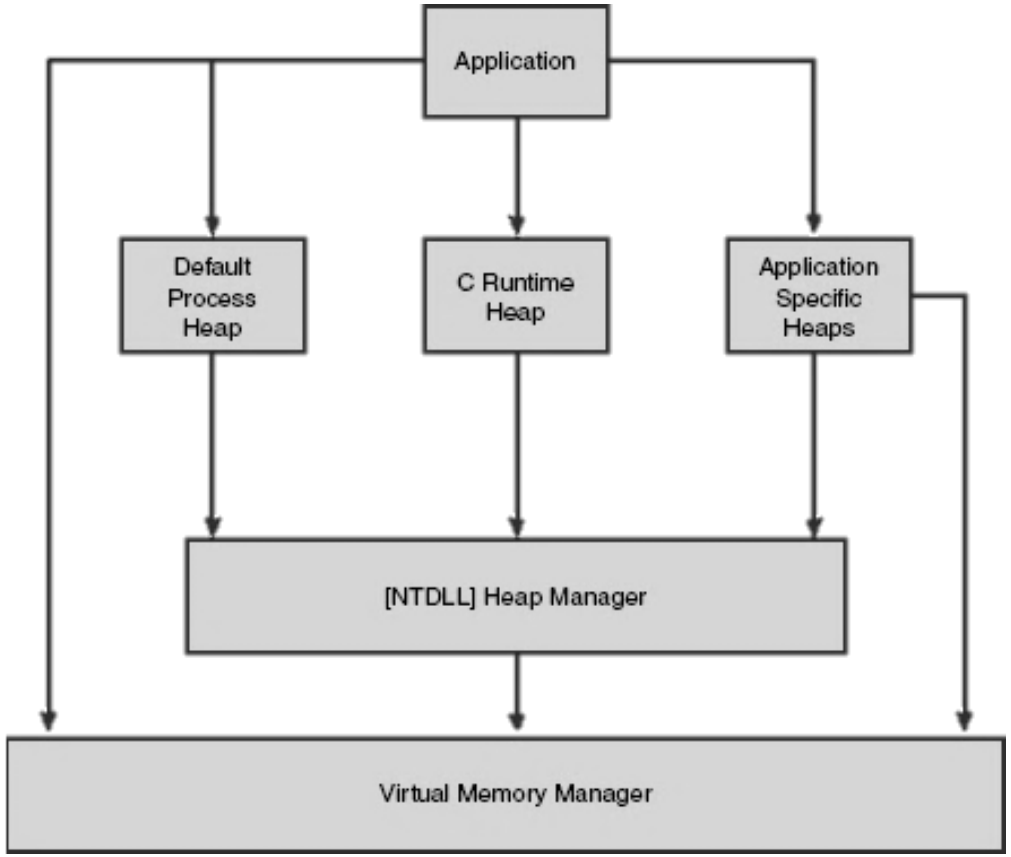
Optimized C++

Ch12. Optimize Memory Management part 2

Window Heap Management



Default Process Heap

프로세스마다 기본적으로 가지고 있는 힙.

GetProcessHeap() 함수를 사용해서 핸들을 얻을 수 있음.

설정을 변경하지 않았다면 기본적으로 1MB.

Address	Type	Size	Committed	Private	Total WS	Private WS	Shar...	Sh...	Lo...	Bl...	Protection	Details
00750000	Heap (Private Data)	1,024 K	112 K	112 K	112 K	112 K				2	Read/Write	Heap ID: 1 [LOW FRAGMENTATION]
0076C000	Heap (Private Data)	912 K									Reserved	Heap ID: 1 [LOW FRAGMENTATION]
00750000	Heap (Private Data)	112 K	112 K	112 K	112 K	112 K					Read/Write	Heap ID: 1 [LOW FRAGMENTATION]

/HEAP Linker Flag로 특정 크기를 지정 가능.

설정한 크기를 넘어서면 힙 확장이 발생.

힙 확장 시 성능저하가 있을 수 있음.

CRT Heap

C/C++ Run-time **라이브러리가 할당하는 전용 힙.**

malloc(), free(), new, delete**로 할당하는 메모리는 해당 힙을 사용.**

```
// vs2008 version
int __cdecl _heap_init (
        int mtflag
        )
{
#if defined _M_AMD64 || defined _M_IA64
        // HEAP_NO_SERIALIZE is incompatible with the LFH heap
        mtflag = 1;
#endif  /* defined _M_AMD64 || defined _M_IA64 */
        //  Initialize the "big-block" heap first.
        if ( (_crtheap = HeapCreate( mtflag ? 0 : HEAP_NO_SERIALIZE,
                                     BYTES_PER_PAGE, 0 )) == NULL )
            return 0;
```

CRT Heap

VS2012 부터 Default Process Heap을 사용.

```
// vs2012 version
int __cdecl _heap_init (void)
{
        //  Initialize the "big-block" heap first.
        if ( (_crtheap = GetProcessHeap()) == NULL )
            return 0;

        return 1;
}
```

VS2015부터는 heapinit.c 를 찾아 볼 수 없었지만…
VMmap을 사용해서 확인해본 결과 VS2015도 마찬가지.

Private Heap

HeapCreate() 함수를 통해서 추가로 생성한 전용 힙.

```
HANDLE privateHeap = HeapCreate( 0, 1, 1 );
```

Address	Type	Size	Committed	Private	Total WS	Private WS	Shar...	Sh...	Lo...	Bl...	Protection	Details
00CD0000	Heap (Private Data)	4 K	4 K	4 K	4 K	4 K				1	Read/Write	Heap ID: 2 [COMPATABILITY]
00CD0000	Heap (Private Data)	4 K	4 K	4 K	4 K	4 K					Read/Write	Heap ID: 2 [COMPATABILITY]

Windows Page 크기에 정렬되어 할당되는 것을 알 수 있음.

```
HANDLE privateHeap2 = HeapCreate( 0, 7 * KB, 7 * KB );
```

Address	Type	Size	Committed	Private	Total WS	Private WS	Shar...	Sh...	Lo...	Bl...	Protection	Details
00660000	Heap (Private Data)	8 K	8 K	8 K	8 K	8 K				1	Read/Write	Heap ID: 3 [COMPATABILITY]
00660000	Heap (Private Data)	8 K	8 K	8 K	8 K	8 K					Read/Write	Heap ID: 3 [COMPATABILITY]

Heap Manager

(주의) 최신 윈도우에 관한 내용은 아님.

