Improving Python's Memory Allocator

Evan Jones <ejones@uwaterloo.ca>
http://evanjones.ca/

Abstract

Python automatically manages memory using reference counting. In order to provide good performance for typical programs, Python provides its own memory allocator for small objects (≤ 256 bytes). However, the original implementation does not release memory to the operating system, which can cause performance problems. This paper describes how the original implementation works, and how the problem was fixed. It also discusses additional areas where Python's memory management can be improved.

1. Introduction

One of Python's many benefits is that it automatically manages memory, as any high level language should. It uses reference counting to determine when the memory occupied by an object can be reclaimed. However, under the hood, that memory is never actually released to the operating system. Instead, the Python interpreter holds on to it, and will reuse it as needed. In many situations this is a good strategy as it minimizes the amount of operating system overhead involved in memory allocation. However, if a Python process is long running, it will occupy the maximum amount of memory that it needed. If an application's peak memory usage is much larger than its average usage, this is wasteful and can hurt the overall system performance, as well as the performance of the application itself.

The way to work around this problem is to avoid allocating significant numbers of temporary objects in a long-lived process. For example, if an application must perform some expensive, one-time computation, it could be computed in another process using `fork()`, and the results passed back via a pipe. Alternatively, instead of using some large temporary list or dictionary, the data could be stored in the file system. These hacks defeat the purpose of automatic memory manangement. A programmer should not need to forced to think about when to return memory to the operating system.

In order to get an understanding of how Python manages memory, we first present an overview of how Python's memory allocator works before describing the modifications to resolve the issue. We then discuss the advantages and disadvantages of this change before concluding with some refinements that would further improve Python's memory efficiency.

2. The pymalloc Allocator

Python's memory allocator, called pymalloc, was written by Vladimir Marangozov and originally was an experimental feature is Python 2.1 and 2.2, before becoming enabled by

default in 2.3. Python uses a lot of small objects that get created and destroyed frequently, and calling `malloc()` and `free()` for each one introduces significant overhead. To avoid this, pymalloc allocates memory in 256 kB chunks, called arenas. The arenas are divided into 4 kB pools, which are in turn subdivided into fixed sized blocks, as shown in Figure 1. The blocks are returned to the application. To understand the details of the allocator, we will step through the process that occurs when a block is needed, and when it is freed.

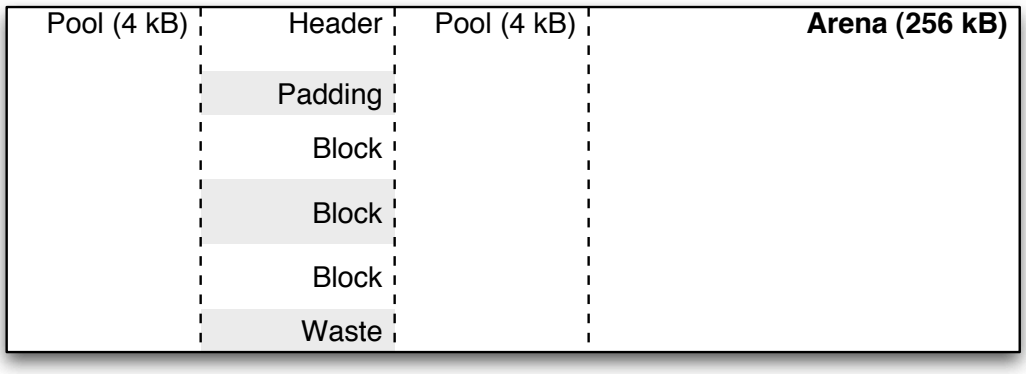

Figure 1: Memory layout of arenas, pools and blocks

2.1. Allocating Memory

When allocating a new object, the first allocator looks to see if there are any pools that have already been divided into blocks of the required size. The `usedpools` array, shown in Figure 2, contains linked lists of pools that have blocks of each size. Each of the pools has a singly linked list of available blocks. If there is a pool, we pop a block off of its list. This is the most common case and it is very fast as it only requires a few memory reads and one write. If the block was the last one in the pool, then it is now completely allocated. In this case, we also pop the pool off. Finally, we return the block to the application.

