System Programming Project 4

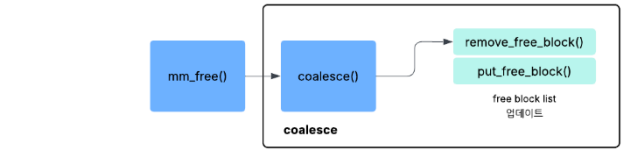
20231609

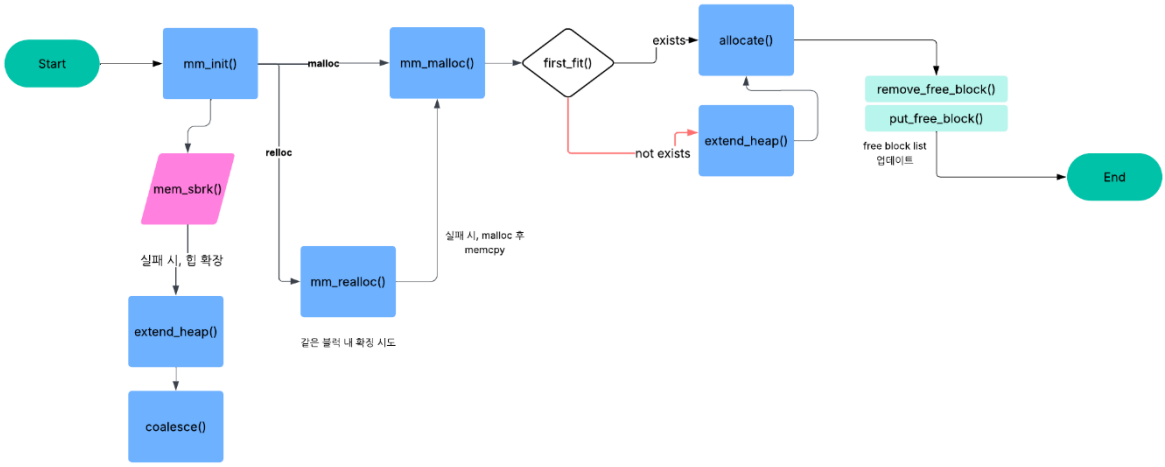
정희선

1. Introduction

본 프로젝트의 목표는 정확성, 효율성, 성능을 갖춘 메모리 할당기(malloc, free, realloc)을 직접 구현하는 것입니다. 최종적으로 explicit free list를 기반으로 한 가용 블록 관리 방식을 구현하였고, first-fit 방식과 coalescing, split 기능을 함께 구현하였습니다.

2. Design of Allocator





전체적인 구조는 위와 같습니다.

1) 블록 구조

Heap 상의 블록은 기본적으로 header, footer, payload로 구성됩니다. Free block의 경우 payload 공간에 이중 연결 리스트용 prev/next 포인터를 추가로 포함하였습니다. Header/footer에는 블록의 크기와 할당 여부가 함께 저장됩니다. 전체 블록은 8-byte 단위로 정렬되어 있습니다.

2) Free list 관리

free block들은 이중 연결 리스트를 통해 관리됩니다. 이 때 free block list는 크기 순 정렬 없이, 앞에서부터 순차적으로 탐색하는 first-fit 전략을 사용했습니다.

새로 반환된 블록의 경우 LIFO 전략을 사용하여 리스트의 가장 앞 쪽(head)에 삽입됩니다. 또한, splitting 전략을 사용하여 선택된 블록이 필요한 크기보다 클 경우, 요청 크기만큼만 할당한 후 나머지는 새로운 free block으로 분할합니다.

3) 블록 병합

외부 단편화를 줄이기 위해, 블록이 free될 때마다 즉시 병합이 수행됩니다. 즉, 현재 블록의 앞이나 뒤 블록이 free 상태라면 병합을 통해 하나의 더 큰 블록으로 만듭니다.

4) Realloc

위 동작은 기존 블록보다 더 큰 크기의 블록으로 재할당하는 동작입니다. 우선 현재 위치에서 인접한 free block과 병합하여 확장이 가능한지 확인합니다. 불가능할 경우에는 새로운 블록을 할당하고 데이터를 복사한 후 기존 블록을 free 시킵니다.

5) Heap 초기화

heap을 초기화하는 **init** 동작입니다. 이 동작으로 인해 prologue/epilogue 블록이 생성되고, 초기 가용 블록을 생성합니다. 또한 free list의 head 포인터도 초기화됩니다.

