기반 작업
(6) GCC garbage collection

정원교
weongyo@hotmail.com

2004년 2월 23일

목 차

제 1 절 13 주째 강의를 시작하며

날씨가 너무 덥습니다. 후덥지근한 방에서 지낼려고 하니 숨이 콱콱 막히는 것 같습니다. 그나마 다행인 건 지금 창문 너머로 비가 오고 있네요. 휴가들 즐겁게 보내시기 바랍니다. 이번주는 “이삭줍기” 라고 불리는 “Garbage Collection” 에 대해서 기술하도록 하겠습니다. Garbage Collection 에 대해 더 많은 자료를

원하신다면 요약을 잘 해놓은 “Uniprocessor Garbage Collection Techniques” 를 읽어보시기 바랍니다. 아래의 링크에서 구할 수 있습니다.

ftp://ftp.cs.utexas.edu/pub/garbage/gcsurvey.ps

제 2 절 기본적인 내용

GCC 에서는 simple garbage collection 방식인 mark-sweep 방식을 사용하고 있는데, mark-sweep 방식의 경우 아래와 같이 동작하게 된다.

1. 주기적으로 전체 메모리를 탐색하여 garbage 를 찾음
2. Local 변수들과 배열, stack 에서 참조하는 각 object 를 ‘유효함’ 상태로 체크
3. 유효하다고 체크된 각 객체들이 참조하는 다른 객체들도 역시 ‘유효함’

그래서 ‘mark 단계’에서는 유효한 object 에 대해 구분을 하는 단계를 거쳐 ‘sweep 단계’에서는 유효하다고 판단되지 않은 것들에 대한 청소를 하게 된다. 그림 1 은 Mark-Sweep 에 대한 간단한 예제를 나타낸 것으로써, 회색으로 색칠된 것이 mark 되지 못한 것을 가르킨다.

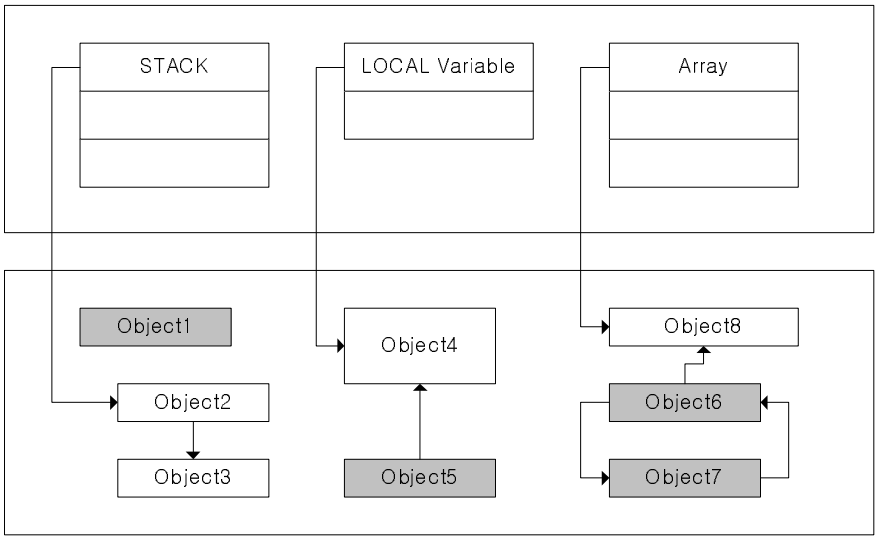

그림 1: Mark-Sweep 방식에 대한 간단한 그림

GCC 에서 Garbage Collection 을 어떤 방법으로 사용할 것인가에 대한 고민을 해야 한다. 선택의 폭은 2 가지가 존재한다. 하나는 MMAP 을 사용하는 것과 또 다른 하나는 Malloc 를 사용하되 page_group 으로 나누어 생각하는 방법이다. 기본적으로 gcc 에서는 MMAP 을 사용하는 것을 추천하고 있다. USING_MMAP 매크로와 USING_MALLOC_PAGE_GROUPS 는 동시에 선언될 수 없으며 둘 중 하나만 선택될 수 있다.

2.1 전략

아래 설명은 $prefix/gcc/ggc-page.c 파일에 설명되어 있는 구현 방식에 대해서 언급한 내용이다. 이 설명을 이해하기 위해서는 아래에 나와 있는 struct globals * 구조체 (3.1 절, 4 쪽 참고)와 struct page_entry * 구조체 (3.2 절, 5 쪽 참고) 를 한번 보길 바란다.

> 이 garbage-collecting allocator 는 page 들의 집합 중 하나에다 object 들을 할당한다. 각 page 는 single size 의 object 들만 할당할 수 있다; 즉, 이용 가능한 size 들은 4 바이트에서 시작하는 2 의 제곱값이다. ‘할당 요청’의 size 는 다음 2 (‘order’) 의 제곱까지 순회하게 되며 적당한 page 에 만족하게 된다.

각 page 는 page-entry 내에 기록되게 되는데, 이것은 또한 그 page 상의 object 위치에 관한 in-use bitmap 을 유지하고 있다. 이것은 특정 object 의 '할당 상태' 를 그 page 자신을 만지는 일 없이 알 수 있게 한다.

각 page-entry 또한 context depth 를 가지고 있는데, 이것은 '할당 context' 들의 push 혹은 pop 되는 상황을 추적하는데 사용된다. 오직 현재 (가장 높은 수를 가진) context 내에 할당된 object 들만 수집될 것이다.

Page entry 들은 단순-linked list 들의 배열속에 정리된다. 그 배열은 그것 상에 존재하는 page 들의 (bits 로) '할당 size' 순으로 나열된다; 예를 들면, 목록상의 모든 page 들은 같은 size 의 object 들을 할당한다. Page 들은 목록상에 다음과 같이 순서화된다. 모든 non-full page 들은 모든 full page 들보다 선행하며 non-full page 들은 context depth 가 감소하는 순서대로 정리된다.

(모든 순서에서의) empty page 들은 single page cache 목록상에 유지되며 새로운 page 들이 요청되었을 때 가장 먼저 고려된다; 이것들은 다음 collection 의 시작되는 시점까지 재활용되지 않는다면 시작하는 부분에서 사라지게 된다.

