18.10.14
OS - gc

gc: garbage collection

단골 질문이라고 할 수 있음. 많은 it기업의 tech blog에서 심심치 않게 관련된 내용을 설명하고 있다. 어느 분야에 지원하느냐에 따라 다르겠지만, 대용량 처리나 백앤드 쪽에서는 꼭 알아두어야 함. java 개발자(우리 중에 이걸 희망하는 사람이 있을까 싶긴 하지만 여튼)로 직군을 넣은 경우에는 java gc에 대해서 꼭 알아두자.

garbage:
주소를 잃어버려서 참조가 불가능한 memory나 object정도. 다른 말로 dangling object라고 한다.
gc:
동적 할당된 메모리 영역 가운데 더 이상 사용하지 않는 영역을 자동으로 해제하는 기법이다. c, c++에서는 coder가 수동적으로 free를 해주어야하는 반면, gava, c#같은 언어들은 처음부터 gc를 염두에 두고 설계되어 gc가 포함되어 있다.
실제로 심각한 취약점들이 double free(free한 영역을 다시 free 하는 것)같은 개발자의 코딩 실수에서 야기되기도 하고, 날이 갈수록 사람보다는 언어(컴퓨터, 컴파일러)가 스마트해져야 한다는 흐름이 있다보니 그에 따른 어쩌구정도라 생각함.

장점:
1. double free, invalid free를 막을 수 있다.
2. memory leak을 막을 수 있다.

단점:
1. 어떤 메모리를 해제해야 할지 결정하는 cost가 overhead가 됨.
2. gc가 일어나는 타이밍이나 점유 시간을 예측하기가 어려움 (-> real time에 좋지 않다는 말이 있네)
3. 할당된 메모리가 언제 해제되는 지 알 수 없다.

가장 기본적인 알고리즘은 tracing과 reference counting.

Tracing:
references chain(특정한 root object부터 시작하는)에 의해서 'reachable'한지 아닌지로 garbage 결정을 한다. tracing 기법을 활용한 gc algorithm은 겁나 많고 많다.

1. mark and sweep: 모든 변수 공간을 1bit checking하고 그 변수가 가르키는 공간도 checking한다. 이렇게 모든 공간을 checking하고 남은 공간은 garbage로 취급하여 해제한다. 이 방법은 stop-the-world 방식을 사용해야한다는 단점이 있다.
2. white, gray, black
3. 객체이동 기법. 2,3번은 skip.
4. 세대 단위 gc: java gc할 때 상세히 다룰 것이다.

Reference counting:
객체마다 count라는 변수 하나를 둔다고 생각하면 된다. 어떤 프로세스에서 그 객체를 생성하거나 사용할때 +1을 해주고 더 이상 사용하지 않을 때 -1을 해주면 count가 0이 되었을 때, garbage로 판단되어 destroyed됨. 이러한 방법은 file system에서도 사용됨.

gc 알고리즘이 고려해야 할 사항.

cycles:
reference counting의 경우에,
만약 2개의 객체가 오로지 서로만을 참조하고 있다고 가정해보자.(좀 로맨틱한데?) 이런 경우 cycle이 생겼다고 말할 수 있고 이 cycle을 끊어내고 2개 다 쓰레기통으로 보내버려야 할 것이다.
이런 경우(CPython) cycle-detecting algorithm을 사용한다. 다른 방법으로 weak reference가 있다. strong reference가 garbage로 절대 만들지 않는 개념이라고 하면 weak reference는 reference count를 증가시키지 않아서 ref되어도 garbage로 만들 수 있는 개념. 그리고 object가 garbage가 되면 'dangling'되는 것이 아니라 null처럼 예측 가능한 값으로 바뀐다고 함. 근데 이거는 좀 더 내가 공부를 해볼게.

space overhead:
reference count는 memory, object의 metadata가 되니깐, space overhead가 발생하는 것은 당연하다. memory, object의 header같은 곳에 저장할 수도 있고, 따로 table로 관리를 할 수도 있지만 space를 쓰긴 하지. 대충 1word의 크기가 필요하다고 한다.
space overhead를 줄이기 위해서 tagged pointer(pointer긴 한데 다른 정보를 담고 있다고 생각하자. access 가능한 memory 영역이 한정되어 있다는 것을 이용하면 1word 전체가 주소를 담고 있을 필요가 없자나)라는 것을 사용. 실제로 ob-c에서 씀.

speed overhead:
숫자 늘리고 낮추는데 드는 cost가 있다. 그래서 c++ 에서는 smart pointer라는 것을 사용해서 생성자, 소멸자, assignment op할 때 reference를 바꾼다. 이 부분은 뒤에 추가로 tcmalloc을 공부하면서 보충하기로 하자.