크게 두가지로 나눌 수 있음.

Front-end Allocator

Back-end Allocator

Front-end Allocator

Back-end Allocator**의 최적화 계층.**

Windows XP**까지** Lookaside Lists

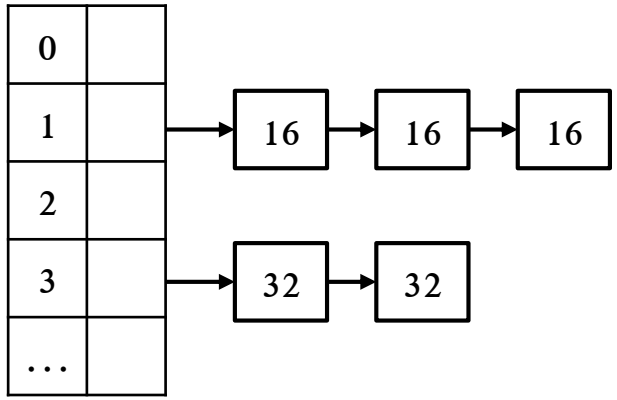
Windows Vista **이후** Low Fragmentation Heap

Lookaside Lists

128개의 단방향 연결 리스트 테이블로 구성.

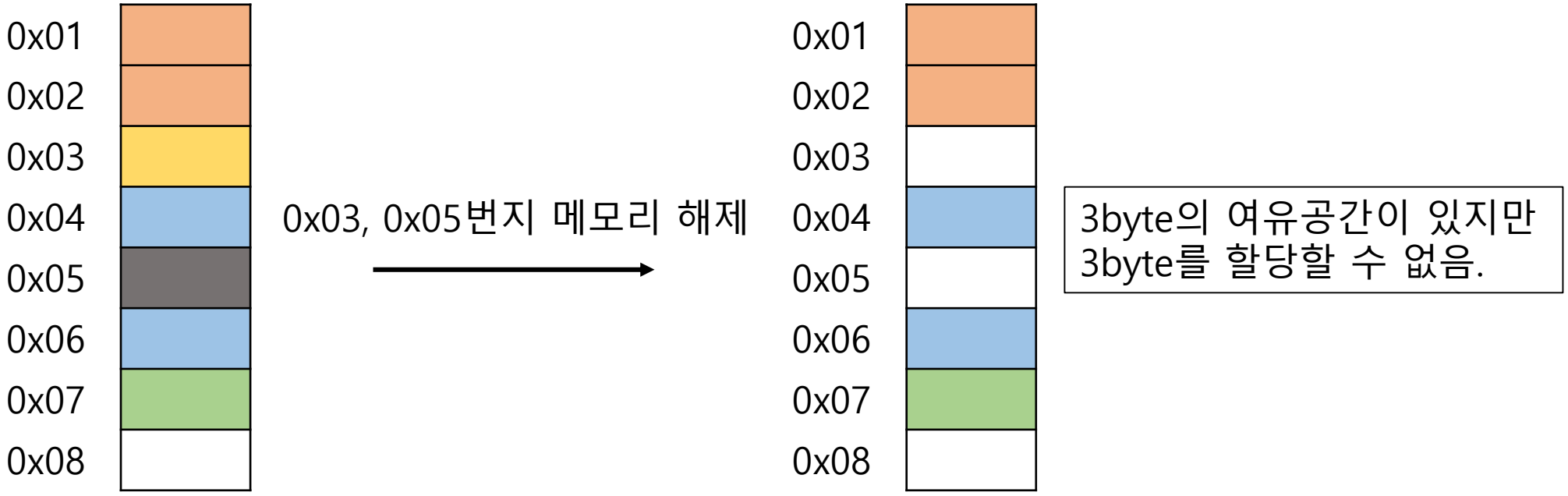
테이블 내의 단방향 연결 리스트에는 16byte에서 특정 크기에 이르는 메모리 블록이 포함.

메모리 블록의 8byte는 메타 데이터로 사용함. 실제 사용 공간은 16byte 블록의 경우 8byte가 됨.



Low Fragmentation Heap

메모리 단편화는 잦은 할당과 해제로 인해 메모리가 여러 조각으로 나뉘는 현상.
큰 크기의 메모리를 할당 받으려고 할 때 할당이 실패할 수 있음.



Low Fragmentation Heap

단편화를 피하기 위한 할당자.

미리 정의된 크기가 다른 블록을 할당. 이를 Bucket 이라고 함.

요청된 크기에 가장 잘 맞는 Bucket을 반환.

Buckets	Granularity	Range
1~32	8	1~256
33~48	16	257~512
49~64	32	513~1024
65~80	64	1025~2048
81~96	128	2049~4096
97~112	256	4097~8194
113~128	512	8195~16384

Low Fragmentation Heap

최대 16KB 까지의 할당요청에 대응.

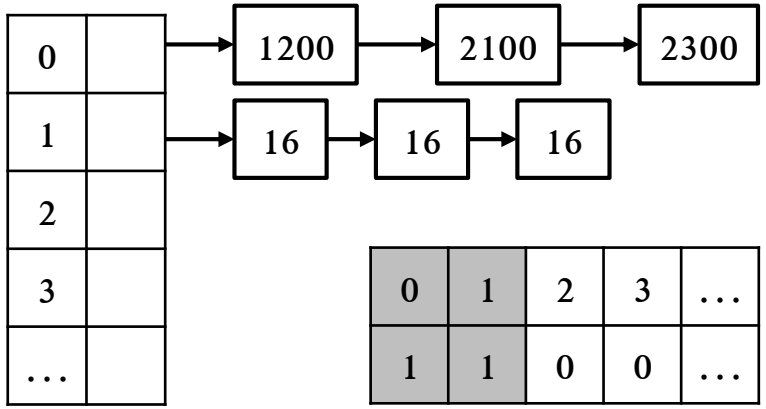
HEAP_NO_SERIALIZE 혹은 고정크기로 생성된 힙에는 적용되지 않음.

