한국산업기술대학교 게임공학과

포트폴리오

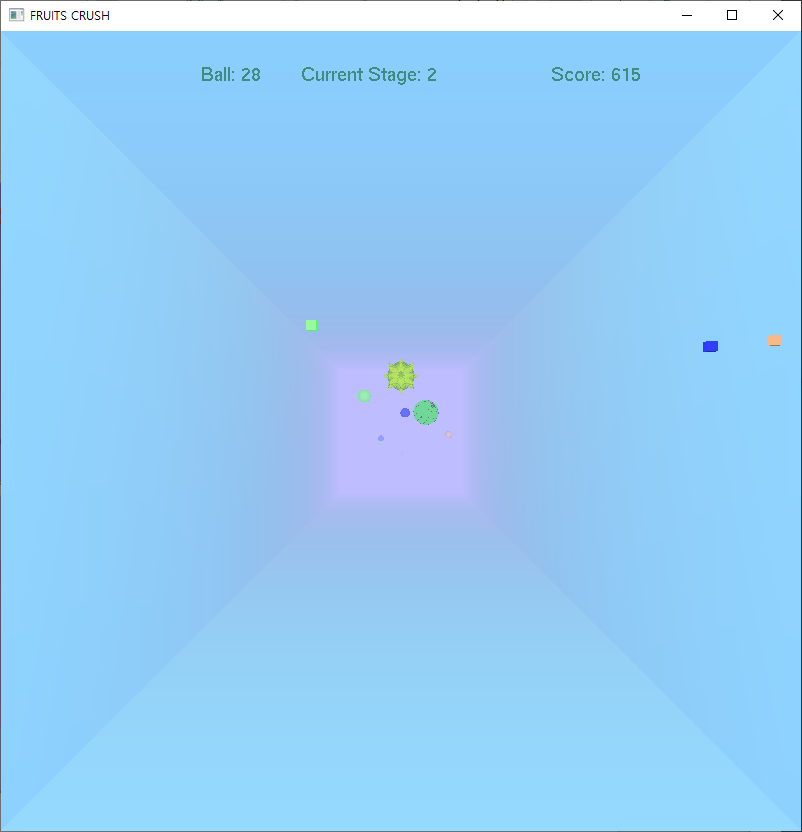
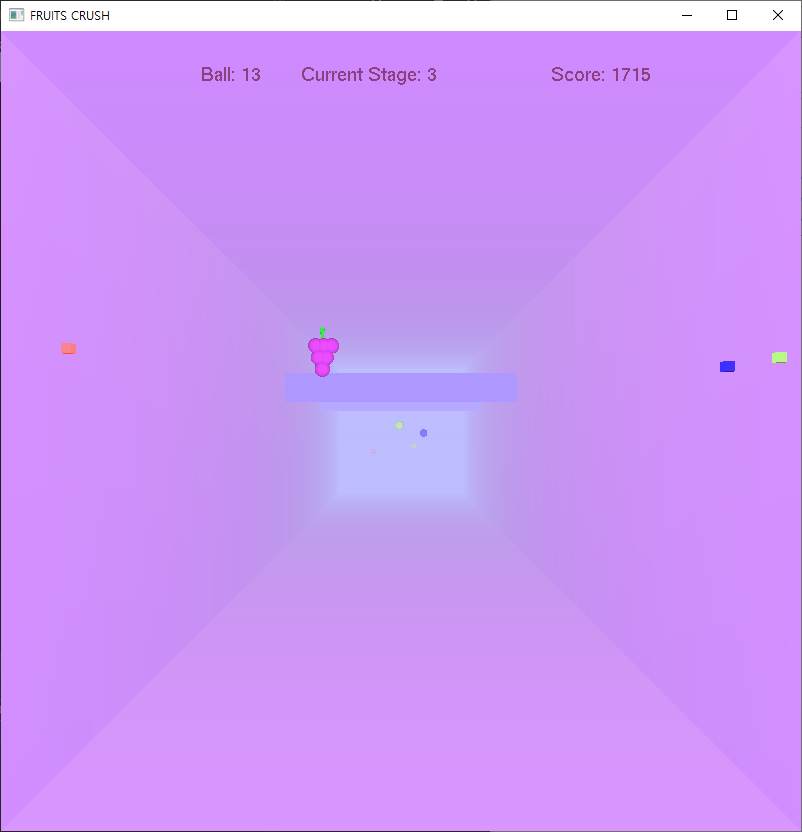
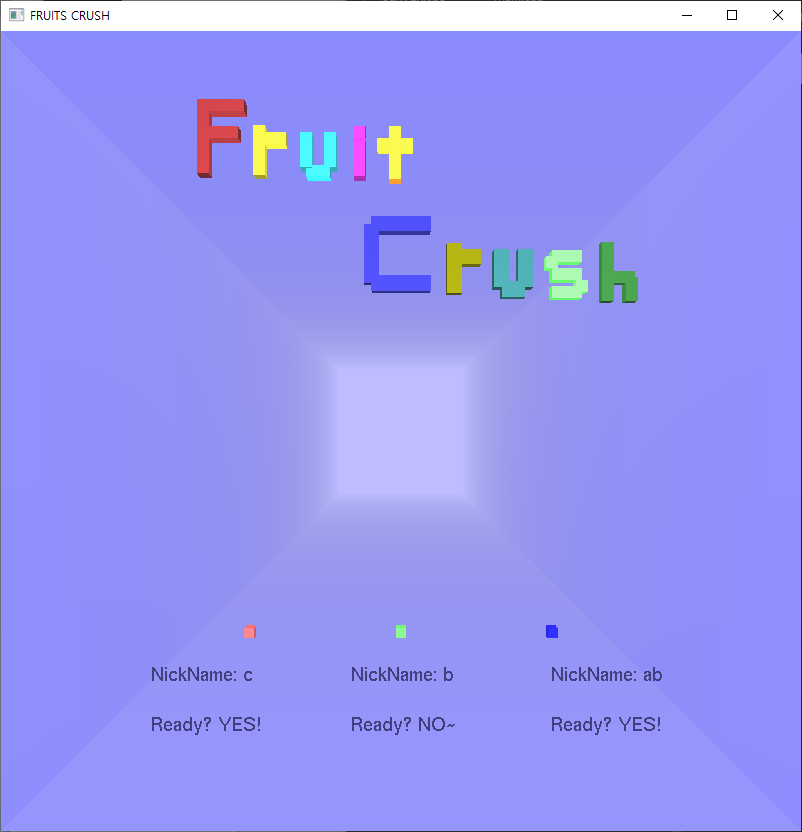
서버 프로그래밍 부분

