게임 서버 개발 포트폴리오

이준기

jungi.therich@gmail.com

목차

메모리 풀

풀의 구조와 할당 방식에 따른 성능 비교 및 동적 할당 방식 확인

Ⅰ. 개요

온라인 게임 서버는 세션 관리와 송수신을 위한 버퍼 작업을 위해 메모리의 동적 할당 및 해제를 수행해야 합니다. 동적 할당 및 해제는 시스템 호출을 필요로 하는 작업입니다. 따라서 빈번한 요청은 서버 성능 저하를 불러옵니다. 동적 할당은 멀티 스레드 환경에서의 객체 관리에도 부정적인 영향을 줄 수 있습니다. 관리되는 객체 내부에 동기화 객체가 존재할 경우 할당 해제 과정에서 코드 설계에 난해함이 생길 수 있기 때문입니다.

이러한 문제점들은 메모리 풀을 사용하여 해결할 수 있습니다. 메모리 풀은 필요한 메모리를 미리 확보하고 재사용하는 방법입니다. 미리 할당된 메모리를 받아 사용하고 사용이 끝난 후에 반납하기 때문에 서버의 부하가 적습니다. 이 연구에선 오브젝트 메모리 풀을 제작하고, 내부 구조에 따라 발생하는 성능 차이를 측정하였습니다.

Ⅱ. 연구 방법

오브젝트 메모리 풀 성능 측정 기준은 다음 두 가지 (1) 메모리 풀 생성 방법 (2) 메모리 분배 방법의 차이로 분석하였습니다. (1)은 배열 방식과 리스트 방식을 의미합니다. 배열은 메모리 풀 최초 생성 시에 요청된 크기를 한 번에 할당하여 모든 객체가 연속적인 공간에 존재하도록 보장합니다. 리스트는 객체 생성을 요구한 만큼 반복하여 생성하기에 연속적이지 않은 공간에 존재할 가능성이 생깁니다. 기준 (2)는 스택과 큐를 나타냅니다. 사용된 메모리가 반납되었을 때 해당 메모리의 즉각적인 재사용 여부에서 차이가 있습니다.

성능은 스레드가 주어진 작업을 완료하는데 걸린 시간을 확인합니다. 시간의 단위는 마이크로 초(us)로 설정하였으며, ‘QueryPerformanceCounter’ 함수를 사용하였습니다.

다음 상황에서 메모리풀이 관리하는 객체 크기에 따른 수행 시간의 차이가 존재하는지 확인하였습니다. 이 때 싱글 스레드 상황에서는 단일 프로세서의 작업을 기대하기 위해 'SetProcess(Thread)AffinityMask' 함수를 사용하였고, 멀티 스레드 상황에서는 모든 스레드를 동시에 깨우기 위해 이벤트를 사용하였습니다. 프로세스는 주어진 작업을 500,000번 반복하여 평균 시간을 구합니다. 이 때 평균의 오차를 줄이기 위해 작업에 걸린 시간 중 최대 시간과 최소 시간 일부는 전체 시간에서 제외됩니다.

(1) 싱글 스레드가 메모리를 할당 받은 후 즉시 반납하는 상황

(2) 싱글 스레드가 메모리를 할당 받은 후 랜덤으로 반납하는 상황

(3) 멀티 스레드가 특정 횟수의 할당과 해제를 반복하는 상황

```
class cMemPool
{
...
public:
        DATA* Alloc();
        void Free(DATA* pFree);

private:
        stDataNode<DATA>* _Top;
        stDataNode<DATA>* _pStart;

        int _totalSize;
        int _usedSize;
        bool _bPlacementNew;
        SRWLOCK _srw;
};
```

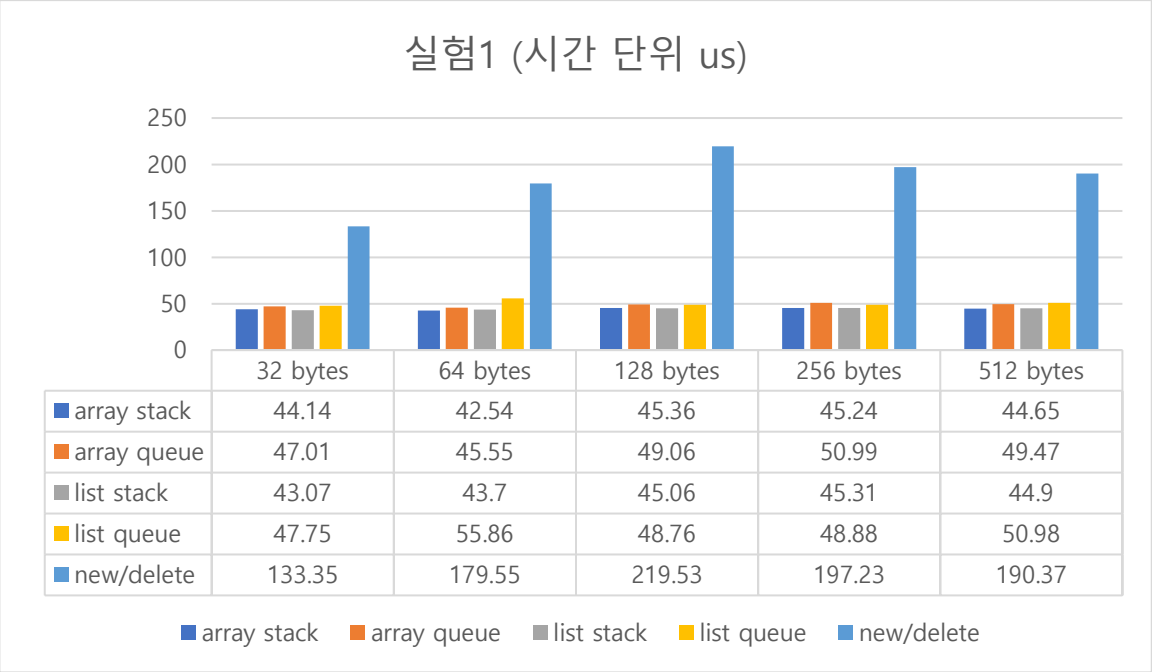
[배열 스택 방식의 메모리풀 코드]

```
struct stDataNode
{
#ifndef NDEBUG
        __int64 _underflowGuard;
        bool _blnUse;
#endif // !NDEBUG

        DATA _data;
        stDAataNode<DATA>* _pNext;

#ifndef NDEBUG
        __int64 _overflowGuard;
#endif // !NDEBUG
};
```

[풀 내부 노드의 코드]

추가적으로 동적 할당 내부의 동작을 파악하기 위해 (4) 크기별로 구분되는 객체들을 10000번 할당하는데 걸린 시간을 측정하였습니다.