LFH를 힙에 적용했을 경우 다시 끌 수 없음.

Back-end Allocator

Front-end Allocator가 할당 할 수 없는 요청을 처리하는 할당자.

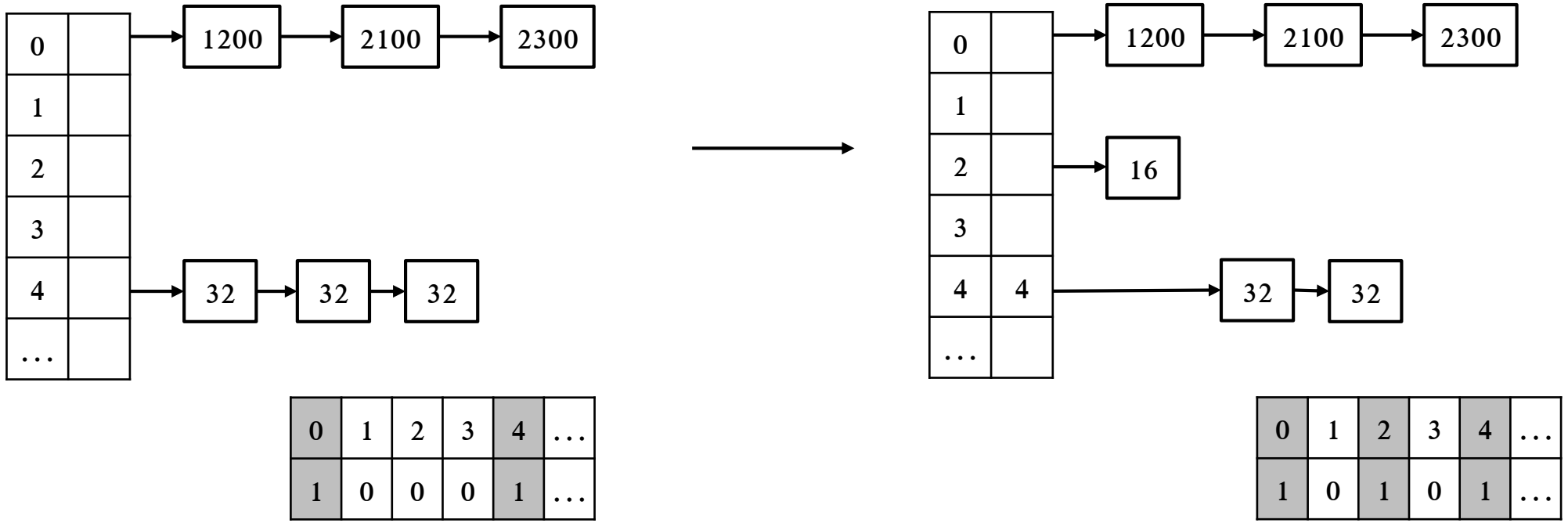
일반적으로 Free List로 불리는 리스트 테이블과 Free List Bitmap이라고 불리는 자료구조를 사용.



Back-end Allocator

요청 받은 크기의 프리 힙 블록을 찾을 수 없을 경우 Chunk Splitting**을 통해 작은 크기로 나눠 반환.**

8byte 할당 요청이 왔다면...

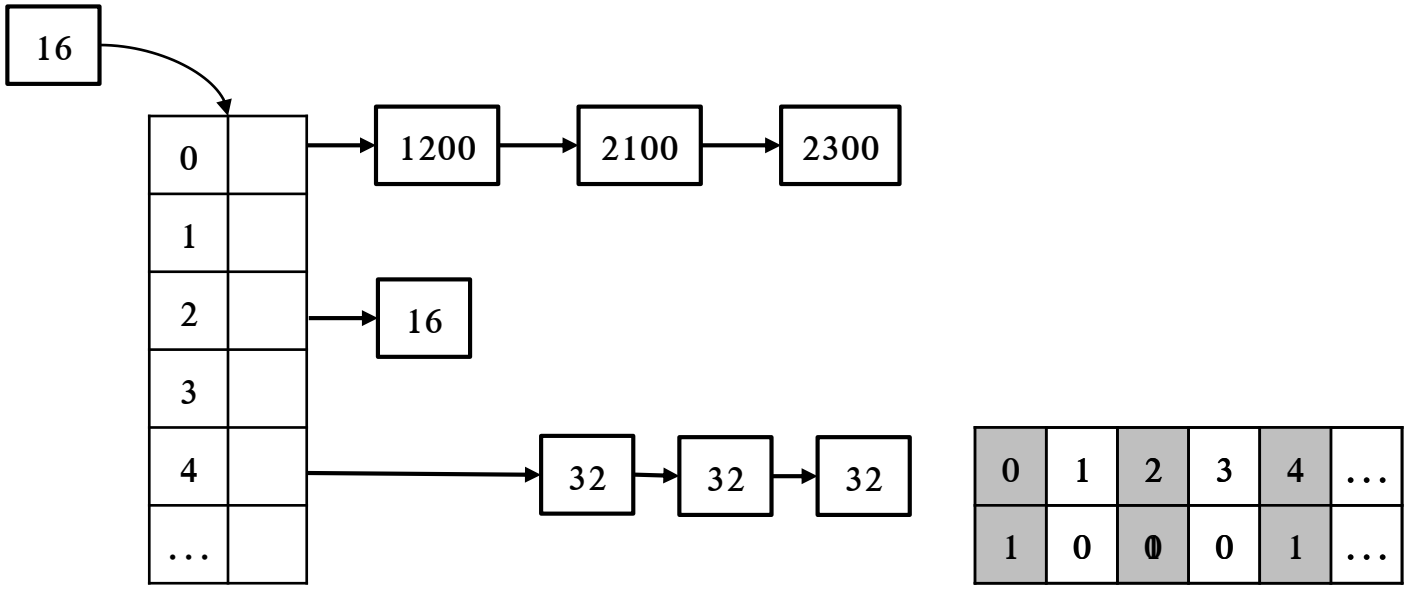


Back-end Allocator

메모리 해제시에는 주변의 프리 블록을 살펴보고 힙을 합병함.

