

System Programming Project 3

담당 교수 : 김영재

이름 : 최원빈

학번 : 20181693

1. 개발 목표

나만의 dynamic memory allocator를 구현한다. Malloc, free, realloc 함수를 직접 구현한다. 이때 필요한 다른 static 함수나 전역변수를 선언해도 되지만 배열이나 구조체 같은 자료구조는 쓰지 않도록 한다. 여러 trade-off를 고려해, 공간을 효율적으로 잘 사용하면서, 동시에 가장 정확하고 효율적이며 빠른 형태로 만들도록 한다.

2. 개발 범위 및 내용

A. 개발 범위

- malloc 요청이 들어오면 메모리의 heap 영역의 공간을 할당해주고 그 블록을 가리키는 포인터를 반환한다. Free 요청이 들어오면 이전에 할당된 영역이라면 할당되지 않았음을 표시하고 그곳을 다시 free list에 넣는다. Realloc 요청이 들어오면 크게 세 가지 경우로 나뉘는데, 포인터가 NULL일 경우 malloc처럼, size가 0일 경우 free처럼 행동한다. 만약 둘 모두에 해당 안 된다면 원래 가리키고 있던 영역의 크기를 size로 바꿔준다. 이때 크기가 커질 경우 원래 내용을 앞에서부터 모두 복사하고, 작아질 경우 일부만 복사해 붙여넣는다.

B. 개발 내용

- ✓ 위와 같은 기능을 하는 함수들을 구현하기 위해 동적 메모리 할당에 대한 전반적인 이해가 필요하다. 얼마나 free를 해야할지, 할당할 알맞은 free 블록을 어떻게 찾을지, 할당된 블록과 할당되지 않은 블록을 어떻게 구별할지 등에 대한 것들을 모두 고려해 코드를 작성해야 한다. 또 내부 단편화와 외부 단편화에 대한 대책을 마련해야 한다. 이런 문제들을 해결하기 위해 나는 이번 프로젝트에서 header와 footer가 모두 존재하는 explicit free list를 사용할 것이다. Header와 footer의 size로 블록의 크기를 알 수 있어 양방향 coalescing이 가능하고, align되어 있기 때문에 마지막 비트에 할당되어 있는지 아닌지 표시할 수 있다. 또 free block들만 양방향으로 연결하여 관리할 것이고, 할당할 블록은 first-fit 방식으로 찾을 것이다. Insertion policy로는 LIFO와 address-ordered 정책을 모두 사용해보고 space utilization과 throughput을

비교해 더 나은 정책을 채택할 것이다.

C. 개발 방법

위의 내용을 구현하기 위해 수정 또는 작성한 함수 및 변수들이다.

MINIMUM : 24로 define되었다. 기본적으로 어떤 블록의 최소 사이즈를 뜻한다.

NEXT_FP(bp) : bp의 다음 free block의 주소를 반환해준다.

PREV_FP(bp) : bp의 이전 free block의 주소를 반환해준다.

static char* free_listp : free list의 첫번째 블록을 가리키는 포인터이다.

int mm_init(void) : 나의 memory manager를 초기화한다. 먼저 mem_sbrk 로 힙영역을 MINIMUM + DSIZE 만큼 늘려준다. 이후 8바이트 단위로 블록포인터를 맞춰주기 위해 처음에 0을 넣어 padding을 하고 그 다음은 헤더를 만들어준다. 이후 previous pointer와 next pointer를 위한 영역에 0을 넣어주고, footer까지 만들어주고, 마지막 표시를 하는 헤더까지 만들어준다. 이때 첫 블록은 나의 free list의 마지막 종단점으로 만들어 주기위해 헤더와 푸터의 마지막 비트를 1로 표시해준다. 즉 실제로 할당된 영역이 아니고 할당해주지도 않을 것이다. Free list에서 알맞은 블록을 찾을 때 끝을 나타내는 영역일 뿐이다. 이후 extend_heap으로 영역을 늘려 새 free_listp가 그곳을 가리키게 하고, 그블록은 나의 종단점을 가리킨다.

