System Programming Project 4

Dynamic Allocator

이름 : 박지민

학번 : 20231552

1. 개발 목표

이번 프로젝트의 목표는 libc malloc, free, realloc 함수의 동작을 구현하면서, 메모리 활용도와 실행 속도 측면에서 효율적인 동적 메모리 할동기를 설계·구현하는 것이다. 특히 다음의 세 가지 목표를 달성하고자 했다:

- **정확성:** 모든 테스트 케이스에서 valid = yes를 달성.
- **공간 효율성:** fragmentation을 최소화하여 높은 utilization 확보.
- **처리 속도:** throughput을 높여 성능 지수를 극대화.

2. 개발 내용

본 할당기는 Segregated Explicit Free List 방식을 기반으로 하며, 크기별로 구간을 나눠 free block을 분리 관리하여 탐색 효율을 높이고 fragmentation을 최소화하였다. 주요 설계 및 구현 내용은 다음과 같다.

2.1 설계 개요

- **Segregated Free Lists (seg_free_lists)**
    - 총 10개의 크기 구간(Class)을 정의하여, 크기 범위에 따라 free block을 나눠 보관.
    - 각 리스트는 pred, succ 포인터로 연결된 explicit doubly linked list 형태로 구현.
- 블록 구성 (Block Layout)
  모든 블록은 다음과 같은 구조를 따른다:
    - [Header | Pred | Succ | Payload | ... | Footer]
    - 할당된 블록: Header + Payload + Footer
    - Free 블록: Header 뒤에 pred, succ 포인터 영역을 추가로 사용함
    - Header/Footer는 블록 크기 및 할당 여부 비트를 포함

2.2 전역 변수 및 매크로

- heap_listp: 힙의 시작 주소를 가리키는 포인터
- seg_free_lists[10]: 크기 구간별 free list를 저장하는 배열
- #define을 통한 정렬, 접근 편의 매크로 정의:
    - HDRP, FTRP, NEXT_BLKP, PREV_BLKP, GET_SIZE, GET_ALLOC 등

2.3 주요 함수 설명

1. mm_init()

- 힙 초기화 및 prologue/epilogue 블록 생성
- 초기 힙 크기만큼 extend_heap()으로 free block 확보
- seg_free_lists 배열을 NULL로 초기화

2. mm_malloc(size_t size)

- 요청 크기를 ALIGN() 및 최소 블록 크기(16B) 이상으로 조정
- class_index()로 적절한 free list 선택 후 find_fit() 수행
- 적절한 block이 없으면 extend_heap() 호출
- 선택된 블록은 place() 함수로 할당 처리
- 특이 처리: 요청 크기가 정확히 DSIZE일 경우, 오버프로비저닝을 통해 4 * DSIZE 크기 block을 먼저 탐색하는 예외 로직 구현

3. mm_free(void *ptr)

- 해당 블록의 header/footer를 0으로 marking
- coalesce() 함수를 호출해 인접 free block들과 병합
- 병합된 결과 블록을 insert_free_block()으로 free list에 삽입

4. mm_realloc(void *ptr, size_t size)

- 축소 요청 시 분할 가능한 경우 분할 후 나머지 영역을 free 처리
- 다음 블록이 free이거나 epilogue일 경우, in-place 확장 시도
- in-place가 불가능할 경우 새 블록을 할당하고 데이터 복사 후 기존 블록 free

5. extend_heap(size_t words)

- mem_sbrk로 힙을 확장하고 새 블록 생성
- 생성된 block을 free로 marking 후 coalesce로 병합하여 free list에 추가

6. coalesce(void *bp)

- 이전/다음 블록의 할당 상태에 따라 4가지 케이스로 병합
- 병합 전 해당 block들이 free list에 있다면 remove_free_block()으로 제거
- 병합 후 결과 블록을 insert_free_block()으로 삽입

7. find_fit(size_t asize)

- class_index(asize) 이상의 구간에서 순차 탐색
- 다음 조건 중 하나에 따라 바로 return:
    - perfect match (bsize == asize)
    - split 손실이 작음 (bsize - asize <= DSIZE)
- 일반적인 경우엔 best-fit 방식으로 최소한의 손실 블록 선택

8. place(void *bp, size_t asize)

- 블록을 할당하고 남는 공간이 4 * DSIZE 이상이면 분할
- 분할된 나머지 블록은 free로 만들고 insert_free_block 호출

2.4 보조 함수

class_index(size_t size)

- 16B 이하부터 4096B 초과까지 총 10개 구간으로 분류
- 작은 요청은 빠르게 처리되고, 큰 요청은 탐색 범위를 줄여 성능 향상

insert_free_block(bp)

- 해당 블록을 크기 구간에 맞는 리스트 맨 앞에 삽입
- pred 및 succ 포인터 초기화 처리 포함

remove_free_block(bp)

