Python Object Sharing

Steffen Viken Valvåg

December 2002

Abstract

Language support for threads was introduced at a late stage in the Python development process. To accomodate that the majority of Python code was not thread-safe, an approach using a global interpreter lock was introduced. This lock has remained a major scalability bottleneck when executing multi-threaded Python programs on multi-processor architectures. A common workaround to achieve parallelism on such architectures is to use multiple processes instead of multiple threads. Typically, these types of applications use ad-hoc shared memory or messaging APIs for interprocess communication.

This thesis presents POSH, an extension to Python that combines the advantages of the shared memory programming model in a multi-threaded Python program with the scalability of a multi-process Python program.

POSH allows Python objects to be transparently shared across multiple processes using shared memory. Shared objects are indistinguishable from regular objects, as seen by Python code. The POSH system is implemented as an extension module and requires no changes to the Python runtime.

Acknowledgements

The author would like to thank his supervisors, Åge Kvalnes and Kjetil Jacobsen, for all their invaluable technical and moral support. This thesis has benefited greatly from their insights and contributions.

Contents

List of Figures

Chapter 1

Introduction

1.1 Background

Python is a dynamically typed, high-level programming language that is used for a wide range of applications. Python has evolved from a simple scripting language to a full-fledged object oriented programming language.

Language support for threads was introduced at a late stage in the Python development process. At this point, the majority of Python code was not thread-safe. As a consequence, an approach using very coarse grained locking was adopted. In fact, the approach was to use a single global lock, known as the *global interpreter lock*, to serialize execution of byte codes. To this day, this lock has remained a major bottleneck when executing multi-threaded Python programs on multi-processor architectures. A common workaround to achieve parallelism on multi-processor architectures is to use multiple processes instead of threads. In such multi-process applications, inter-process communication is performed using ad-hoc shared memory or a messaging API.

POSH is an extension to Python that attempts to address the above problems by enabling placement of Python objects in shared memory. The design and implementation of POSH is the subject of this thesis.

1.2 Problem definition

The goal of this thesis is to extend the Python programming language to support placement of objects in shared memory. To the extent possible, the

support should satisfy the following requirements.

- Objects in shared memory should be interchangeable with regular objects.
- Objects in shared memory should be accessible to multiple concurrent processes.
- Constraints on which kind of objects that can be placed in shared memory should be minimized.
- The support should require minimal changes to the Python runtime.

1.3 Method and approach

The problem will be investigated through researching existing Python documentation. There are no references on the implementation of the Python runtime, so a thorough analysis of the Python source code is also required. Based on this research and analysis, an actual design and implementation will be performed.

1.4 Limitations

Due to time constraints, we have had to omit performance evaluation of POSH abstractions and mechanisms.

1.5 Outline

Chapter 2 describes important abstractions, data-structures and algorithms used internally in the Python runtime environment. This chapter provides a background for understanding the design and implementation of POSH.

Chapter 3 describes the design and implementation of POSH.

Chapter 4 concludes the thesis. An outline for future work is provided.

Appendix A contains a complete listing of the source code for the POSH system.

Chapter 2

Python internals

This chapter provides a background for understanding the design and implementation of POSH. Important abstractions, data-structures and algorithms used internally in the Python runtime environment are presented.

2.1 Introduction

Python is a dynamically typed, high-level programming language that is used for a wide range of applications. It is maintained as an open source project, continuously evolving to include new language features as well as libraries. Its popularity has increased steadily as it has become a powerful tool for both scripting, functional programming and object-oriented programming.

This chapter is not intended as an introduction to the Python language, but rather as a view behind the scenes, examining the implementation of the Python interpreter and runtime. A basic understanding of the topics treated in this chapter is required in order to understand the implementation of POSH.

There exist no official resources describing the internals of Python. The Python documentation is targeted at programmers using the language, and also includes a description of the Python/C API, which is for authors of extension modules. However, this is a high-level API that only provides limited insight into low-level implementation details. In addition, the documentation tends to be outdated, since Python is an open source project under continuous development. The only real way of gaining a thorough understanding of Python's internals is by examining the source code. The standard distribution of Python currently has approximately 275000 lines

of C code, in addition to the modules that are implemented in Python. This chapter primarily focuses on the implementation details of particular relevance with regard to POSH.