2.2 2 단계 paging

주어진 포인터에 대한 page-entry 를 찾는데 2 단계 tree 가 사용된다. 포인터의 bit 들의 두 chunk 들은 아래와 같이 tree 의 첫번째와 두번째 단계를 찾음으로써 추출된다.

```
                          HOST_PAGE_SIZE_BITS
                  32           |      |
msb +----------------+----+------+------+ lsb
                    |    |      |
                PAGE_L1_BITS   |
                         |     |
                     PAGE_L2_BITS
```

```
#define PAGE_L1_BITS    (8)
#define PAGE_L2_BITS    (32 - PAGE_L1_BITS - G.lg_pagesize)
#define PAGE_L1_SIZE    ((size_t) 1 << PAGE_L1_BITS)
#define PAGE_L2_SIZE    ((size_t) 1 << PAGE_L2_BITS)

#define LOOKUP_L1(p) \
  (((size_t) (p) >> (32 - PAGE_L1_BITS))
   & ((1 << PAGE_L1_BITS) - 1))

#define LOOKUP_L2(p) \
  (((size_t) (p) >> G.lg_pagesize)
   & ((1 << PAGE_L2_BITS) - 1))
```

page-entry page 들이 시스템 page boundary 들상에 align 되기 때문에 최하단 HOST_PAGE_SIZE_BITS 는 무시된다. 다음으로 가장 중요한 PAGE_L2_BITS 와 PAGE_L1_BITS 는 lookup table 에서는 각각 두번째와 첫번째 index 값들이다.

32-bit architecture 들과 설정들은 아래에 있으며 남는 bit 들은 없다. 더 넓은 포인터를 사용하는 architecture 에 대해서는 lookup tree 는 올바른 것을 찾기 위해 반드시 스캔되어야 하는 page 들의 list 를 가르키고 있다.

제 3 절 GC 를 이루는 구조체들

아래부터는 GCC 에서 GC 를 구현하기 위해 사용한 구조체에 대해 알아 보도록 하겠다.

3.1 struct globals

이 구조체는 GC root 정보를 가지는 전역변수 roots, d_htab_roots 를 제외한 GC 의 전체적인 정보를 가지고 있는 구조체이다. (물론 모든 정보를 가지는 것은 아니다.)

```
#define NUM_ORDERS (HOST_BITS_PER_PTR + NUM_EXTRA_ORDERS)

static struct globals {
  page_entry *pages[NUM_ORDERS];
  page_entry *page_tails[NUM_ORDERS];
  page_table lookup;

  size_t pagesize;
  size_t lg_pagesize;
  size_t allocated;
  size_t allocated_last_gc;
  size_t bytes_mapped;

  unsigned short context_depth;

#if defined (HAVE_MMAP_DEV_ZERO)
  int dev_zero_fd;
#endif

  page_entry *free_pages;

#ifdef USING_MALLOC_PAGE_GROUPS
  page_group *page_groups;
#endif

  FILE *debug_file;
} G;
```

표 1: struct globals

page_entry *pages[NUM_ORDERS];

> 이 배열의 N 번째 element 는 2^N 크기의 object 들을 가지는 page 이다. 만약 free object 들을 가진 어떤 page 들이 있다면 그것들은 list 의 head 에 있을 것이다. 만약 이 object size 를 위한 page-entry 들이 없다면 NULL 이다.

page_entry *page_tails[NUM_ORDERS];

> 이 배열의 N 번째 element 는 2^N 크기의 object 들을 가지는 마지막 page 이다. 만약 이 object size 를 위한 page-entry 가 없다면 NULL 이다.

page_table lookup;

> Object 주소들을 가지는 할당 page 들을 연합시키기 위한 lookup table.

size_t pagesize;
size_t lg_pagesize;

> 시스템의 page 크기. 기본적인 32비트 컴퓨터에서는 4096 값을 가지며, 64 비트 컴퓨터의 경우 8096 값을 가지는 경우도 있다. lg_pagesize 는 $\log_2(pagesize)$ 를 나타낸다.

size_t allocated;

현재 할당된 byte 수.

size_t allocated_last_gc;

마지막 collection 의 끝에 현재 할당된 byte 수.

size_t bytes_mapped;

메모리가 mapping 되어진 총 용량.

unsigned short context_depth;

Context stack 내에서의 현재 depth.

int dev_zero_fd;

읽기위해 /dev/zero 를 여는 file descriptor.

page_entry *free_pages;

Free system page 들의 cache.

page_group *page_groups;

아직 설명이 없음

FILE *debug_file;

Ouput 을 debugging 하기 위한 file descriptor.

3.2 struct page_entry

하나의 page_entry 는 '할당 page 의 상태' 를 기록하는 구조체로써 bitmap in_use_p 를 조정하여 동적으로 크기가 변할 수 있다.
struct page_entry *next;

같은 크기의 object 들을 가지는 다음 page-entry, 만약 이것이 마지막 page-entry 라면 NULL 을 가짐.

size_t bytes;

할당된 byte 들의 양. (이것은 항상 host system page size 의 두배 크기일 것이다.)

char *page;

메모리가 할당된 address 위치.

struct page_group *group;

이 page 가 왔던 page group 으로의 back pointer.

unsigned long *save_in_use_p;

Collection 동안은 가장 상단 context 에는 있지 않은 page 들을 위한 저장된 in-use bit vector.

unsigned short context_depth;

이 page 의 context depth.

unsigned short num_free_objects;

```
typedef struct page_entry
{
  struct page_entry *next;

  size_t bytes;

  char *page;

#ifdef USING_MALLOC_PAGE_GROUPS
  struct page_group *group;
#endif

  unsigned long *save_in_use_p;
  unsigned short context_depth;
  unsigned short num_free_objects;
  unsigned short next_bit_hint;
  unsigned char order;

  unsigned long in_use_p[1];
} page_entry;
```

표 2: struct page_entry 구조체

이 page 상에 남아있는 free object 들의 갯수.

unsigned short next_bit_hint;

이 page 로부터 다음 할당을 위한 free object 의 bit 위치 중 선호되는 것.

unsigned char order;

이 page 로 부터 할당된 object 들의 size 의 lg 값.

unsigned long in_use_p[1];

object 가 사용중인지 아닌지를 나타내는 bit vector. 이 page 상의 N 번째 object 가 할당되었다면 N 번째 비트는 1 이다. 이 배열은 동적으로 크기가 변한다.

