Chapter 6. 단일 쓰레드 메모리 풀링

- 메모리를 자주 할당하고 해제하는 것은 성능 저하의 주요 원인이다.

- 메모리 관리에 특화된 관리자를 개발하여 이런 사항을 극복할 수 있다.

- 메모리 관리자의 디자인시 고려되어야 할 것들은 아래와 같다.

**크기 차원**

- 고정 크기 : 메모리 관리자는 단일 고정 크기의 메모리 블록을 할당한다.

- 가변 크기 : 메모리 관리자는 어떤 크기의 메모리 블록이라도 할당 가능, 요청 크기를 사전에 알 수 없다.

**동시성 차원**

- 단일 스레드 : 메모리 관리자는 단일 스레드로 한정된다. 하나의 스레드가 메모리를 사용하며, 메모리는 스레드 경계를 넘어서지 않는다. 여러 스레드가 동시에 수행하는 상황과 관계가 없다.

- 멀티 스레드 : 여러 스레드가 동시에 이 메모리 관리자를 사용한다. 구현 코드는 상호 배제적인 실행을 수행한다. 코드 부분의 한 시점에서 오직 하나의 스레드만이 실행할 수 있다.

**버전 0: 전역 new()와 delete()**

- 기본 메모리 관리자라는 것은 전역 new(), delete()를 호출할 때 작동하는 것이다.

- 이 두 함수의 구현은 간소화된 가정을 만들 수 없다. 프로세스 컨텍스트에서 메모리를 관리하고, 여러 스레드로 확장할 수 있기에, 멀티스레드를 고려해야 되며, 요청된 메모리 크기는 요청이 발생할 때 마다 변할 수 있다. 이러한 유연성은 속도와 절충된다.

- 클라이언트는 이런 new(), delete()의 모든 기능을 필요로 하지 않을 수 있다. 만일 단일 스레드에서 동작하며, 오직 특정 크기의 메모리 청크(chunk)를 필요로 할 수 있다.

- 이 경우 이 함수의 모든 기능을 사용하는 것은 cpu 주기의 낭비일 수 있다.

- new(), delete() 메소드를 오버로드하여 고유의 메모리 관리자를 구현할 필요가 있다.

**버전 1: 특화된 Rational 메모리 관리자**

- 기본 관리자(일반 new, delete) 너무 자주 호출되는 것을 방지하기 위해 Rational 클래스를 미리 연결 리스트로 할당해 놓음

- 이 연결 리스트는 가용 객체들의 자유 리스트 이다.

- 새로운 Rational 객체가 필요하면, 자유 리스트에서 하나를 가져오고 사용이 완료되면 리스트로 반환

|  |
| --- |
| class NextOnFreeList  {  public:  NextOnFreeList \* next;  };  class Rational  {  private:  static NextOnFreeList\* freeList;  }; |

- 자유 리스트는 두 가지 역할을 수행, 1. Rational 객체의 시퀀스, 2. NextOnFreeList 요소들의 시퀀스

- Rational new(), delete() 연산자를 이용하여 정적 자유 리스트를 조작한다. 이 연산자는 전역 연산자들을 오버로드

|  |
| --- |
| class Rational  {  public:  Rational(int a = 0, int b = 1) : n(a), d(b) {}  inline void \*operator new(size\_t size)  {  if (freeList == nullptr)  expandTheFreeList();  NextOnFreeList \*head = freeList;  freeList = head->next;  return head;  }  inline void operator delete(void \*doomed, size\_t size)  {  NextOnFreeList \*head = static\_cast<NextOnFreeList\*> (doomed);  head->next = freeList;  freeList = head;  }  static void newMemPools() { expandTheFreeList(); }  static void deleteMemPool()  {  NextOnFreeList\* nextptr;  for (nextptr = freeList; nextptr != nullptr; nextptr = freeList)  {  freeList = freeList->next;  delete[] nextptr;  }  }  private:  static NextOnFreeList\* freeList;  static void expandTheFreeList()  {  // 다음 포인터를 포함할 수 있도록 객체를 크게 할당해야 한다.  size\_t size = (sizeof(Rational) > sizeof(NextOnFreeList\*)) ? sizeof(Rational) : sizeof(NextOnFreeList\*);  NextOnFreeList \* runner = static\_cast<NextOnFreeList\*> (new char [size]);  freeList = runner;  for (int i = 0; i < EXPANSION\_SIZE; ++i)  {  runner->next = static\_cast<NextOnFreeList\*> (new char[size]);  runner = runner->next;  }  runner->next == nullptr;  }  enum { EXPANSION\_SIZE = 32 };  int d, n;  }; |

- 연산자 new()는 자유 리스트로부터 새로운 Rational 객체를 할당, 만약 자유 리스트가 비어있다면 확장, 자유 리스트의 헤드를 선택하고 자유 리스트 포인터를 수정한 다음 반환

- 연산자 delete()는 Rational 객체를 자유 리스트로 반환, 이 객체를 자유 리스트의 앞에 추가

- 자유 리스트가 비었다면, Rational 객체를 힙으로부터 할당, 하지만 Rational 형식에서 NextOnFreeList 형식으로 전환하는 것은 위험할 수 있음, 자유 리스트의 요소가 두 가지 형식을 모두 만족 시킬 수 있도록 충분히 크다는 것을 확인

- expandTheFreeList의 구현이 최적은 아님, new를 호출할 때 한번에 커다란 메모리 블록을 할당해 다음 여러 요소로 나누도록 하는게 더욱 효율적이다.