2.2 The Python execution environment

2.2.1 Interpretation of Python code

Python is an interpreted language, which means that the code is parsed and executed at run-time, without any prior compilation. However, Python source code is transformed to an intermediate format known as byte-codes before interpretation. The byte-codes are cached on disk to optimize repeated execution of the same source code. They also speed up interpretation in general given the presence of looping constructs. Byte-codes may be viewed as instructions for a stack-based virtual machine known as the byte-code interpreter. When the Python interpreter executes a program, the byte-code interpreter is its main loop, repeatedly executing byte-codes from the instruction stream. Each byte-code may push or pop objects off the interpreter's operand stack.

Python also allows *extension modules*[1] to be implemented in C, using the Python/C API. When Python code invokes a function defined in an extension module, the call is transformed into a single byte-code in the instruction stream. Since the extension module may perform calculations of arbitrary duration, the time to execute a single byte code may vary greatly.

2.2.2 Threading model

Language support for threads was introduced at a late stage in the Python development process. At that point, the majority of Python code was not thread-safe. As a consequence, an approach using very coarse grained locking was adopted. In fact, the approach was to use a single global lock to serialize execution of byte codes. To this day, the lock has remained a major bottleneck when executing multi-threaded Python programs on multi-processor architectures, since the contention on the lock all but neutralizes the performance gain of multiple processors. Given the presence of extension modules, the execution time of a single byte code is actually unbounded, so the degree of parallelism achieved in multithreaded programs may be unsatisfactory. Extension modules may voluntarily release the global

[1] Also known as *built-in modules*.

```
typedef struct {
    PyTypeObject* ob_type;
    int ob_refcnt;
} PyObject;
```

Figure 2.1: Definition of the `PyObject` data structure.

interpreter lock while executing thread-safe code, but not all modules do this, as it requires extra vigilance with regard to thread safety.

A common workaround employed by Python programs to achieve parallelism on multi-processor architectures is to use multiple processes instead of threads. However, such multi-process applications require some form of inter-process communication, and cannot employ the convenient shared memory programming model of multi-threaded programs.

2.2.3 Object model

As described in Section 2.2.1 the entries on the byte-code interpreter's operand stack are all objects. In fact, all values in the Python language are objects, including entities such as functions, types and even stack frames. Being able to manipulate these objects at run-time, just like any other object, enables programming paradigms that are in many cases inapplicable in statically typed languages.

Being a dynamically typed language, Python treats all objects generically, having no *a priori* information about their types. In general, the emphasis in a Python program does not lie on what type an object has, but rather on what interface the object supports. Objects that support the same interface are generally interchangeable. Explicitly checking the types of objects is considered bad programming practice, since it defeats the genericness inherent in the language.

2.3 Python objects

2.3.1 Representation of objects

All Python objects have a common physical representation, defined by the `PyObject` C structure shown in figure 2.1. The structure only defines two fields, which are common to all objects. They are the *type pointer* (`ob_type`)

```
typedef struct {
    PyTypeObject* ob_type;
    int ob_refcnt;
    int ob_size;
} PyVarObject;
```

Figure 2.2: Definition of the `PyVarObject` data structure.

and the *reference count* (`ob_refcnt`). The reference count is used for garbage collection, as described in Section 2.6. The type pointer points to the object's *type object*, which is described in Section 2.4.

2.3.2 Fixed-size vs. variable-size objects

Python distinguishes between two classes of objects, according to their requirements with respect to memory allocation.

Fixed-size objects are uniformly sized, meaning that all objects of a given type have the exact same size in bytes. Examples of fixed-size objects include integers and floats.

Variable-size objects are special in that objects of the same type may differ in size. Each variable-size object contains an extra field that defines its size. This is defined by the `PyVarObject` C structure shown in figure 2.2, which extends the `PyObject` structure with a field named `ob_size`. The `ob_size` field is initialized when the object is created, and subsequently never changes. Examples of variable-size objects include strings (which contain a variable number of characters) and tuples (which contain a variable number of references to other objects).