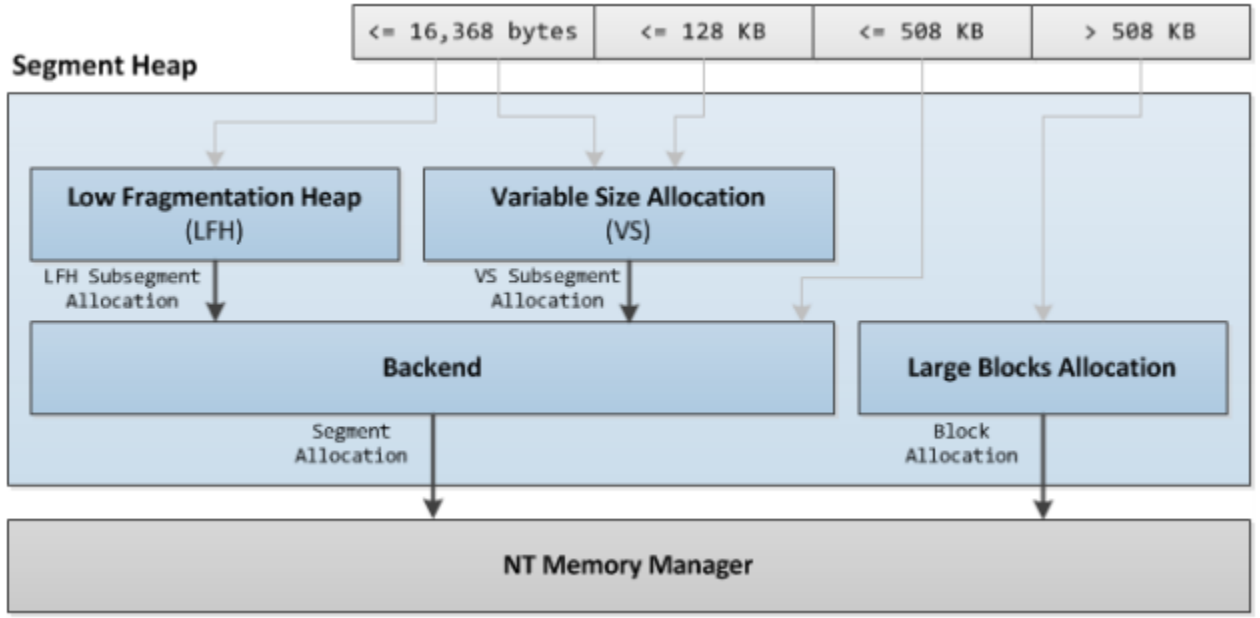
→ Heap Coalescing

이는 단편화를 피하기 위함.



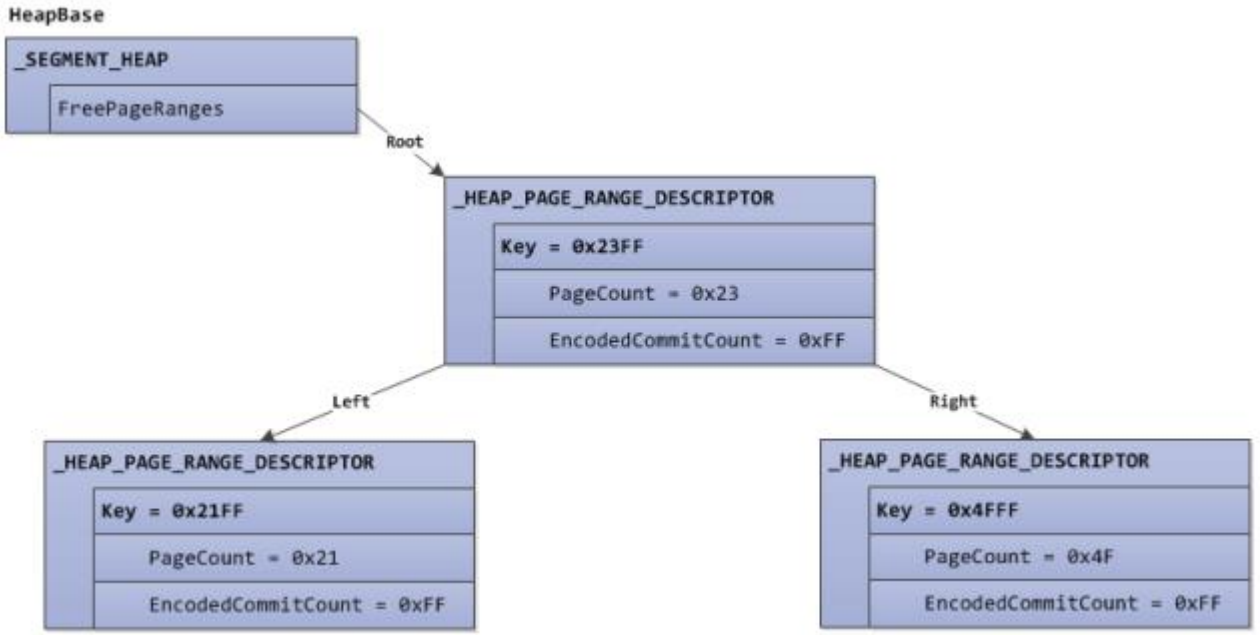
Heap Manager

윈도우 10의 Segment Heap의 경우 이런 모양



Heap Manager

Free List**도** Free Tree**로 변화**



Heap Manager

윈도우 Heap Manager는 동기화를 위해 싱글 스레드로 동작.
메모리 할당, 해제에 거는 Lock이 성능에 병목.
HEAP_NO_SERIALIZE flag로 생성하면 동기화 과정이 생략.

```
HeapCreate( HEAP_NO_SERIALIZE, ..., ... );
```

하지만 지금은 멀티 코어 시대.
여러 스레드에서의 메모리 할당, 해제 요청에 대한 빠른 처리가 필요.

