중간 레벨 과제

간단한 메모리 관리자 - 계획 보고서

제출일: 2022.10.24	전공: 컴퓨터공학과			
어드벤처디자인_02	학번	20190602	20190622	20190788
담당 교수: 김경수 교수님	이름	설진영	손현락	이근탁

목 차

Ⅰ. 프로젝트의 목표

동적할당이란 프로그램으로부터 요청을 하면 OS의 메모리 관리 메커니즘에 의해 컴퓨터의 메모리 중 사용 가능한 부분에서 요청된 크기만큼을 사용할 수 있도록 하는 작업이다. 이를 이용하여 개발자는 상황에 따라 원하는 크기만큼의 메모리를 할당받으며 경제적인 방법으로 컴퓨터의 자원을 효율적으로 사용할 수 있다. 또한, 언제든지 할당받은 메모리를 해제하고, 이미 할당된 메모리라도 언제든지 크기를 조절하는 등 유동적으로 메모리를 관리하며, 자원의 낭비를 최대한 줄일 수 있다.

이번 프로젝트에서는 C언어에서의 메모리 동적할당, 해제와 관련된 함수 malloc()과 free()의 전반적인 메커니즘을 모방하는 mymalloc(), myfree() 함수를 구현하여 메모리 관리를 수행하는 간단한 프로그램을 만들고자 한다. 직접 OS에게 요청하는 방법으로 기능을 구현하는 것이 아니라, 동적할당의 원리를 이해할 수 있는 시뮬레이션 프로그램을 만드는 것을 기본 목표로 한다.

Ⅱ. 문제정의

본 과제의 문제정의는 다음과 같다.

1. mymalloc() 함수와 myfree() 함수를 작성하고 main 함수에서 그림의 예와 같이 mymalloc와 myfree를 호출하여 수행되는 결과를 출력하라.

2. 주어진 예 이외의 다른 시나리오 하나를 더 만들어 결과를 출력하라.

3. 주어진 예시에서는 할당 시 가장 앞에 있는 덩어리 크기가 충분하면 바로 잘라 할당하고, 그렇지 않으면 차례로 리스트 노드를 검사한다. 하지만 주어진 덩어리가 특정 비어있는 공간에서 크기가 딱 맞는다면, 첫 번째 덩어리를 나누는 것보다 그 공간을 대신 할당하는 방법은 어떤지 두 방법을 모두 구현해보고 장단점을 논해라.

Ⅲ. 아이디어 / 접근방법

실제 메모리 할당, 메모리 해제 함수 구현의 제한사항

어느 운영체제에서든 자신 들이 사용하는 메모리에 대한 보호 기법들이 존재한다. 그중 하나가 ASLR(Address Layout Randomization) 보호 기법, 직역하면 메모리 주소 난독화이다. 해당 보호 기법은 스택 영역의 시작 주소와 메모리가 할당이 될 힙 영역의 시작 주소를 아래와 같이 프로그램이 실행될 때마다 바꿔주는 역할을 한다. 즉, 정말로 malloc 및 free 함수의 동작을 구현하기 위해서는 프로그램의 시작 때마다 해당 주소를 알아야 할 필요가 있는데 이는 취약점이 있는 프로그램에서만 가능할뿐더러 이를 알아내는 방법도 쉬운 방법이 아니기에 일반 프로그램의 변수들이 매핑될 스택 영역에서 마치 실제로 동작하는 것처럼 보이도록 구현을 할 것이다.

```
buf_stack addr: 00000007BEAFF9F8
buf_heap addr: 00000195DEAAE3A0
main addr: 00007FF761EF1258
```

```
buf_stack addr: 000000A8E9D3F908
buf_heap addr: 0000020079D93230
main addr: 00007FF761EF1258
```

```
buf_stack addr: 00000092ED79F928
buf_heap addr: 00000267F6441510
main addr: 00007FF761EF1258
```

```
buf_stack addr: 000000C9453DFA68
buf_heap addr: 0000020A5B7B1510
main addr: 00007FF761EF1258
```

메모리 관리 시뮬레이션 시나리오

따라서 실제로 OS에게서 메모리를 할당받는 것을 보여주는 프로그램이 아닌, 동적할당, 해제의 기본적인 메커니즘을 시뮬레이션하는 프로그램을 만든다. 우선 시뮬레이션을 체험할 사용자에게 시뮬레이션에서 사용할 최대한의 메모리 공간을 입력받아 일종의 메모리 공간을 설정하고 메모리 공간을 할당할지 해제할지 추가적인 요청 사항을 받아서 일종의 메모리 할당, 해제 작업을 수행한 후, 그 결과인 메모리 할당 상태를 사용자에게 출력하여 사용자에게 하여금 다소 어려울 수 있는 간단한 동적할당, 해제에 관한 전반적인 이해도를 높이는 것에 초점을 맞춘다.