Note that referring to an object as variable-size does not imply that the object's size may actually change. It simply states that separate objects of that type may differ in size. *Every* Python object keeps its initial size throughout its lifetime; once it is allocated, it stays the same size until it is deallocated. If objects were to change size throughout their lifetime, the Python memory allocator would need the ability to move them, which would be difficult at best, since objects are referenced throughout the Python runtime using direct pointers. Instead, mutable objects such as lists and dictionaries "grow" by allocating additional memory in auxiliary data structures, not by resizing the memory region in which they are allocated.

```
typedef struct {
    PyObject_HEAD
    long ob_ival;
} PyIntObject;
```

Figure 2.3: Object structure of `int` objects.

```
typedef struct {
    PyObject_VAR_HEAD
    digit ob_digit[1];
} PyLongObject;
```

Figure 2.4: Object structure of `long` objects.

Thus, lists and dictionaries are actually classified as fixed-size, since their objects always have a uniform size.

2.3.3 Extending object structures

Depending on its type, an object may require data fields that are not defined by `PyObject`. For example, integer objects need to store their actual value as a C variable of type `long`. Python generally defines a data structure for each type of object, known as its *object structure*, that extends the basic `PyObject` or `PyVarObject` data structures with more fields. As an example, figure 2.3 shows the object structure of integer objects. The `PyObject_HEAD` macro defines the `ob_type` and `ob_refcnt` fields from `PyObject`. The `ob_ival` field is specific to integer objects. Figure 2.4 shows the data structure of long integer objects (objects of type `long`). The `PyObject_VAR_HEAD` macro defines the fields from `PyVarObject`. The structure declares that the object contains an array holding 1 digit. In reality, the number of digits is stored in the `ob_size` field, and the allocation of the object will reserve sufficient memory to hold that many digits. This approach to implementing variable-sized arrays is common in C, since the language does not enforce out-of-bounds checks on array indexes.

2.3.4 Referencing objects from C code

The Python runtime, which is implemented in C, views all Python objects as variables of type `PyObject*`. Only when implementing operations specific to a particular type of objects, are the pointers cast to more specific types such

as `PyListObject*`. This casting is generally only done after checking the object's type explicitly (by examining its type pointer), or in places where the type of the object can be logically deduced (in which case an assertion prior to the cast would be appropriate).

2.4 Python types

As noted in Section 2.3.1, every object contains a pointer to its *type*, which is actually a regular Python object known as a *type object*. The type object contains a collection of function pointers referred to as *method slots*, which define the effects of various generic operations on objects of that type. When the interpreter executes a byte-code that performs some generic operation, such as addition, it examines the type object of the operand(s) and calls one of the functions pointed to by its method slots. For example, when a statement such as "`print x`" is compiled into byte-codes, the `PRINT_ITEM` byte-code is generated. This byte-code operates on a single operand, which is popped off the top of the interpreter's operand stack. When the `PRINT_ITEM` byte-code is interpreted, the type object of the operand is examined. In this particular case, a method slot named `tp_print` is invoked, and it is the function pointed to by this method slot that defines the actual effect of printing the object.

The type object contains other vital information about an object, such as the number of bytes it occupies in memory. For fixed-size objects, this equals the size of the object's data structure. For variable-size objects, the type object defines the size in bytes of each item in the object, as well as its base size. The actual size of an object may be calculated using this information in combination with the `ob_size` field from the object.

2.4.1 Inheritance

Python allows types to subtype other types using single or multiple inheritance, and forces all types to inherit from the fundamental `object` type if no other base types are specified. This means that all types can be represented in a hierarchy rooted at the `object` type. Since multiple inheritance is allowed, this hierarchy will take the form of a directed acyclic graph.[2]An example type hierarchy showing some of the standard built-in types as well as a few fictitious types is depicted in figure 2.5. In the figure, the `document` type uses multiple inheritance to subtype both the `searchable` type and the built-in `string` type.

[2]In single inheritance languages, type hierarchies are always trees.

Figure 2.5: A sample type hierarchy.