Requires atomicity:
multi threading 환경에서 각 thread 마다 local count를 주고 global count를 따로 두는 것으로 해결할 수 있다. local count가 0이 되는 경우에만 global을 확인한다.

java gc:
jvm이 있다. java는 X같지만 이참에 java에 대해서… java는 machine independent하다. android에서 자바를 놓지 못하는데 이런 이유도 있지싶다. c언어에서는 컴파일 하면 어셈블리로 컴파일 되는 반면(그래서 c는 machine에 맞는 컴파일러를 써야한다.) java에서는 컴파일을 하면 java bite code로 바뀐다. 그러면 이 java bite code를 실행하기 위해서는 jvm이라는 게 필요하다. java virtual machine으로 os에 바로 실행되는게 아니라 jvm이 os위에 올라가고 java bite code가 이 jvm위에 올라가서 실행되는 것이다. 그래서 java bite code만 있으면 어떤 머신이든지 jvm만 설치되어 있으면 실행할 수 있는 것이다.
근데 jvm도 os로 부터 메모리를 받아야 한다. 그리고 실행하다가 memory가 부족하면 요청도 보낸다. 바로 이 요청을 보내기전에 java gc가 실행된다고 생각하면 된다.

그리고 java로 코딩하다가 system.gc()를 실행하는 멍청한 짓은 제발 하지 말자. 맘대로 gc를 부르는 코드는 시스템 성능에 큰 영향을 줄 것이 분명하다. 우리의 아름다운 gc는 아름답게 알아서 행동하게 두자.

2가지의 가정에 입각하여 만들어졌다. (weak generational hypothesis)
1. 대부분의 객체는 금방 unreachable 상태가 된다.

2. 오래된 객체가 젊은 객체를 참조하는 일은 아주 적다.
이런 가설로 나온게 '세대 분리해서 gc를 돌리자!'이다.

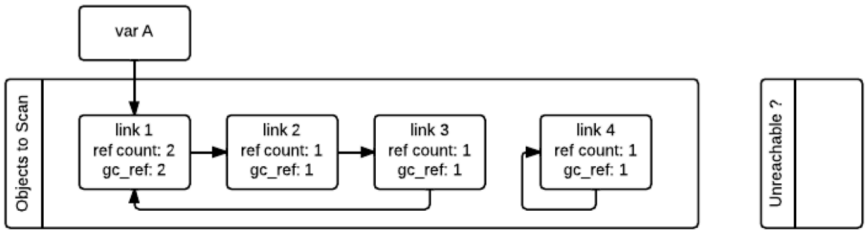
young 영역, old 영역으로 나뉜다.
young: 새롭게 생성된 객체, 많은 객체가 금방 unreachable되기 때문에 여기서 생성되었다가 사라진다.
old: young에서 살아남은 객체가 여기로 복사됨. young보다 크게 할당함. young보다 gc가 적게 발생한다.
old 영역에 있는 객체가 young영역의 객체를 참조하는 경우를 처리하기 위한 'card table'이 존재한다. old 영역의 512바이트 씩 잘라서 표시됨. 이때 young 영역의 gc를 실행할 때 old 전체를 뒤지지 않고 이 card table만 뒤져서 garbage인지 판단한다.

그리고 추가적으로 compaction을 해줌.

cycle-detecting-algorithm (CPython)

cycle에 대한 문제는 ref_count 기법을 사용하는 gc의 경우 고려해야할 대상이다.
사실 directed graph에서 cycle을 탐지하는 여러가지 알고리즘이 존재하는 듯 하다. O(E+V)에 찾아준다고 한다.
근데 실제 활용 분야에서 어떻게 쓰이는 지 한번 보고싶었다.
https://hg.python.org/cpython/file/eafe4007c999/Modules/gcmodule.c#l335
위의 링크는 CPython gc 코드이다. 코드 참 잘 짰더라…;; 뒤에서 중요한 부분만 좀 스크랩해둠.