Ⅲ. 연구 결과 및 결론

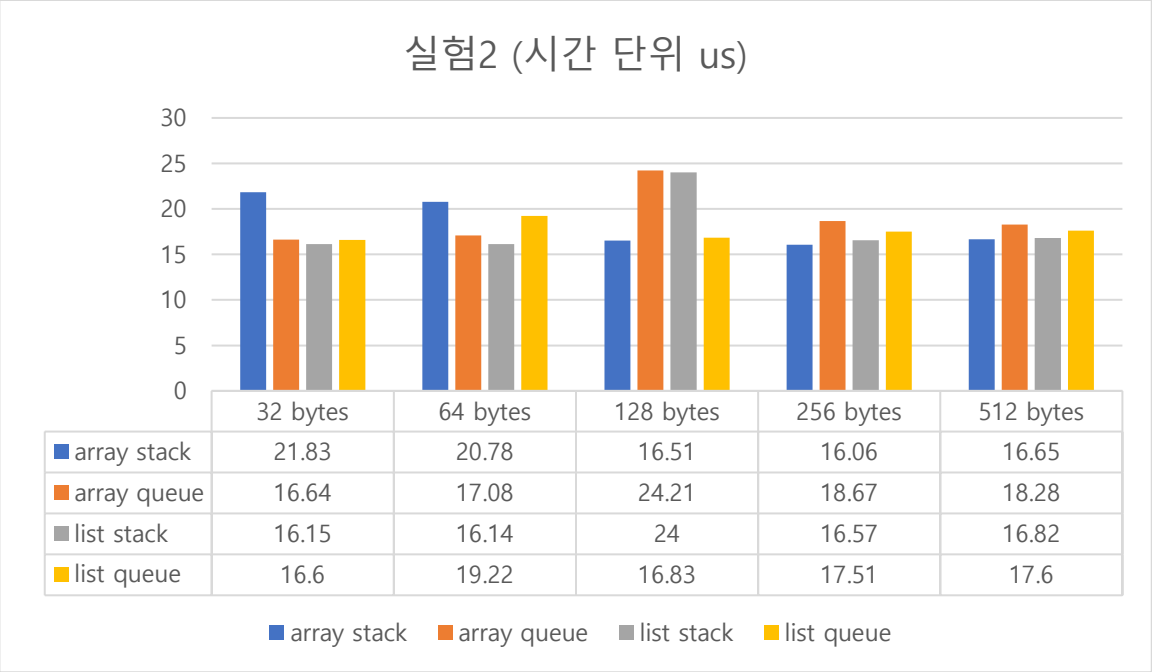


	32 bytes	64 bytes	128 bytes	256 bytes	512 bytes
array stack	44.14	42.54	45.36	45.24	44.65
array queue	47.01	45.55	49.06	50.99	49.47
list stack	43.07	43.7	45.06	45.31	44.9
list queue	47.75	55.86	48.76	48.88	50.98
new/delete	133.35	179.55	219.53	197.23	190.37

첫 번째 실험은 단일 스레드가 하나의 객체를 할당 받은 후에 바로 반납하는 상황에 대한 측정입니다. 목적은 시스템 호출이 불러오는 성능 하락의 정도와 메모리 재사용으로 인한 캐시 히트가 불러오는 이점을 찾는 것입니다.

실험 결과를 통해 두 가지를 확인할 수 있습니다. (1) 시스템 호출은 성능에 분명한 영향을 줍니다. 메모리 풀을 사용하는 것이 필요할 순간마다 동적 할당을 하는 방식보다 3-4배 정도 빠른 속도를 기대할 수 있습니다. (2) 스택 방식의 할당은 큐 방식의 할당보다 더 나은 성능을 기대할 수 있습니다. 이는 캐시 히트 측면에서 분명한 이점이 있기 때문입니다. 풀의 일부가 오랜 시간 사용되지 않아 페이지 아웃이 되는 상황까지 가정한다면 성능적인 이점은 더 커질 수 있습니다.

동일한 측정을 반복하였을 때 메모리풀은 비슷한 결과를 보여줬지만 동적 할당 방식은 결과의 편차가 컸습니다. 따라서 안정적인 성능을 위해선 메모리풀을 사용할 필요가 있습니다.

