CPython's Private (Memory) Heap

Jonathon Ogden - jeog.dev@gmail.com

March 30, 2016

One of the many advantages of using a high-level language like Python is not having to worry about managing memory. At times, optimizing code or building extension modules for instance, a basic understanding of python's memory management model can prove very helpful.

A core component of any such model is the allocation of raw memory, handled in CPython by a private heap. The following will attempt to explain what CPython's private heap is, what it does, and how – its interface and its implementation. Fortunately, the Python developers have included fairly comprehensive (and surprisingly comprehensible) comments in the source code.[1][2] A basic understanding of python, C, and data structures is recommended.

Layers of the Python memory architecture

```
       Object-specific allocators
       _____    ______    ______       ________
      [ int ] [ dict ] [ list ] ... [ string ]       Python core         |
   +3 | <----- Object-specific memory -----> | <-- Non-object memory --> |
       _______________________________       |                           |
      [   Python's object allocator   ]      |                           |
   +2 | ####### Object memory ####### | <------ Internal buffers ------> |
       ______________________________________________________________    |
      [          Python's raw memory allocator (PyMem_ API)          ]   |
   +1 | <----- Python memory (under PyMem manager's control) ------> |   |
       __________________________________________________________________
      [    Underlying general-purpose allocator (ex: C library malloc)   ]
    0 | <------ Virtual memory allocated for the python process -------> |

      =========================================================================
       _______________________________________________________________________
      [                OS-specific Virtual Memory Manager (VMM)               ]
   -1 | <--- Kernel dynamic storage allocation & management (page-based) ---> |
       __________________________________   __________________________________
      [                                  ] [                                  ]
   -2 | <-- Physical memory: ROM/RAM --> | | <-- Secondary storage (swap) --> |
```

(Objects/obmalloc.c:441)

That seems a bit overwhelming at first glance, so let's simplify things:

```
     ----------------------------------------------------------------------
+2/3 |    Python's Object Allocator(s)       |        Python core         |
     |                   -------------------------------------------------|
 +1  |                   |       Python's Raw Memory Allocator(s)         |
     |--------------------------------------------------------------------|
     |                   |        C-lib calls (e.g. malloc, free)         |
  0  |                   -------------------------------------------------|
     |              OS calls (e.g. mmap, VirtualAlloc)                    |
     |====================================================================|
     |             OS stuff we don't have to worry about                  |
     ----------------------------------------------------------------------
```

We'll break the heap into two parts: Raw Memory Allocator(s) and Object Allocator(s). It's important to stress that – although in many cases it doesn't result in problems[3] - none of the API calls we discuss should be mixed with C-lib allocation functions (malloc, free etc.), shared memory calls, operator new etc. In fact, the Raw and Object Allocator calls should not even be mixed.

1 github.com/python/cpython -- branch 3.5 -- commit a62eb19db8653a0a9315268083dc5059956ce987 (Mar 25 2016)
2 We are not using a debug build of python, as doing so defines a different set of low-level memory management functions.
3 For instance: on linux an extension module using PyMem_ uses the same heap as malloc; objects > 512 bytes all end up at a Raw Allocator; if _PyObject_Free doesn't recognize an address it will be sent to PyMem_RawFree.

CPython's Private (Memory) Heap

Jonathon Ogden - jeog.dev@gmail.com

March 30, 2016

RAW MEMORY ALLOCATOR(S)[4]

		Args	Description
PyMem_RawMalloc	PyMem_Malloc	(size)	allocate size bytes
PyMem_RawCalloc	PyMem_Calloc	(nelem,elsize)	allocate array of nelem*elsize bytes
PyMem_RawRealloc	PyMem_Realloc	(*ptr,new_size)	change size of *ptr to new_size bytes
PyMem_RawRealloc	PyMem_Free	(*ptr)	deallocate *ptr

Each low-level API function calls their respective Allocator struct's 'method'[5].

```
typedef struct {
    void  *ctx;
    void* (*malloc) (void *ctx, size_t size);
    void* (*calloc) (void *ctx, size_t nelem, size_t elsize);
    void* (*realloc) (void *ctx, void *ptr, size_t new_size);
    void  (*free) (void *ctx, void *ptr);
} PyMemAllocatorEx;
```