Jemalloc

Facebook, Firefox, Redis 에서 사용하는 메모리 할당자.
단편화 회피와 확장성 있는 동시성 지원을 목적으로 함.
메모리 사용에 관련된 여러 문제를 해결하기 위한 툴도 제공.
보편적으로 쓰이는지 모르겠지만 책에는 TCMalloc만 언급하여 소개.

Jemalloc

Jemalloc은 메모리를 Chunk 단위(4MB)로 할당하는데 메모리 단편화를 피하기 위해서 LFH와 같이 미리 정의된 크기로 메모리 공간을 나눠 놓음.

64bit system의 기본 설정에서 일부분을 살펴보면 아래와 같음.

Category	Spacing	Size
Small	lg	[8]
	16	[16, 32, 48, ..., 128]
	...	...
Large	4 KiB	[4 KiB, 8 KiB, 12 KiB, ..., 4072 KiB]
Huge	4 MiB	[4 MiB, 8 MiB, 12 MiB, ...]

Jemalloc

Arena라고 불리우는 메모리 공간을 복수 할당하여 Lock 경합을 완화.

Arena의 최대 개수는 CPU 코어 수에 따라서 다름.

Single-core CPU에서 1개 Multi-core CPU에서 코어 수의 4배.

```
ncpus = malloc_ncpus();


…

if (opt_narenas == 0) {
    if (ncpus > 1)
        opt_narenas = ncpus << 2;
    else
        opt_narenas = 1;
}
```

Jemalloc

Arena는 Small, Large **카테고리의 메모리 요청을 처리.**

Huge**의 경우 별도로 처리 됨.**

```
void* imalloct(size_t size, bool try_tcache, arena_t *arena)
{
    assert(size != 0);

    if (size <= arena_maxclass)
        return (arena_malloc(arena, size, false, try_tcache));
    else
        return (huge_malloc(size, false, huge_dss_prec_get(arena)));
}
```

Jemalloc

스레드에 대한 Arena의 배정은기본적으로 Round Robin.

한번 배정되면 TSD (Thread Specific Data)라고 불리는 전용 자료구조에 의해서 연결되어 다시 아레나를 찾지 않음.

하나의 Arena를 여러 스레드가 배정받을 수도 있음.

따라서 Arena에 대한 할당 요청은 lock이 필요.

```
void* arena_malloc_large(arena_t *arena, size_t size, bool zero)
{
    void *ret;
    UNUSED bool idump;

    /* Large allocation. */
    size = PAGE_CEILING(size);
    malloc_mutex_lock(&arena->lock);
    ret = (void *)arena_run_alloc_large(arena, size, zero);
    ...
```

Jemalloc

TCMalloc과 같이 스레드 별 캐시 자료구조를 두어(tCache) Lock이나 아레나를 탐색하지 않고 메모리를 할당 받을 수 있게 함.

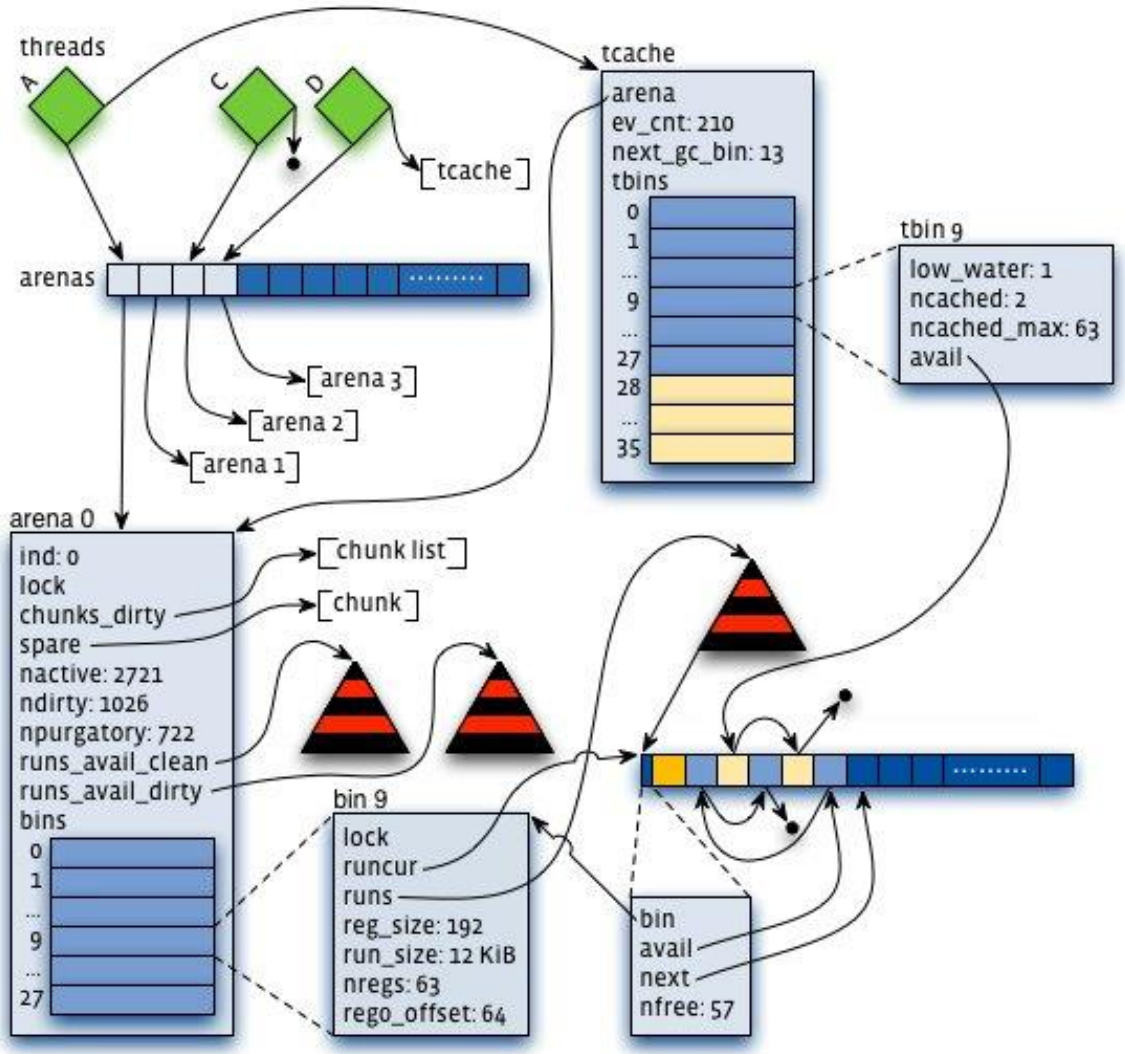
```
void* tcache_alloc_small(tcache_t *tcache, size_t size, bool zero)
{
    void *ret;
    size_t binind;
    tcache_bin_t *tbin;

    binind = SMALL_SIZE2BIN(size);
    assert(binind < NBINS);
    tbin = &tcache->tbins[binind];
    size = arena_bin_info[binind].reg_size;
    ret = tcache_alloc_easy(tbin);
    ...
```