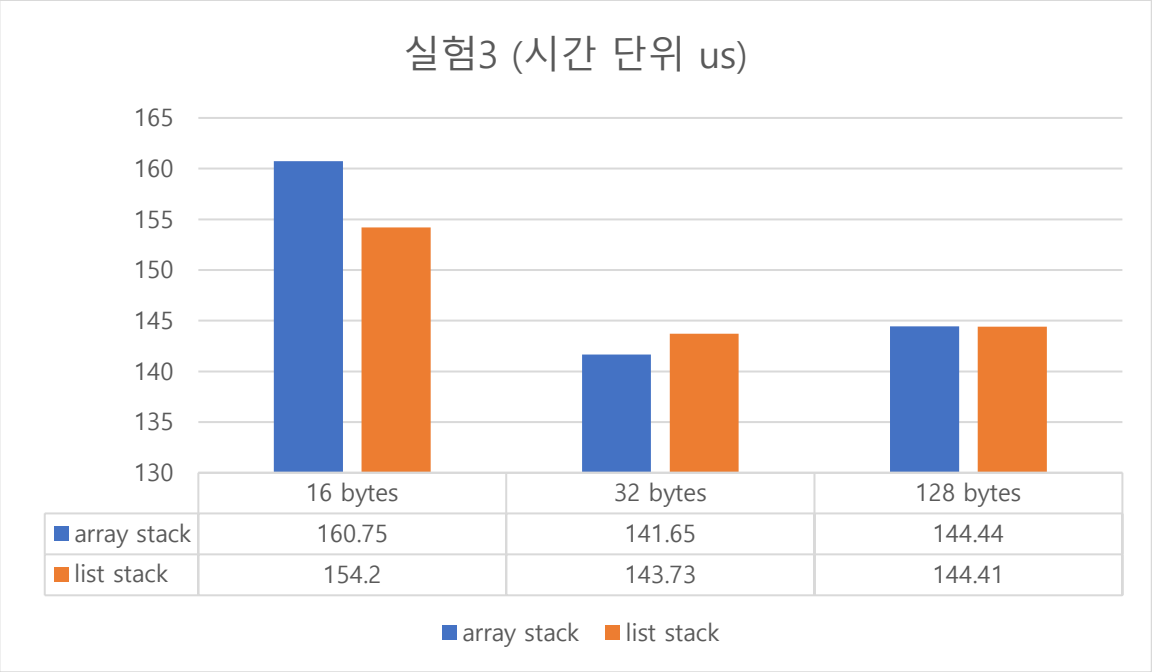


	32 bytes	64 bytes	128 bytes	256 bytes	512 bytes
array stack	21.83	20.78	16.51	16.06	16.65
array queue	16.64	17.08	24.21	18.67	18.28
list stack	16.15	16.14	24	16.57	16.82
list queue	16.6	19.22	16.83	17.51	17.6

두 번째 실험은 4000개의 객체를 할당 받은 후에, 무작위의 2000개 객체를 삭제하는 데에 걸린 시간을 측정한 결과입니다. 목적은 연속되지 않은 객체들이 해제되는 상황에서 배열과 리스트 방식에서 어떤 성능적인 차이가 나타나는지 확인하는 것입니다.

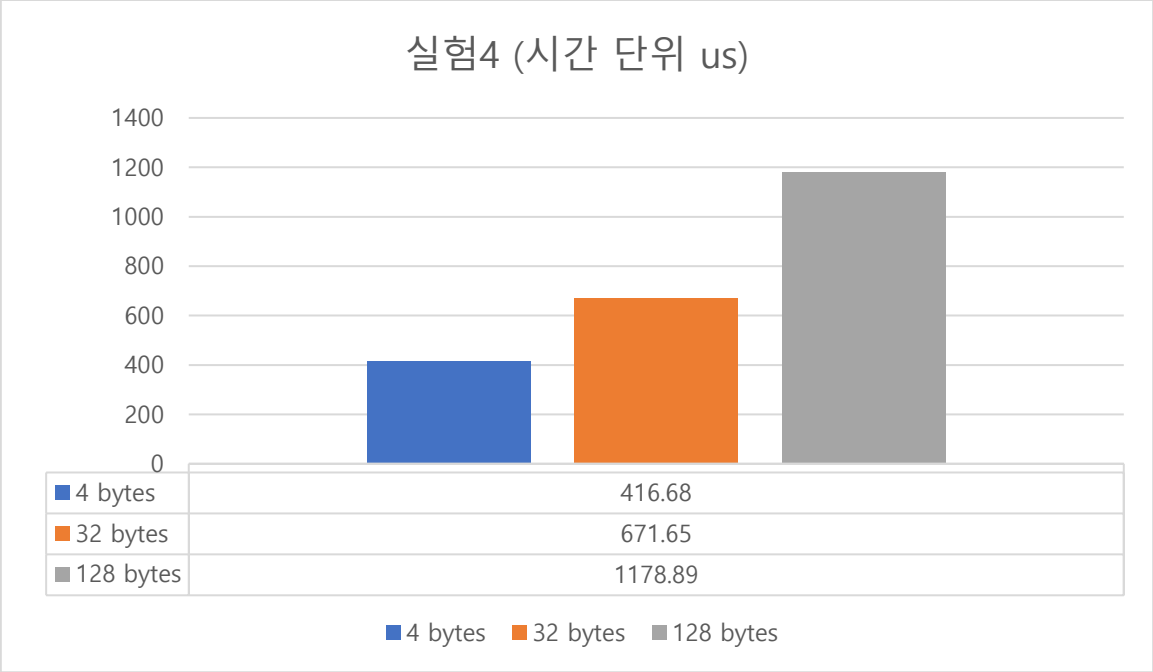
실제 온라인 서버에서는 실험1과 같이 하나의 객체만이 관리되고 해제되는 상황은 없습니다. 단일 스레드로 서버가 동작한다면 어떤 객체가 사용되고 해제될지 예측할 수 없으며 이런 요소로 인해 배열이 갖는 연속된 메모리의 이점이 사라집니다.

위 표에서 보여지는 차이는 유의미하다고 보기 어려우며 측정을 반복했을 때마다 더 빠른 속도를 보여준 메모리풀의 종류가 달랐습니다. 즉, 단일 스레드에서 여러 무작위 객체에 접근하는 상황에선 배열과 리스트의 수준 차이는 미미합니다. 다만 이 실험은 해제를 하는데 걸린 시간만을 측정한 것입니다. 해제 이후 할당을 받는 작업을 고려한다면 스택 방식이 갖는 이점은 분명히 존재합니다.



세 번째 실험은 4개의 스레드가 동일한 메모리풀에 진입하여 각자 1000개의 객체를 할당 받아 객체의 내용을 수정하고 반납하기까지 걸리는 시간을 측정하였습니다. 이 때 이벤트 객체와 인터락을 사용하여 주어진 시행 횟수동안 모든 스레드가 동시에 출발하여 메모리 풀에 할당을 요청하도록 했습니다. 실험 목적은 멀티 스레드 환경에서 동일한 메모리풀에 접근할 때 좀 더 효율적인 구조를 찾는 것입니다. 할당되는 객체는 16 bytes, 32 bytes, 128 bytes 크기입니다.

실험 결과 크기가 작은 객체에 대해서 여러 스레드가 메모리풀을 사용할 때 불리한 점이 있음을 확인하였습니다. 이는 하나의 캐시 라인 안에 다수의 객체가 들어갔기 때문입니다. 두 개 이상의 프로세서가 동일한 하나의 캐시 라인을 들고 있을 때 한 쪽에서 수정이 발생하면 무효화로 인해 다른 프로세서는 캐시 라인을 갱신해야 하기 때문입니다. 배열은 연속된 메모리를 보장하기 때문에 그 여파가 더 크게 나타납니다.