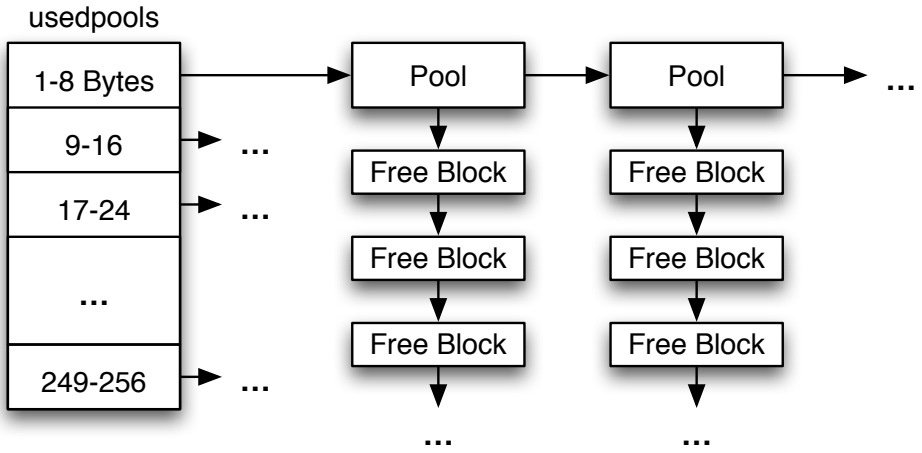

Figure 2: The `usedpools` array for storing partially allocated pools

If there are no pools of the correct size, we need to find an available pool. The `freepools` linked list, shown in Figure 3, contains available pools. If there is a pool on the list, we pop it off. Otherwise, we need to create a new pool. If there is space left at the end of the last arena we allocated, we can cut another pool off using the `arenabase` pointer as indicated shown in Figure 3. If there is no space, we call `malloc()` to allocate a new arena. Now that we finally have a pool, we divide it into fixed size blocks, place the pool in `usedpools` and finally return one block to the application.

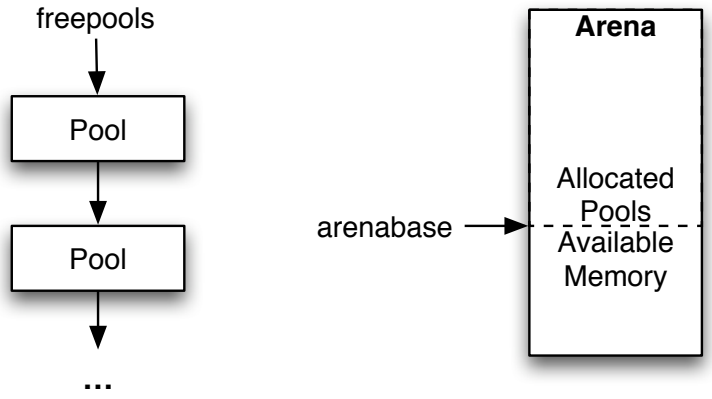

Figure 3: The `freepools` list and allocating new pools

2.2. Freeing Memory

When an application frees a block, the procedure is very similar. First, some magic is performed in order to determine what pool a block belongs to (see `obmalloc.c` for the gory details). The block is then placed on the pool's free list. If the pool was completely allocated, it is added to the `usedpools` array. If the pool is now completely available, it is removed from `usedpools` and added to the `freepools` list.

3. Solving the Problem

The problem is that the process stops there. There is no code to match free pools with the arenas they belong to, and then to return free arenas to the operating system. To do this, a free pool needs to be put on a list specifically for the arena that it comes from. Once all the pools in the arena are on that list, the entire arena can be released.

This means that some data structure must be associated with each arena to track available and allocated pools. An array of arena structures is maintained to provide good memory locality when manipulating arenas. The next challenge is to get from a pool to the arena it was allocated from. To do this, the pool header was extended to include the index of the arena in the array. Additionally, the `freepools` list was changed into a `partially_allocated_arenas` list. This list, shown in Figure 4, keeps track of arenas that have pools available. With these changes, the procedure for allocating memory described in Section 2.1 needs to be modified when a free pool is needed. Instead of

taking the pool from the `freepools` list, the pool is taken from the first arena in the `partially_allocated_arenas` list.