GCC 내용을 살펴보면 'the one-past-the-end' bit 라는 개념이 사용되는데, 이는 OBJECT_PER_PAGE (order) 에 따른 in_use_p 의 크기에서 마지막을 가르키는 비트를 항상 1 로 설정해 준다는 말이다. 부연한다면, OBJECT_PER_PAGE (order) 의 크기가 8 이고 이에 대해 ont-page-the-end bit 를 설정한다는 말은 아래와 같은 의미가 된다. 아래에서 num_objects 가 OBJECT_PER_PAGE (order) 의 크기를 가르킨다.

```
p->in_use_p[num_objects / HOST_BITS_PER_LONG]
  = ((unsigned long) 1 << (num_objects % HOST_BITS_PER_LONG));
```

3.3 struct page_group

page_group 은 malloc 으로부터의 큰 할당을 표현하거나, align 되어 있는 page 들을 분해한 큰 할당을 표현할 때 사용된다.

```
#ifdef USING_MALLOC_PAGE_GROUPS
typedef struct page_group {
```

```
  struct page_group *next;

  char *allocation;

  size_t alloc_size;

  unsigned int in_use;
} page_group;
#endif
```

struct page_group *next;

현존하는 모든 page group 들에 관한 연결 리스트

char *allocation;

malloc 로부터 받은 주소

size_t alloc_size;

block 의 크기

unsigned int in_use;

사용중인 page 들에 대한 bitmask 이다.

제 4 절 GC root 들과 통계 구조체

이 절에서는 $prefix/gcc/ggc-common.c 파일에 선언되어 있는 구조체와 전역 변수에 대해서 알아 보자.

4.1 struct ggc_root

이 구조체는 GC 가 수행되는 동안에 보존할 global root 들을 유지한다. 구조체의 내용은 아래와 같다.

```
struct ggc_root {
  struct ggc_root *next;
  void *base;
  int nelt;
  int size;
  void (*cb) PARAMS ((void *));
};

static struct ggc_root *roots;
```

이 구성요소로의 추가는 ggc_add_root () 함수에서 담당하게 되고, 삭제의 경우 ggc_del_root () 함수에서 이루어지게 된다.

4.2 struct d_htab_root

struct d_htab_root 구조체는 모든 root 들이 처리되었을 경우 scan 될 hash table 을 추가한다. 우리는 그 때까지 mark 되어 있지 않은 table 내의 어떤 entry 에 대해서 삭제한다. 내부적으로 포함되어 있는 요소에 대해서도 나열해 놓았다.

```
struct d_htab_root {
  struct d_htab_root *next;
  htab_t htab;
  ggc_htab_marked_p marked_p;
  ggc_htab_mark mark;
};

struct htab {
  htab_hash hash_f;
  htab_eq eq_f;
  htab_del del_f;

  PTR *entries;

  size_t size;
  size_t n_elements;
  size_t n_deleted;

  unsigned int searches;
  unsigned int collisions;

  int return_allocation_failure;
};

typedef int (*ggc_htab_marked_p) PARAMS ((const void *)); typedef
void (*ggc_htab_mark) PARAMS ((const void *));
```

struct htab 구조체에 대해 설명을 해보면 Hash table 은 아래의 type 이다. 이 type 의 구조체 (수행) 은 hash table 들의 사용을 필요로 하지 않는다. hash table 에 관한 모든 수행은 아래에 언급되어 있는 함수들만 통해서 수행되어져야 한다.

각 구성요소에 대한 설명은 아래와 같다.

htab_hashhash_f;

Hash 함수로의 포인터.

htab_eq eq_f;

비교 함수로의 포인터.

htab_del del_f;

Cleanup 함수로의 포인터.

PTR *entries;

Table 그 자신.

size_t size;

Hash table 의 현재 (entry 로) 크기

size_t n_elements;

삭제된 element 들을 포함하여 현재 element 들의 갯수

size_t n_deleted;

Table 에서 현재 삭제된 element 들의 갯수

unsigned int searches;

다음의 구성요소는 debugging 에 사용된다. 이것의 값은 hash table 용 'htab_find_slot' 의 호출된 모든 횟수이다.

unsigned int collisions;

다음의 구성요소는 debugging 에 사용된다. 이것의 값은 hash table 의 수행 시간동안 수정된 collision 들의 갯수이다.

int return_allocation_failure;

우리가 메모리 할당을 하는 함수 호출에 대해 NULL 을 반환하는 것이 허락된다면 0 이 아닌 값을 가짐.

4.3 struct ggc_statistics

이 구조체는 모든 collector 들에서 공통되는 통계들을 표현하는데, 특히 collector 들은 이 구조체를 확장할 수 있다.

```
typedef struct ggc_statistics {
  unsigned num_trees[256];
  size_t size_trees[256];
  unsigned num_rtxs[256];
  size_t size_rtxs[256];
  size_t total_size_trees;
  size_t total_size_rtxs;
  unsigned total_num_trees;
  unsigned total_num_rtxs;
} ggc_statistics;
```

위 구조체에 대한 설명은 아래와 같다.

unsigned num_trees[256];

I 번째 element 는 code I 에 할당된 node 들의 갯수이다.

size_t size_trees[256];

I 번째 element 는 code I 에 할당된 byte 의 크기이다.

unsigned num_rtxs[256];

I 번째 element 는 code I 에 할당된 node 들의 갯수이다.

size_t size_rtxs[256];

I 번째 element 는 code I 에 할당된 byte 의 크기이다.

size_t total_size_trees;

할당된 tree node 들의 총 크기.

size_t total_size_rtxs;

할당된 RTL node 들의 총 크기.

unsigned total_num_trees;

할당된 tree node 들의 총 갯수.

unsigned total_num_rtxs;

할당된 RTL node 들의 총 갯수.