Figure 2.6: Inheritance graph for the `document` type.

```
typedef struct {
    PyIntObject base;
    char a;
    float b;
} SubTypeOfIntObject;
```

Figure 2.7: Object structure of a type subtyping the built-in `list` type.

```
class name(bases):
        body
```

Figure 2.8: Syntax of the class statement.

An edge in the type hierarchy graph represents inheritance, where the edge leads from the base type to the subtype. Conversely, an *inheritance graph* for a type may be produced by recursively drawing edges from a type to its base types. Figure 2.6 shows the inheritance graph as it would appear for the `document` type.

Type objects keep track of their relationship to other types using two data structures. Firstly, each type keeps a simple list of its base types, accessible from Python code through the special `__bases__` attribute. Secondly, each type maintains a list of its immediate subtypes, which can be retrieved by calling the special `__subclasses__` method.

In general, the concept of inheritance means that the methods and fields of the base types apply to its subtypes, too, unless explicitly overridden by the subtype. In a statically typed language, the compiler will resolve all method calls and field references by examining the type and then its base types in some predefined order until the method or field in question is found. This is how inheritance is implemented. Python, being a dynamically typed language, essentially does the same, except at run-time. The implementation of inheritance in Python is described in more detail in Section 2.5.5.

2.4.2 Built-in vs. user-defined types

As mentioned in Section 2.2, Python modules may be implemented in C or Python. Likewise, Python types may be created using C or Python code. In the same way that modules implemented in C are sometimes referred to as built-in modules, types implemented in C are called *built-in types*. Most of the types provided by the standard distribution, such as `int` or `dict`,

are built-in types. They are created by statically allocating a type object in a built-in module, and binding it to the module's namespace (giving it a name) in the module's initialization function. The methods of the type are implemented as C functions, and pointed to by the type object's method slots.

Built-in types typically store data required for their implementation in the object structure, which is defined similarly to those in Figure 2.3 and 2.4. If the type has a built-in base type, the subtype's object structure must extend that of its base type, enabling the subtype to rely on the implementation provided by the base type. Figure 2.7 shows an example of this technique, where a subtype of the built-in `int` type extends its object structure with two more fields. The data structures associated with the subtype are appended to the existing object structure defined by the base type. One way of looking at the `PyObject` data structure is as the object structure of the fundamental `object` type. All object structures consequently extend this structure, since all types are subtypes of the `object` type.

Since subtypes of built-in types must have an object structure that extends the base type's object structure, a type inheriting from multiple built-in types is faced with a conflict. It cannot extend more than one object structure at a time, which means that it is unable to rely on the implementation of more than one built-in base type. As a consequence, a general restriction on Python types is that they cannot have more than one built-in base type.

Naturally, new types may also be created by Python code, in which case the *class statement* is used. Types created in this way are called *user-defined types*. The syntax of the class statement is shown in figure 2.8. In the figure, `name` is the name of the new type, and `bases` is a comma-separated list of bases. `body` is a block of statements that form the body of the class statement. In practice, the body often consists of a series of function definitions (or *def statements*), which define the methods of the new type.

Informally, the effect of the class statement is to create a new type of the specified name, which inherits from the given base types, and has the methods defined in the body. A new type object is created and bound to the specified name in the current namespace. Section 2.4.4 goes into more detail about the semantics of a class statement.

Note that there are no fundamental differences between built-in and user-defined types, except for the way they are created, and the fact that types may only have one built-in base type.

```
# Execute the statements in the body and store
# the results in a dictionary d
d = {}
exec body in d
del d['__builtins__']

# Create the new type by calling the meta-type
name = type('name', bases, d)
```

Figure 2.9: Creating a new type by calling the meta-type.

2.4.3 Meta-types

Since the type object is itself a regular object, it also has a type of its own. The type of a type is generally referred to as a meta-type. In the same way that types define the behavior of objects, meta-types define the behavior of types. Of course, meta-types have types, too, so the terminology goes on to include the terms meta-meta-type and so forth. However, those terms are rarely needed, since the chain of type pointers normally terminates fairly soon in a type that is its own type. Indeed, this is exactly the case for Python's most common meta-type, `type`.[3] As explained in the next section, meta-types play a central role in the creation of types.