void* mm_malloc(size_t size) : size 만큼 프리블록에 할당해주고 포인터를 반환해준다. 먼저 size가 0일시 NULL값을 반환해준다. 이후 size를 align해준 값과 MINIMUM 을 비교해 더 큰 값을 asize에 저장한다. 이후 알맞은 영역을 찾으면, 해당 영역을 가리키는 포인터에 asize 만큼 할당해주고 반환한다. 찾지 못한 경우 heap영역을 늘린 뒤 할당해주고 반환한다.

void mm_free(void* bp) : bp가 0이면 바로 반환해준다. 아닐 경우 size를 구해 0으로 표시해주고, 합병이 필요한 경우 합병해준다. 이때 free list의 처음이 된다.

void* mm_realloc(void* ptr, size_t size) : size가 0이면 free를 해주고, ptr가

NULL이면 malloc을 해준다. 둘 다 아닐 시 해당 포인터의 원래 크기를 구해 align된 사이즈와 비교해준다. 같으면 그대로 반환하고 작으면 그 차이가 MINIMUM 보다 클 시 그 부분을 free list에 넣어준다. 아닐 시 그냥 반환한다. 더 크면 새로 블록을 할당하고 필요한 만큼 내용을 앞에서부터 복사해 앞에서부터 넣어준다. 이후 새 포인터 반환.

static void* extend_heap(size_t words) : 힙영역을 align한 워드 단위로 늘려준다. 이때 최소 사이즈는 MINIMUM 이다. 만약 mem_sbrk가 실패하면 NULL을 리턴한다. 성공했다면 새 영역의 첫 블록에 free block임을 표시해주고 coalesce 해준다.

static void place(void* bp, size_t asize) : 프리블록 bp의 원래 사이즈와 할당될 사이즈를 비교해 그 차가 MINIMUM 이상이면 남은 부분을 free list에 넣어준다. 그게 아니라면 그대로 사용해준다.

static void* find_fit(size_t asize) : free_list의 처음부터 시작해 알맞은 블록을 찾아준다. 찾지 못할 시(다 asize보다 작음) NULL을 리턴해준다.

static void* coalesce(void* bp) : 블록의 이전 블록과 다음 블록이 할당되었는지 확인하고 그에 따라 케이스를 4가지로 나눠 합병을 해준다. 합병을 해줄 시 이미 free list에 있는 이전 혹은 다음 블록을 list에서 빼준다. 이후 표시도 해준다. 이후 합병을 했든 안 했든 free_list에 넣어주고 포인터를 반환한다.

static void insert_front(void* bp) : 해당 포인터를 free list의 첫 블록으로 한다. 이를 위해 원래 free_listp의 prev-next pointer와 bp의 그 둘을 수정해준다. Bp의 prev는 NULL로 하고 free_listp에 bp를 넣는다.

static void insert_ordered(void* bp) : 해당 포인터를 free_list 내에 적절한 위치를 찾아 넣어준다. 여기서 적절한 위치란 주소순으로 정렬되어 있는 리스트를 만들기 위해 이전 포인터보다는 주소가 작고, 다음 포인터보다는 큰 위치를 말한다.(마지막이 가장 작은 주소임)

static void remove_block(void* bp) : 해당 블록을 free list에서 제거한다. 만약 prev pointer가 NULL이면 첫 블록이므로 free_listp로 지정한다.

3. 구현 결과

- 구현결과 mdriver를 실행 시 다음과 같은 성능을 보이며 정확히 동작한다. 먼저 LIFO 삽입 정책을 사용시 다음과 같다.

```
Results for mm malloc:
trace  valid  util    ops      secs  Kops
0      yes   89%    5694  0.000856  6654
1      yes   92%    5848  0.000728  8033
2      yes   95%    6648  0.001167  5698
3      yes   96%    5380  0.000939  5731
4      yes   88%   14400  0.001348 10682
5      yes   88%    4800  0.001235  3887
6      yes   85%    4800  0.000920  5220
7      yes   55%   12000  0.026755   449
8      yes   55%   24000  0.024110   995
9      yes   26%   14401  0.105737   136
10     yes   28%   14401  0.004121  3494
Total                73%  112372  0.167915   669

Perf index = 44 (util) + 40 (thru) = 84/100
```