- new()와 delete()를 호출하면 Rational::operator new()와 delete()가 호출된다. **성능향상을 기대할 수 있을 것**

**버전2. 고정 크기 객체 메모리 풀**

- 버전 1은 Rational 객체만을 관리하는 것에 제한, 다른 크기의 어떤 클래스를 위한 메모리 관리자가 필요한 경우, 메모리 풀을 템플릿으로 구현하면 좋을 것.

|  |
| --- |
| template<class T>  class MemoryPool  {  public:  MemoryPool(size\_t size = EXPANSION\_SIZE)  {  expandTheFreeList();  }  ~MemoryPool()  {  MemoryPool<T> \*nextptr = next;  for (nextprt = next ; nextptr != nullptr ; nextptr = next)  {  next = next->next;  delete[] nextptr;  }  }  // 자유 리스트로부터 T요소를 할당  inline void\* alloc(size\_t size)  {  if (!next)  expandTheFreeList();  MemoryPool<T> \*head = next;  next = head->next;  return head;  }  // 자유 리스트로 T요소를 반환  inline void\* free(void \*someElement)  {  MemoryPool<T> \*head = static\_cast<MemoryPool<T>\*>(someElement);  head->next = next;  next = head;  }  private:  // 자유 요소를 자유리스트에 추가  static void expandTheFreeList()  {  // 다음 포인터를 포함할 수 있도록 객체를 크게 할당해야 한다.  size\_t size = (sizeof(T) > sizeof(MemoryPool<T>\*)) ? sizeof(T) : sizeof(MemoryPool<T>\*);  MemoryPool<T> \* runner = static\_cast<MemoryPool<T>\*> (new char[size]);  next = runner;  for (int i = 0; i < EXPANSION\_SIZE; ++i)  {  runner->next = static\_cast<MemoryPool<T>\*> (new char[size]);  runner = runner->next;  }  runner->next == nullptr;  }  // 자유 리스트가 비었다면 이 양만큼 확장  enum { EXPANSION\_SIZE = 32 };  // 다음 요소  MemoryPool<T> \*next;  }; |

- MemoryPool 생성자는 자유리스트를 초기화한다. size 인자는 자유 리스트의 초기 길이를 지정한다.

- 소멸자는 자유리스트를 따라 모든 요소를 제거한다.

- alloc() 함수는 T요소에 충분한 공간을 할당한다. 만약 리스트가 비었다면 expandTheFreeList를 호출한다.

- free() 함수는 T요소를 자유 리스트에 다시 붙이는 방식으로 T 요소를 해지한다.

- expandTheFreeList()를 사용하여 새로운 요소를 자유 리스트에 추가한다. 새로운 요소를 힙에서 할당하여 연결 리스트에 붙인다.

|  |
| --- |
| class Rational  {  public:  Rational(int a = 0, int b = 1) : n(a), d(b) {}  inline void \*operator new(size\_t size) { return memPool->alloc(size); }  inline void operator delete(void \*doomed, size\_t size) { memPool->free(doomed); }  static void newMemPools() { memPool = new MemoryPool<Rational>; }  static void deleteMemPool() { delete memPool; }  private:  static MemoryPool<Rational> \*memPool;  int d, n;  }; |

- Rational 클래스는 자신의 자유 리스트를 더 이상 유지 할 필요가 없고, MemoryPool 클래스가 이 역할을 대신 한다.

- 정적 freeList 멤버 포인터와 관련 함수를 모두 없애고 메모리 풀에 대한 포인터로 교체

|  |
| --- |
| MemoryPool <Rational> \*Rational::memPool = nullptr;  int main()  {  Rational\* array[1000];  Rational::newMemPools();  for (int i = 0; i < 500; ++i)  {  for (int j = 0; j < 1000; ++j)  array[i] = new Rational(j);  for (int j = 0; j < 1000; ++j)  delete array[j];  }  Rational::deleteMemPool();  } |

