Malloc Lab Report

2019-11730 신현지

Implement

`static void *coalesce(void *bp)`

`coalesce` 함수는 주어진 블록(bp)과 인접한 블록이 free인 경우, 이들을 하나로 합친 후 free list에 삽입한다. 총 4가지 경우가 존재한다.

1. 둘 다 할당된 경우 : `bp`만 반환하고 아무것도 하지 않는다.
2. 이전 블록은 할당된 상태이고, 다음 블록은 비어있는 경우 : 해당 블록과 다음 블록을 합쳐 하나의 블록으로 만든다.
3. 이전 블록은 비어있고, 다음 블록은 할당된 상태인 경우 : 해당 블록과 이전 블록을 합쳐 하나의 블록으로 만든다.
4. 둘 다 비어있는 경우 : 이전 블록, 해당 블록, 다음 블록을 합쳐 하나의 블록으로 만든다.
5. 합쳐진 블록은 free list에 다시 삽입한다. 함수는 합쳐진 블록의 포인터를 반환한다.

`static void *extend_heap(size_t words)`

`extend_heap` 함수는 초기 heap 영역을 확장하는 함수이다. `mem_sbrk` 시스템 콜을 호출하여 새로운 가용 블록을 힙 끝에 추가한다. 확장된 블록은 현재 블록의 epilogue 헤더 뒤에 위치하며, 반환된 포인터는 블록의 헤더 포인터이다.

1. 함수의 첫 부분에서는 새로 추가될 블록의 크기를 계산한다. 추가할 블록은 두 워드의 배수로 크기가 조정되어야 하므로, 만약 홀수라면 다음 워드로 크기를 조정한다.
2. `mem_sbrk`를 호출하여 힙 끝에 블록을 추가한다. 만약 호출이 실패하면 NULL을 반환한다.
3. 새로운 블록의 헤더와 푸터를 초기화한다. 추가된 블록은 가용 블록이므로, 할당 비트를 0으로 설정한다.
4. 블록이 heap의 끝에 위치하므로, 새로운 epilogue 헤더를 설정한다.
5. 새로운 블록을 이전의 가용 블록과 병합한다 (`coalesce` 함수 호출).

`static void *find_fit(size_t size)`

`find_fit` 함수는 요청된 크기에 맞는 빈 공간을 찾는 함수이다.

1. 가용 블록 리스트의 head부터 시작하여, 블록이 가용인 경우 while문을 진행한다.
2. 만약 찾은 가용 블록의 사이즈가 요청한 size보다 크거나 같은 경우, 해당 블록의 주소를 반환한다.
3. 그렇지 않으면, 다음 가용 블록으로 이동하여 반복한다.
4. 만약 요청한 크기에 맞는 가용 블록이 존재하지 않으면, `NULL`을 반환한다.

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

`place` 함수는 할당하려는 블록을 free 리스트에서 제거하고, 만약 분할 가능한 경우 분할하여 남은 부분을 free 리스트에 추가하는 함수이다.

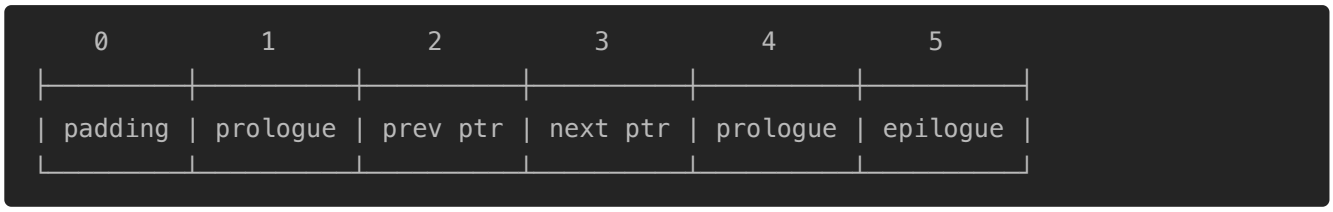
1. 할당 가능한 블록 크기(csize)를 가져온다.

2. 해당 블록을 free 리스트에서 제거한다.
3. 만약 `csize - size`가 `2 * DSIZE`보다 크다면, 새로 할당된 블록의 크기를 `size`로 설정하고, 남은 공간에 대해서 새로운 블록을 만든다. 그리고 남은 공간에 대한 블록을 free 리스트에 추가한다.
4. 그렇지 않으면, 해당 블록 전체를 할당하고 할당 요청을 완료한다.