- 해당 블록이 속한 리스트에서 제거
- 리스트 맨 앞, 중간, 마지막 블록일 경우를 모두 고려

mm_check()

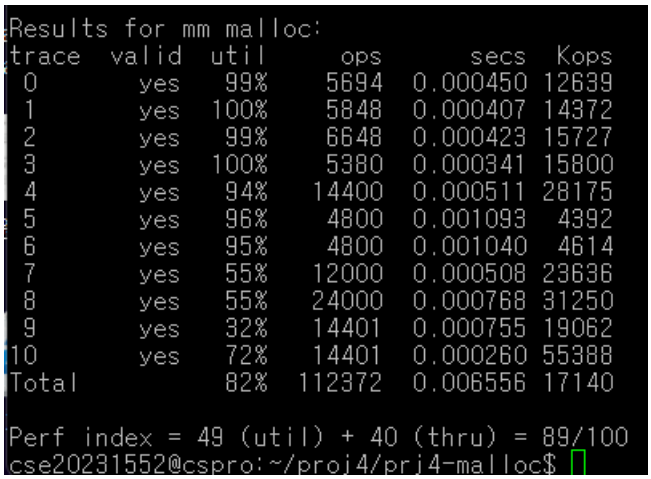
- free block이 실제로 free인지, header/footer가 일치하는지 등 검증
- free list 내 consistency 확인 및 디버깅에 활용.

3. 구현 결과

본 할당기는 모든 테스트 케이스에서 valid = yes를 만족했으며, 성능 점수는 약 89점으로 확인되었다.

3.1 정량적 성능 결과

아래는 ./mdriver -v 명령어를 통해 측정한 trace별 결과이다.



3.2 성능 향상을 위한 구체적 노력

(1) 할당 전략 개선: First-fit 제한 탐색 기반 Best-fit 하이브리드

Best-fit은 작은 블록 낭비를 줄여 util 점수는 높았지만 throughput이 떨어졌다. 반면 First-fit은 처리량은 높지만 불필요한 큰 블록 사용으로 util 점수가 낮아지는 문제가 있었다. 따라서 split 손실이 작을 경우(≤ DSIZE 차이)에는 **First-fit**처럼 동작하게 하고, 그 외에는 **Best-fit**을 사용하여 두 전략의 장점을 절충한 탐색 방식을 find_fit()에서 구현하였다.

(2) Segregated Free List 구조의 활용

블록 크기별로 10개의 클래스에 구분하여 free list를 분리하였고, class_index()를 통해 적절한 크기의 free list부터 탐색하도록 구현하여 탐색 시간을 줄이고 throughput을 향상시켰다.

(3) 블록 분할 조건 최적화

place() 함수에서 (csize - asize) >= 4 * DSIZE일 경우에만 분할하도록 하여, 너무 작은 블록이 생기는 것을 방지하였다. 이 임계값은 실험을 통해 조정되었으며, split으로 인한 fragmentation을 줄여 utilization을 향상시키는 데 도움이 되었다.

(4) Realloc 최적화

mm_realloc()에서는 in-place 확장 가능 시 coalesce를 호출하지 않고 직접 header/footer만 갱신하고 free list 조작을 피했다. 특히, epilogue 직전 블록일 경우 mem_sbrk()를 호출하여 in-place 확장을 시도해 불필요한 메모리 복사와 추가 할당을 줄여 throughput을 개선했다.

(5) 디버깅 및 성능 실험

remove_free_block() 오류를 방지하기 위해, 블록 상태를 확인하고 할당된 블록을 제거하려는 경우 명시적으로 exit(1) 하도록 설정하여 문제를 조기에 발견할 수 있도록 했다.

다양한 조합의 CHUNKSIZE, place 임계값, 클래스 수에 대한 실험을 수행했지만, 최종적으로 CHUNKSIZE = 512, CLASS_NUM = 10, split 임계값 = 4 * DSIZE가 가장 안정적인 성능을 보였다.

(6) 성능 향상 시도 및 포기한 방법

PREV_ALLOC을 사용하는 header-only 방식으로 전환하려다 coalesce() 시 오류가 빈번히 발생하고, 디버깅 비용이 커져 최종 구현에서는 유지하지 않았다.

find_fit()에서 탐색 블록 수를 제한하는 제한 First-fit도 시도했지만, 오히려 utilization 점수가 감소해 제외하였다.

place()의 임계값을 낮춰 더 자주 split하려 했으나, 작은 block들이 쌓이며 util 점수가 오히려 하락해 원래 임계값을 사용하였다.

3.3 결론

최종 코드는 간결한 구조를 유지하면서도 실험적으로 조정된 매개변수와 알고리즘 선택을 통해 성능을 최대치로 끌어올린 구현이다. 결과적으로 성능 지수 89점을 획득하였고, 이는 실용성과 안정성 측면에서 가장 균형 잡힌 결과라고 판단되어 최종 제출한다.