구태균

2020.11.13

목차

1. [게임] Fruit Crush
2. [게임] Alone Dungeon
3. [논문] Implement Lock-Free shared\_ptr and weak\_ptr
4. **[게임] Fruit Crush**



**기본 설명**

사용 언어: C++11

개발 환경: Visual Studio 2017 64bit

장르: 3인 협동 슈팅

제작 인원: 3인 (서버 – 동기화 [본인], 서버 – 통신, 클라이언트)

제작 기간: 약 2개월

목표: 윈도우 소켓, mutex, UDP와 TCP를 활용한 협동 슈팅게임 제작

**세부 설명**

1. 게임

Fruit Crush는 3인용 협동 슈팅게임으로 3명의 플레이어가 공유하는 공을 모두 소모되지 않도록 장애물을 부수며 점수를 획득하는 게임입니다. 플레이어가 접속한 뒤 닉네임을 입력하면 로비에 접속되며 모든 유저가 준비 완료되면 게임이 시작됩니다. Fruit Crush는 총 3라운드가 있으며, 라운드가 올라갈수록 장애물이 좌우로 이동하고 더 빠른 속도로 접근해 장애물을 맞추기 어려워집니다. 공의 수가 0이되거나 3라운드를 클리어하면 게임이 종료되며, score를 통해 획득한 점수를 확인할 수 있습니다.

