Writing a Dynamic Storage Allocator

정성태 2018-15515

Outline

전체적인 구조는 segregated free list 방법을 따랐다. 이는 free block들을 추적할 수 있도록 별도의 list 형태로 보관하되, 여러 개의 size set을 두어 free block의 크기에 따라 적절한 set에 속할 수 있도록 함으로써 search에 소요되는 시간을 줄일 수 있는 방법이다.

heap 내부의 한 free block은 "header – next link pointer – empty space – footer"의 구조로 구성되어 있다. header와 footer는 해당 block이 allocated 되었는지의 여부와 block의 크기를 저장하는 공간으로, coalesce의 구현 등에 필요하다. next link pointer는 각 free list의 다음 block을 가리키는 포인터이다. allocate block의 경우 이 부분이 필요가 없으므로 "header – payload – padding – footer"의 구조를 띠게 된다.

Macros

코드 작성을 간편하게 하기 위해 교과서에 나와 있는 것들에 더해 몇 가지 macro를 추가로 정의해 사용했다.

```
#define NSETS 15                // must be an odd number
```

segregated free list에 포함되어 있는 size set의 개수를 나타낸다. heap 내부 구현과 alignment condition에 의해 size set은 홀수 개여야 한다는 제약이 생기게 된다. 여러 값을 대입해 실험해본 뒤 util 점수가 가장 높게 나온 15를 선택했다.

```
/* gets the address of the next link */
#define GETNEXT(bp) ((void *)GET(bp))
#define SETNEXT(bp, next) (PUT(bp, (addr)(next)))
```

explicit free list를 구현하기 위해 사용되는 macro이다. 각각 heap 내부 특정 block에 저장되어 있는 next link pointer를 읽고 쓸 때 사용된다.

```
#define PREVFTRP(bp) ((char *)(bp) - 8)
```

특정 block의 pointer로부터 이전 block의 footer는 항상 8 byte만큼 떨어져 있다는 점을 이용해 정의한 macro이다. FTRP(PREVBLKP(ptr)) 대신 PREVFTRP(ptr)로 쓸 수 있게 된다.

```
typedef unsigned int addr;      // address value of pointer
```

각 부분의 next link pointer 부분에는 pointer의 값이 저장되어 있어야 하므로, void*와 unsigned int 간에 반복적인 형변환이 필요하다. 이를 위한 코드를 직관적으로 작성하기 위해 addr라는 새로운 type을 정의했다.

mm_init

malloc, free, realloc 등 기능을 사용하기 전에 호출해야 하는 함수로 heap 메모리의 초기화를 담당한다. heap 의 가장 앞쪽에는 각 free list 의 첫 번째 node 를 가리키는 pointer 를 저장하는 부분이 위치한다. 그 뒤에는 메모리 공간의 처음과 끝을 나타내는 prologue 와 epilogue block 이 위치한다. prologue block 은 allocated block 으로 간주되며 payload 에는 아무것도 들어 있지 않다.

```
/* Put the root links of free lists */
for (i=0; i<NSETS; i++) {
    PUT(heappt, 0);   // initially null
    heappt += WSIZE;
}

/* Put the prologue & epilogue blocks */
PUT(heappt, PACK(16, 1));               // prologue header
PUT(heappt + 1*WSIZE, 0);               // prologue payload
PUT(heappt + 2*WSIZE, 0);               // prologue payload
PUT(heappt + 3*WSIZE, PACK(16, 1));     // prologue footer
PUT(heappt + 4*WSIZE, PACK(0, 1));      // epilogue header
```

이렇게 초기화가 완료되고 나면 extend_heap 을 호출해 heap 의 크기를 확장하고 종료된다.

mm_malloc

입력된 size 만큼의 heap 메모리를 할당해주는 함수이다. 이때 alignment condition 을 충족하기 위해 입력된 size 를 8 의 배수로 조정하는 작업이 필요하다. 또한 header 와 footer 를 저장할 공간도 추가로 확보해야 한다.

```
/* Adjust block size to meet the alignment condition */
newsize = ALIGN(size + 2*WSIZE);    // minimum block size: 4 words

/* Find an appropriate space to allocate */
if ((ptr = find_fit(newsize)) != NULL) {
    place(ptr, newsize, 1);
    return ptr;
}

/* No fit space; Extend heap to get more space */
extendsize = MAX(newsize, CHUNKSIZE);
if ((ptr = extend_heap(extendsize/WSIZE)) == NULL) {
    return NULL;
}
place(ptr, newsize, 1);

return ptr;
```

먼저 find_fit 을 호출해 지정된 size 를 만족하는 공간을 탐색한다. 만약 적절한 공간이 발견되면 place 를 호출해 그 부분에 새로운 block 을 할당하고 포인터를 반환한다. 만약 탐색에 실패하면 extend_heap 을 호출해 heap 메모리를 확장하고, 확장된 부분에 새로운 block 을 할당한다.

