M1522.000800 System Programming
Fall 2019

System Programming Memory Lab Report

Jinje Han
2018-18786

1. Introduction

이번 Lab에서는 컴퓨터에서 프로세스를 실행하는 과정에서 heap의 메모리를 동적으로 할당하고 관리하는 함수들을 직접 구현한다. 메모리를 관리하는 방법에는 여러 가지가 있는데, 각 방법을 직접 구현하면서 이들의 차이점을 알아보고, 주어진 test input들을 넣어서 테스트해 봄으로써 각 방식의 장단점을 익힐 수 있을 것이다.

2. Important Concepts

프로그램이 원하는 크기의 메모리를 할당해 달라고 요청하면, heap을 관리하는 라이브러리 또는 모듈은 heap의 공간 중에서 사용 가능한 공간을 요청한 크기만큼 잘라서 프로그램에게 할당해 준다. 이때 각각의 요청에 대해 할당해 준 메모리 공간을 block이라고 한다. 할당된 block들은 서로 중첩되지 않아야 한다. 반면 할당이 해제된 공간들은 굳이 구분할 필요가 없어 하나로 합쳐 관리하는 것이 더 효율적이다. 이때 할당이 해제된 메모리를 합치는 것을 coalescing이라고 한다.

각각의 block을 할당하고 coalescing을 수행하는 방식은 여러 가지가 있을 수 있다. 이번 lab에서 나는 두 가지 방식을 사용해 보았다. 첫 번째는 Implicit list 방식이다. 이 방식에서는 block이 header, payload(+padding), footer로 이루어져 있다. header와 footer에는 이 block이 지금 할당되어 있는지, 이 block의 크기는 얼마인지가 기록된다. payload 부분은 실제로 프로그램에게 할당되는 부분이고, padding은 정렬과 같은 요구사항이 있을 때 이를 충족시키기 위해 존재하는 추가적인 공간이다. 두 번째는 segregated list 방식이다. 이는 크기가 서로 비슷한 block끼리 linked list로 묶어서 관리한다.

각각의 방식은 두 가지 평가 요소를 갖는다. 하나는 space utilization으로, heap 영역을 얼마나 효율적으로 사용하고 있는지에 대한 것이다. Heap의 크기에 비해 프로그램에 할당된 부분이 적을 때, 특히나 할당이 해제된 영역을 제대로 활용하지 않으면 space utilization이 낮게 평가된 것이다. 다른 하나는 throughput으로, 초당 몇 개의 할당/할당해제 연산을 수행할 수 있는지를 말한다. 할당 요청들에 대해 heap의 적절한 빈 공간을 최대한 효율적으로 할당하여 space utilization을 높인다고 해도, 그 빈 공간을 찾는 데에 시간이 오래 걸린다면 throughput이 낮게 측정될 것이다. 이번 lab의 test data를 보니 랜덤으로 생성된 query를 포함하여 다양한 형태의 data가 있으므로 throughput을 잘 측정할 수 있을 것이다.

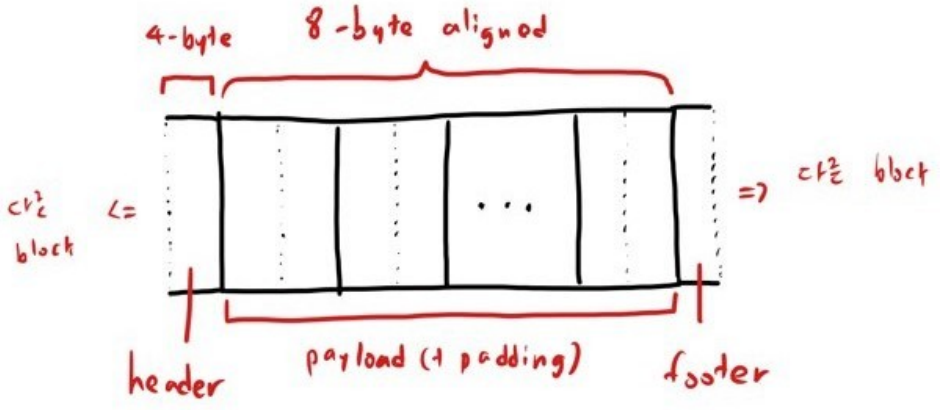
구현해야 하는 함수들은 총 4가지다. 첫째, malloc() 함수는 할당하고 싶은 메모리의 크기를 받고 그만큼의 메모리를 할당한 다음 해당 block의 시작 지점을 가리키는 pointer를 반환한다. 이때 고려해야 할 점은 할당해 주는 메모리의 크기가 특정 숫자의 배수에 맞춰서 align되어야 한다는 것이다. 이번 lab에서는 alignment 값이 8로 되어 있으므로, 어떤 크기의 메모리를 요청하면 8의 배수 주소에서 시작하는, 요청한 값보다 크거나 같은 8의 배수 크기의 메모리를 할당해 주어야 한다는 것이다. 둘째, free() 함수는 어느 할당된 block을 가리키는 pointer를 받아서 block의 할당을 해제한다. 이때 input으로 들어가야 하는 pointer는 malloc()에서 반환한 것과 같이 해당 block의 시작 지점을 가리키고 있어야 한다. 셋째, init() 함수는 본격적으로 할당/할당 해제를 하기 전에 heap의 기본적인 상태를 마련해 주는 코드이다. 이는 어떤 메모리 관리 방식을 사용했느냐에 따라 달라지므로 아래

에서 더 자세히 말하겠다. 마지막으로 exit() 함수는 프로그램을 종료하기 전에 아직 할당이 해제되지 않은 block들을 전부 찾아서 할당 해제하는 역할을 한다.