1. 서버

Fruit Crush의 서버는 총 3종류의 스레드를 이용합니다.

1. Update thread: 30프레임에 맞도록 시간을 측정한 뒤 Update()를 실행한다.

* 업데이트는 장애물 생성, 장애물 위치 갱신, 공의 위치 갱신하고 장애물과 공의 충돌여부를 검사해 공의 수를 증가시킨다.

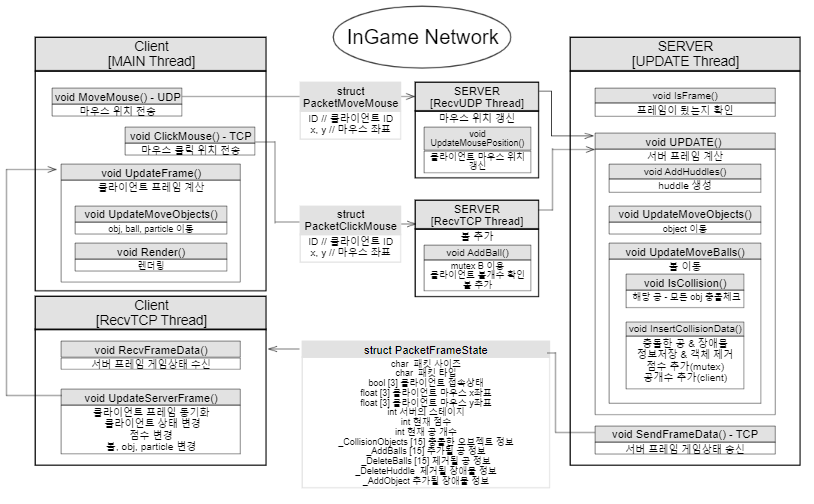
1. Recv UDP thread: 클라이언트가 마우스를 움직일 때 마다 클라이언트의 마우스 좌표를 갱신하고 다른 클라이언트에 마우스 좌표를 전송한다.

* 각 클라이언트는 모든 클라이언트의 마우스 위치를 볼 수 있다.

1. Recv TCP thread: 클라이언트가 마우스를 클릭할 때 마다 공의 개수를 줄인 뒤 해당 좌표에 공을 추가하고 모든 클라이언트에게 공의 좌표를 전송한다.

* 각 클라이언트가 클릭해 생성된 공을 모든 클라이언트가 볼 수 있다.