제 5 절 init_ggc

'언어 독립적인 초기화' 과정을 담당하는 lang_independent_init () 함수에 의해 호출되는 이것은 그림 2 에서 볼 수 있는 것처럼 기본적인 struct globals * 구조체의 구성요소와 새로운 struct page_entry * 구조체를 할당하고 초기화한다. 그리고 GC 에서 사용되는 독립적인 전역변수 (예를 들면, object_size_table, extra_order_size_table 등등) 들을 초기화하게 된다.

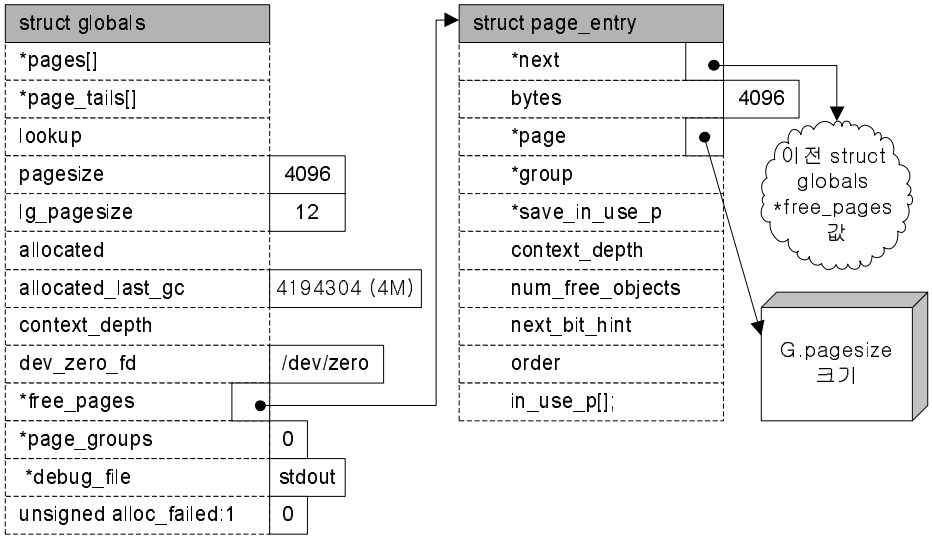

그림 2: init_gcc 에서 struct globals 초기화한 후 모습

아래부터는 GC 에서 구조체내에 포함되는 변수가 아닌 독립적으로 선언되어 있는 전역변수에 대한 설명과 init_ggc () 함수에 의해 초기화가 된후의 값을 나열하도록 하겠다.

object_size_table 변수

I 번째 entry 는 order I 의 page 상에서의 object 의 size 이다.

이 변수의 크기는 NUM_ORDERS 이다. 아래와 같이 설정됨. 앞의 숫자는 order 를 나타내며 뒤는 삽입되는 값이다.

```
00 - 0x1
01 - 0x2
02 - 0x4
03 - 0x8
04 - 0x10
05 - 0x20
06 - 0x40
07 - 0x80
08 - 0x100
09 - 0x200
10 - 0x400
11 - 0x800
12 - 0x1000
13 - 0x2000
14 - 0x4000
15 - 0x8000
```

```
16 - 0x10000
17 - 0x20000
18 - 0x40000
19 - 0x80000
20 - 0x100000
21 - 0x200000
22 - 0x400000
23 - 0x800000
24 - 0x1000000
25 - 0x2000000
26 - 0x4000000
27 - 0x8000000
28 - 0x10000000
29 - 0x20000000
30 - 0x40000000
31 - 0x80000000
32 - 0x70
33 - 0x14
```

extra_order_size_table 변수

I 번째 entry 는 I 번째 extra order 에 저장된 object 의 최대 size 이다. 이 배열에 새로운 entry 를 추가하는 것은 당신이 새로운 특별한 allocation size 를 추가해야 할 필요가 있을 경우에만 하십시오.

objects_per_page_table 변수

I 번째 entry 는 order I 의 page 상에서의 object 의 번호 이다.

다음과 같이 설정된다. 앞의 숫자는 order 를 나타내며 뒤는 삽입되는 값이다.

```
00 - 0x1000     (4096)
01 - 0x800      (2048)
02 - 0x400      (1024)
03 - 0x200      (512)
04 - 0x100      (256)
05 - 0x80       (128)
06 - 0x40       (64)
07 - 0x20       (32)
08 - 0x10       (16)
09 - 0x8        (8)
10 - 0x4        (4)
11 - 0x2        (2)
12 - 0x1        (1)
13 - 0x1        (1)
14 - 0x1        (1)
15 - 0x1        (1)
16 - 0x1        (1)
17 - 0x1        (1)
18 - 0x1        (1)
19 - 0x1        (1)
20 - 0x1        (1)
21 - 0x1        (1)
22 - 0x1        (1)
```

```
23 - 0x1        (1)
24 - 0x1        (1)
25 - 0x1        (1)
26 - 0x1        (1)
27 - 0x1        (1)
28 - 0x1        (1)
29 - 0x1        (1)
30 - 0x1        (1)
31 - 0x1        (1)
32 - 0x24       (36)
33 - 0xcc       (204)
```

size_lookup 변수

이 테이블은 '할당 요청' 에 대해 $\lceil \log_2(size) \rceil$ 의 결정을 빠르게 하기 위해 제공한다. 최소 할당 크기는 8 바이트이다.

기본적으로 설정되어 있는 size_lookup[257] 의 값

```
static unsigned char size_lookup[257] =
{
  3, 3, 3, 3, 3, 3, 3, 3, 3, 4, 4, 4, 4, 4, 4, 4,
  4, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5,
  5, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6,
  6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6,
  6, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7,
  7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7,
  7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7,
  7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7,
  7, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8,
  8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8,
  8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8,
  8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8,
  8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8,
  8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8,
  8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8,
  8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8,
  8
};
```

다음과 같이 변한다.

```
static unsigned char size_lookup[257] =
{
  3, 3, 3, 3, 3, 3, 3, 3, 3, 4, 4, 4, 4, 4, 4, 4,
  4,33,33,33,33, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5,
  5, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6,
  6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6,
  6,32,32,32,32,32,32,32,32,32,32,32,32,32,32,32,
 32,32,32,32,32,32,32,32,32,32,32,32,32,32,32,32,
 32,32,32,32,32,32,32,32,32,32,32,32,32,32,32,32,
 32, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7,
  7, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8,
```

```
    8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8,
    8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8,
    8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8,
    8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8,
    8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8,
    8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8,
    8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8,
    8
  };
```