3. Implicit list

앞서 말했듯 Implicit list 방식에서는 block이 header, payload(+padding), footer로 이루어져 있다. Header와 footer는 각각 4byte인데, 이 안에 block의 크기 정보만 담으면 block의 크기는 항상 8의 배수이기 때문에 하위 bit 3개가 0으로 고정된다. 따라서 가장 하위 bit를 이 block이 할당되었지 아닌지 기록하는 용도로 사용한다. 이 둘은 block의 양쪽 끝에서 에서 동일한 내용을 담고 있는데, 양쪽으로 인접한 block들에서 이 block의 크기 정보를 알 수 있기 때문이다.

하나의 block은 alignment를 아래 그림과 같이 payload는 8-byte align, 그 양쪽에 header와 footer를 각각 4byte씩 붙이기로 했다. 즉, 한 block의 header(또는 footer)와 그 옆 block의 footer(또는 header)가 모여서 8byte를 만드는 방식이다. 이렇게 하면 header와 footer 각각이 8byte씩 사용하는 것에 비해 조금이라고 메모리 사용량을 줄일 수 있다.

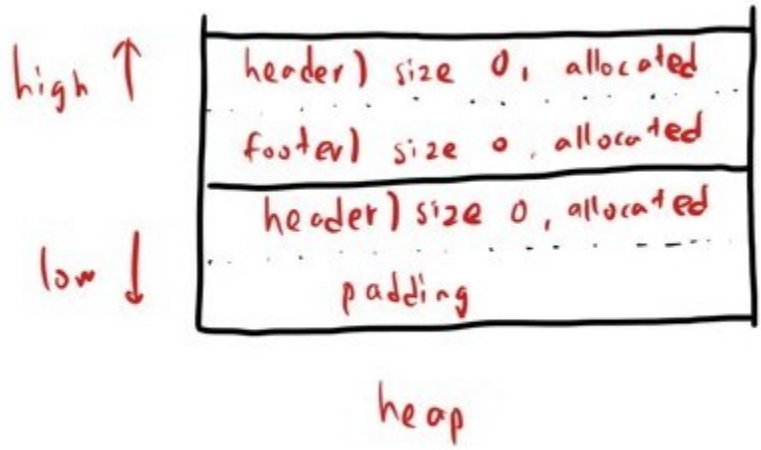


△ block design(left-right)

Block을 이렇게 관리하기로 정했다면, 일단 주요 함수들을 무작정 작성하기 시작한 것이 아니라 이 코드 전반에 거쳐 자주 사용하게 될 함수들을 먼저 구현했다. 우선 payload가 시작하는 부분의 pointer를 주면 header pointer, header, footer pointer, footer, block size, allocation or deallocation을 반환하는 함수를 각각 만들었다. 이 함수들은 헷갈리기 쉬운 bitwise 연산들을 하지 않아도 되게 할뿐더러 이 방식 말고 다른 방식으로 메모리를 관리하는 코드를 짤 때도 사용할 수 있다. 다음으로 payload 시작부분의 pointer와 함께 int값을 주면 그 block의 크기를 변경시켜 주는 함수와, 0/1을 주면 그 block의 allocation 상태를 변경시켜 주는 함수를 만들었다. 이 함수들을 통해 block의 상태를 쉽게 변경할 수 있다. 마지막으로 coalesce관련 함수 2개이다. coalesce_front()는 인자로 받은 pointer가 가리키는 block이 unallocated이고 이 block 바로 앞에 있는 인접한 block이 있을 때, 이 두 block을 하나로 합치고 block의 payload를 가리키는 pointer를 반환하거나, 합치지 못하면 인자로 받은 pointer를 그대로 반환한다. coalesce_back()은 같은 일을 뒤쪽에 인접한 block에 대해 수행하고, 아무것도 반환하지 않는다는 것을 제외하면 같은 함수이다. 이렇게 만들어 놓으면 coalescing 과정을 coalesce_back(coalesce_front(pointer)) 한 줄로 축약할 수 있다.