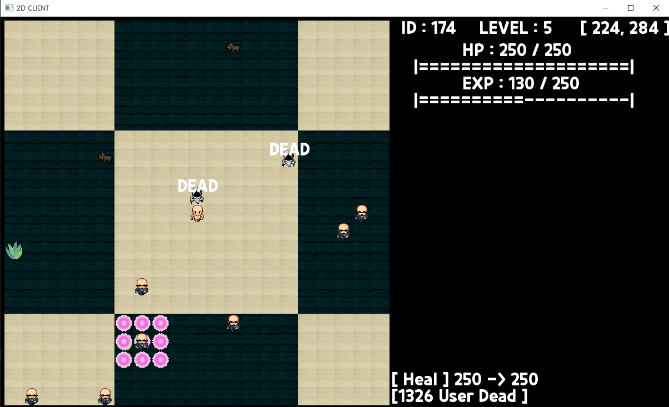
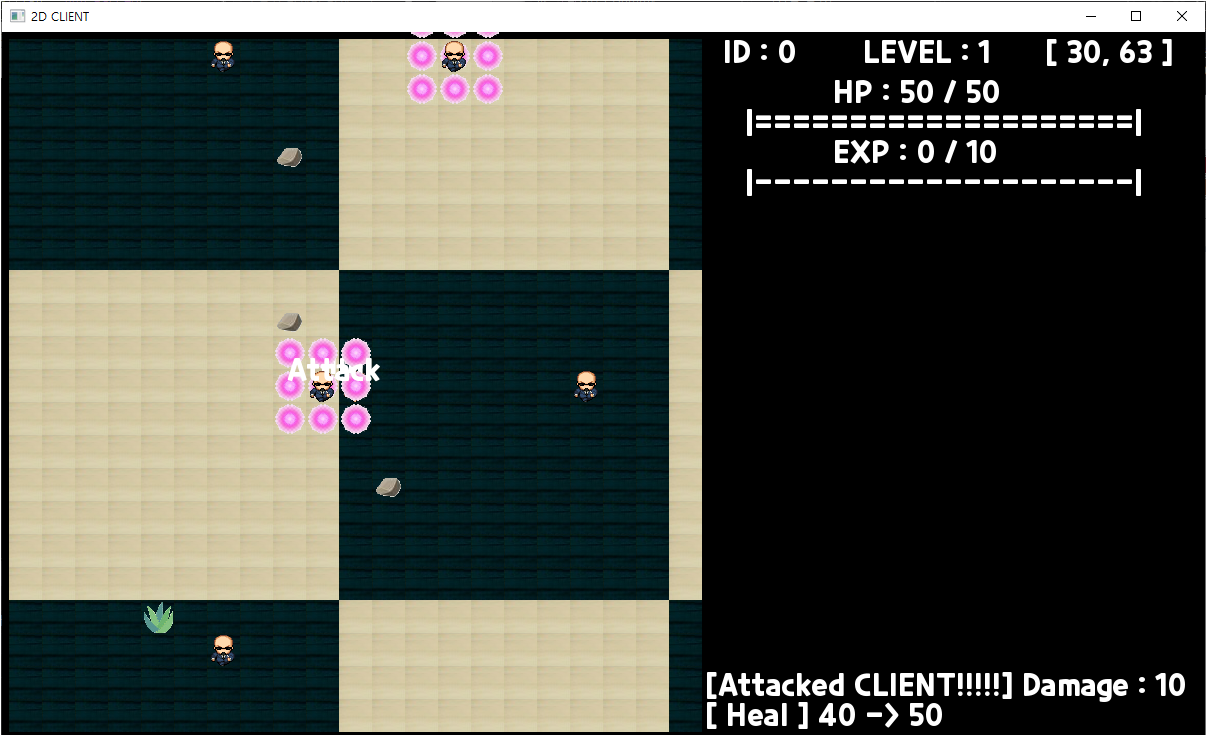
서버는 총 5개의 스레드로 구현되며, 1개의 Update thread와 Recv UDP thread, 3개의 Recv TCP thread로 구현되었으며, 흐름도는 다음과 같습니다.



1. 구현

저와 팀원들은 Fruit crush를 제작할 때 각자의 컴퓨터에서 개발을 진행하였습니다. 하지만 팀원들과 모여 게임을 실행해본 결과 각 클라이언트에서 보는 공의 위치가 다른 문제점을 발견하였습니다. 처음에는 방화벽으로 인한 네트워크 레이턴시 증가가 원인이라 생각했지만 방화벽을 끈 뒤에도 개선되지 않았고, 클라이언트들의 마우스 움직임이 생길 때 유독 심해지는 것을 보면서 불필요한 UDP통신이 빈번하게 발생한다고 생각하게 되었습니다. UDP통신으로 인한 네트워크 과부하가 TCP통신에도 영향을 주게 되었고 이로 인해 클라이언트에 패킷이 도착한 시간을 기준으로 공을 생성하는 방식이 잘못되었음을 알게 되었습니다. 저는 이러한 문제점을 해결하기 위해 서버에 고정 프레임을 추가하였고 매 프레임마다 서버를 업데이트하였습니다. 그 결과 모든 클라이언트가 렉없이 동일한 화면을 보였고 30,45와 60 프레임 중 30 프레임에서 게임 품질이 가장 쾌적한 것을 보며 게임 품질에 대한 프레임의 중요성을 확인할 수 있었습니다. 이를 통해 네트워크에 대한 지식이 서버 프로그래밍에 필요하다는 것을 확인할 수 있었습니다.