제 6 절 ggc_alloc

실제로 GC 에 어떠한 object 를 할당하는 부분은 모두 ggc_alloc () 함수를 사용해서 이루어지게 되는데, 단독으로 사용될 경우도 있고, 아래와 같이 정해져 있는 매크로를 사용하여 할당될 수도 있다. 하지만 이 모두는 GC 의 한 object 로써 할당되기 때문에 실제로 mark-sweep 이 수행될 때는 이에 따른 검사가 이루어 지게 된다.

```
#define ggc_alloc_rtx(NSLOTS)                                    \
  ((struct rtx_def *) ggc_alloc (sizeof (struct rtx_def)        \
                                 + ((NSLOTS) - 1)               \
                                 * sizeof (rtunion)))

#define ggc_alloc_rtvec(NELT)                                    \
  ((struct rtvec_def *) ggc_alloc (sizeof (struct rtvec_def)    \
                                   + ((NELT) - 1)
                                   * sizeof (rtx)))

#define ggc_alloc_tree(LENGTH)                  \
  ((union tree_node *) ggc_alloc (LENGTH))
```

위에서 볼 수 있듯이, GC 를 통해 object 를 사용하는 것은 tree 혹은 rtx, rtvec 등등을 위한 node 를 할당할 경우 이를 통해서 사용하는 것을 알 수 있다.

어떠한 크기의 object 를 할당하는데 있어 그것의 order 를 구한 후 구한 order 를 위한 page entry 가 존재하지 않을 경우 alloc_page () 함수를 통해서 새로운 entry 를 할당받게 된다.

alloc_page () 함수에서 하는 수행에 대해서 약간 언급한다면 다음과 같이 나타낼 수 있을 것이다.

1. 우리가 사용할 수 있는 것이 free page 목록에 존재하는지를 검사한다.
2. entry 를 나머지 구성요소에 대해 설정해 준다.
3. one-past-the-end in-use bit 를 설정한다.
4. set_page_table_entry () 함수를 사용하여 entry 를 등록한다. 이를 통해서 2 단계 paging 에 값이 들어가게 된다. 알아야 할 내용으로는 paging 을 할 때 각 단계를 구분하는데 사용되는 주소는 entry 의 구성요소인 page 의 주소를 기반으로 나눠진다는 사실이다.

제 7 절 ggc_collect

GCC 에서 사용하는 mark-sweep Garbage Collection 방식의 경우 주기적으로 수행되면서 mark 단계와 sweep 단계를 거치게 된다. 위에서 모든 모든 root 들은 여기서 수행되며 비교 검색되게 된다. 이 함수가 주기적으로 수행되며 두 단계를 수행할 실질적인 함수가 되는 것이다. 이것이 호출되는 시점은 c-parse.in 에서 extdefs 구문까지 처리한 직후와 $prefix/gcc/toplev.c 파일의 rest_of_compilation 에서, 그리고 cse 단계에서 수행된다.

수행은 아래와 같은 순서로 실행된다.

1. 만약 총 할당들이 마지막 collection 이후 더 이상 확장되지 않았다면 collection 을 건너뛰는 것으로 빈번한 불필요한 작업을 피한다.

   수행되는 비교 연산은 아래와 같으며 이를 만족하지 못할 경우 반환된다.

```
#define GGC_MIN_EXPAND_FOR_GC (1.3)

#ifndef GGC_ALWAYS_COLLECT
  if (G.allocated < GGC_MIN_EXPAND_FOR_GC * G.allocated_last_gc)
    return;
#endif
```

2. 이 단계까지 도달했다는 것은 garbage collection 을 실행시켜야 할 단계이다. garbage collection 에서의 소유 시간을 재기 위해 timevar_push (TV_GC) 한다.

3. 할당된 총 바이트 (G.allocated = 0) 를 0 으로 한다. 이것은 sweep 단계에서 재계산될 것이다.

4. release_pages () 함수가 수행된다. 이는 G.free_pages 에 연결되어 있는 struct page_entry * 포인터들을 모두 해제하게 되는데, 인접한 page 들을 모은 후 한꺼번에 unmap 하는 방식을 사용하고 있다. 그림 3 에 방식에 대한 간략한 그림이 있는데, 이는 USING_MMAP 를 사용할 경우에 대한 그림이다.

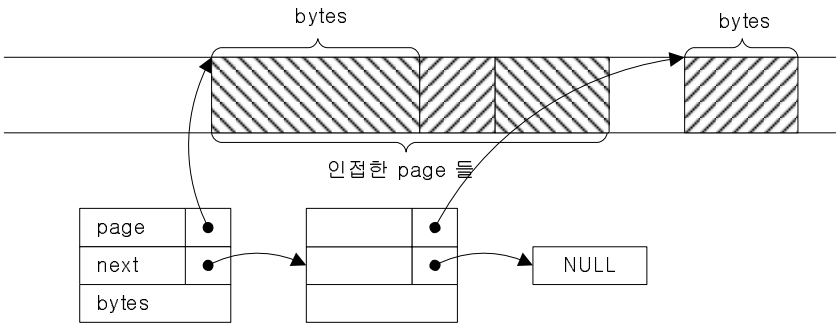

그림 3: 인접한 page 들

5. clear_marks () 함수가 수행된다. 이 함수에서는 모든 object 들을 unmark 하게 된다. 그리고 G.context_depth 보다 P->context_depth 가 작다면 P->in_use_p 를 P->save_in_use_p 에 백업해 놓게 된다.

6. ggc_mark_roots () 함수가 수행된다. 이 과정에서는 GC root 로 등록된 모든 root 를 반복적으로 살펴봐서 각 element 를 mark 한다.

   아래와 같은 단계로 수행된다.

   (a) ggc_pending_trees 편수에 대해 VARRAY_TREE_INIT 를 사용하여 초기화 시킨다.