Jemalloc

그만큼 메모리 사용량은 증가.

그래서인지 Firefox는 꺼놓았다고 함.

Jemalloc


threads
A
C
D
tcache
arenas
arena 3
arena 2
arena 1
tcache
arena
ev_cnt: 210
next_gc_bin: 13
tbins
0
1
9
27
28
35
tbin 9
low_water: 1
ncached: 2
ncached_max: 63
avail
arena 0
ind: 0
lock
chunks_dirty
spare
nactive: 2721
ndirty: 1026
npurgatory: 722
runs_avail_clean
runs_avail_dirty
bins
0
1
9
27
chunk list
chunk
bin 9
lock
runcur
runs
reg_size: 192
run_size: 12 KiB
nregs: 63
rego_offset: 64
bin
avail
next
nfree: 57

Provide Class-Specific Memory Mangers

지금까지 살펴본 내용은 범용적인 메모리 할당에 관한 이야기.

만약 특정 클래스에서 new operator를 구현하면 전역 new operator가 호출되지 않아 커스터마이즈 된 메모리 할당 방식이 사용가능.

특정 클래스에 대한 메모리 할당자는 모든 메모리 할당 요청이 동일한 크기로 요청된다는 점을 이용하여 효율적인 메모리 할당이 가능.

Fixed-Size-Block Memory Manager

고정크기 메모리 매니저는 아래와 같은 이유로 매우 효율적.

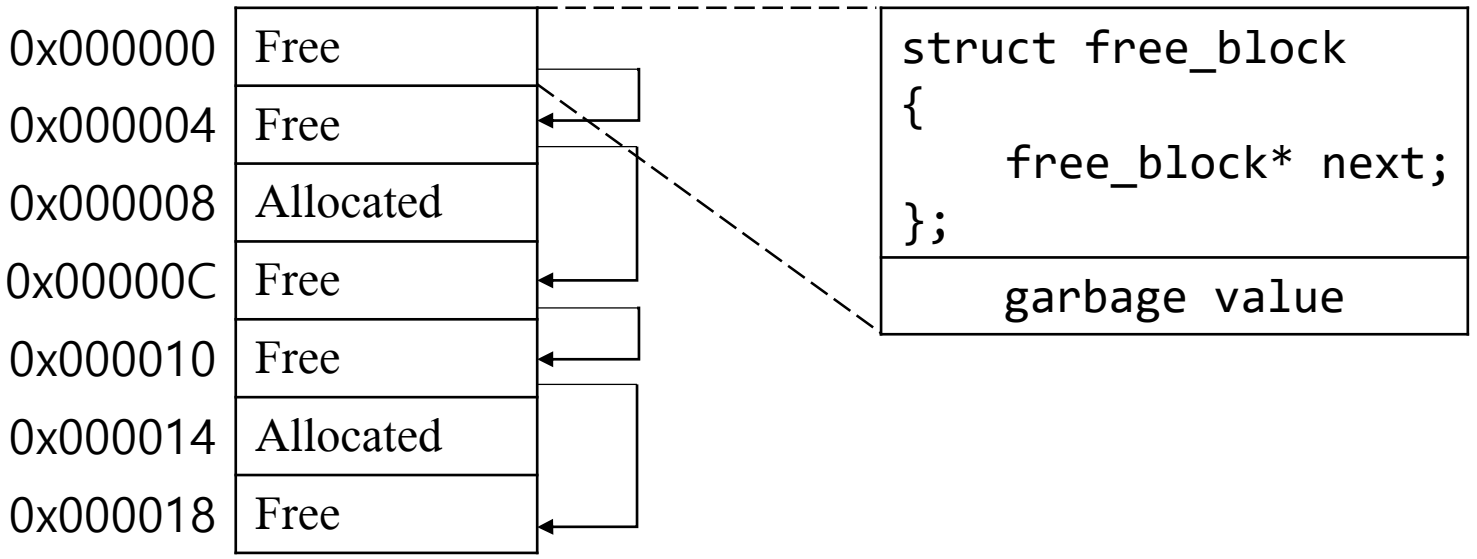
1. 모든 요청의 크기가 같으므로 단편화가 발생하지 않음.
2. 메모리 오버헤드가 적게 구현가능.
3. 메모리 상한선을 지정가능.
4. 할당, 해제 작업이 매우 간단하여 효과적으로 인라인화 가능.
5. 캐시 친화적.