As currently implemented, PYMEM_FUNCS is #defined as PYRAW_FUNCS so both default allocators - _PyMem and _PyMemRaw - are defined with the same set of calls. These calls essentially wrap the familiar C-lib functions (malloc, free etc.).[6]

Python	`>>> print('something')`

Call Stack	`#0  __GI___libc_malloc (bytes=21) at malloc.c:2881`
	`#1  0x000000000041bab3 in _PyMem_RawMalloc (ctx=0x0, size=21)`
	`    at Objects/obmalloc.c:139`
	`#2  0x000000000041bf44 in PyMem_Malloc (size=21) at Objects/obmalloc.c:404`
	`#3  0x00000000005aef49 in translate_newlines (`
	`    s=0x28b6000 "print('something')\n", exec_input=0, tok=0x28b5210)`
	`    at Parser/tokenizer.c:708`
	`...`
	`#15 0x000000000041ba2f in main (argc=1, argv=0x7ffd85ddf9e8)`
	`    at ./Programs/python.c:65`

obmalloc.c	`Void* PyMem_Malloc(size_t size)`
	`{`
	`    if (size > (size_t)PY_SSIZE_T_MAX)`
	`        return NULL;`
	`    return _PyMem.malloc(_PyMem.ctx, size);`
	`}`

'Type-oriented' convenience macros, that check for overflows, are provided: PyMem_New(type,n), PyMem_Resize(p, type, n). They allocate - and reallocate, respectively - 'n' * (size of the underlying object) bytes, returning a pointer to an uninitialized object of the passed-in type, NULL on failure.

There are also MACRO versions that may be more efficient but DO NOT guarantee binary compatibility between versions: PyMem_MALLOC, PyMem_REALLOC, PyMem_FREE, PyMem_NEW, PyMem_RESIZE. Extension modules that include them should be recompiled with each new version of Python.

4 Include/pymem.h, Objects/obmalloc.c

5 Custom allocators can be inserted with PyMem_SetAllocator for a particular domain: PYMEM_DOMAIN_RAW, PYMEM_DOMAIN_MEM, or PYMEM_DOMAIN_OBJ - enum PyMemAllocatorEx (Include/pymem.h). Building custom Allocators is beyond the scope of this paper, see the comments in pymem.h for more information.

6 There's a slight change in the behavior: *“PyMem_RawMalloc(0) means malloc(1). Some systems would return NULL for malloc(0), which would be treated as an error. Some platforms would return a pointer with no memory behind it, which would break pymalloc. To solve these problems, allocate an extra byte.”* - (Objects/obmalloc.c:56)

OBJECT ALLOCATOR(S)[7]

The Object Allocators, particularly PYMALLOC, are much more involved. We'll separate them into three distinct groups:

```
IF OF A CERTAIN BUILT-IN TYPE   -->   Specialized Object Allocators (1)
ELSE IF > 512 bytes             -->   PyMem_RawMalloc (2)
ELSE                            -->   Generic Object Allocator - PYMALLOC (3)
```

We, and the interpreter more generally, tend to a create a lot of 'small' objects. Some of these objects, like integers and lists, benefit from unique space-time trade-offs and specialized allocators (#1 above). An example of how lists are allocated:

Python:	`>>> my_list = [1,2,3,4,5]`

Call Stack:	`#0  PyList_New (size=5) at Objects/listobject.c:169`
	`#1  0x00000000004fe54c in PyEval_EvalFrameEx (f=0x7fb6b55383c8, throwflag=0)`
	`    at Python/ceval.c:2496`
	`  ...`
	`#11 0x000000000041ba2f in main (argc=1, argv=0x7ffc1eef7958)`
	`    at ./Programs/python.c:65`