- 그리고 address-ordered 정책 사용시 다음과 같다. 수업 때 배운 내용과 같이 단편화가 적어지면서 공간을 더 효율적으로 사용하지만, 내 코드의 경우 처리량이 급격히 줄어 채택하기 힘들다.

```
Results for mm malloc:
trace  valid  util    ops      secs  Kops
0      yes   99%    5694  0.001501  3793
1      yes   99%    5848  0.001641  3563
2      yes   99%    6648  0.000779  8533
3      yes   99%    5380  0.001355  3971
4      yes   88%   14400  0.001996  7215
5      yes   92%    4800  0.005307   904
6      yes   88%    4800  0.005912   812
7      yes   55%   12000  0.038955   308
8      yes   55%   24000  0.065084   369
9      yes   30%   14401  0.107624   134
10     yes   28%   14401  0.004403  3271
Total                76%  112372  0.234558   479

Perf index = 45 (util) + 32 (thru) = 77/100
```

- 또 chunksize에 따라 다음과 같이 달라지는데

```
Results for mm malloc:
trace  valid  util    ops      secs  Kops
0      yes   90%   5694  0.000864  6591
1      yes   92%   5848  0.000651  8982
2      yes   95%   6648  0.000893  7445
3      yes   96%   5380  0.000561  9595
4      yes   96%  14400  0.000865 16647
5      yes   88%   4800  0.000882  5441
6      yes   85%   4800  0.000888  5404
7      yes   55%  12000  0.033853   354
8      yes   55%  24000  0.051256   468
9      yes   25%  14401  0.106152   136
10     yes   30%  14401  0.002633  5470
Total                73% 112372  0.199498   563
```

- Perf index = 44 (util) + 38 (thru) = 82/100 chunksize = 256

```
Results for mm malloc:
trace  valid  util    ops      secs  Kops
0      yes   89%   5694  0.000758  7511
1      yes   92%   5848  0.000680  8600
2      yes   94%   6648  0.001142  5819
3      yes   96%   5380  0.000978  5499
4      yes   66%  14400  0.001298 11098
5      yes   88%   4800  0.000891  5385
6      yes   85%   4800  0.000928  5171
7      yes   55%  12000  0.008801  1363
8      yes   51%  24000  0.005387  4455
9      yes   26%  14401  0.105438   137
10     yes   30%  14401  0.004147  3473
Total                70% 112372  0.130448   861
```

- Perf index = 42 (util) + 40 (thru) = 82/100 chunksize = 4096

- 줄일 경우 처리량이 줄어들고, 늘릴 시 공간사용량이 줄어든다. 제출용 코드의 경우 가장 좋은 성능을 보이는 1024일 경우다. 적당한 크기를 늘려 호출을 최소화 하면서도 공간을 너무 낭비하지 않아야 되는 trade-off가 존재한다.
- 다음은 프로젝트를 하던 중 사용한 gprof 결과로 나온 프로파일링 리포트의 일부다. 이것을 확인하여 mm_malloc과 coalesce, extend_heap 위주로 최적화를 하게 됐다.

Each sample counts as 0.01 seconds.

% time	cumulative seconds	self seconds	calls	self us/call	total us/call	name
83.98	1.52	1.52	607446	2.50	2.53	mm_malloc
9.94	1.70	0.18				add_range
4.97	1.79	0.09				remove_range
0.55	1.80	0.01	1444974	0.01	0.01	coalesce
0.55	1.81	0.01	276960	0.04	0.04	extend_heap
0.00	1.81	0.00	710220	0.00	0.00	place
0.00	1.81	0.00	513870	0.00	0.00	mm_free
0.00	1.81	0.00	95980	0.00	2.47	mm_realloc
0.00	1.81	0.00	110	0.00	0.05	mm_init