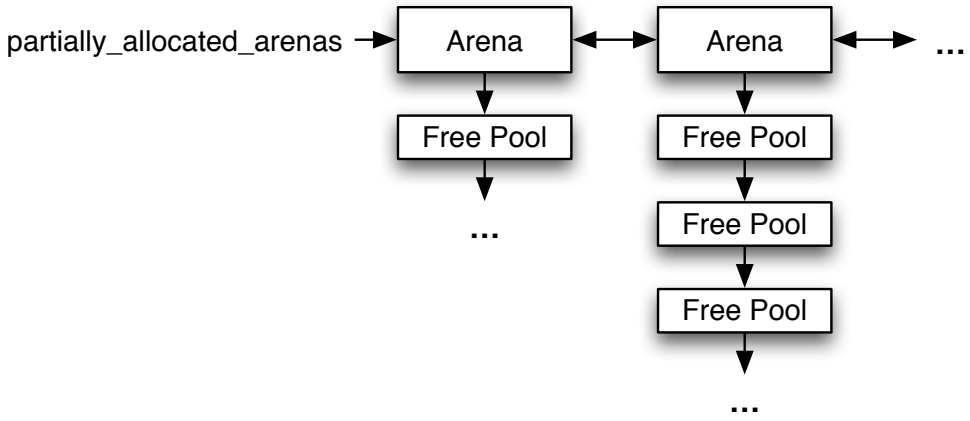

Figure 4: The `partially_allocated_arenas` list contains arenas with free pools

3.1. Freeing Memory Take Two

More significant changes are made to the procedure for freeing memory. If a pool is completely free, the pool is placed on a free list specifically for its arena. If the arena was completely allocated, it is added to the `partially_allocated_arenas` list. This linked list contains arenas that contain some free pools. If the arena is now completely available, it is removed from the `partially_allocated_arenas` list and it is released to the operating system by calling `free()`. One additional tweak is that the `partially_allocated_arenas` list is kept sorted so that the most allocated arenas are used first when a pool is needed. This gives the arenas that are nearly empty a chance to become completely deallocated and returned to the operating system, and was experimentally determined to improve the amount of memory that could be freed.

4. Impact

These changes will not affect the correctness of programs, only their performance. The new version of the allocator is capable of returning memory to the operating system, which should improve the overall system performance for applications that allocate memory in a bursty fashion. The disadvantage is that is some additional overhead required when freeing blocks, to keep track of where they came from. For programs that cyclically allocate and deallocate memory, there is also the additional overhead of repeatedly calling `malloc()` and `free()`. The overhead did not exist before because Python would hold on to the maximum amount of RAM that was required. However, this additional overhead should be small because the common code paths in the memory allocator are unchanged.

Unfortunately, this patch can only free an arena if there are no more objects allocated in it anymore. This means that fragmentation is a large issue. An application could have many

megabytes of free memory, scattered throughout all the arenas, but it will be unable to free any of it. This is a problem experienced by all memory allocators. The only way to solve it is to move to a compacting garbage collector, which is able to move objects in memory. This would require significant changes to the Python interpreter.

4.1. Example Application

To illustrate the effect of this patch, consider the simple program shown in Figure 5. It allocates a large number of dictionaries, frees them, and finally allocates them again. The memory usage over time is shown in Figure 6. The ideal allocator shows that the program allocates approximately 550 MB, then frees all but 50 MB, reallocates it, and finally exits. The original allocator maintains the maximum amount of memory the entire time, while the new allocator closely follows the ideal allocator behaviour. For this application, the new allocator is actually harmful because the memory is reused immediately. However, if the time between the two allocations were long, the new allocator would be very useful.

```
iterations = 4000000

l = []
for i in xrange( iterations ):
l.append( None )

for i in xrange( iterations ):
l[i] = {}

for i in xrange( iterations ):
l[i] = None

for i in xrange( iterations ):
l[i] = {}
```