2.4.4 The Don Beaudry hook

Section 2.4.2 describes the class statement, and how it is used to create new types. However, that is a simplified explanation, which assumes that the meta-type in use is `type`. In fact, the class statement may be used to create any kind of object, depending on which meta-type is specified. The key mechanism that allows this flexibility is called the *Don Beadry hook*, after its inventor, and is a simple, yet elegant rule. Just like new objects are created by calling their type, new types are created by calling their *meta-type*. So the effect of a class statement is actually a call to a meta-type, and the return value from this call is bound to the name specified in the statement. The parameters passed to the meta-type are:

- The name specified in the class statement, as a string.
- A tuple containing the bases.

[3] Readers may find it confusing that the name of a *meta-type* is `type`. If so, consider that the type of integers is named `int` and the type of lists is named `list`. Correspondingly, it makes sense for the type of types to be named `type`.

- A dictionary, which is the namespace that results from executing the statements in the body of the class statement. In practice, this will often be a dictionary containing the methods of the new type.

Figure 2.9 shows a code snippet that essentially does the same as a class statement, except that it calls the meta-type manually. The results of that code and a regular class statement (using the `type` meta-type) are identical.

In order to create other kinds of objects than types using the class statement, a different meta-type must be specified. This can be done by assigning a value to the special `__metaclass__` attribute somewhere in the class body, or by inheriting the meta-type from the base(s). When executing a class statement, the Python runtime will examine the bases listed, and use their special `__class__` attributes to determine the meta-type to use. As described in Section 2.4.5, the default meta-type used when no bases are specified is in fact *not* `type`. This is for backwards compatibility reasons, and will change in future versions of Python. For the moment, however, the most common technique for creating types is to inherit from the built-in `object` type, or from another built-in type whose meta-type is `type`. This is actually required to make sure that the class statement does produce a new type.

It is possible (and quite easy) to define custom meta-types, and this actually has many applications, since meta-types are a very powerful mechanism. For instance, a custom meta-type can add logging calls to all methods of the types it creates, or even (ab)use the class statement to create something entirely different from a type.

2.4.5 Classes

Python was not originally designed as an object-oriented language, and types initially lacked the ability to inherit from other types. The concept of subtyping[6] was actually added quite recently, even though the language has supported object-orientation for some time. This is because Python's first approach to supporting object-orientation was adding a new kind of objects called *classes*, without changing the fundamentals of how types worked. Classes and types thus became separate entities that were in no way interchangeable. This has later been recognized as a bad design choice, since it introduced an artificial distinction between types and classes known as the *class/type dichotomy*. In the most recent versions of Python, great changes have been made to the type system, essentially rendering classes obsolete. The process of revising the type system while maintaining backwards compatibility for classes is often referred to as the *type/class unification*[5].

```
>>> class A: pass
...
>>> class B: pass
...
>>> class C(object): pass ## C is a type, not a class
...
>>> a = A(); b = B(); c = C()
>>> a.__class__, b.__class__, c.__class__
(<class __main__.A>, <class __main__.B>, <type '__main__.C'>)
>>> type(a), type(b), type(c)
(<type 'instance'>, <type 'instance'>, <type '__main__.C'>)
```

Figure 2.10: An interpreter session illustrating the difference between types and classes.

When the class statement was introduced in Python, its effect was to create a new object of type `ClassType`, also known as a class. (Hence the name and syntax of the statement). The Don Beadry hook described in Section 2.4.4 was implemented later, and classes are now created by specifying `ClassType` as the meta-type to use. For backwards compatibility, `ClassType` is the default meta-type used when no bases are specified in the class statement. In addition, if all the bases are classes, the class statement will use the `ClassType` meta-type, thus creating a class. Like types, classes (or `ClassType` objects) have a special `__bases__` attribute that refers to its base classes. To support "instantiation", classes are callable, and return a new `InstanceType` object when called. `InstanceType` objects are also referred to as class instances, and have a special `__class__` attribute that refers to the class that created them. However, all `InstanceType` objects are in fact instances of the same type (namely `InstanceType`), so there is no correspondence between the class and type of an object. Figure 2.10 shows an interpreter session that illustrates this point. Inheritance for class instances is implemented by the `InstanceType` type, which overrides the attribute lookup methods to traverse the inheritance graph of the object's class. This is similar to the way inheritance is implemented for types, as described in Section 2.5.5.