1. **[게임] Alone Dungeon**



**기본 설명**

사용 언어: C++11

개발 환경: Visual Studio 2017 64bit

장르: MMO 생존 서바이벌

제작 인원: 1인

제작 기간: 약 1개월

목표: 윈도우 소켓, mutex, IOCP를 활용한 MMO 서버 제작

**세부 설명**

1. 게임

Alone Dungeon은 PVP가 가능한 MMO 생존 서바이벌 게임으로 다른 플레이어에게 죽지 않고 몬스터를 사냥해 성장하는 게임입니다. 800x800 크기의 월드에 2000개의 지형 오브젝트와 5000마리의 몬스터가 있으며 1500명의 클라이언트가 동시접속 가능합니다. 플레이어는 몬스터를 사냥해 경험치를 획득하고 레벨이 높을수록 체력과 데미지가 상승해 다른 플레이어보다 우위의 능력치를 가질 수 있습니다. 모든 플레이어는 다른 플레이어의 레벨을 확인할 수 없어 다른 플레이어의 사냥 속도를 보면서 레벨을 가늠해 싸워야 하며, 사망하면 레벨이 초기화된 상태로 부활하므로 체력 회복을 이용해 최대한 죽지 않고 오래 살아남아야 합니다.

1. 서버

1. 구현

1. **[논문] Implement Lock-Free shared\_ptr and weak\_ptr**

**기본 설명**

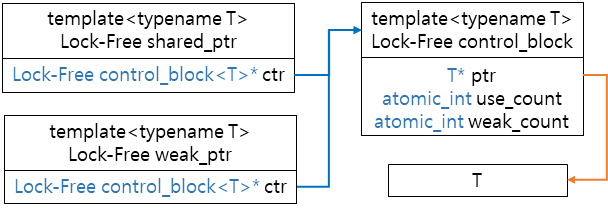
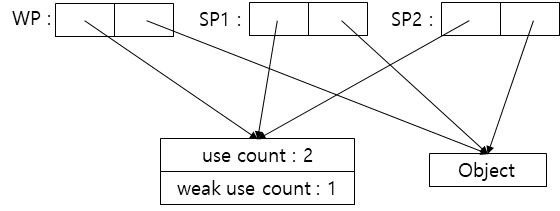
준비 기간: 약 11개월