mm_free

malloc 이나 realloc 으로 할당된 공간을 해제하는 함수이다. 입력된 포인터가 가리키는 block 을 찾아 header와 footer를 수정해 free 상태로 만든다. 만약 앞뒤에 이웃한 free block이 있다면 병합해 하나의 block 으로 만들어야 하는데, 이 작업은 coalesce 가 담당한다. 해제된 block 을 free list 에 추가하는 작업도 coalesce 에서 수행된다.

mm_realloc

지정된 block 의 크기를 수정하여 재할당하는 함수이다. 만약 입력된 포인터가 NULL 이라면 malloc 과 동일하게 동작하고, size 가 0 이라면 free 처럼 동작한다.

일반적인 경우, malloc 과 유사하게 alignment condition 과 header, footer 가 차지하는 공간을 고려해 크기를 조정해야 한다. 이렇게 산출된 새로운 크기와 기존 block 의 크기의 관계에 따라 세 가지 경우가 나타날 수 있다.

```
/* Case 1: Same size */
if (newsize == orig_size) {
    return ptr;                     // do nothing
}

/* Case 2: Decreased size */
else if (newsize < orig_size) {
    place(ptr, newsize, 0);         // split the original block
    return ptr;
}
```

먼저 새로운 block 의 크기가 기존 block 의 크기와 동일한 경우, 아무런 작업도 할 필요가 없다. 따라서 입력된 포인터를 그대로 반환하면 된다.

만약 새로운 크기가 기존 크기보다 작은 경우, 기존 block 을 둘로 쪼개어 한쪽을 free 처리하면 된다. 이 작업은 malloc 에서도 사용했던 place 가 담당한다. 다만 malloc 에서는 원래 free 상태였던 block 을 분할하는 반면, 여기서는 원래 allocated 상태였던 block 을 분할해야 한다. place 의 마지막 인자인 0 은 이 차이를 표시하는 역할을 한다.

```
/* Case 3: Increased size */

/* Case 3-1: Can merge with the next block to get enough size */
if (!GET_ALLOC(HDRP(NEXTBLKP(ptr)))) {
    nextblock_size = GET_SIZE(HDRP(NEXTBLKP(ptr)));

    if (orig_size + nextblock_size >= newsize) {
        increase(ptr, newsize);
        return ptr;
    }
}

/* Case 3-2: Need to find a new space */
newptr = mm_malloc(size);           // allocate a new block
memmove(newptr, ptr, size);         // copy the contents of the original block
mm_free(ptr);
return newptr;
```

만약 새로운 크기가 기존 크기보다 큰 경우 block 을 확장해야 한다. 만약 바로 뒤에 다른 free block 이 존재해서 새로운 크기를 충분히 소화할 수 있는 경우라면 단순히 크기만 확장하면 된다. 이 작업은 increase 에서 수행된다.

뒤쪽에 free block 이 없거나 있음에도 공간이 부족한 경우, 새로운 block 을 찾아 기존 block 에 들어 있던 데이터를 복사해야 한다. 이때는 malloc 과 free 를 호출하는 방식을 사용했다

extend_heap

mem_sbrk 를 이용해 heap 메모리를 주어진 크기만큼 확장하고 적절한 초기화를 수행한다. 여기에는 새로 확장된 heap 메모리의 끝에 epilogue block 을 추가하고, 확장된 부분의 바로 앞에 free block 이 존재할 경우 병합을 수행하는 등의 과정이 포함된다. 구체적인 구현은 대부분 교재에 나와 있는 것을 따랐다.

coalesce

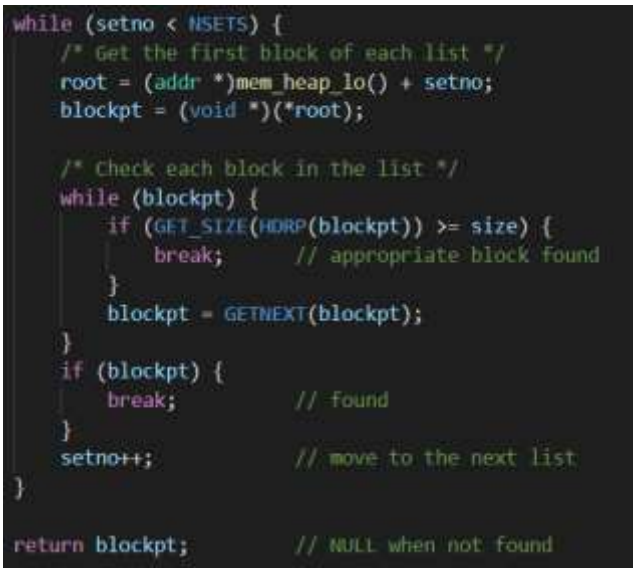
입력된 포인터가 가리키는 free block 을 앞뒤로 이웃하는 다른 free block 과 병합해 하나로 만들고, 적절한 free list 에 추가하는 작업까지 수행하는 함수이다. 앞뒤로 이웃하는 block 이 free 인지의 여부에 따라 4 가지 경우가 존재한다.