표는 싱글 스레드가 동적 할당을 10000번 요청했을 때, 요청한 크기에 따른 시간 차이를 측정한 결과입니다. 현재 'Visual Studio 2022의 릴리즈 모드'에서는 4, 32 바이트 크기의 할당이 요청되면 내부적으로 'RtlpLowFragHeapAllocFromContext' 함수가 호출되고, 128 바이트 크기의 객체는 'RtlpAllocateHeap' 함수가 호출됩니다.

저단편화힙(Low Fragmentation Heap)은 일종의 메모리 풀입니다. 할당 요청된 크기가 LFH에서 담당할 수 있는 크기라면 유저 블록이라는 풀에서 꺼내 리턴합니다. 이 풀은 크기별로 존재하기 때문에 두 개의 스레드가 각각 4 바이트와 32 바이트를 요청하더라도 한 스레드가 블락 되는 과정 없이 동작합니다. 동일한 크기를 LFH에 요청하게 될 경우 인터락과 스핀락을 통해 동기화를 유지해줍니다. 스핀락을 시도하는 스레드는 일정 횟수 안에 유저 블록에 접근하지 못할 경우 스레드는 아예 새로운 유저 블록을 할당 받습니다.

LFH로 할당 받은 메모리의 해제는 블록의 상태를 나타내는 비트를 바꾸고 크기에 맞는 풀에 넣는 방식으로 진행됩니다. 메모리 해제 후 재할당 요청 시 LFH는 사용 가능한 풀에 랜덤으로 접근하기 때문에 동일한 메모리의 할당이 보장되지 않습니다.

이 유저 블록의 크기는 한계치가 존재합니다. 그래서 LFH가 담당할 수 있는 크기임에도 불구하고 유저 블록에서 넘겨줄 수 있는 메모리가 없다면 LFH는 'AllocateUserBlocks' 함수를 호출하여 새로운 유저 블록을 할당 받습니다.

이러한 로직으로 인해 위 실험과 같은 결과가 발생합니다. 우선 4 바이트와 16 바이트 할당은 LFH가 담당합니다. 4 바이트 할당은 크기가 작기 때문에 유저 블록을 새로 할당 받는 수도 적어 그만큼 힙 할당 호출이 적습니다. 16 바이트는 상대적으로 힙 할당 호출이 많기 때문에 소요 시간이 커집니다. 128 바이트 할당은 LFH가 동작하지 않으므로 그 속도가 더 느립니다.

LFH가 동작하지 않는 힙 할당 요청은 함수 내부에서 'EnterCriticalSection'으로 동기화를 하고 'RtlpHeapRemoveListEntry'를 통해 할당 받을 수 있는 크기가 존재하는지 확인합니다. 할당 받을 수 있는 블록이 존재한다면 그 블록을 그대로 받고, 없을 경우 할당이 일어납니다. 이로 인해 ListEntry에 들어갈 수 있는 크기의 객체를 해제한 후 바로 할당을 요청하면 이전에 사용했던 메모리가 그대로 반납됩니다.

배열은 연속된 메모리를 공간을 통해 높은 캐시 적중율을 기대할 수 있는 구조입니다. 그러나 서버 입장에서는 객체를 관리할 때 이러한 효과를 보기 어렵습니다. 반면 스택 할당이 큐 형식의 관리보다 유리한 부분은 분명 존재합니다. 따라서 메모리풀 설계는 스택 방식을 사용한 배열과 리스트 무엇을 사용하던 상관없습니다. 다만 배열 방식은 고정된 크기를 갖기 때문에 유동적인 크기의 풀이 필요하다면 리스트 방식을 선택하는 방법 밖에 없습니다.

동적 할당을 줄이기 위해 자주 사용되지 않는 객체를 미리 할당해두면 메모리 낭비라는 단점이 생깁니다. 크기가 크더라도 가끔 사용되는 객체라면 필요한 순간 동적 할당을 하는 것이 구조적으로 나을 수도 있습니다. 또한 할당을 요청하는 크기가 작다면 시스템 내부적으로 LFH를 사용하기 때문에 성능 저하가 크지 않습니다.

길 찾기 알고리즘

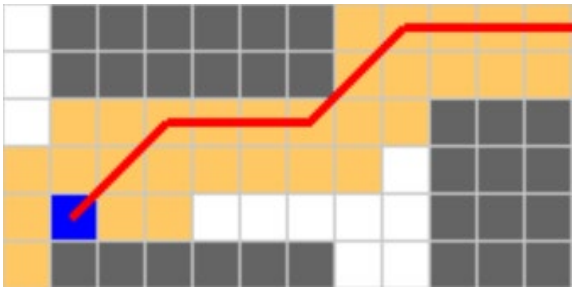
맵 유형에 따른 A*와 JPS 알고리즘의 성능 비교

Ⅰ. 개요

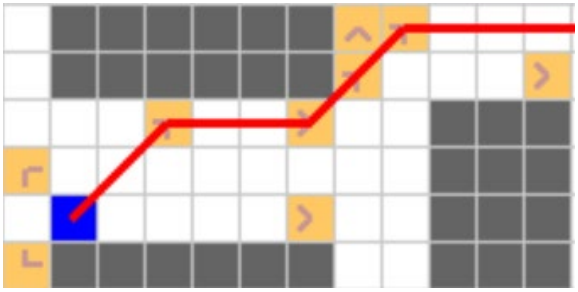
온라인 게임 서버 캐릭터들의 위치는 이차원 그리드를 통해 표현할 수 있습니다. 이 경우 맵 내의 모든 객체를 순회하는 대신 목표로 하는 좌표를 기준으로 충돌 처리나 공격 로직을 계산하면 되기 때문에 쉽고 빠르게 처리가 가능합니다.

길 찾기 알고리즘 또한 맵을 이차원 배열로 표현하면 더 쉽게 목표 지점까지의 최단 경로를 찾아낼 수 있습니다. 이 연구에선 길 찾기 방식 중에서 A* 알고리즘과 JPS 알고리즘을 구현한 후에 길 찾기에 걸리는 시간을 통해 성능 차이를 확인하였습니다.