목표: C++ 멀티스레드 프로그램에서 Lock-Free로 동작하는 shared\_ptr 구현

**세부 설명**

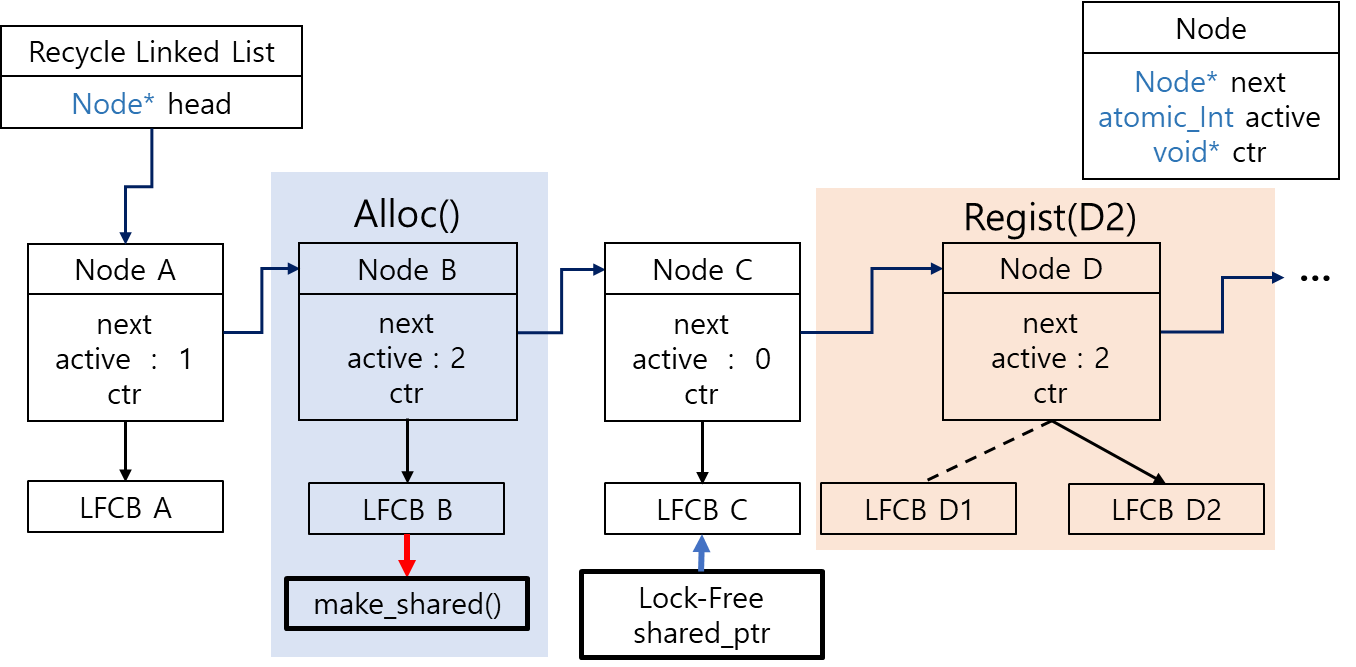
1. 동기

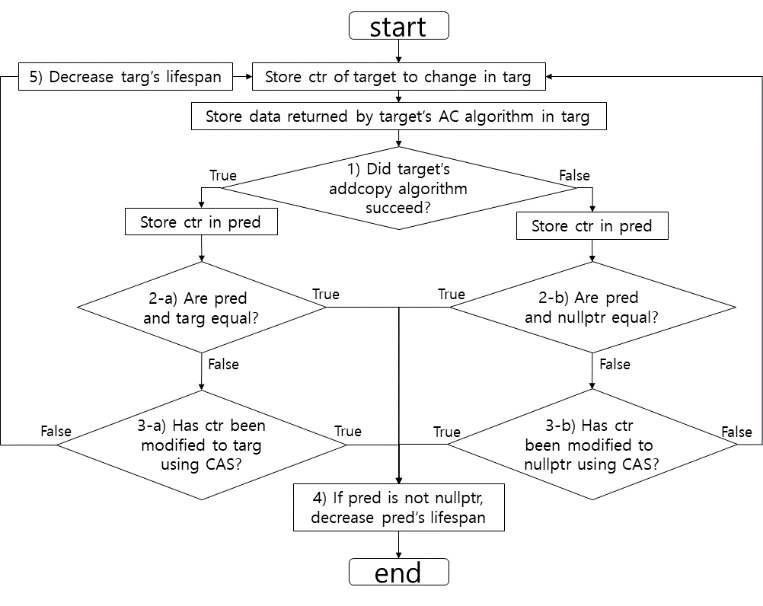
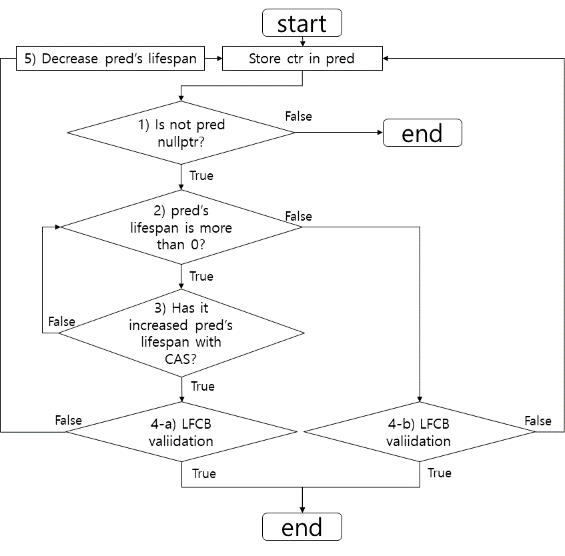
저는 Alone Dungeon의 서버를 구현하면서 Mutex를 이용한 멀티스레드 프로그램 성능의 한계를 볼 수 있었고, 더 높은 성능의 게임서버를 구현하기 위해서 Lock-Free 알고리즘에 대해 관심을 갖게 되었습니다. 이후 stack과 queue와 같은 기존의 자료구조를 Lock-Free로 수정해보면서 Lock-Free 구현의 어려움을 느낄 수 있었고, garbage collection을 제공하지 않는 C++에서는 프로그래머가 직접 메모리를 관리해야 하는 것을 알 수 있었습니다. 하지만 저는 Lock-Free를 이해하지 못한 상태였기 때문에 메모리를 해제하지 않는 방법을 이용했고, Lock-Free을 이해하기 시작한 뒤 메모리 관리에 대해 생각할 수 있었습니다. 이후 Hazard Pointer와 EBR을 접하게 되었지만 두 방법을 정확히 이해하지 못할 뿐더러 Lock-Free에 대한 지식도 부족해 프로그램에서 에러가 발생했을 때 원인을 찾는 것에 어려움을 느꼈습니다. 결국 저에게 직접 메모리를 관리하지 않는 방법이 필요하다 생각하게 되었고, C++11에서 제공하는 shared\_ptr와 weak\_ptr를 Lock-Free로 동작하도록 구현해 이용하면 Lock-Free 구현의 난이도를 낮춰 Lock-Free 알고리즘을 이해하는데 도움이 될 것이라 생각하게 되었습니다.