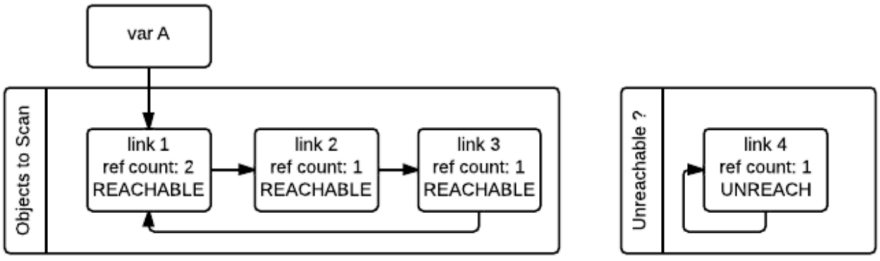
1.



일단 초기 상태이다. ref_count는 진짜 count를 의미하고 gc_ref는 cycle-detecting을 위한 field이다. gc_ref := ref_count로 초기화한다. 그리고 큰 틀은 의심되는 것을 모두 unreachable로 취급했다가 reachable함이 판단되면 reachable로 확정짓는 것이다.

2. 모든 gc_ref를 -1한다. 다시 말하면 outside에서 들어오는 edge만 고려하겠다는 것이다.
3. 그 다음 다시 gc가 scan을 하면서 gc_ref값이 0인 objects를 모두 UNREACH로 취급한다.
4. scan을 하다가 gc_ref > 0인 object를 만나면 REACHABLE로 확정하고 traversing되는 objects를 REACHABLE로 확정한다.

5. 모든 objects를 scan하고 나면, 아래와 같이 unreachable을 판단할 수 있게 되고 gc를 작동시킬 수 있다.



details in code:

The `tp_traverse` handler must have the following type:

int **(*traverseproc)**(PyObject *self*, visitproc *visit*, void *arg*)

Traversal function for a container object. Implementations must call the *visit* function for each object directly contained by *self*, with the parameters to *visit* being the contained object and the *arg* value passed to the handler. The *visit* function must not be called with a *NULL* object argument. If *visit* returns a non-zero value that value should be returned immediately.

To simplify writing `tp_traverse` handlers, a `Py_VISIT()` macro is provided. In order to use this macro, the `tp_traverse` implementation must name its arguments exactly *visit* and *arg*:

gc가 double linked list로 관리됨.

```
/** GC information is stored BEFORE the object structure. */
80 union PyGC_Head {
82     struct _gc {
84         PyGC_Head *gc_next;
86         PyGC_Head *gc_prev;
88         Py_ssize_t gc_refs;
89     }
91     _gc gc;
93     real dummy;
94 }
```

```
static void
move_unreachable(PyGC_Head *young, PyGC_Head *unreachable)

while (gc != young) {
        PyGC_Head *next;
        if (_PyGCHead_REFS(gc)) {
            PyObject *op = FROM_GC(gc);
            traverseproc traverse = Py_TYPE(op)->tp_traverse;
            assert(_PyGCHead_REFS(gc) > 0);
            _PyGCHead_SET_REFS(gc, GC_REACHABLE);
            (void) traverse(op,
                            (visitproc)visit_reachable,
                            (void *)young);
            next = gc->gc.gc_next;
            if (PyTuple_CheckExact(op)) {
                _PyTuple_MaybeUntrack(op);
            }
        }
        else {
            next = gc->gc.gc_next;
            gc_list_move(gc, unreachable);
            _PyGCHead_SET_REFS(gc, GC_TENTATIVELY_UNREACHABLE);
        }
        gc = next;
    }
```

```
static int
visit_reachable(PyObject *op, PyGC_Head *reachable)
{
    if (PyObject_IS_GC(op)) {
        PyGC_Head *gc = AS_GC(op);
        const Py_ssize_t gc_refs = _PyGCHead_REFS(gc);
        if (gc_refs == 0) {
            _PyGCHead_SET_REFS(gc, 1);
        }
        else if (gc_refs == GC_TENTATIVELY_UNREACHABLE) {
            gc_list_move(gc, reachable);
            _PyGCHead_SET_REFS(gc, 1);
        }
```

References:

https://en.wikipedia.org/wiki/Garbage_collection_(computer_science)
https://ko.wikipedia.org/wiki/쓰레기_수집_(컴퓨터_과학)
https://docs.microsoft.com/ko-kr/dotnet/standard/garbage-collection/weak-references
https://d2.naver.com/helloworld/1329
https://pythoninternal.wordpress.com/2014/08/04/the-garbage-collector/
https://hg.python.org/cpython/file/eafe4007c999/Modules/gcmodule.c#l335
https://docs.python.org/3/c-api/gcsupport.html#c.traverseproc
http://pyd.dpldocs.info/source/deimos.python.objimpl.d.html#L60

Contact:

jinious1111@naver.com
이진명