listobject.c	`PyObject* PyList_New(Py_ssize_t size)`
	`{`
	`    ...`
	`    nbytes = size * sizeof(PyObject *);`
	`    if (numfree) {`
	`        numfree--;`
	`        op = free_list[numfree];` **`<<< (A)`**
	`        _Py_NewReference((PyObject *)op);`
	`    } else {`
	`        op = PyObject_GC_New(PyListObject, &PyList_Type);` **`<<< (B)`**
	`        if (op == NULL)`
	`            return NULL;`
	`    }`
	`    if (size <= 0)`
	`        op->ob_item = NULL;`
	`    else {`
	`        op->ob_item = (PyObject **) PyMem_MALLOC(nbytes);` **`<<< (C)`**
	`        if (op->ob_item == NULL) {`
	`            Py_DECREF(op);`
	`            return PyErr_NoMemory();`
	`        }`
	`        memset(op->ob_item, 0, nbytes);`
	`    }`
	`    ...`
	`    return (PyObject *) op;`
	`}`

Looking at PyList_New we notice a simple 'stack' named free_list that stores up to PyList_MAXFREELIST[8] recycled list objects. If possible it 'pops' an old list **(A)**; if not it uses PYMALLOC via PyObject_GC_New[9] to get a new one **(B)**. It finishes by using the Raw Allocator to allocate space for the individual list items **(C)**.

7 Inculde/objimpl.h, Objects/obmalloc.c, Objects/object.c
8 Objects/listobject.c:96 – currently defined as 80
9 PyObject_GC_New (Modules/gcmodule.c) prepends a garbage collection header, PyGC_Head (Include/objimpl.h), to the object and eventually uses PyObject_Malloc (Object/obmalloc.c) to allocate the list object. A detailed explanation of garbage collection is beyond the scope of this paper, and its author.

CPython's Private (Memory) Heap

Jonathon Ogden - jeog.dev@gmail.com

March 30, 2016

PYMALLOC

What if we need to individually allocate space for a large number of small, custom class instances; details of which can't be known until the class is defined? In cases like this, and many others, CPython uses a 'generic' Object Allocator – PYMALLOC.

INTERFACE

PyObject_Malloc/Calloc/Realloc/Free are all similar to the Raw Allocator versions.[10]

PyObject_New(type,typeobj) allocates typeobj->tp_basicsize bytes. 'type' is the actual object's type (e.g. PyFloatObject), 'typeobj' is the type object (e.g. PyFloat_Type). The object is initialized automatically via PyObject_INIT: reference count and type pointer of the object are set, the rest is undefined.

PyObject_NewVar(type,typeobj,n) does the same for variable-size objects, allocating extra space of (n * typeobj->tp_itemsize) bytes, rounded up to the closest multiple of sizeof(void*).[11] PyObject_INIT_VAR calls PyObject_INIT and sets the ob_size field.

IMPLEMENTATION

Now things get a little tricky. Rather than simply wrapping a C-lib call or caching removed objects for reuse, we have to deal with a more sophisticated allocation strategy: *“...a variant of what is known as "simple segregated storage based on array of free lists".”*[12]

There are three object types at the heart of PYMALLOC's implementation: pools, blocks, and arenas. Pools that are in-use are stored in the usedpools array of circular, doubly-linked lists.

-POOLS-

Pools are 4KB in size and contain a header followed by as many same-size blocks as can fit.

```
struct pool_header {
    union { block *_padding;
            uint count; } ref;
    block *freeblock;
    struct pool_header *nextpool;
    struct pool_header *prevpool;
    uint arenaindex;
    uint szidx;
    uint nextoffset;
    uint maxnextoffset;
};
```

'freeblock' is guaranteed to point at a valid block if the pool is not full. It's cleverly implemented as a singly-linked list that contains the address of the next free block, at the beginning of the raw memory of its own (yet to be allocated) free block.

10 Macro versions are also exposed – PyObject_MALLOC, PyObject_REALLOC, PyObject_FREE, PyObject_NEW, PyObject_NEW_VAR – with the same caveats about binary compatibility.