   (b) 전역 변수 roots 에 등록된 모든 element 들에 대해서 각 element 가 가지는 size 와 nelt 에 따라, 등록될 당시에 같이 등록된 *cb 함수를 적당한 인자값과 함께 호출하게 된다. 여기서 각 *cb 의 행동에 대해서 좀 더 구체적으로 아래에서 설명하도록 하겠다..
   어쨌든, *cb 함수를 수행하면서 mark 해야 할 목록을
   ggc_pending_trees 에 넣게 된다.

   (c) ggc_mark_trees () 함수를 사용하여 모든 queue 되어 있는 tree 들과 그들의 자식들을 mark 한다. 여기서 부터는 아직 글을 작성하지 않은 'TREE' 에 대한 지식을 가지고 있어야 하므로 대략적으로 'TREE' 가 가지고 있을 자식들도 mark 된다는 사실만 생각해라. 'TREE' 에 대해서는 다음 강의에서 보도록 하겠다.

GC 의 사용에 관한 통계를 업데이트 하고, 각각의 자식을 mark 한 후에 언어에 따라 각각 'TREE' 를 달리 mark 할 수 있도록 하는 callback 함수인 lang_mark_tree () 를 호출하게 된다. 살펴 볼만한 다른 점이 있다면 몇몇 TREE node 들은 특별한 관리가 필요로 하기 때문에 개별적으로 수행하는 것이 있는 반면 대부분의 일반적인 node 들은 class 단위로 다룰 수 있다는 점이다.

(d) 이제 모든 ggc_pending_trees 내에 있는 element 들을 mark 하였으므로 ggc_pending_trees 를 위해 할당되었던 공간만 할당한다. 그것에 포함되어 있던 내부 element 들의 공간은 해제하지 않는다.

(e) 이제 object 가 이미 mark 되어 있다면 지워 질 수 있는 object 들을 가진 모든 hash table 들을 훑어본다. 여기서 살펴볼 수 있는 것은 많은 tree 들을 mark 하고 있을 수 있기 때문에 varray 를 재초기화할 필요가 있다.

(f) 이제 전역변수 d_htab_roots 에 등록되어 있는 요소들을 순회하며 htab_traverse () 함수를 호출하게 된다. 이 함수는 각 live entry 에 대해 CALLBACK 을 호출함으로써 전체 hash table 을 모두 훑어본다. ggc_mark_roots () 함수에서 호출하는 CALLBACK 은 ggc_htab_delete () 함수이다.

(g) 다시 ggc_mark_trees () 함수를 사용하여 ggc_pending_trees queue 되어 있는 모든 tree 들과 그들의 자식들을 mark 한다.

(h) VARRAY_FREE (ggc_pending_trees) 를 사용하여 메모리 공간을 다시 해제한다.

7. GGC_POISON 이 정의되어 있다면 poison_pages () 함수를 수행한다. 현재 page 방식을 사용하는 collector 의 경우 이것을 사용하지 않는다. poison 하는 방식은 사용되지 않는 공간에 0xa5 를 채워 넣는 방식으로 이루어 진다.

8. sweep_pages () 함수가 수행된다. 이 과정에서는 모든 empty page 들을 해제한다. 'mark' 비트가 'unused' 비트와 이중이기 때문에 부분적으로 empty page 들을 필요로 하지 않을 수 있다.

9. G.allocated_last_gc 값을 아래와 같이 만든다.

```
G.allocated_last_gc = G.allocated;
if (G.allocated_last_gc < GGC_MIN_LAST_ALLOCATED)
  G.allocated_last_gc = GGC_MIN_LAST_ALLOCATED;
```

10. 여기까지 하면 ggc_collect 함수가 모두 수행되었다. 시간의 끝을 나타내야 하므로 timevar_pop (TV_GC) 로 시간을 마무리 짓는다.

제 8 절 ggc_pending_trees 의 사용

위에서 언급했듯이 ggc_pending_trees 에 element 가 추가되는 상황을 추적해 나가보자.

```
VARRAY_TREE_INIT (ggc_pending_trees, 4096, "ggc_pending_trees");

for (x = roots; x != NULL; x = x->next)
  {
    char *elt = x->base;
    int s = x->size, n = x->nelt;
    void (*cb) PARAMS ((void *)) = x->cb;
    int i;

    for (i = 0; i < n; ++i, elt += s)
      (*cb)(elt);
  }
```

```
ggc_mark_trees ();
VARRAY_FREE (ggc_pending_trees);
```

위와 같은 방법으로 전역변수에 대한 virtual array 를 초기화하고 해제하는 것을 알 수 있다. 하지만 ggc_mark_trees () 함수를 보면 알겠지만 바로 ggc_pending_trees 에 할당된 element 를 사용한다는 것을 알 수 있다. 그럼 for 구문이 수행되는 동안 이 전역변수에 어떤 element 들이 추가되었다는 것은 쉽게 유추할 수 있을 것이다.

struct ggc_root * 구조체의 CALLBACK 함수를 살펴볼 필요가 있을 것이다. $prefix/gcc/ggc-common.c 파일을 보면 struct ggc_root * 구조체에 구성요소를 추가시키는 루틴이 있는데, 아래와 같이 선언되어 있다는 것을 볼 수 있다.

```
ggc_add_root (base, nelt, sizeof (rtx), ggc_mark_rtx_ptr);
ggc_add_root (base, nelt, sizeof (tree), ggc_mark_tree_ptr);
ggc_add_root (base, nelt, sizeof (varray_type),
              ggc_mark_rtx_varray_ptr);
ggc_add_root (base, nelt, sizeof (struct hash_table *),
             ggc_mark_tree_hash_table_ptr);
```

여기서 CALLBACK 에 해당하는 부분은 세번째 인자이다. 이를 추적해 나가보자. ggc_mark_rtx_ptr () 함수는 단지 (rtx 인) 인자를 ggc_mark_rtx 로 전달하는 구성요소 일 뿐이다. ggc_mark_tree 는 $prefix/gcc/ggc.h 에 선언되어 있는 매크로 인데, 다음과 같이 선언되어 있다는 것을 알 수 있다.

```
#define ggc_mark_tree(EXPR)                             \
  do {                                                  \
    tree t__ = (EXPR);                                  \
    if (ggc_test_and_set_mark (t__))                    \
      VARRAY_PUSH_TREE (ggc_pending_trees, t__);        \
  } while (0)
```