- 메모리풀의 템플릿 코드가 추가되어, 시간 비교를 해보면 **버전 1보다 약간 느려졌다.**

**버전 3. 단일 스레드 가변 크기 메모리 관리자**

- MemoryChunk 클래스는 이전 버전에서 사용한 NextOnFreeList 를 대신하고 있다. 이것은 다양한 크기의 메모리 청크들을 묶어 하나의 시퀀스로 만드는 역할을 수행

|  |
| --- |
| class MemoryChunk  {  public:  MemoryChunk(MemoryChunk\* nextChunk, size\_t reqSize)  {  chunkSize = (reqSize > DEFAULT\_CHUNK\_SIZE) ? reqSize : DEFAULT\_CHUNK\_SIZE;  next = nextChunk;  bytesAlraedyAllocated = 0;  mem = new char[chunkSize];  }  ~MemoryChunk() { delete[] mem; }  inline void \*alloc(size\_t reqSize)  {  void \*addr = static\_cast<void\*> (static\_cast<size\_t\*>(mem) + bytesAlraedyAllocated);  bytesAlraedyAllocated += reqSize;  return addr;  }  // 메모리 조각을 해지 하는 것은 고려 하지 않는다. 객체가 삭제되면 메모리 청크를 해지하고 힙으로 돌려줌  //inline void free(void\* someElement) {}    // 리스트의 다음 메모리 청크에 대한 포인터  MemoryChunk\* nextMemChunk() { return next; }  // 이 메모리 청크에 얼마나 많은 공간이 남아 있는가?  size\_t spaceAvailable() { return chunkSize - bytesAlraedyAllocated; }  // 이것은 단일 메모리 청크의 기본 크기  enum { DEFAULT\_CHUNK\_SIZE = 4096 };    private:  MemoryChunk \* next;  void \*mem;  // 단일 메모리 청크의 크기  size\_t chunkSize;  // 이 바이트 수만큼의 메모리가 현재 메모리 청크에 할당되어 있다.  size\_t bytesAlraedyAllocated;  }; |

- MemoryChunk 클래스는 NextOnFreeList의 개선된 버전

- MemoryChunk 생성자는 메모리 블록의 적절한 크기를 결정, 이 크기를 사용해 전용 저장소에 메모리를 할당

- 소멸자는 생성자가 할당한 메모리를 삭제

- alloc() 메소드가 메모리할당 요청을 처리, mem이 가리키고 있는 MemoryChunk 전용 저장소의 가용 공간에 대한 포인터를 반환

- MemoryChunk 는 단지 보조 클래스일 뿐이다. ByteMemoryPool 클래스가 MemoryChunk를 사용하여 가변 크기 메모리 관리자를 구현

|  |
| --- |
| class ByteMemoryPool  {  public:  // 전용 저장소를 만든다.  ByteMemoryPool(size\_t initSize = MemoryChunk::DEFAULT\_CHUNK\_SIZE)  {  expandStorage(initSize);  }  ~ByteMemoryPool()  {  MemoryChunk\* memChunk = listofMemoryChunks;  while (memChunk)  {  listofMemoryChunks = memChunk->nextMemChunk();  delete memChunk;  memChunk = listofMemoryChunks;  }  }  // 전용 풀로부터 메모리를 할당  inline void\* alloc(size\_t reqSize)  {  size\_t space = listofMemoryChunks->spaceAvailable();  if (space < reqSize)  expandStorage(reqSize);  return listofMemoryChunks->alloc(reqSize);  }  // 풀로부터 할당된 메모리를 해제  inline void\* free(void\* someElemnet)  {  listofMemoryChunks->free(someElemnet);  }  private:  // 하나의 메모리 청크를 전용 저장소에 추가  void expandStorage(size\_t reqSize)  {  listofMemoryChunks = new MemoryChunk(listofMemoryChunks, reqSize);  }  // 메모리 청크의 리스트, 이것은 전용 저장소  MemoryChunk \* listofMemoryChunks;  }; |

- 생성자는 initSize 인자를 받아들여 단일 메모리 청크의 크기를 지정한다. 생성자는 이 값에 맞게 단일 메모리 청크의 크기를 지정

- expandStorage() 메소드는 listofMemoryChunks가 할당된 MemoryChunk 객체를 가리키도록 설정

- alloc() 은 우리가 충분한 가용메모리르 가지고 있음을 보장하고 메모리 할당 작업을 MemoryChunk로 위임

|  |
| --- |
| class Rational  {  public:  Rational(int a = 0, int b = 1) : n(a), d(b) {}  inline void \*operator new(size\_t size) { return memPool->alloc(size); }  inline void operator delete(void \*doomed, size\_t size) { memPool->free(doomed); }  static void newMemPools() { memPool = new ByteMemoryPool; }  static void deleteMemPool() { delete memPool; }  private:  static ByteMemoryPool \*memPool;  int d, n;  }; |

- 버전 3은 버전 1, 2보다 느리다. 할당 논리가 좀더 복잡해 졌기 때문이다.

**키 포인트**

- 유연성은 속도와 절충된다. 메모리 관리의 강력함과 유연성이 높아질수록 실행 속도는 떨어진다.

- 전역 메모리 관리자 (new()와 delete()에 의해 구현되는)는 일반적인 목적을 가지고 있으며, 그러므로 부하를 많이 준다.

- 고정된 크기의 메모리 블록을 주로 할당한다면, 특화된 고정 크기 메모리 관리자가 성능을 현저하게 높여줄 것이다.

- 단일 스레드에 한정된 메모리 블록을 주로 할당한다면 위와 동일한 성능 개선이 가능하다. 단일 스레드 메모리 관리자는 전역 new()와 delete()가 처리해야 하는 동시성을 고려할 필요가 없기 때문에 성능에 많은 도움을 준다.