설정  size\_t NEXT\_ALLOC = GET\_ALLOC(HDRP(NEXT\_BLK(bp)));  size\_t PREV\_ALLOC = GET\_ALLOC(FTRP(PREV\_BLK(bp))) || PREV\_BLK(bp) == bp;  size\_t size = GET\_SIZE(HDRP(bp));  //다음블록만 free인경우  if (PREV\_ALLOC && !NEXT\_ALLOC) {  size += GET\_SIZE(HDRP(NEXT\_BLK(bp)));  node\_remove(NEXT\_BLK(bp));  PUT(HDRP(bp), PACK(size, 0));  PUT(FTRP(bp), PACK(size, 0));  }  //이전블록만 free인 경우  else if (!PREV\_ALLOC && NEXT\_ALLOC) {  size += GET\_SIZE(HDRP(PREV\_BLK(bp)));  bp = PREV\_BLK(bp);  node\_remove(bp);  PUT(HDRP(bp), PACK(size, 0));  PUT(FTRP(bp), PACK(size, 0));  }  //양쪽다 free인 경우  else if (!PREV\_ALLOC && !NEXT\_ALLOC) {  size += GET\_SIZE(HDRP(PREV\_BLK(bp))) + GET\_SIZE(HDRP(NEXT\_BLK(bp)));  node\_remove(PREV\_BLK(bp));  node\_remove(NEXT\_BLK(bp));  bp = PREV\_BLK(bp);  PUT(HDRP(bp), PACK(size, 0));  PUT(FTRP(bp), PACK(size, 0));  }//최종으로 넣은 bp를 return  node\_insert(bp);  return bp;  } |

coal여부는 전 후의 노드의 allocate여부에 따라 나누어서 coal해주고 헤더와 푸터를 조정한다.

|  |
| --- |
| static void \*extend\_heap(size\_t words) {  char \*bp;  size\_t size;  //짝수번호를 할당하여 word사이즈단위로 할당  size = (words % 2) ? (words + 1) \* WSIZE : words \* WSIZE;  //미니멈 블록사이즈만큼은 되야하니 16보다 작은 할당은 16유지  if (size < 16) {  size = 16;  }  //예외처리  if ((int)(bp = mem\_sbrk(size)) == -1) {  return NULL;  }  //초기화를 시킨다.  PUT(HDRP(bp), PACK(size, 0));  PUT(FTRP(bp), PACK(size, 0));  PUT(HDRP(NEXT\_BLK(bp)), PACK(0, 1));  //초기화후 합침  return coalesce(bp);  } |

에스비알케이 함수처럼 extend를 한다. 워드사이즈단위로 해주고 초기화시킨 후 합류시키는 방식으로 coal을 해준다.

|  |
| --- |
| static void \*find\_fit(size\_t asize) {  //이 함수에서는 fit되는 블록위치를 찾아준다.  void \*bp;  static int last\_malloced\_size = 0;  static int repeat\_counter = 0;  if (last\_malloced\_size == (int)asize) {  if (repeat\_counter>30) {  int extendsize = MAX(asize, 4 \* WSIZE);  bp = extend\_heap(extendsize / 4);  return bp;  }  else  repeat\_counter++;  }  else  repeat\_counter = 0;  for (bp = free\_list\_start; GET\_ALLOC(HDRP(bp)) == 0; bp = GET\_NEXT\_PTR(bp)) {  if (asize <= (size\_t)GET\_SIZE(HDRP(bp))) {  last\_malloced\_size = asize;  return bp;  }  }  return NULL;  } |

fit되는 블록의 위치를 찾는다. Malloc된 블록을 기준으로 서치하여 fit되는 블록을 찾아서 이를 asize만큼 단위로 이동한다.

|  |
| --- |
| static void place(void \*bp, size\_t asize) {  size\_t csize = GET\_SIZE(HDRP(bp));  //이 블록에서는 asize 블록을 bp에 위치시켜주도록 함  if ((csize - asize) >= 4 \* WSIZE) {  PUT(HDRP(bp), PACK(asize, 1));  PUT(FTRP(bp), PACK(asize, 1));  node\_remove(bp);  bp = NEXT\_BLK(bp);  PUT(HDRP(bp), PACK(csize - asize, 0));  PUT(FTRP(bp), PACK(csize - asize, 0));  coalesce(bp);  }  else {  PUT(HDRP(bp), PACK(csize, 1));  PUT(FTRP(bp), PACK(csize, 1));  node\_remove(bp);  }  } |

Assize 단위의 블록을 bp(할당정보)를 가진 블록에 위치해주도록 해주는 함수

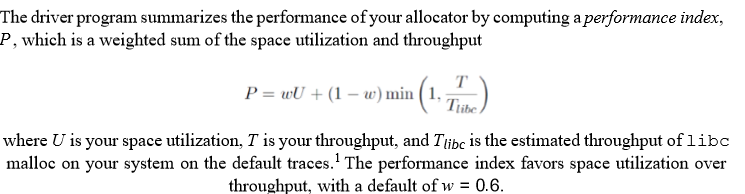
|  |
| --- |
| static void node\_insert(void \*bp) {  SET\_NEXT\_PTR(bp, free\_list\_start);  SET\_PREV\_PTR(free\_list\_start, bp);  SET\_PREV\_PTR(bp, NULL);  free\_list\_start = bp;  //global 포인터인 free\_list\_start에 노드를 넣어준다.  }  static void node\_remove(void \*bp) {  if (GET\_PREV\_PTR(bp))  SET\_NEXT\_PTR(GET\_PREV\_PTR(bp), GET\_NEXT\_PTR(bp));  else  free\_list\_start = GET\_NEXT\_PTR(bp);  SET\_PREV\_PTR(GET\_NEXT\_PTR(bp), GET\_PREV\_PTR(bp));  //global 포인터인 free\_list\_start에 노드를 빼준다.  } |

글로벌 변수인 freelist부분에 노드를 추가와 제거해주도록 하는 함수이다.

이와같이 mm.c를 설치하여 trace파일들을 돌려 결과를 확인해보면 다음과 같다.

|  |
| --- |
| Results for mm malloc:  trace valid util ops secs Kops  0 yes 90% 5694 0.000269 21175  1 yes 82% 5848 0.000217 26912  2 yes 92% 6648 0.000377 17620  3 yes 92% 5380 0.000335 16041  4 yes 99% 14400 0.000330 43663  5 yes 88% 4800 0.000761 6309  6 yes 85% 4800 0.000778 6172  7 yes 55% 12000 0.000408 29433  8 yes 51% 24000 0.000623 38517  9 yes 93% 14401 0.000363 39672  10 yes 21% 14401 0.000357 40384  Total 77% 112372 0.004818 23325  Perf index = 46 (util) + 40 (thru) = 86/100 |

7번과8번그리고10번파일에서 낮은 효율을 보여 util점수가 낮아 86퍼센트의 효율을 보임을 알 수 있다.



위 p점을 보아 검사해보면 util의 점수가 낮아진 이유를 알 수 있다.

위 코드는 explcit list기법을 기반으로 했는데 seglist가 explicitlist기반보다 best-fit에 가깝기에 그 효율의 차이가 나는 것같다.