`int mm_init(void)`

`mm_init` 함수는 malloc package를 초기화한다. 먼저 초기 힙을 생성한 다음 확장 가능한 힙을 만들기 위해 이를 확장한다.

1. `mem_sbrk` 함수를 사용하여 초기 힙을 할당한다. 힙의 크기는 6 words이다. 초기 힙은 다음과 같은 구조를 가진다.



2. `extend_heap` 함수를 사용하여 초기 힙을 확장한다. 초기 힙의 크기는 `CHUNKSIZE`이다. 초기 힙을 확장하면 빈 블록이 추가된다. 이 블록은 free 리스트의 첫 번째 블록이 된다.

`void *mm_malloc(size_t size)`

`mm_malloc` 함수는 메모리 블록을 할당하기 위해 호출되는 함수이다. 주어진 크기에 맞는 블록을 찾거나 힙을 확장하여 새로운 블록을 할당한다.

1. 할당할 블록의 크기를 정렬하기 위해 `ALIGN` 매크로를 사용한다.
2. 블록 크기를 비교하기 위해 `find_fit` 함수를 호출한다. 해당 함수는 가장 작은 적합한 블록을 찾아 포인터를 반환한다.
3. 적합한 블록을 찾을 수 없으면 힙을 확장한다.
4. 확장된 블록을 할당하고 사용할 부분을 반환한다.

`void mm_free(void *ptr)`

`mm_free` 함수는 할당되어 있던 메모리 블록을 반환하고, 반환된 블록과 주변의 빈 블록을 병합한다.

1. 해당 블록의 사이즈를 구한다.
2. 해당 블록의 헤더와 푸터에 alloc을 0으로 표시하고, 병합한다.

`void *mm_realloc(void *ptr, size_t size)`

`mm_realloc` 함수는 주어진 `ptr`이 가리키는 메모리 블록을 `size` 바이트로 다시 할당하고, 원래 블록의 내용을 새 블록으로 복사한다.

1. 입력으로 주어진 포인터 `ptr`이 `NULL` 인지 확인한다. 만약 `ptr`이 `NULL`이면, `mm_malloc`을 호출하여 새 메모리 블록을 할당하고 `NULL`을 반환한다.
2. `size`가 `0`인지 확인한다. 만약 `size`가 `0`이면, `mm_free`를 호출하여 해당 메모리 블록을 해제하고 `NULL`을 반환한다.
3. 새로운 메모리 블록을 할당하고 할당이 실패하면 `NULL`을 반환한다.
4. `copySize` 변수를 설정하고, 이전 블록의 크기와 `size` 중 더 작은 값으로 설정한다.

5. `memcpy` 함수를 사용하여 원래 메모리 블록에서 새 메모리 블록으로 데이터를 복사한다.
6. `mm_free`를 사용하여 이전 메모리 블록을 해제한다.
7. 새로 할당된 메모리 블록에 대한 포인터를 반환한다.

What was difficult

블록 분할, 병합 등 heap 메모리 관리 알고리즘을 이해하고 구현하는 것이 어려웠다. 특히, 블록이 할당 및 해제될 때의 모든 상황에 대해 처리해야 했기 때문에 코드 구현이 까다로웠다. implicit list에서 explicit list로 변경하면서 생겨난 버그가 너무 많은 것도 어려운 점이었다. 세그멘테이션 에러가 자주 발생했는데, 어떻게 디버깅을 하는 게 좋을지 몰라서 문제점을 발견하기가 어려웠다.

Something new and surprising

링크드리스트를 이용한 단순한 알고리즘을 사용해도 프로그램이 제대로 작동한다는 것이 놀라웠다. 힙 메모리 관리를 위한 기본 알고리즘을 구현하면 충분히 기능할 뿐만 아니라, 대부분의 케이스에서 매우 빠른 속도로 메모리를 할당할 수 있었다. 또한 red-black tree를 사용하면 적당한 free block을 빠르게 찾을 수 있다는 점을 알게 되었다. 이번 과제를 진행하면서는 시간과 구현 난이도의 문제로 red-black tree를 사용하지는 못했지만 빠른 이유를 이해할 수 있었다.