Ⅱ. 연구 방법



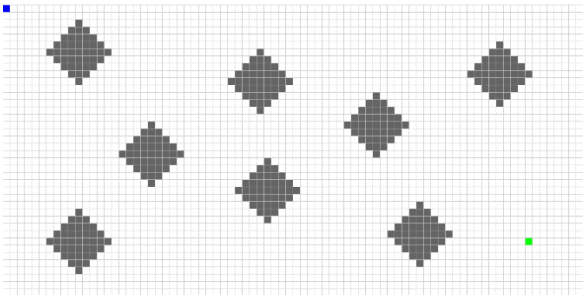
[A* 알고리즘에서의 노드 생성]



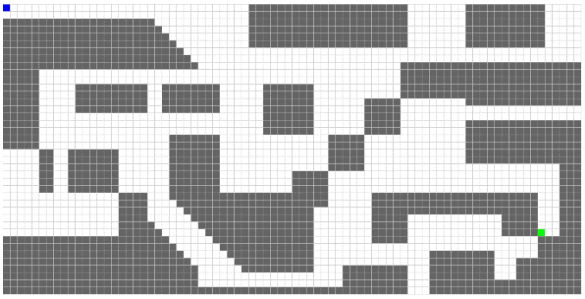
[JPS 알고리즘에서의 노드 생성]

두 알고리즘 모두 f-value라는 값을 두고 목표 노드까지 진행하는 방식입니다. A*는 현재 노드를 기준으로 주변에 생성 가능한 모든 좌표에 새로운 노드를 만듭니다. 이 노드들은 A* 내부에서 관리해야 할 노드 리스트에 들어가며, 이 리스트 안에서 가장 작은 f-value를 갖는 노드를 찾아 새롭게 주변 노드를 생성하며 목적지를 찾아갑니다. JPS는 현재 노드를 기준으로 방향성을 갖는 노드들을 생성하는 방식입니다. 이 때 방향성을 갖는 노드들의 생성 기준은 현재 노드의 위치에서 추가적인 탐색이 필요한 곳이 존재할 때 생

성됩니다. 예를 들어, 직진만이 가능한 공간이라면 노드 생성을 하지 않고, 직진 이후에 추가적인 공간(코너)이 나와 새로운 탐색을 해야 할 때 노드를 생성하는 방식입니다. 이렇게 생성된 노드들은 리스트에 보관되며, JPS 또한 저장된 리스트 내부 노드들 중에서 가장 작은 f-value를 갖는 노드를 기준으로 탐색을 재개합니다.



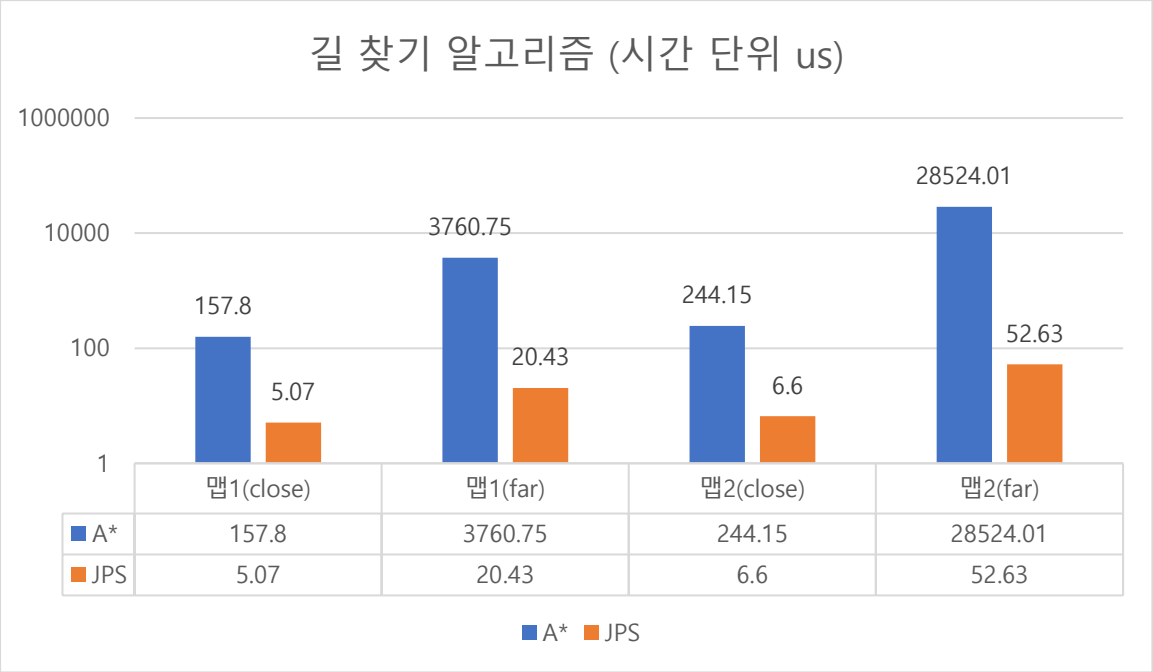
[맵1 – 개방형 지형]



[맵2 – 미로형 지형]

실험은 위 두 가지 종류의 맵을 기준으로 목표 지점까지 도달하는데 걸린 시간을 측정하여 진행했습니다. 맵(1)은 개방된 지형으로 약간의 장애물만 존재합니다. 맵(2)는 일종의 미로 느낌의 지형입니다. 이 두 가지 맵을 기준으로 목표 지점까지의 거리가 멀거나 가까운 경우를 테스트하였습니다. 출발 지점은 모두 (0, 0)으로 동일하였으며 거리가 먼 경우 도착 지점은 (70, 35), 거리가 가까운 경우 도작 지점은 (20, 10) 좌표였습니다. 시간은 'QueryPerformanceCounter'를 사용하여 마이크로 초단위로 측정하였습니다. 동일한 작업을 100000번 반복한 후 이에 대한 평균을 구했습니다.

Ⅲ. 연구 결과 및 결론



거의 대부분의 상황에서 JPS 알고리즘이 A* 알고리즘에 비해 우수한 성능을 보여줬습니다. A* 알고리즘이 JPS 보다 성능적으로 좋았던 상황은 직선으로 이동 가능한 두 좌표의 거리가 7 개의 타일 이내일 때였습니다.