이 부분과 연관있는 것을 좀 더 살펴 보자.

```
#define ggc_test_and_set_mark(EXPR) \
  ((EXPR) != NULL && ! ggc_set_mark (EXPR))

#define VARRAY_PUSH_TREE(VA, X)     VARRAY_PUSH (VA, tree, X)
#define VARRAY_GROW(VA, N) ((VA) = varray_grow (VA, N))

#define VARRAY_PUSH(VA, T, X)                                   \
  do                                                            \
    {                                                           \
      if ((VA)->elements_used >= (VA)->num_elements)            \
        VARRAY_GROW ((VA), 2 * (VA)->num_elements);             \
      (VA)->data.T[(VA)->elements_used++] = (X);                \
    }                                                           \
  while (0)
```

역시나 VARRAY_PUSH_TREE 매크로에 의해서 ggc_pending_trees 에 element 가 추가된다는 사실을 알수 있다. ggc_pending_trees 는 'TREE' 관련 내용만 다루기 때문에 다른 rtx 나 rtvec 과 같은 것은 상관없으며, ggc_mark_tree 매크로나 ggc_mark_nonnull_tree 매크로가 element 추가에 관여하는 작업을 한다.

제 9 절 Garbage Collection 함수들의 요약

9.1 ggc-common.c 에서의 함수들

void **ggc_add_root** (void **base*, int *nelt*, int *size*, void (**cb*) PARAMS ((void *)))

BASE 를 새로운 garbage collection root 로 추가한다. 각 element 의 크기가 SIZE byte 길이를 가지는 NELT 길이의 배열이다. CB 는 각 배열의 요소로의 포인터를 가진채 호출되는 함수이다; 이것은 CB 가 그 요소(element)에 대한 gc-able(GC 가능한) memory 를 mark 하기 위한 적당한 routine 을 호출함을 내포한다.

void **ggc_add_rtx_root** (rtx **base*, int *nelt*)

GC root 로써 rtx 배열에 등록한다.

void **ggc_add_tree_root** (tree **base*, int *nelt*)

GC root 로써 tree 배열에 등록한다.

void **ggc_add_rtx_varray_root** (varray_type **base*, int *nelt*)

GC root 로써 rtx 들의 varray 를 등록한다.

void **ggc_add_tree_varray_root** (varray_type **base*, int *nelt*)

GC root 로써 tree 들의 varray 를 등록한다.

void **ggc_add_tree_hash_table_root** (struct hash_table ***base*, int *nelt*)

GC root 로써 tree 들의 hash table 을 등록한다.

void **ggc_del_root** (void **base*)

이전에 등록된 GC root 가 BASE 인 것을 제거한다.

void **ggc_add_deletable_htab** (PTR *x*, ggc_htab_marked_p *marked_p*, ggc_htab_mark *mark*)

GC 메모리에 의해 할당된 object 들을 포함하는 'htab 들의 목록'에 htab 인 X 를 추가한다. 이전에 모든 다른 root 들이 mark 되었다면 우리는 그것이 이미 mark 되었는지를 보기 위해 htab 내의 각 object 를 검사한다. 만약 mark 되어 있지 않다면 그것은 삭제된다.

(만약 지정될 경우) MARKED_P 는 만약 entry 가 "marked" 로써 고려되어 진다면 1 을 반환하는 함수이다. 만약 존재하지 않는다면, htab slot 에 의해 가르켜지는 data 구조체는 테스트될 것이다. 만약 몇몇 다른 object (그 object 에 의해 가르켜지는 어떤 것들과 같은) 가 mark 되었는지 아닌지를 (ggc_marked_p 를 사용하여) 검사해야 하는 경우와 만약 그렇다면 0 이 아닌값을 반환해야 할 경우 이 함수를 제공해야만 한다.

(만약 지정될 경우) MARK 는 (위의 함수를 통해) "marked" 로 결정되어진 slot 의 내용들을 건내주고 그 object 에 의해 가르켜지는 어떤 다른 object 들을 mark 하는 함수이다. 예를 들면, 우리가 MEM RTL 에 의해 가르켜지지만 DECL 로의 포인터를 가지고 있는 memory attribute block 들의 hash table 을 가지고 있다고 합시다. 이러한 경우 MARKED_P 는 우리가 attribute block 이 MEM 에 의해 가르켜 지는지를 알기 원하기 때문에 지정되지 않을 것이지만 block 이 이미 mark 되었다면 우리는 DECL 을 mark 할 필요성 때문에 MARK 는 지정되어야만 한다.

static int **ggc_htab_delete** (void ***slot*, void **info*)

htab 의 slot 이 mark 되어 있지 않다면 그것을 삭제함으로써 처리한다.

void **ggc_mark_roots** ()

모든 등록된 root 들을 반복적으로 살펴봐서 각 element 를 mark 한다.

void **ggc_mark_rtx_children** (rtx *r*)

R 는 이전에 한번도 mark 된적이 없지만 ggc_set_mark 를 통해 이제 mark 되었다. 이제 recursion 을 수행하고 children 을 처리한다.

static void **ggc_mark_rtx_children_1** (rtx *r*)

재귀호출을 위한 **ggc_mark_rtx_children** 의 하위 함수.

void **ggc_mark_rtvec_children** (rtvec *v*)

V 는 이전에 한번도 mark 된적이 없지만 ggc_set_mark 를 통해 이제 mark 되었다. 이제 recursion 을 수행하고 children 을 처리한다.

static void **ggc_mark_trees** ()

재귀적 방법으로 GCC_PENDING_TREES 의 children 의 모든것을 mark 설정한다.

void **ggc_mark_rtx_varray** (varray_type *v*)

rtx 들을 포함하는 varray V 의 모든 element 들을 mark 한다.

void **ggc_mark_tree_varray** (varray_type *v*)

tree 들을 포함하는 varray V 의 모든 element 들을 mark 한다.

static bool **ggc_mark_tree_hash_table_entry** (struct hash_entry **he*, hash_table_key *k* ATTRIBUTE_UNUSED)

hash table-entry HE 를 mark 한다. 그것의 key field 는 실제로 tree 이다.

void **ggc_mark_tree_hash_table** (struct hash_table **ht*)

tree 들을 포함하는 hash-table H 의 모든 element 들을 mark 한다.