이제 필요한 함수들을 구현하기 시작했다. 우선 가장 처음에는 init()함수를 사용하여 초기 구조를 설정했는데, 이는 아래와 같다. 맨 위쪽 header)size 0, allocated는 항상 heap의 top에 고정한 채로, 두 크기 0짜리 block 사이에 block들을 만드는 형태로 구현이

되었는데, 이러한 구조는 coalescing을 할 때 heap의 boundary를 고려하지 않아도 되는 장점이 있다(allocated된 block은 coalescing 대상이 아니므로)



△ Heap의 초기 상태

malloc() 함수는 기본적으로 현재 heap의 block들을 가장 아래서부터 순서대로 훑어 가며, 현재 요청 받은 크기 이상이면서 unallocated인 block을 찾는다. 이때 한 block의 header를 읽으면 이 바로 다음 block(의 header)가 어디에 있는지를 바로 알 수 있으므로 offset만큼 jump하는 방식으로 진행한다. Block을 하나라도 찾았으면 그 block을 알맞게 자르고 하나의 state를 위에서 정의한 함수를 사용해 allocated로 바꾼다. 그리고 allocated인 block의 시작 주소를 반환한다. 만약 그러한 block이 없으면 heap을 확장해서 새로운 block을 만들어 할당해야 하는데, 이 때도 두 가지 경우가 있다. Heap으 가장 위에 allocated된 block이 있다면 요청 받은 크기만큼 heap을 늘리고 그 block을 할당한다. 반면 가장 위에 unallocated된 block이 있다면 heap을 그만큼 적게 늘릴 수 있으므로 약간의 bitwise 연산을 직접 하여 처리해 준다.

free()함수는 두 가지 일을 하다. 우선 인자로 받은 pointer를 직접 정의한 함수에 넣어 state를 unallocated로 바꾼다. 다음으로 free된 block에 인접한 unallocated block이 있다면 coalesce를 진행하는데, 이는 위에서 소개한 방식으로 한 줄만 적으면 된다. 마지막으로 exit()함수는 malloc()에서 진행한 방식으로 모든 block들을 순회하며 아직 allocated인 block이 있다면 deallocate해 준다.

4. Segregated List

Implicit List를 성공적으로 구현한 다음에는 throughput을 늘리기 위해 Segregated list 방식을 사용한 구현을 시도해 보았다. Segregated List에서는 각각의 list에 들어가는 block들의 크기를 정해야 하는데, 4btye로 표시 가능한 범위 안에서 malloc 요청값이 들어오므로 2의 거듭제곱으로 끊기로 했다. 기본적으로 8의 배수만큼 할당해 줄 수 있으므로 8byte짜리 칸을 2^0=1개, 2^1 ~ 2^1개, 2^0 ~ 2^2개, ... 2^{27} ~ 2^{28} 개만큼 가지고 있는 block들끼리 묶어서 하나의 list로 관리하기로 했다.

여러 개의 list를 하나의 heap에 넣어야 하므로 각각의 list의 원소들은 인접하게 위치할 수가 없다. 따라서 하나의 block에서, 동일한 list의 다음 block을 찾기 위해 block에 offset을 2개 추가했다. 어떤 block을 가리키는 pointer에, 각각의 offset을 더하면, 동일한 list의 바로 앞과 뒤에 있는 block에 접근할 수 있게 된다. 이외의 block 구조는 implicit list과 동일하다.

이 구조에서 특히나 복잡한 점은 coalescing이다. 서로 물리적으로 인접한 두 block이 모두 unallocated이면 하나로 합치는 것은 동일하지만, 이 과정에서 linked list의 offset들

을 잘 처리해 주어야 하기 때문이다. 이를 위해 함수를 몇 개 만들어 주었다. 우선 cut_off()라는 함수인데, 이는 block을 가리키는 pointer를 주면 이 block을 linked list 중간에서 자르고 이 block 앞뒤에 있던 block들을 서로 이어 준다. 이때, 이 block이 list의 마지막 원소였다면 추가적인 처리를 따로 해 주어야 한다. 그리고 set_in()이라는 함수가 있는데, 이 함수를 block으로의 pointer와 block의 size를 주면 자동으로 적합한 linked list를 찾아 맨 뒤에 이 block을 넣어 준다. 이 두 함수를 통해 linked list를 잇고 끊는 구조였다. coalescing 함수들도 이에 맞게 적절히 변형해 주었다.

init() 함수에서는 heap의 가장 아래 부분에, 각 linked list로 통하는 offset의 배열을 넣어 주었다. 어떤 block이 주어졌을 때, 이 block의 크기로부터 몇 번 linked list를 사용해야 하는지도 알려주는 기능 또한 만들었다.

그러나 이러한 함수를 구현하지만 코드의 debugging에 실패하여 실제로 구현을 끝마치지 못하였다. 현재 gitlab에 올라와 있는 코드는 implicit list의 코드이다.

5. Conclusion

이 lab을 하기 위해 메모리를 관리하는 다양한 방식을 알아보았는데 각각의 방식의 장단점을 파악할 수 있었다. 이를테면 red-black tree를 사용하는 방법도 있었는데, 구현이 매우 어려운 대신 throughput이 매우 크다고 한다. 전반적으로 throughput과 구현 난이도가 반비례하는 양상이었다.

이번 lab은 다른 lab들에 비해 코드를 짜는 것이 특히나 어려웠는데, 이 때문에 계획한 구현을 최종적으로 완수하지 못한 것에 대한 아쉬움이 크다. 이외에도 splay tree 등의 방식을 사용할 수도 있다고 들었는데 시간 관계상 구현하지 못한 것이 아쉽다.