이러한 현상의 이유는 A*는 로직상 매우 많은 수의 노드를 생성할 수밖에 없기 때문입니다. 동적 할당에 소요되는 비용이 더 클 뿐 아니라, std::list 의 자료 구조 특성 상 탐색에 O(N)의 시간 복잡도를 갖기 때문입니다. 이러한 단점은 노드를 위한 메모리풀을 사용하고, f-value 값을 key 로 갖는 std::multimap 자료 구조를 사용하는 방식으로 완화할 수 있습니다.

IOCP 에코 서버

IOCP 에코 서버 설계 방식에 따른 성능 비교

Ⅰ. 개요

IO Completion Port(IOCP)는 사용자가 요청한 비동기 입출력에 대한 완료 통지를 OS가 알려주는 방식 중 하나입니다. 비동기 IO 방식의 설계 방법 중 IOCP 모델을 서버 설계에 사용한 이유는 다음과 같습니다. 이벤트 방식은 완료 통지를 받는 함수에 등록 가능한 이벤트의 수가 64개입니다. 하나의 소켓에 송신 완료, 수신 완료 이벤트를 따로 등록할 경우 32개의 소켓 등록이 최대입니다. 각 스레드마다 고유한 APC큐에 입출력 결과를 저장하고 완료 루틴을 호출하는 방식은 IO를 요청한 스레드만이 완료 통지를 처리할 수 있습니다. 즉, 효율적인 스레드의 관리가 어렵습니다.

반면 IOCP는 동시에 수행되는 스레드 수를 조절할 수 있고, 완료 통지를 요청한 스레드와 관계없이 어떤 스레드라도 그 결과를 처리할 수 있습니다. 스레드는 FIFO 방식으로 나오기 때문에 스레드 문맥 교환이 줄어듭니다. 이 연구에선 에코 서버와 클라이언트를 제작하고, 설계 방식에 따라 루프백 상황에서 송수신되는 패킷의 차이를 통해 성능 차이를 확인하였습니다.

Ⅱ. 연구 방법

에코 서버는 IOCP로 동작하는 네트워크 라이브러리를 설계하여 만들었습니다. 네트워크 라이브러리는 세션의 관리와 메시지 송수신 등 모든 기능을 전적으로 담당합니다. 콘텐츠 차원에서는 라이브러리가 넘겨주는 세션의 ID값을 통해 세션을 간접적으로 관리하고 접근할 수 있습니다. 간접적인 접근은 네트워크 라이브러리에서 선언한 순수 가상 함수를 통해 이루어지기 때문에 콘텐츠단에서는 네트워크 라이브러리의 클래스를 상속받아 사용합니다.

콘텐츠 단에서는 텍스트 파일을 통해 네트워크 라이브러리의 여러 옵션을 설정할 수

있습니다. 이 때의 텍스트 파서는 다양한 인코딩 형식을 받아 해석할 수 있는 형태로 제작하였습니다.

스트레스 클라이언트는 서버에 연결을 시도할 최대 클라이언트 수와 송신을 시도할 최대 패킷 수를 입력 받아 동작합니다. 설정에 따라 클라이언트가 연결 해제를 시도할 수 있고 송신하는 패킷의 시간 간격도 조절할 수 있습니다. 스트레스 클라이언트의 워커스레드는 select 방식으로 자신만이 관리하는 클라이언트를 위 설정에 맞춰 송신 및 수신 작업을 수행합니다.

스트레스 클라이언트와 주고받는 패킷의 차이를 확인하기 위해 서버는 최적화 여부에 따라 두 가지가 제작되었습니다.

```
HANDLE _hIOCP;
WSADATA _wsa;
SOCKADDR_IN _serverAddr;
SOCKET _listenSocket;

DWORD64 _uniqueId;
BOOL _bRun;

unordered_map<DWORD64, stSession*> _sessionMap;
CRITICAL_SECTION _cs_mapLock;

HANDLE _hAcceptThread;
HANDLE* _hIOCP_workerThread;
HANDLE _hMonitorResetThread;

DWORD _numberOfWorkerThread;
```

```
HANDLE _hIOCP;
WSADATA _wsa;
SOCKADDR_IN _serverAddr;
SOCKET _listenSocket;

DWORD _uniqueId;
BOOL _bRun;

cPoolArr_stack<DWORD>* _pValidIdxPool;
vector<stSession*> _sessionArr;

HANDLE _hAcceptThread;
HANDLE* _hIOCP_workerThread;
HANDLE _hMonitorResetThread;

DWORD _numberOfWorkerThread;
```

[서버 멤버 변수 최적화 전(상)과 후(하)]

```
SOCKET _sock;
DWORD64 _id;
DWORD _ioCount;
DWORD _sendFlag;
CHAR _ipAddr[30];

OVERLAPPED _recvOverlapped;
OVERLAPPED _sendOverlapped;

cRingBuffer _recvBuffer;
cRingBuffer _sendBuffer;

CRITICAL_SECTION _cs_sessionLock;
```

```
DWORD _sessionStatus;
SOCKET _sock;
uld _id;
DWORD* _pIdxFree;
CHAR _ipAddr[30];

OVERLAPPED _recvOverlapped;
OVERLAPPED _sendOverlapped;

alignas(64) cRingBuffer _recvBuffer;
alignas(64) cRingBuffer _sendBuffer;

alignas(64) DWORD _ioCount;
alignas(64) DWORD _sendFlag;

SRWLOCK _srw_sessionLock;
```

[세션 멤버 변수 최적화 전(상)과 후(하)]

네트워크 라이브러리 최적화 첫 번째 방법은 동적 할당을 줄이는 것입니다. 서버 생성

시에 최대로 접속 가능한 세션의 수를 정하고, 그 수만큼 미리 할당하여 '_sessionArr'에 저장하였습니다. 모든 세션들은 고정된 위치에 존재하며, 사용 가능한 세션들의 인덱스 값은 '_pValidIdxPool'에 들어있게 됩니다. 세션의 고유한 아이디는 8바이트 크기의 유니온으로 관리됩니다. 상위 4바이트에는 세션이 존재하는 인덱스 값이 들어가고 하위 4바이트는 매 세션의 접속마다 증가하는 정수 값이 들어갑니다. 콘텐츠 단에서는 8바이트 크기의 아이디를 받지만, 서버에서는 이 아이디를 통해 한 번에 세션 배열에 접근할 수 있는 구조입니다. 메시지 버퍼 또한 동적 할당 없이 풀 형태로 존재합니다.