Figure 5: Example program that allocates and frees large amounts of memory

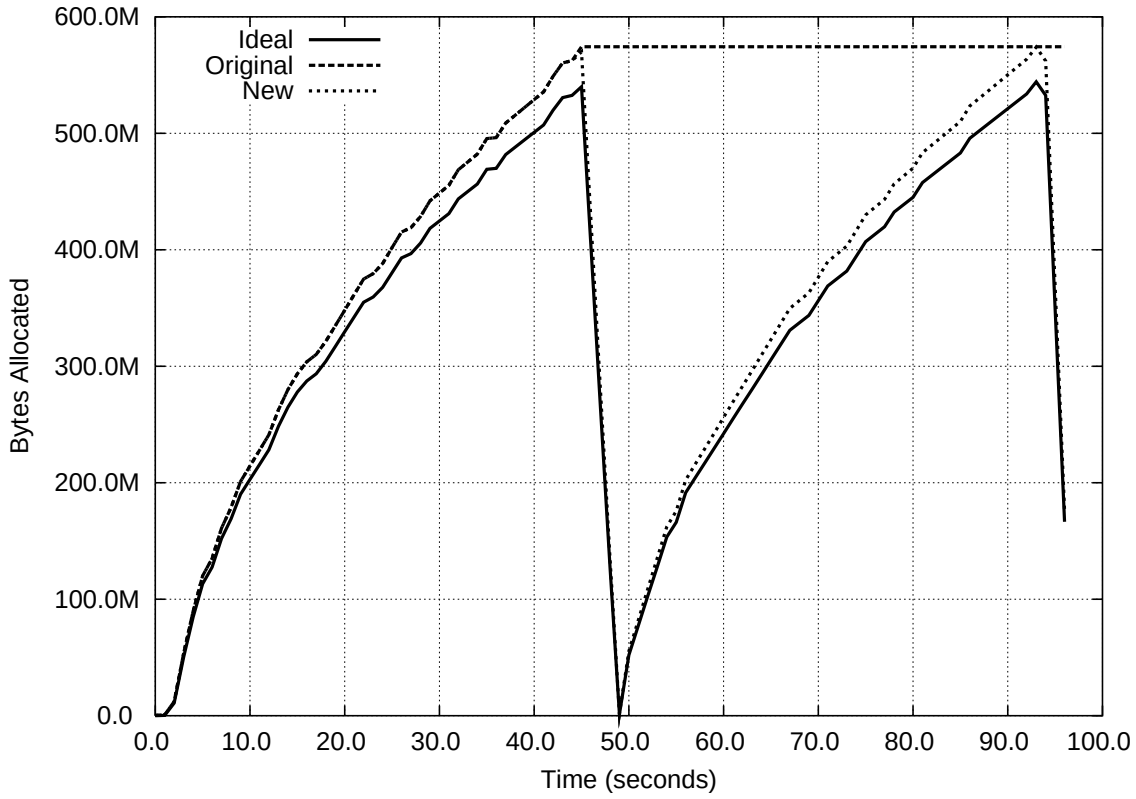

Figure 6: Memory allocator behaviour for the example program

5. Future Improvements

There is still one area where Python's memory management can be improved. There are a few objects that do not use the pymalloc allocator. The most important ones are Python's integers, floats, lists and dictionaries. These data types maintain their own list of free objects in order to make more efficient use of space and time for these very common objects. However, the current implementation can lead to the same problem that the changes described in this paper were trying to solve.

The worst offenders are integers and floats. These two object types allocate their own blocks of memory of approximately 1kB, which are allocated with the system `malloc()`. These blocks are used as arrays of integers and float objects, which avoids waste from pymalloc rounding the object size up to the nearest multiple of 8. These objects are then linked on to a simple free list. When an object is needed, one is taken from the list or a new block is allocated. When an object is freed, it is returned to the free list.

This scheme is very simple and very fast, however, it exhibits a significant problem: the memory that is allocated to integers can never be used for anything else. That means if you write a program which goes and allocates 1 000 000 integers, then frees them and allocates 1 000 000 floats, Python will hold on to enough memory for 2 000 000 numerical objects. The solution is to apply a similar approach as was described above.

Pools could be requested from pymalloc, so they are properly aligned. When freeing an integer or a float, the object would be put on a free list for its specific pool. When the pool was no longer needed, it could be returned to pymalloc. The challenge is that these types of objects are used frequently, so care is required to ensure good performance.

Dictionaries and lists use a different scheme. Python always keeps a maximum of 80 free lists and dictionaries, any extra are freed. This is not optimal because some applications would perform better with a larger list, while others need less. It is possible that self-tuning the list size could be more efficient.

6. Conclusions

The changes to the memory allocator make it possible for Python return memory to the operating system. This is an issue for long running applications whose peak memory usage is much greater than their average usage. For these applications, this can greatly improve the overall system performance. It is possible to further improve Python's memory allocator. In particular, the free lists that are maintained for integers and floats can be a significant area of wasted memory in some Python programs.