

# dynamic allocation file buffer

---

*Source:* <http://coding.derkeiler.com/Archive/Python/comp.lang.python/2008-09/msg00818.html>

---

- *From:* castironpi <castironpi@xxxxxxxx>
  - *Date:* Tue, 9 Sep 2008 14:59:19 -0700 (PDT)
- 

I will try my idea again. I want to talk to people about a module I want to write and I will take the time to explain it. I think it's a "cool idea" that a lot of people, forgiving the slang, could benefit from. What are its flaws?

A user has a file he is using either 1/ to persist binary data after the run of a single program (persistence) or 2/ share binary data between concurrently running programs (IPC / shared memory). The data are records of variable types and lengths that can change over time. He wants to change a record that's already present in the file. Here are two examples.

Use Case 1: Hierarchical ElementTree-style data

A user has an XML file like the one shown here.

```
<a>
<b>
<c>Foo</c>
</b>
...
```

He wants to change "Foo" to "Foobar".

```
<a>
<b>
<c>Foobar</c>
</b>
...
```

The change he wants to make is at the beginning of a 4GB file, and recopying the remainder is an unacceptable resource drain.

Use Case 2: Web session logger

A tutor application has written a plugin to a webbrowser that records the order of a user's mouse and keyboard activity during a browsing session, and makes them concurrently available to other applications in a suite, which are written in varying languages. The user takes

## dynamic allocation file buffer

some action, such as surfing to a site or clicking on a link. The browser plugin records that sequence into shared memory, where it is marked as acknowledged by the listener programs, and recycled back into an unused block. URLs, user inputs, and link text can be of any length, so truncating them to fit a fixed length is not an option.

### Existing Solutions

- Shelve – A Python Standard Library shelf object can store a random access dictionary mapping strings to pickled objects. It does not provide for hierarchical data stores, and objects must be unpickled before they can be examined.
- Relational Database – Separate tables of nodes, attributes, and text, and the relations between them are slow and unwieldy to reproduce the contents of a dynamic structure. The VARCHAR data type still carries a maximum size, no more flexible than fixed-length records.
- POSH – Python Object Sharing – A module currently in its alpha stage promises to make it possible to store Python objects directly in shared memory. In its current form, its only entry point is 'fork' and does not offer persistence, only sharing. See: <http://poshmodule.sourceforge.net/>

### Dynamic Allocation

The traditional solution, dynamic memory allocation, is to maintain a metadata list of "free blocks" that are available to write to. See:

[http://en.wikipedia.org/wiki/Dynamic\\_memory\\_allocation](http://en.wikipedia.org/wiki/Dynamic_memory_allocation)

<http://en.wikipedia.org/wiki/Malloc>

<http://en.wikipedia.org/wiki/Mmap>

[http://en.wikipedia.org/wiki/Memory\\_leak](http://en.wikipedia.org/wiki/Memory_leak)

The catch, and the crux of the proposal, is that the metadata must be stored in shared memory along with the data themselves. Assuming they are, a program can acquire the offset of an unused block of a sufficient size for its data, then write it to the file at that offset. The metadata can maintain the offset of one root member, to serve as a 'table of contents' or header for the remainder of the file. It can be grown and reassigned as needed.

An acquaintance writes: It could be quite useful for highly concurrent systems: the overhead involved with interprocess communication can be overwhelming, and something more flexible than normal object persistence to disk might be worth having.

### Python Applicability

The usual problems with data persistence and sharing apply. The format of the external data is only established conventionally, and conversions between Python objects and raw memory bytes take the usual overhead. 'struct.Struct', 'ctypes.Structure', and 'pickle.Pickler' currently offer this functionality, and the buffer offset obtained

from 'alloc' can be used with all three.

Ex 1.

```
s= struct.Struct( 'III' )
x= alloc( s.size )
s.pack_into( mem, x, 2, 4, 6 )
Struct in its current form does not permit random access into
structure contents; a user must read or write the entire converted
strucutre in order to update one field. Alternative:
s= struct.Struct( 'I' )
x1, x2, x3= alloc( s.size ), alloc( s.size ), alloc( s.size )
s.pack_into( mem, x1, 2 )
s.pack_into( mem, x2, 4 )
s.pack_into( mem, x3, 6 )
```

Ex 2.

```
class Items( ctypes.Structure ):
    _fields_ = [
        ( 'x1', ctypes.c_float ),
        ( 'y1', ctypes.c_float ) ]
x= alloc( ctypes.sizeof( Items ) )
c= ctypes.cast( mem+ x, ctypes.POINTER( Items ) ).contents
c.x1, c.y1= 2, 4
The 'mem' variable is obtained from a call to PyObject_AsWriteBuffer.
```

Ex 3.

```
s= pickle.dumps( ( 2, 4, 6 ) )
x= alloc( len( s ) )
mem[ x: x+ len( s ) ]= s
'dumps' is still slow and nor does permit random access into contents.
```

## Use Cases Revisited

Use Case 1: Hierarchical ElementTree–style data

Solution: Dynamically allocate the tree and its elements.

```
Node: tag: a
Node: tag: b
Node: tag: c
Node: text: Foo
```

The user wants to change "Foo" to "Foobar".

```
Node: tag: a
Node: tag: b
Node: tag: c
Node: text: Foobar
```

Deallocate 'Node: text: Foo', allocate 'Node: text: Foobar', and store the new offset into 'Node: tag: c'. Total writes 6 bytes 'foobar', a one–word offset, and approximatly 5– 10–word metadata update.

#### Use Case 2: Web session logger

Dynamically allocate a linked list of data points.

Data: 'friendster.com'

Data: 'My Account'

Allocate one block for each string, adding it to a linked list. As listeners acknowledge each data point, remove it from the linked list. Keep the head node in the 'root offset' metadata field.

#### Restrictions

It is not possible for persistent memory to refer to live memory. Any objects it refers to must also be located in file. Their mapped addresses must not be stored, only their offsets into it. However, live references to persistent memory are eminently possible.

#### Current Status

A pure Python alloc-free implementation based on the GNU PAVL tree library is on Google Code. It is only in proof-of-concept form and not commented, but does contain a first-pass test suite. See: <http://code.google.com/p/pymmapstruct/source/browse/#svn/trunk>  
The ctypes solution for access is advised.

.