실제로, 이 시뮬레이션을 만들기 위하여 후술할 자료구조를 이용해 구현을 진행할 계획이기 때문에 훗날 코드를 살펴보면 프로그램이 구동되는 동안에도 실제 동적할당과 해제가 이루어지고 있다는 사실을 확인할 수 있게 될 것이다.

Ⅳ. 방법론이나 구현 언어

방법론

우선 메모리 관리와 관련된 노드를 구성하는 구조체를 정의한다. 구조체 안에는 할당되어있는 해당 메모리의 시작 주소와 메모리의 공간이 포함된다. 공간이 추가되거나 삭제될 때 연결되거나 해제할 수 있어야 하므로 다음 구조체, 이전 구조체를 가리키는 포인터 변수도 존재한다. 다음으론 전체 메모리 하나를 의미하는 클래스(객체)를 정의한다. 해당 클래스의 맴버 변수에는 할당 가능한 메모리 공간에 대한 정보를 나타내는 노드 들을 가리키는 available 포인터 변수와 할당되어 사용 중인 데이터 공간에 대한 정보를 나타내는 노드 들을 가리키는 malloced 변수가 있다. 해당 클래스의 멤버 함수에서 동적할당, 해제 과정이 진행되는 동안 앞서 만들었던 메모리 공간의 정보를 가진 두 맴버 변수들을 수정하며 메모리 관리가 수행될 것이다.

구현 언어

전반적으로 시뮬레이션을 구동하기 위해 필요한 전체적인 메모리 공간 하나가 있고, 그 메모리 공간에 대해서 필요한 크기만큼 할당 및 해제하는 방식으로 프로그램이 작동한다 볼 수 있다. 전체 메모리 공간을 객체로 보고, 그 객체의 맴버 함수를 통해 메모리 할당, 해제 그리고 메모리에 대한 정보를 조회하는 형식으로 프로그램을 구성하는 것이 적절하다고 판단하였다.

메모리 속에서 메모리의 할당, 해제가 이루어지고, 데이터끼리의 연산을 하는 과정이 필요하기 때문에 절차적 프로그래밍을 활용하여 문제를 해결하기는 무리가 있다고 보았다. 포인터를 통해서 값을 전달하는 과정에서 포인터를 잘못 사용하는 등의 경우를 방지하고, 프로그램을 조금 더 직관적으로 개발하기를 원했다. 또한, 포인터와 구조체를 통해 자료구조를 표현할 수 있고, 사용자가 직접 명시적으로 동작 할당을 수행해야 하는 언어에서 해당 프로그램이 잘 적용될 수 있다고 판단하였다.

따라서, C 기반의 객체지향 프로그래밍 언어인 C++ 언어를 사용하여 해당 프로그램을 구현하기로 결정했다. 문제정의의 1번인 main 함수에서 myalloc()와 myfree()를 단순히 호출하여 수행하는 것이 아닌, 사용자가 동적할당을 선택했을 때 메모리 공간의 myalloc() 메소드가, 사용자가 동적 해제를 선택했을 때 메모리 공간의 myfree() 메소드가 실행되는 방식으로 구현이 진행될 것이다.

Ⅴ. 추가하고 싶은 기능

현재 메모리의 상황을 시각적으로 보여주는 함수

사용자의 입장에서 자신이 메모리를 할당하거나 해제하는 행위를 반복하다 보면 현재 메모리의 상태를 보고 싶은 마음이 들기 마련이다. 이를 위해 현재 메모리의 상태를 보다 가시적이고 직관적으로 보여줄 수 있는 출력함수를 구현해볼 것이다.

ex) 100의 크기로 설정되어있는 메모리에서 10~19번, 20~29번, 90~94번이 myalloc으로 할당되어있다고 하면 각 메모리를 1단위로 나뉘어 해당 메모리에 할당이 되어있다면 *을 채워 넣고 비어있다면 해당 칸도 비우는 방식이다.

잔여 메모리의 양과 전체 메모리의 할당 크기를 반환하는 함수

사용자가 사용하는 메모리 공간에 대한 다음 세 가지 정보를 얻을 수 있는 함수를 작성해볼 것이다.