3. Description of Key Components

- **전역 변수**

```
static char *heap_listp;  // prologue 이후 첫 번째 블록 포인터
static void *free_listp;  // 가용 리스트의 시작 포인터
```

- **매크로**

1) 크기 및 정렬 관련

```
#define ALIGNMENT 8
#define WSIZE 4
#define DSIZE (WSIZE*2)              // 8바이트: 최소 블록 단위
#define MIN_BLOCK (DSIZE*2)          // 최소 블록 크기
#define CHUNKSIZE (1<<13)            // heap 확장 시 기본 크기 (8KB)
```

2) 기본 연산

```
#define ALIGN(size) (((size) + (ALIGNMENT-1)) & ~0x7)  // 8바이트 정렬
#define MAX(x,y) ((x) > (y) ? (x) : (y))               // 최대값
```

```
#define PACK(size, alloc) ((size) | (alloc))          // size와 할당 여부를 통합 값 생성
#define GET(p) (*(unsigned int *)(p))                 // p가 가리키는 워드 값 읽기
#define PUT(p, val) (*(unsigned int *)(p) = (val))    // p가 가리키는 워드에 값 저장
#define GET_SIZE(p) (GET(p) & ~0x7)                   // p에서 블록 크기 추출
#define GET_ALLOC(p) (GET(p) & 0x1)                   // p에서 할당 여부 추출
```

3) 블록 포인터 연산

```
// 블록 포인터 → 헤더 주소
#define HDRP(bp) ((char *)(bp)-WSIZE)
// 블록 포인터 → 풋터 주소
#define FTRP(bp) ((char *)(bp) + GET_SIZE(HDRP(bp)) - DSIZE)
// 다음 블록 포인터
#define NEXT_BLKP(bp) ((char *)(bp) + GET_SIZE(((char *)(bp) - WSIZE)))
// 이전 블록 포인터
#define PREV_BLKP(bp) ((char *)(bp) - GET_SIZE(((char *)(bp) - DSIZE)))
```

4) Explicit free list 포인터 매크로

```
#define NEXT(bp) (*(void **)(bp))                  // 현재 블록의 next 포인터
#define PREV(bp) (*(void **)((char *)(bp) + WSIZE))  // 현재 블록의 prev 포인터
#define SET_PTR(p, val) (*(void **)(p) = (val))      // 포인터 p에 val 저장
```

- **함수**

1) 기본 함수: mm_init(), mm_malloc(size_t size), mm_free(void *ptr), mm_realloc(void *ptr, size_t size)

2) 서브 루틴 함수

- extend_heap(size_t words): heap 공간 부족 시 추가적인 힙 공간을 요청하고, 해당 공간을 free block으로 설정합니다.
- static void *coalesce(void *bp): 인접한 블록들과의 병합을 수행하고, 병합된 최종 블록을 free list에 삽입합니다.
- static void *first_fit(size_t asize): free list에서 첫 번째로 적합한 블록을 찾는 전략입니다.
- static void *next_fit(size_t asize): 직전 탐색 위치부터 순회하는 전략입니다. 구현 이슈로 해당 프로그램에서는 사용하고 있지 않습니다.
- static void allocate(void *bp, size_t asize): 해당 블록을 요청받은 크기로 할당하고, 남은 부분은 새로운 free block으로 분할하여 리스트에 삽입합니다.

- int mm_check(void): heap consistency를 점검하기 위한 디버깅용 함수입니다. 성능 저하로 인해 현재 프로그램에서는 해당 코드가 주석처리 되어 있습니다.

3) free list 관리 함수

- static void put_free_block(void *bp): free block을 free list의 앞에 삽입합니다. 삽입 시 next/prev 포인터를 갱신합니다.
- static void remove_free_block(void *bp): 해당 블록을 free list에서 제거합니다. Prev/next 포인터를 적절히 연결하여 list 연결을 유지합니다.

4. Special Considerations

성능 향상을 위한 구현 과정을 서술하겠습니다.

시도 1. Implicit list 사용

./mdriver로 성능 테스트 시 40점 정도의 성능 밖에 나오지 않았습니다.

시도 2. Explicit free list 사용

Explicit free list의 경우, 새로운 블록을 할당할 때, 모든 블록을 탐색하지 않고 free 블록만 순회하므로 성능을 높일 수 있습니다. 그 결과 82점으로 성능이 급격히 올랐습니다.

시도 3. Realloc 방식 변경

기존의 항상 새 블록을 malloc하여 동작하던 방식을 크기가 허용되는 경우 같은 블록 내에서 크기를 확장하는 방식으로 변경하였습니다. 그 결과 84점으로 성능 점수가 2점 올랐습니다.

시도 4. Chunck size 조정

Chunck size를 1<<12에서 1<<13으로 조정하여 성능 점수를 2점 높였습니다. 할당 요청이 많은 경우 큰 chunck가 성능에 유리합니다. 하지만 chunck size를 1<<14로 높이자 성능이 4점이나 떨어졌습니다. 이 경우 외부 단편화가 너무 커지게 되어 성능이 줄었다는 결론을 도출했습니다.