1. 구현

Lock-Free shared\_ptr(LFSP)와 Lock-Free weak\_ptr(LFWP)은 멀티스레드 환경에서 기존의 shared\_ptr와 weak\_ptr가 발생시키는 데이터 레이스를 방지하고, Lock-Free로 동작할 수 있도록 Lock-Free control\_block(LFCB)이라 하는 고유의 메모리 관리 객체만을 참조하는 구조를 가집니다.

LFSP/LFWP는 다른 LFSP/LFWP의 LFCB를 공유하기 위해 해당 LFCB의 카운터를 증가시킨 뒤 참조하며, 이후 본래 LFCB의 카운터를 감소시켜 다른 스레드의 영향을 받는 상황에서도 정상적으로 동작하도록 구현하였습니다.

LFSP/LFWP 구현에서 가장 어려웠던 점은 내부에서 발생하는 메모리 관리의 문제를 해결하는 것이었습니다. 처음에는 기존의 shared\_ptr/weak\_ptr와 동일하게 카운터가 0인 LFCB의 메모리를 해제했지만 LFCB가 공유 메모리이기 때문에 해제된 메모리에 접근하는 문제가 발생했습니다. 이러한 문제를 해결하기 위해 Hazard pointer나 EBR을 이용할 수 있었지만 프로그래머의 부가적인 작업을 요구하는 이러한 방법은 적절하지 않다 생각했고, LFCB를 재사용하는 방법을 통해 문제점을 해결할 수 있었습니다.

LFCB를 안전하게 재사용하기위한 LFCB 재사용 관리 객체인 Recycle Linked List(RLL)는 Lock-Free로 동작하는 연결 리스트로 Hazard Pointer에서 영감을 받았습니다. Hazard Pointer의 연결리스트는 Node 삽입이 존재하지만 Node 제거는 존재하지 않기 때문에 내부적으로 메모리 해제와 관련된 메모리 문제가 발생하지 않고 사용 가능/불가능한 노드를 검색해 빠르게 Node의 상태를 변경할 수 있다는 특징을 가져 RLL에 적합하다고 생각했기 때문입니다.

LFCB를 재사용하기 위해서는 LFSP/LFWP가 다른 LFCB를 이용할 때 사용 가능한 LFCB인지 확인하는 과정이 필요하다 생각하였습니다. 이용할 LFCB의 카운터를 증가시키는 과정 도중 LFCB가 재사용된다면 이는 전혀 관계없는 LFCB를 의미하기 때문입니다. 이를 확인하기 위해 서 해당 LFSP/LFWP가 여전히 LFCB를 참조하는지 검사하도록 하였고, 이때 발생할 수 있는 모든 경우를 논증하였습니다.