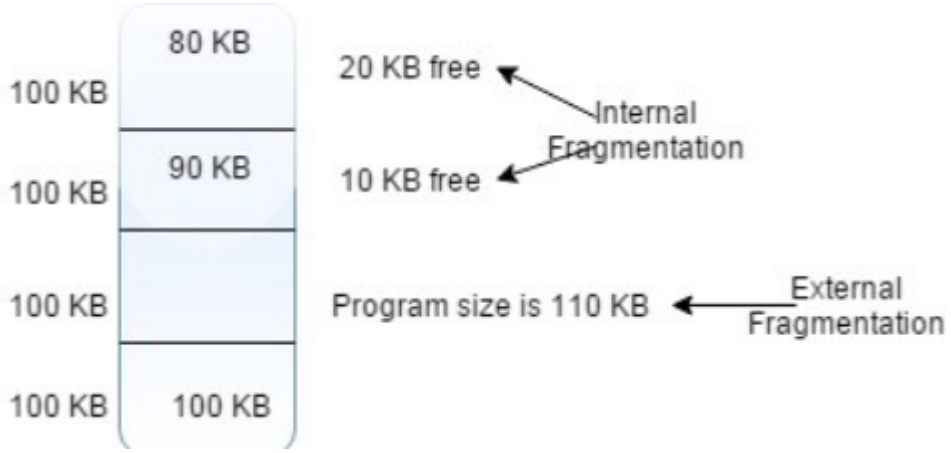
첫 번째, 사용자가 프로그램을 시작할 때 최초로 지정했던 총 메모리 크기를 int 형을 반환한다.

두 번째, 사용자가 실제로 할당한 크기를 제외한 현재 가용한 공간을 int 형을 반환한다.

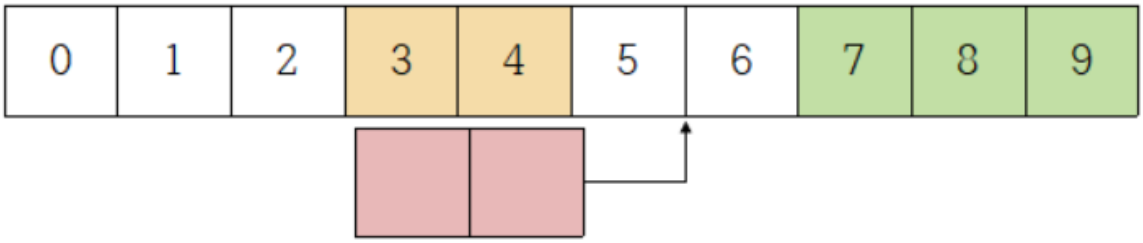
세 번째, 현재 메모리가 가득 차 있는지 판별하여 Boolean 형을 반환한다.

외부 단편화를 해소할 수 있는 방법

메모리를 관리하는 차원에서 '단편화'라는 문제점이 있다. 이러한 단편화는 내부단편화와 외부 단편화로 나눌 수 있다. 먼저 내부단편화는 메모리에 적재될 어느 프로세스가 필요로 하는 공간이 단위가 되는 용량보다 작을 때 그 offset만큼 메모리 낭비가 일어나는 것을 의미한다. 밑의 그림에서는 각 100KB씩 적재할 수 있는 메모리 단위에 80, 90KB의 프로세스가 적재되어 각 20, 10KB씩 낭비되었다. 다음으로 외부 단편화는 밑의 그림에서 총 130KB가 남아있음에도 110KB의 프로세스를 담을 '연속적인' 공간이 없기에 적재하지 못하는 것처럼 특정한 프로세스를 실행할 충분한 용량이 있음에도 실제로는 실행하지 못하는 문제점이다.



이러한 외부 단편화를 최소화할 수 있는 방법에는 이러한 방법이 있다. 예를 들어 다음과 같이 100의 크기로 정해진 메모리가 있고 한 칸당 10의 크기로 묶어놓았다고 하자. 그리고 30~49, 70~99번지에 메모리가 할당되어있다고 하자.



이 상황에서 20 크기의 메모리를 할당한다고 하면 제일 앞쪽에 할당하는 법도 있고, 50~69번지에 할당하는 방법이 있다. 이 중에서 나중에 있을 메모리 할당을 위해서 딱 20만큼 남은 50~69번지에 메모리를 할당하는 방법이다. 즉, 메모리를 할당할 수 있는 범위 내에서 메모리를 적재했을 때 남은 공간이 가장 적을 곳에 할당하는 것이다. 이러한 방법을 'Best-fit'이라고 하며 해당 기능을 구현해볼 것이다.

Ⅵ. 참고문헌

https://m.blog.naver.com/PostView.naver?isHttpsRedirect=true&blogId=qbxlvnf11&logNo=221367010754