만약 앞뒤의 block 이 모두 allocated 인 경우, 단순히 해당 block 을 free list 에 추가하기만 하면 된다. 이 작업은 add_block 을 호출해 수행하게 된다.

만약 앞이나 뒤의 block 이 free 라면, 우선 병합을 위해 그 block 을 free list 에서 제거해야 한다. 이 작업은 pop_block 을 호출해 수행한다. 이제 header 와 footer 의 size 정보를 적절하게 수정해 두 block 을 병합하고, 새로운 block 을 다시 add_block 을 호출해 free list 에 추가하면 완료된다.

find_fit

새로 할당할 block 의 크기가 주어지면 free list 를 순회하며 주어진 크기를 만족할 수 있는 적절한 free block 을 찾아내는 함수이다.



```
while (setno < NSETS) {
    /* Get the first block of each list */
    root = (addr *)mem_heap_lo() + setno;
    blockpt = (void *)(*root);

    /* Check each block in the list */
    while (blockpt) {
        if (GET_SIZE(HDRP(blockpt)) >= size) {
            break;      // appropriate block found
        }
        blockpt = GETNEXT(blockpt);
    }
    if (blockpt) {
        break;          // found
    }
    setno++;            // move to the next list
}

return blockpt;         // NULL when not found
```

free list 가 segregated 방식으로 구현되었으므로 먼저 주어진 크기가 포함되는 size set 을 찾아 순회한 뒤, 만약 적절한 block 을 찾지 못하면 계속해서 다음 size set 을 순회하면 된다. 이때 각 size set 내부에서는 first fit 방식으로 block 을 찾아내는데, 이렇게 하면 전체 free list 를 best fit 방식으로 탐색하는 것과 유사한 효과를 낼 수 있다. 만약 모든 size set 을 순회했음에도 적절한 block 을 발견하지 못한 경우 NULL 을 반환한다. 이 경우 malloc 에서 extend_heap 을 호출하게 된다.

place

주어진 공간을 둘로 쪼개어 한쪽을 할당하고 나머지는 free 처리하는 함수이다. malloc 과 realloc 에서 이 함수를 호출한다. 다만 malloc 에서는 free block 에 대해, realloc 에서는 allocated block 에 대해 호출하게 된다. 이를 구별하기 위해 마지막 인자로 int was_free 를 두었다.

was_free 가 1 인 경우, 즉 malloc 에서 호출된 경우 입력된 포인터가 가리키는 block 을 free list 에서 제거해야 한다. 이때 pop_block 을 호출하여 이 작업을 수행한다.

```
if (rest < 4*WSIZE) {                  // remaining space is too small
    size = blocksize;                  // give the entire block
    rest = 0;
}

PUT(HDRP(ptr), PACK(size, 1));         // update header
PUT(FTRP(ptr), PACK(size, 1));         // update footer
if (rest == 0) {
    return;                            // exactly fit
}

/* Handle the remaining block block */
ptr = NEXTBLKP(ptr);
PUT(HDRP(ptr), PACK(rest, 0));         // update header
PUT(FTRP(ptr), PACK(rest, 0));         // update footer
add_block(ptr, rest);                  // add to free list
```

만약 필요한 크기를 확보한 후 남은 공간이 4 words 보다 작다면, header 와 next link pointer, footer 를 저장할 수 없게 된다. 이때는 남은 공간을 하나의 free block 으로 만드는 것이 불가능해지므로 block 을 분할하지 않고 전부 할당하도록 했다.

이제 header 와 footer 의 크기 정보를 수정해 block 을 분할 처리하고, 뒤에 남는 공간이 존재할 경우 역시 header 와 footer 를 수정해 독립된 free block 으로 만든다. 마지막에는 add_block 을 호출해 새로 만들어진 이 free block 을 free list 에 추가하면 된다.

increase

realloc 에서 호출되는 함수로, allocated block 의 크기를 확장하기 위해 바로 뒤에 위치한 free block 의 일부를 흡수한다. 전반적인 동작 방식은 place 와 유사하다. 뒤에 위치한 free block 을 free list 에서 제거한 뒤, header 와 footer 의 크기 정보를 수정한 뒤 새로 생겨나는 free block 을 다시 free list 에 추가해주면 된다.

add_block, pop_block

free list 에 block 을 추가하고 제거하는 역할을 하는 함수 add 와 pop 의 wrapper 이다. add_block 과 pop_block 은 block 을 가리키는 포인터와 해당 block 의 크기를 인자로 받는데, 내부에서 which_set 을 호출해 어떤 free list 에 속해야 하는지를 판단한 뒤 add 와 pop 을 호출해 입력된 block 을 해당 list 에 추가한다.