Fixed-Size-Block Memory Manager

구현은 헤더 하나로 매우 간단함.

Fixed-Size-Block Memory Manager

결국 고정크기 메모리 매니저는 메모리 블록의 Single Linked-list.



Fixed-Size-Block Memory Manager

전역 메모리 관리자와의 성능 측정을 두 가지 방식으로 진행.

1. 1,000,000번의 할당을 연속적으로 수행해서 성능을 측정.

2. 길이가 100인 포인터 배열을 만들어 임의의 위치에 객체를 할당, 이미 할당되어 있으면 객체를 해제. 이 과정을 1,000,000번 반복.

확실한 성능개선이 있음.

단위 : millisecond

	저자의 측정		직접 측정	
	Fixed-Size	Global	Fixed-Size	Global
1.	4	64	10	73
2.	25	107	39	83

Fixed-Size-Block Memory Manager

여기서 구현한 고정크기 메모리 관리자는 매우 간단하여 몇가지 아쉬운 부분이 있을 수 있음.

프로그램에 따라서 다음과 같은 변형을 생각해 볼 수 있음.

1. 고정 크기의 Arena로 초기화 하지 않고 free list가 비어 있으면 malloc으로 메모리를 할당.
2. 필요할 경우 malloc이나 new를 호출하여 추가적인 Arena를 할당.
3. 클래스의 인스턴스가 한동안 사용되고 모두 버려진다면 고정 크기 메모리 관리자를 메모리 풀과 같이 사용가능. 메모리 풀은 할당은 기존과 동일하게 수행되지만 메모리를 전혀 해제하지 않음.

Fixed-Size-Block Memory Manager

고정 크기 메모리 관리자는 스레드 안전하지 않기 때문에 효율적이기도 함.

스레드 안전하지 않은 메모리 관리자가 효율적인 이유는 두가지가 있음.

1. 임계 영역에 대한 동기화가 필요 없음. Memory fence는 동기화의 핵심인데 비용이 높음.
2. 전역 메모리 관리자를 사용하기 위해서 다른 스레드와 경합하지 않아도 됨.

스레드 안전하지 않은 메모리 관리자는 스레드 안전한 메모리 관리자보다 쉽게 작성할 수 있음.

임계 영역을 줄이기 위해서 스레드 안전한 메모리 관리자는 복잡해 질 수 있음.

Provide Custom Standard Library Allocators

STL 컨테이너는 동적 메모리를 많이 사용.

STL 컨테이너에 고정 크기 메모리 관리자와 같은 커스텀 메모리 관리자를 포함하여 성능향상을 기대해 볼 수 있음.

다만 STL 컨테이너가 유저가 제공한 자료형이 아닌 숨겨진 자료형을 동적으로 할당하는 경우가 있음.

```
template<class _Value_type, class _Voidptr>
struct _List_node
{// list node
_Voidptr _Next;// successor node, or first element if head
_Voidptr _Prev;// predecessor node, or last element if head
_Value_type _Myval;// the stored value, unused if head

private:
_List_node& operator=(const _List_node&);
};
```

Provide Custom Standard Library Allocators

STL의 코드를 고칠 수는 없으므로 해당 자료형에 operator new()와 operator delete()를 추가하는 방식은 사용할 수 없음.

다행이도 STL 컨테이너는 Allocator 인자를 통해서 특정 클래스에 operator new()를 추가하여 커스텀 메모리 관리자를 사용하는 것과 같은 효과를 누릴 수 있음.

Allocator는 메모리를 관리하는 템플릿 클래스로 3가지 일을 수행함.

1. 메모리 관리자로부터 저장 공간을 가져옴.
2. 저장 공간을 메모리 관리자에 반납.
3. 연관된 메모리 관리자로부터 자신을 복사 생성.

Provide Custom Standard Library Allocators

기본 할당자인 std::allocator<T>는 ::operator new()를 감싼 얇은 래퍼 클래스로 개발자가 원한다면 다른 동작을 하는 별도의 할당자를 제공할 수 있음.

할당자는 기본적으로 2종류로 나눌 수 있음.

1. 상태를 가지지 않은 할당자
2. 상태를 가진 할당자

std::allocator<T>는 상태를 가지지 않은 할당자임.

Stateless Allocator

상태를 가지지 않은 할당자는 몇 가지 매력적인 속성을 가지고 있음.

1. 기본 생성자를 가짐.
2. 컨테이너 클래스의 크기를 증가시키지 않음 (zero-byte base class)
3. 서로 간에 구별이 없음. 어떤 할당자에서 할당한 메모리를 다른 할당자에서 해제할 수 있음. 즉 AllocX<T>와 AllocX<U>는 같음.

다만 모든 할당자가 같은 메모리 관리자로부터 메모리를 가져와야 함으로 전역 의존성을 가짐.

Allocator with internal state

상태를 가지는 할당자는 다음과 같은 이유로 만들고 사용하기에 좀 더 복잡함.

1. 대부분의 경우 기본 생성자를 통해 생성할 수 없음.
2. 할당자의 상태를 모든 변수에 저장해서 크기가 증가함.
3. 같은 타입의 할당자를 비교했을 때 같지 않을 수 있음.

다만 다양한 목적을 위한 여러 크기의 메모리를 쉽게 생성할 수 있음. 유연함.

Minimal C++11 Allocator

C++11 표준을 완전히 따르는 컴파일러를 사용한다면 약간의 정의로 할당자를 사용할 수 있음.

예제 코드는 std::allocator와 거의 유사한 역할을 하는 할당자 임.