11 *“_PyObject_VAR_SIZE returns the number of bytes (as size_t) allocated for a vrbl-size object with nitems items, exclusive of gc overhead (if any). The value is rounded up to the closest multiple of sizeof(void *), in order to ensure that pointer fields at the end of the object are correctly aligned for the platform...”* - (Include/objimp.h:146)

12 I have little doubt that those in the know refer to it as SSSBOAOFL.

CPython's Private (Memory) Heap

Jonathon Ogden - jeog.dev@gmail.com

March 30, 2016

-BLOCKS-

Blocks represent areas within a pool, addresses of which are returned to the caller as allocated memory. They can be between 8 and 512 bytes, in 8-byte multiples. A particular pool only contains blocks of a certain size; different pools hold different size blocks (e.g. pool #32 may contain blocks of 16 bytes, or blocks of 128 bytes).

-USEDPOOLS-

This is the first place PYMALLOC looks for allocatable pools/blocks. Unfortunately, the code defining it is a hot mess.[13] I fear trying to explain its implementation would evince dormant sadomasochistic tendencies; so here's a simplified, although not technically correct, way to think about it:

```
index
                          <----------->         <----------->
 [0]  --> [8-byte-block pool ]   [8-byte-block pool ]   [8-byte-block pool ]
                          <----------->         <----------->
 [1]
                          <-------
 [2]  --> [16-byte-block pool]   |    (<- not a real pool, points to itself, i.e. NULL)
                          <-------
 [3]
 ...
                          <----------->
[126] --> [512-byte-block pool]   [512-byte-block pool]
                          <----------->
[127]
```

The fast(est) way X bytes (0, 512] are allocated:

```
  1) szindx = (X - 1) // ALIGNMENT
  2) poolp = usedpools[2 * szindx]
  3) return poolp->freeblock
```

As you can imagine this is very fast: a read from usedpools, a read from the pool's freeblock, and a write to the pool's freeblock (not shown). If a pool is not on its respective list we have to find a usable arena to get one from...

-ARENAS-[14]

Arenas represent large (virtual) chunks of memory allocated by the Operating System.[15]

```
struct arena_object {
    uptr address;
    block* pool_address;
    uint nfreepools;
    uint ntotalpools;
    struct pool_header* freepools;
    struct arena_object* nextarena;
    struct arena_object* prevarena;
};
```

13 *"... The excruciating initialization code below fools C so that usedpool[i+i] "acts like" a genuine poolp, but only so long as you only reference its nextpool and prevpool members ... So what usedpools[i+i] really contains is a fudged-up pointer p such that *if* C believes it's a poolp pointer, then p->nextpool and p->prevpool are both p (meaning that the headed circular list is empty). It's unclear why the usedpools setup is so convoluted... "* - Objects/obmalloc.c:809

14 Don't be confused, there is an unrelated arena in Python/pyarena.c

15 How exactly this happens is platform dependent.

The address field points to 256KB that contains up to 63 or 64 Pools[16]. The pool_address field points to the first (POOL_SIZE aligned) pool. This alignment allows the POOL_ADDR macro to return the pool from which a particular block belongs.[17]

```
           Pool 1           Pool 2               < ... >             Pool 63 or 64
------------------------------------------------------------------------------------------
|  A  |    HEADER     |     HEADER    |                            |               |  A  |
|  L  |---------------|---------------|                            |               |  L  |
|  I  |   block 1     |               |                            |               |  I  |
|  G  |---------------|               |                            |               |  G  |
|  N  |   block 2     |    block 1    |                            |               |  N  |
|     |---------------|               |                            |               |     |
|  P  |               |               |          < ... >           |               |  P  |
|  A  |     ...       |---------------|                            |               |  A  |
|  D  |               |               |                            |               |  D  |
|  D  |---------------|               |                            |               |  D  |
|  I  |   block X     |     ...       |                            |               |  I  |
|  N  |---------------|               |                            |               |  N  |
|  G  |   - waste -   |               |                            |               |  G  |
------------------------------------------------------------------------------------------

      [----  4KB  ----]
[------------------------------------   256 KB   ----------------------------------------]
^     ^
^     arena_object->pool_address (assuming none of these pools exist yet)
^
arena_object->address
```

The nextarena and prevarena fields allow 'usable_arenas' to serve as a doubly-linked list, 'unused_arena_objects' as a singly-linked list. usable_arenas are kept in order of ascending nfreepools - those with less free pools are accessed first.[18]

As mentioned earlier, when a pool is not in usedpools we have to see if there's a usable arena available to find or create an empty pool. If there isn't we need to get one by calling new_arena(); if there is and its freepools list turns up empty – or we had to get a new arena - we carve a new pool off of the arena. The entire process is detailed in the next section.