which_set

block 의 크기를 입력 받아 어떤 size set 에 속해야 하는지를 결정해주는 함수이다. 여기서는 n 번째(n=0, 1, …, NSETS-1) size set 의 크기 상한을 2^{n+5}로 설정했기 때문에 간단하게 적절한 size set 을 찾아낼 수 있다. 5 라는 값은 여러 값을 대입해 실험해본 후 최적의 결과를 내는 값을 선택한 결과이다.

add

size set 번호와 block 포인터를 받아 해당 block 을 size set 의 list 에 추가해주는 함수이다. heap 의 앞부분에 위치한 각 list 의 root node 를 수정해 입력된 block 을 가리키게 함으로써 block 을 list 의 맨 앞에 추가한다.

pop

size set 번호와 block 포인터를 받아 해당 block 을 list 에서 제거하는 함수이다. root 에서 출발해 node 를 하나씩 순회하며 해당 block 을 발견하면 제거한다.

```
/* Find the requested block in the list */
if (prev == block) {                // first block matched
    next = GETNEXT(prev);
    *root = (addr)next;
    return;
}

while ((next = GETNEXT(prev)) != block) {
    if (next == NULL) {             // not found
        printf("pop(): requested block not found\n");
        exit(1);
    }
    prev = next;
}

/* Remove the block */
SETNEXT(prev, GETNEXT(next));
```

이때 list 를 일방향으로 구현했기 때문에 제거 작업을 위해 순회 과정에서 previous node 를 계속 저장해둘 필요가 있다. 또한 list의 첫 번째 block을 제거해야 하는 경우 root의 값을 수정해야 하므로 이 경우는 따로 다루었다.

Debugging Tools

디버깅에 활용하기 위해 mm_check_init, mm_check_heap, mm_check_segregated 라는 함수를 정의해 사용했다.

mm_check_init 은 mm_init 에서 호출해 heap 메모리에 대한 initialization 이 잘 수행되었는지를 확인하기 위해 사용한다. prologue block 과 epilogue block 이 잘 만들어졌는지를 확인할 수 있다.

mm_check_heap 은 malloc, free, realloc 이 수행된 뒤 heap 메모리에 대한 변경이 정상적으로 이루어졌는지를 확인하기 위해 쓸 수 있다. header 에 저장된 크기 정보를 이용해 heap 내부의 모든 block 을 순회하며 정보를 출력한다. 또한 header 의 값을 footer 와 비교해 불일치가 발생하는 경우를 포착할 수 있도록 했다.

mm_check_segregated 는 free list 의 각 size set 을 모두 순회하며 free block 들의 정보를 출력한다. free block 들이 정상적으로 free list 에 추가되었는지를 확인하기 위해 사용할 수 있으며, 각 size set 에 block 들이 추가되는 양상을 관찰함으로써 size set 의 개수와 크기 상한을 조절하고자 할 때 참고로 활용할 수도 있다.

Result

```
~/system_programming/lab03_malloc
> ./mdriver -v
Using default tracefiles in ./traces/
Measuring performance with gettimeofday().

Results for mm malloc:
trace  valid  util     ops      secs  Kops
 0       yes   98%    5694  0.000963  5911
 1       yes   98%    5848  0.000771  7588
 2       yes   97%    6648  0.000860  7728
 3       yes   99%    5380  0.000622  8644
 4       yes   66%   14400  0.001416 10170
 5       yes   93%    4800  0.000946  5076
 6       yes   90%    4800  0.000848  5663
 7       yes   55%   12000  0.006984  1718
 8       yes   51%   24000  0.038099   630
 9       yes   38%   14401  0.001912  7533
10       yes   51%   14401  0.000938 15359
Total          76%  112372  0.054358  2067

Perf index = 46 (util) + 40 (thru) = 86/100
```

어려웠던 점

메모리 공간을 직접 조작해야 하다 보니 포인터를 직접 다룰 필요가 있는데, 각 동작의 결과를 일일이 추적하기가 어려워 디버깅이 간단하지 않았다. mdriver 의 실행 결과로 출력된 각종 오류 메시지나 프로그램 자체에서 발생하는 segmentation fault 등을 해결하기 위해 직접 정의한 mm_check 함수들을 적극 활용했다. 또한 디버깅 툴인 gdb 의 도움을 받기도 했다.

새롭게 알게 된 점

수업시간에 간단하게만 다루었던 segregated free list 를 직접 구현해보며 작동 방식을 더욱 구체적으로 이해할 수 있었다.

gdb 와 gprof 등 C 프로그래밍을 할 때 사용할 수 있는 도구들을 새로 알게 되었다.