네트워크 라이브러리 최적화의 두 번째 방법은 'alignas'를 사용하는 것입니다. '_ioCount'와 '_sendFlag' 멤버 변수는 인터락 함수에 영향을 받습니다. 인터락의 동작은 변수가 있는 캐시 라인을 잠그는 방식입니다. 따라서 두 변수가 붙어있게 된다면 두 스레드가 서로 다른 변수에 접근해야 하는 상황에서 지연이 될 수 있습니다. 이를 해결하기 위해 두 변수의 간격을 조절하였습니다. 이와 유사한 이유로 링버퍼 또한 'alignas'를 사용했습니다. 현재 링버퍼 크기는 40바이트이며 내부는 읽기와 쓰기 작업이 필요한 변수들로 이루어져 있습니다. 송신과 수신 작업은 동시에 발생할 수 있으며 송신 버퍼 멤버 변수의 수정으로 인해 같은 캐시 라인에 우연히 존재하던 수신 버퍼의 멤버 변수를 읽기 시도하려던 스레드가 있다면 무효화가 발생하며 속도가 느려 집니다.

네트워크 라이브러리 최적화의 마지막 방법은 동기화 객체를 줄이고, 기능을 구분하는 것입니다. 미리 할당된 세션을 고정된 위치에 두는 방식으로 동기화 객체를 없애고, 읽기와 쓰기를 구분하는 동기화 객체를 사용하였습니다.

Ⅲ. 연구 결과 및 결론

```
==============================
q: quit server  d: disconnect first session test

Total Session Accepted: 260
Current Session Accepted: 100
Session Accept TPS: 10
Message Recv TPS: 32567
Message Send TPS: 32679
Recv Bytes Per Sec: 16306660
Send Bytes Per Sec: 16327190

Maximun Recv TPS: 38989
Maximun Send TPS: 39083

Current Cpu Usage: 22.739361
Committed Memory Size by this Process: 20893696(0.314651)
Current Non-paged Pool Usage: 82840
```

[최적화 전 서버 모니터]

```
------------------------------------------------

s: Stop Echo / Resume Echo
q: Quit

Desired Client: 100     Desired Send: 100
Send Delay: 0

Total Connect: 370
Current Connect: 100
Message Send TPS: 1899983
Message Recv TPS: 1900199

Login Packet Not Recv: 0
Duplicated Login Packet: 0

Max Latency(ms): 13
Error - Connect Failed: 0
Error - Echo Not Recv(1sec.): 0
Error - Invalid Packet Recv: 0
Error - Disconnected From Server: 0
```

[최적화 전 스트레스 에코 모니터]

```
==============================
q: quit server  d: disconnect first session test

Total Session Accepted: 175
Current Session Accepted: 100
Session Accept TPS: 5
Message Recv TPS: 55867
Message Send TPS: 55909
Recv Bytes Per Sec: 27972990
Send Bytes Per Sec: 27995150

Maximun Recv TPS: 83205
Maximun Send TPS: 83215

Current Cpu Usage: 19.384828
Committed Memory Size by this Process: 135053312(0.000000)
Current Non-paged Pool Usage: 83792
Message Pool Usage Rate: 2 / 5000
```

[최적화 후 서버 모니터]

```
------------------------------------------------

s: Stop Echo / Resume Echo
q: Quit

Desired Client: 100     Desired Send: 100
Send Delay: 0

Total Connect: 280
Current Connect: 100
Message Send TPS: 3956455
Message Recv TPS: 3956762

Login Packet Not Recv: 0
Duplicated Login Packet: 0

Max Latency(ms): 4
Error - Connect Failed: 0
Error - Echo Not Recv(1sec.): 0
Error - Invalid Packet Recv: 0
Error - Disconnected From Server: 0
```

[최적화 후 스트레스 에코 모니터]

스트레스 클라이언트는 100 개의 클라이언트를 관리하며 100 개의 패킷을 딜레이 없이 보내도록 하였습니다. 동시에 0.5 초 마다 연결 종료와 재연결을 시도합니다.

최적화를 진행한 이후 클라이언트는 서버와 주고받는 패킷의 수가 크게 증가했고 송신한 패킷을 받기까지 걸리는 최대 지연 시간이 감소했습니다.

네트워크 라이브러리에서는 네이글 옵션을 끄거나 켤 수 있습니다. 네이글 알고리즘은 크기가 작은 여러 개의 패킷에 대해 송신 작업을 수행할 때 네트워크 트래픽을 감소시키는 알고리즘입니다. 현재 서버 테스트 환경은 루프백이었기 때문에 송신되는 패킷에 따른 네이글 알고리즘의 효과를 확인하기 어려웠습니다. 그러나 이론적으로 상대방으로부터 수신 확인 패킷을 받기 전까지 송신을 진행하지 않으므로 반응 속도가 떨어질 수 있습니다. 이는 FPS 게임과 같이 빠른 응답 속도가 송신 효율보다 중요한 구조일 경우 치명적입니다.

이진 탐색 트리와 레드 블랙 트리

자료 구조의 구현 및 성능 비교

Ⅰ. 개요

게임 서버는 생성된 모든 객체에 접근하는 순간보단 작업이 필요한 일부 객체에만 접근하는 경우가 많습니다. 이는 곧 목표로 한 객체를 빠르게 찾는 것이 서버의 성능과 직결된다고 볼 수 있습니다. 이 연구에선 이진 탐색 트리와 레드 블랙 트리를 구현하고 삽입, 탐색, 삭제 과정에서 소요되는 시간을 비교하였습니다.

Ⅱ. 연구 방법

이진 탐색 트리는 이진 트리에 탐색의 기능을 강화한 자료 구조입니다. 자료의 삽입은 키 값을 기준으로 이루어지기 때문에 최악의 경우 리스트 자료 구조와 같은 선형 형태로 삽입될 가능성이 있습니다. 레드 블랙 트리는 노드에 RED/BLACK이라는 색상의 개념을 추가하여 한 쪽으로 노드가 쏠리지 않도록 밸런싱 작업을 수행하는 자료 구조입니다.

성능 비교는 두 자료 구조를 구현한 후에 삽입, 탐색, 삭제에 걸리는 평균 시간을 구하여 비교하였습니다. 'QueryPerformanceCounter' 함수를 사용하였으며 500000번의 동일한 로직을 반복하여 평균값을 구했습니다.