Example of how things look immediately after new_arena() is called.[19]

```
          [        ***   old   ***           ] [        ***   new   ***           ]
          -------------------------------------------------------------------------
arenas -> |  arena 1  |  arena 2  |  arena 3  |  arena 4  |  arena 5  |  arena 6  |
          -------------------------------------------------------------------------
                                              |          -|-> NULL
usable_arenas ----------------------->        |  arena 4  |
                                       NULL <-|-          |
                                              -------------------------------------
                                                          |          -|->        -|-> NULL
unused_arena_objects ------------------------------->     |  arena 5  |  arena 6  |
                                                          -------------------------
```

16 If the address returned is not POOL_SIZE aligned we loose a pool from the padding needed.

17 By masking certain bits you can 'back-up' to the pool address. (see Objects/obmalloc.c:725)

18 *“...This means that the next allocation will come from a heavily used arena, which gives the nearly empty arenas a chance to be returned to the system. In my unscientific tests this dramatically improved the number of arenas that could be freed.”* - Objects/obmalloc.c:894

19 To be clear: there are only six arena struct objects (in this example) that exist in actual memory; arenas, usable_arenas, and unused_arena_objects point to different spots in the 'logical' list.

CPython's Private (Memory) Heap

Jonathon Ogden - jeog.dev@gmail.com

March 30, 2016

-ALLOCATION-

PyObject_Malloc/Calloc -> _PyObject_Alloc

```
--- IF THERE IS A PARTIALLY USED POOL FROM THE USEDPOOL LIST ---

   (1)  increment the pool's ref.count and save a pointer to the first free block

   --- IF THERE IS MORE THAN ONE FREE BLOCK ON THE POOL'S FREEBLOCK LIST ---

         •  set freeblocks to address at the beginning of the block it points to
         •  return the address from (1)

   --- ELSE IF THERE IS UNUSED SPACE IN THE POOL (we can extend the free list) ---

         •  adjust the nextoffset field
         •  set the freeblock list to NULL
         •  return the address from (1)

   --- ELSE (pool is now full) ---

         •  unlink the pool from the usedpools doubly-linked list
         •  return the address from (1)

--- ELSE ---

    --- IF NO USABLE ARENAS ---

        --- GOTO NEW ARENA (BELOW) ---

    --- IF THERE IS A FREEPOOL FROM USABLE_ARENAS ---

        (1)   unlink from the arena's freepools list and decrement nfreepools

        --- GOTO ARENA CHECK (BELOW) ---

        --- GOTO POOL INITIALIZATION (BELOW) ---

    --- ELSE ---

        (1)   get a new pool (pointer) from the arena's pool_address field
        (2)   store the arena index and a dummy size index[20] in the pool header
        (3)   increment pool_address field by POOL_SIZE and decrement nfreepools

       --- GOTO ARENA CHECK (BELOW) ---

       --- GOTO POOL INITIALIZATION (BELOW) ---

--- DONE ---
```

20 If the pool header's size index field equals the allocation request size index, the old pool header is still valid and most of pool initialization is skipped; this dummy value assures that a newly 'carved' pool – that can't have a valid header, but may 'randomly' have a matching size index field – is initialized regardless.