Minimal C++11 Allocator

최소한의 할당자는 다음과 같은 함수를 포함하고 있어야 함.

1. Default Constructor
2. Copy Constructor
3. T* allocate(size_type n, const void* hint = 0)
4. deallocate(T* p, size_t n)
5. bool operator==(const allocator& a) const
   bool operator!=(const allocator& a) const

Minimal C++11 Allocator

Default Constructor

할당자가 기본 생성자를 가지고 있으면 개발자는 명시적으로 할당자의 인스턴스를 생성해서 컨테이너의 생성자로 건내 줄 필요가 없음.

상태를 가지지 않는 할당자에서는 대부분 비어 있음.

보통 상태를 가진 할당자에서는 존재하지 않음.

Minimal C++11 Allocator

Copy Constructor

할당자를 연관된 할당자로 변환할 때 사용.
대부분의 컨테이너가 전달된 자료형 T의 노드를 할당하지 않기 때문에 중요.
상태를 가지지 않는 할당자에서는 대부분 비어 있음.
상태를 가진 할당자에서는 내부 상태를 복사해야 함.

Minimal C++11 Allocator

T* allocate(size_type n, const void* hint = 0)

n 바이트를 담는데 충분한 저장 공간의 포인터를 반환하고 실패하면 std::bad_alloc예외를 던짐.
hint는 지역성을 높이기 위해서 메모리 관리자가 참고할 수 있도록 제공하는 주소.

`pointer allocate( size_type n, std::allocator<void>::const_pointer hint = 0 );`	(1)	(until C++17)
`T* allocate( std::size_t n, const void * hint);`	(1)	(since C++17) (deprecated)
`T* allocate( std::size_t n );`	(2)	(since C++17)

Minimal C++11 Allocator

deallocate(T* p, size_t n)

주소 p에서 n 만큼의 저장 공간을 해제.
n 은 allocate() 주소 p의 저장 공간을 할당할 때 사용한 크기와 동일해야 함.

Minimal C++11 Allocator

bool operator==(const allocator& a) const

bool operator!=(const allocator& a) const

두 할당자의 인스턴스가 서로 같은지를 비교하기 위한 비교 연산자.
비교 결과 같다면 서로간 메모리 해제가 자유로움.
상태를 가지지 않는 할당자에서는 그냥 true를 반환.
상태를 가진 할당자에서는 내부 상태를 비교하거나 간단하게 false를 반환.

Additional Definitions for C++98 Allocator

C++11 에서는 할당자를 쉽게 작성할 수 있지만 반면에 컨테이너의 구현이 복잡해 짐.
C++11 이전에서는 앞에서 제시한 함수에 추가로 다음 사항을 포함해야 함.

```
template <typename T> struct my_allocator_98
{
    typedef T value_type;
    typedef T* pointer;
    typedef const T* const_pointer;
    typedef T& reference;
    typedef const T& const_reference;
    typedef std::size_t size_type;
    typedef std::ptrdiff_t difference_type;

    pointer address( reference r ) { return &r; }
    const_pointer address( const_reference r )
    { return &r; }
```

Additional Definitions for C++98 Allocator

복사 생성자는 비어 놓고 rebind 구조체가 그 역할을 대신함.

```
template <typename U> struct rebind
{
    typedef my_allocator_98<U> other;
};

my_allocator_98( ) {}
template <typename U>
my_allocator_98( const my_allocator_98<U>& ) {}
```

Additional Definitions for C++98 Allocator

또한 명시적으로 생성자와 소멸자를 호출하기 위한 함수가 필요함.

```
void construct( pointer p, const T& t )
{
    new ( p ) T( t );
}

void destroy( pointer p )
{
    p->~T( );
}
```

With Fixed-Block Allocator

STL 컨테이너의 할당자의 allocate(), deallocate() 함수를 조금 고치면 고정 크기 메모리 관리자를 통해 컨테이너의 메모리를 할당할 수 있음.

VS 2015에서 컨테이너의 구현은 Proxy 구조체를 추가적으로 할당하므로 고정 크기 메모리 관리자를 조금 수정할 필요가 있음.

```
template<typename Arena>
inline void *
fixed_block_memory_manager<Arena>::allocate( size_t size )
{
//...
    if ( size > block_size_ )
    {
        throw std::bad_alloc( );
    }
```

With Fixed-Block Allocator

성능 측정 결과는 다음과 같음.

단위 : microsecond

	저자의 측정		직접 측정	
	Fixed-Size	Global	Fixed-Size	Global
1.	11.6	76.2	13	131
2.	67.4	142	108	340

Fixed-Block Allocator for Strings

std::string은 가변적인 길이의 문자열을 다루기 위해 동적인 문자열 배열을 갖고 있음.

고정 크기 메모리 관리자를 적용할 후보가 아닌것 같지만 프로그램 내에서 사용하는 문자열의 최대 길이를 알고 있다면 최대 길이 만큼의 메모리를 고정적으로 할당하게 할 수 있음.

```
T* allocate( std::size_t n, const void* = 0 )
{
    return reinterpret_cast<T*>( string_memory_manager.allocate( 512 ) );
}
```

With Fixed-Block Allocator

성능 측정 결과는 다음과 같음.

단위 : millisecond

	저자의 측정		직접 측정	
	Fixed-Size	Global	Fixed-Size	Global
1.	1124	2693	11	32

충분히 빠르지만 Ch 4.에서 훨씬 빠르게 만드는 방법을 보았음.

커스텀 메모리 관리자를 사용하면 프로그램을 효율적으로 만들 수 있음. 하지만 메모리 관리자 호출 자체를 줄이도록 최적화하는 것 보다는 덜 효과적임.

Reference

- 실전 윈도우 디버깅 (원제: advanced windows debugging)
- WINDOWS 10 SEGMENT HEAP INTERNALS
- Jemalloc Manual Page
- Scalable memory allocation using jemalloc
- Exploiting the jemalloc Memory Allocator: Owning Firefox's Heap