Ⅲ. 연구 결과 및 결론

연구 결과 산출에 앞서 구현한 자료 구조의 검증을 진행하였습니다. 이진 탐색 트리의 검증은 데이터 삽입 후 중위 순회를 했을 때 오름차순으로 잘 정렬되었는지, 그리고 순서대로 삭제하는 과정에서 누락된 노드 없이 모두 정상적으로 오름차순을 지켜가며 삭제됐는지 확인했습니다. 레드 블랙 트리는 이진 트리 검증의 동일한 구조와 함께 모든 경

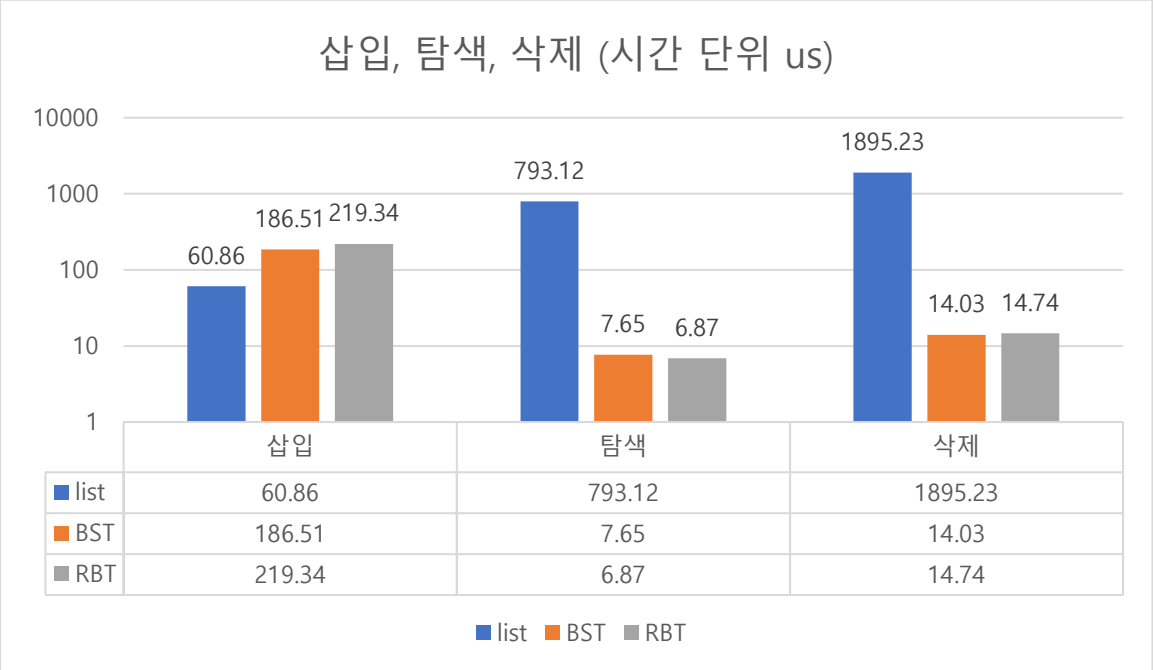
로에 대하여 BLACK 노드의 수가 동일하며 RED 노드가 연속된 경우는 없는지 확인하는 로직을 추가했습니다.

```
// 자료 구조 검증용 함수로 정수 자료형 기준으로만 동작함
static_assert(std::is_integral<DATA>());


for (int i = 1; i <= putSize; ++i)
{
        insert(i);
        Verify_Insequence();
}

for (int i = 1; i <= putSize; ++i)
{
        iterator iter = find(i);
        erase(iter);
        Verify_Insequence();
}
```

[이진 탐색 트리 검증 코드]

```
// 자료 구조 검증용 함수로 정수 자료형 기준으로만 동작함
static_assert(std::is_integral<DATA>());


for (int i = 1; i <= putSize; ++i)
{
        int countBlack = 0;
        insert(i);
        Verify_Insequence();
        Verify_CountBlack(_pRoot, 1, &countBlack);
}

for (int i = 1; i <= putSize; ++i)
{
        int countBlack = 0;
        iterator iter = find(i);
        erase(iter);
        Verify_Insequence();
        Verify_CountBlack(_pRoot, 1, &countBlack);
}
```

[레드 블랙 트리 검증 코드]



첫 번째로 std::shuffle을 통해 무작위 2000개의 데이터를 삽입하였습니다. 그 결과 'push_back'으로 동작한 리스트가 가장 빨랐으며 삽입 이후 밸런싱을 필요로 하는 레드 블랙 트리가 가장 느린 결과를 보여주었습니다.

동일한 방식으로 2000개의 데이터를 삽입한 후에 무작위의 수 100개를 탐색하는 로직

에선 리스트 자료 구조가 가장 느렸고 레드 블랙 트리가 가장 좋은 성능을 보여줬습니다.

2000개의 데이터를 삽입한 후 탐색 대신 100개의 데이터를 삭제하는 로직은 리스트가 가장 느렸고 이진 탐색 트리가 가장 빨랐습니다.

이진 탐색 트리는 자료가 선형적으로 들어가지 않는다면 그 성능이 보장됩니다. 그러나 이는 레드 블랙 트리 또한 마찬가지로 밸런스 작업이 적어져 삽입 및 삭제에서 성능이 떨어진다고 말할 수 없게 됩니다.

이외에도 해시맵을 사용한 자료 구조인 std::unordered_map을 사용하는 방법도 있습니다. 동일한 키에 자료들이 몰리지 않는다면 O(1)의 시간 복잡도를 보장합니다. 다만 이 경우 key 값의 순서를 보장하지 않기 때문에 순서를 고려해야 하는 상황에서는 적합한 자료구조가 아닙니다.

[부록]

모든 측정은 Visual Studio 2022에서 최적화를 끈 Release 모드 프로세스로 진행하였습니다. 측정한 시스템 사양은 아래와 같습니다.

64비트 Windows11 OS, AMD Ryzen 7 6800HS, 3.2GHz, 8 Core
512KB L1 Cache, 4.0MB L2 Cache, 16GB RAM

작성한 코드는 아래에서 확인 가능합니다.

https://github.com/JunGi-theRich/server_portfolio