static void **ggc_mark_rtx_ptr** (void **elt*)

ggc_add_root 로 보내기 위한 type-correct 함수. 단지 (rtx 인) *ELT 를 ggc_mark_rtx 로 전달한다.

static void **ggc_mark_tree_ptr** (void **elt*)

ggc_add_root 로 보내기 위한 type-correct 함수. 단지 (tree 인) *ELT 를 ggc_mark_rtx 로 전달한다.

static void **ggc_mark_rtx_varray_ptr** (void **elt*)

ggc_add_root 로 보내기 위한 type-correct 함수. 단지 (실제로 varray_type * 인) ELT 를 ggc_mark_rtx_varray 로 전달한다.

static void **ggc_mark_tree_varray_ptr** (void **elt*)

ggc_add_root 로 보내기 위한 type-correct 함수. 단지 (실제로 varray_type * 인) ELT 를 ggc_mark_tree_varray 로 전달한다.

static void **ggc_mark_tree_hash_table_ptr** (void **elt*)

ggc_add_root 로 보내기 위한 type-correct 함수. 단지 (실제로 struct hash_table ** 인) ELT 를 ggc_mark_tree_hash_table 로 전달한다.

void * **ggc_alloc_cleared** (size_t *size*)

메모리 블럭을 할당한 후 그것을 깨끗히 한다.

void **ggc_print_common_statistics** (FILE **stream*, ggc_statistics **stats*)

사용중인 collector 와 독립적으로 통계들을 출력한다.

9.2 ggc-page.c 에서의 함수들

static inline int **ggc_allocated_p** (const void **p*)

만약 P 가 GC 가능한 메모리에 할당되어 있다면 0 이 아닌 값을 반환.

static inline page_entry * **lookup_page_table_entry** (const void **p*)

Page table 을 돌아다니며 page 에 대한 entry 를 찾는다. 만약 object 가 GC 를 통해 할당되지 않았다면 (아마도) 없앤다.

static void **set_page_table_entry** (void **p*, page_entry **entry*)

Page 를 위한 page table entry 를 설정한다.

void **debug_print_page_list** (int *order*)

디버깅을 위해, object size ORDER 를 위한 page-entry 를 출력한다.

static inline char * **alloc_anon** (char **pref* ATTRIBUTE_UNUSED, size_t *size*)

SIZE 바이트 크기의 익명 (anonymous) 메모리를 할당하며 (만약 PREF 가 NULL 이 아니라면) PREF 근처의 것으로 한다. 여기 있는 ifdef 구조체가 HAVE_* 중 하나가 정확하게 정의되지 않을 경우 compile 오류를 발생할 수 있다.

static inline size_t **page_group_index** (char **allocation*, char **page*)

Page group 내에서 이 page 에 대한 index 를 계산한다.

static inline void **set_page_group_in_use** (page_group **group*, char **page*)

Page group 내 이 page 에 대한 in_use 비트를 설정하고 지운다.

static inline void **clear_page_group_in_use** (page_group **group*, char **page*)

static inline struct page_entry * **alloc_page** (unsigned *order*)

크기 2^ORDER 의 object 들을 할당하기 위해 새로운 page 를 할당하고 그것에 대한 entry 를 반환한다. entry 는 적당한 page_table list 에 추가되지 않는다.

static inline void **free_page** (page_entry **entry*)

더 이상 필요하지 않는 page 에 대해, free page list 에 그것을 넣는다.

static void **release_pages** ()

시스템으로 free page cache 를 release 한다.

void * **ggc_alloc** (size_t *size*)

SIZE 바이트의 메모리 chunk 를 할당한다. 만약 ZERO 가 0 이 아니면 메모리는 0 으로 초기화되고 그렇지 않다면 그것의 내용은 정의되지 않는다.

int **ggc_set_mark** (const void **p*)

만약 P 가 mark 되지 않는다면 그것을 mark 하고 false 를 반환한다. 그렇지 않다면 true 를 반환한다. P 는 반드시 GC allocator 에 의해 할당되어야 했으며, 이것은 static object 들 혹은 stack variable 들, malloc 에서 할당된 메모리를 절대 가르켜서는 안된다.

int **ggc_marked_p** (const void **p*)

만약 P 가 이미 mark 되어 있다면 1 를 반환, 그렇지 않을 경우 0 을 반환. P 는 GC allocator 로 부터 할당되어 있어야만 한다; 이것은 static object 들 혹은 stack variable 들, malloc 에 의해 할당된 메모리를 가르키면 안된다.

size_t **ggc_get_size** (const void **p*)

gc-able object P 의 크기를 반환한다.

void **init_ggc** ()

ggc-mmap allocator 를 초기화한다.

void **ggc_push_context** ()

static void **ggc_recalculate_in_use_p** (page_entry **p*)

P 안의 SAVE_IN_USE_P 와 IN_USE_P 배열들을 통합하여 그래서 IN_USE_P 가 실제를 반영한다. 또한 NUM_FREE_OBJECTS 도 재계산한다.

void **ggc_pop_context** ()

'GC context' 를 감소시킨다. 이전 ggc_push_context 이후에 할당된 모든 object 들은 외부 context 로 이전된다.

static inline void **clear_marks** ()

모든 object 들을 unmark 한다.

static inline void **sweep_pages** ()

모든 empty page 들을 해제한다. 'mark' 비트가 'unused' 비트와 이중이기 때문에 부분적으로 emptry page 들은 필요하지 않을 수 있다.

static inline void **poison_pages** ()

모든 free object 들을 clobber 한다.

void **ggc_collect** ()

최상위층 mark-and-sweep 루틴.

void **ggc_print_statistics** ()

할당 통계들을 출력한다.

제 10 절 13 차 강의를 마치며

많은 시간이 흘렀습니다. 벌써 13 차를 쓰는 군요. 하지만 갈 길이 너무나 멉니다. GCC 를 모두 분석하는데, 아직도 많은 부분이 남았습니다. 정말 100 차 까지 가야 할 듯합니다. 그 때까지 꾸준히 하는 제 모습을 보고 싶습니다. 조금씩, 꾸준히 전진하겠습니다.

그럼. 행복한 하루 보내시길..