추가적으로 내가 구현한 내용은 아니지만, 구현하려 했던 보다 이상적인 형태를 기술하겠다. 우선 나의 경우 맨 처음 제공된 implicit free list 예제코드를 skeleton code로 삼아 최종적으로 segregated free list로 구현을 하려 했었다. 그러나 하던 중 큰 문제를 직면하게 되는데 바로 전역 포인터 배열이 사용 불가능하다는 이번 과제의 룰이었다. 그래서 처음에는 다음과 같이 여러 free_listp를 전역으로 선언한 뒤 temp_listp로 그때그때 size에 따라 temp_listp에 free_listp를 받아 사용하려 했었다. 사실 조금만 생각해보면 왜 안 되는지 알 법도 했을 텐데, 피로로 인해서인지 그 이유가 잘 보이지 않았다. ./mdriver를 실행해도 어떤 케이스는 통과하고 어떤 케이스만 ran out of memory가 되는지 이해가 가지 않았다.

```
static char* free_listp = 0; //24~127, asize기준
static char* free_listp2 = 0; //128~4095
static char* free_listp3 = 0; //4096~
```

```
static int list_num;
static char* temp_listp;
```

```
static int which_list(size_t asize) {
    if (asize < 128)
        return 1;
    if (asize < 4096)
        return 2;
    else
        return 3;
}
```

```
static void which_list2() {
    switch (list_num) {
```

```

case 1:
    temp_listp = free_listp;
    break;
case 2:
    temp_listp = free_listp2;
    break;
case 3:
    temp_listp = free_listp3;
    break;
}
}

```

이때 처음으로 heap checker를 사용하였다. 기본적으로 제공된 코드의 check_block에 다음과 같은 코드를 추가해 사용했는데, 나타나는 Error: Previous free pointer %p is out of bound 를 보고서야 왜 안 되는지 이유를 깨달았다. 그래서 free가 제대로 안 되고 있는 것을 인지하고 방식을 바꿔보기로 했었다. 그전에 여기서 mm_check에 대해 설명을 하도록 하겠다. 예를 들어 이 두가지의 경우 현재 블록의 다음 혹은 이전 free block이 힙의 영역을 벗어나서 -1이 리턴된다. 그러면 mm_check에서도 -1을 리턴한다. 그게 아닐 시 1을 리턴하고 그 경우 프로그램은 계속 진행된다. 이렇듯 mm_check는 매번 free list를 처음부터 끝까지 잘 align 되어있는지, header footer 도 정확한지, 그리고 힙의 영역에 잘 있는지 확인해주는 함수이다.

```

if (NEXT_FP(bp) < mem_heap_lo() || NEXT_FP(bp) > mem_heap_hi()) {
    printf("Error: Next free pointer %p is out of bound\n", NEXT_FP(bp));
    return -1;
}
if (PREV_FP(bp) < mem_heap_lo() || PREV_FP(bp) > mem_heap_hi()) {
    printf("Error: Previous free pointer %p is out of bound", PREV_FP(bp));
    return -1;
}

```

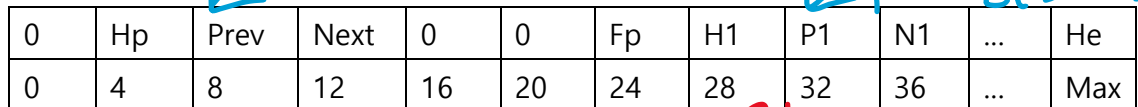
바로 블록의 최소 단위를 24바이트로 하고 다음과 같은 정보를 담는 것이었다.

Header | Previous free pointer | Next free pointer | 가장 가까운 이전 범위의 free list의 멤버 | 가장 가까운 다음 범위의 free list의 멤버 | Footer

예를 들어 위에서라면 내가 구현한 free_listp2에 연결되어 있어야 하는 블록이라면, 여기서 추가적으로 가장 가까운 free_listp, free_listp3에 연결되어 있어야 하

는 블록 중 가장 가까운 주소를 가지고 있는 것이다. 이런 식으로 연결해두면 하나의 free_listp로 범위에 따라 블록을 구분해 저장할 수 있을 것이라 생각했지만, 구현을 하던 중 오류를 도저히 못 잡을 것 같아 포기하고 하나의 리스트로 구현하였다. 이후 최적화를 한 결과가 제출물이다.

+) 추가적으로 하나만 더 설명하도록 하겠다. 나의 explicit free list의 기본형태이다.(처음 init할 때)



0	Hp	Prev	Next	0	0	Fp	H1	P1	N1	...	He
0	4	8	12	16	20	24	28	32	36	...	Max

NULL