CPython's Private (Memory) Heap

Jonathon Ogden - jeog.dev@gmail.com

March 30, 2016

NEW ARENA[21]

--- IF NO UNUSED_ARENAS (unused_arena_objects == NULL) ---

- double the number of arenas via PyMem_RawRealloc[22]
- mark each new arena as unassociated, set nextarena field to the next arena (last to NULL)
- point unused_arena_objects at the first new arena

(1) Allocate 256KB via _PyObject_Arena.alloc[23]
(2) store (1) in the address field of arena pointed to by unused_arena_objects, point unused_arena_objects at the next arena on the list
(3) adjust some of the global arena state variables
(4) set the new arena's fields
(5) set usable_arenas to the arena address returned from (1) and set both of its list pointers to NULL

ARENA CHECK

--- IF ALL THE ARENA'S FREE POOLS ARE ALLOCATED (nfreepools == 0) ---

- move usable_arenas to the next linked arena (or NULL)
- (if not NULL) set the pointer to the previous arena to NULL

POOL INITIALIZATION

(1) 'frontlink' new pool to usedpools

--- IF THE NEW POOL'S SZINDX FIELD IS THE SAME AS THE REQUEST'S INDEX SIZE ---

- no need to initialize, just save a pointer to the first free block
- adjust the freeblock list
- **return the saved block pointer**

--- ELSE ---

- set a number of the pool's header fields based on block size
- save a pointer to the first block
- set freeblock to the second block
- set the second block's freeblock list pointer to NULL[24]
- **return the saved block pointer**

21 obmalloc.c:new_arena()
22 Or allocate INITIAL_ARENA_OBJECTS many if this is the first time
23 struct PyObjectArenaAllocator (Include/objimpl.h) – the alloc field calls down to VirtualAlloc (Windows) or mmap (if available) or malloc (last resort)
24 *"Note that the available blocks in a pool are *not* linked all together when a pool is initialized ... This is consistent with that[sic] pymalloc strives at all levels (arena, pool, and block) never to touch a piece of memory until it's actually needed."* - obmalloc.c:788 (This also applies to an extension of the freelist.)

-DEALLOCATION-

PyObject_Free -> _PyObject_Free

POOL_ADDR returns the pool of a particular block so Py_ADDRESS_IN_RANGE can check to see if we (PYMALLOC) are responsible for it.[25] If we're not, pass the block address off to PyMem_RawFree and **we're done.**

If we are, first re-link it to the start of the pool's freeblock list. By checking whether the old freeblock value (the one we just replaced) is NULL we can determine if the pool *was* full. If it *was* full we need to insert it back into the appropriate usedpools list and **we're done.**

If it wasn't full, we decrement the ref.count, and check to see if it hit zero. If it didn't, we know the pool isn't empty; we can leave it in usedpools and **we're done.**

Now we know the pool is empty. We need to unlink from usedpools, and link to the front of its arena's freepools list. (We know what arena to use because of the pool's arenaindex field.) The last thing to do is manage the arenas:

--- IF ALL THE POOLS IN THE ARENA ARE NOW FREE ---

- remove the arena from the usable_arenas list by adjusting the nextarena and prevarena fields of the surrounding arenas; move usable_arenas over one if the arena in question is not linked to any previous.
- re-link with the unused_arena_objects list
- pass the address to _PyObject_Arena.free[26] and mark as unassociated

--- ELSE IF WE WENT FROM A FULL ARENA TO NOW ONE WITH A FREE POOL ---

- move to the front of usable_arenas

--- ELSE IF THE NEXT ARENA HAS LESS FREE POOLS ---

- unlink arena from usable_arenas
- iterate over the list looking for the first arena's whose nfreepools field is higher, re-link in front of that arena

--- DONE ---

SUMMARY

Hopefully we covered elements of CPython's private heap that are useful to python programmers. Obviously there are other areas that may be of interest: custom Allocators, debugging features, pitfalls of using PYMALLOC under certain situations, and potential improvements, to name a few – not to mention the broader memory management model itself.

As stated, there is plenty of helpful information in the source. A number of good jumping-off points for the interested reader are referenced in the footnotes.

25 An explanation of how can be found in the detailed comments beginning at: Objects/obmalloc.c:1036
26 struct PyObjectArenaAllocator (Include/objimpl.h) – the free field calls down to VirtualFree (Windows) or munmap (if available) or free (last resort)