The type/class unification has entailed quite major changes to the Python runtime, but the changes are nevertheless very transparent to the user. This has unfortunately contributed to a general confusion about the terms *types* and *classes*. Many programmers seem to insist on calling their user-defined types classes, since they are created using a class statement. To distinguish these types from actual classes, terms such as *classic classes* and *old-style classes* are used to describe `ClassType` objects. This thesis attempts to avoid such confusion by consistently reserving the term *class* for actual `ClassType`

objects, and referring to types by their proper name, even though they may be created using class statements.

Since classes remain part of the Python language purely for backwards-compatibility, and there is no need to use them in new programs, support for sharing of class instances was not a high priority in the design of POSH. POSH generally restricts the kind of objects that can be shared in that their type cannot override the special attribute lookup methods. This restriction also precludes sharing of class instances, since the `InstanceType` type overrides attribute lookup in order to implement inheritance.

2.5 Attributes

Python objects may have associated values called *attributes*, which are accessed using the *dot operator*, expressed with the period sign. To access an attribute named "a" of an object `o`, the syntax is simply `o.a`. As with any other operator in Python, the effect of the *dot operator* is determined at run-time. Hence, attribute lookup is performed at run-time, and not at compile-time, which would be the case for a statically typed language.

2.5.1 Object dictionaries

Objects may provide storage for their attributes by maintaining a regular dictionary that maps the attribute names to their values. This is done by reserving space for a pointer to the dictionary in the object's data structure. A special field in the type object, named `tp_dictoffset`, specifies the offset in bytes of the dictionary pointer, relative to the start of the object's structure. If `tp_dictoffset` is 0, objects of that type have no dictionaries. Object dictionaries can generally be accessed from Python code via the special `__dict__` attribute. For certain kinds of objects (notably types), the `__dict__` is read-only, while other types permit direct modification of the contents of the dictionary. In any case, directly modifying an object's dictionary is seldom an appropriate way of accessing its attributes. In addition, objects without dictionaries may still support attributes, as explained in the following sections.

User-defined types automatically reserve storage for a dictionary in their objects, unless explicit steps are taken to avoid this. (Specifically, this is avoided by assigning an empty list to the special `__slots__` attribute someherere in the body of the class statement.)

2.5.2 Attribute lookup methods

The effect of reading and writing the attributes of an object may be customized by a type by defining certain special methods. A brief description of their semantics is given here.

`__getattribute__` is invoked whenever an attribute is read, and accepts the name of the attribute as an argument. It should simply return the value of the attribute. Implementing this method can be cumbersome, since great care must be taken not to accidentally access an attribute of the object, as that will cause infinite recursion.

`__getattr__` is a less intrusive way of overriding attribute access, which works as a fall-back for the default `__getattribute__` method implemented by the `object` type. Whenever the normal lookup algorithm fails, this method is invoked with the name of the attribute, and should return its value. This can be used, for instance, to implement default attribute values.

`__setattr__` defines the effect of attribute assignment, and is invoked with the name and value of the attribute whenever an attribute is assigned a value.

`__delattr__` defines the effect of deleting attributes (using a *del statement* such as `del o.a`), and accepts the name of the attribute being deleted.

2.5.3 Attribute descriptors

Attribute descriptors allows for considerable flexibility in customizing attributes, without redefining the default attribute lookup methods.

Section 2.5.2 describes the attribute lookup methods that a type can define to customize the attributes of its instances. However, the default attribute lookup methods inherited from the `object` type also allow for considerable flexibility, through the concept of *attribute descriptors*, or *descriptors* for short. A descriptor is an object that controls the access to an individual attribute. It resides in the dictionary of a type object, indexed by the name of the attribute it controls. The descriptor must implement the special `__get__` and `__set__` methods, and optionally the `__delete__` method.[4]

When the default `__getattribute__` method looks for an attribute of an object, it first checks the dictionary of the object's type, looking for a

[4] Do not confuse the `__delete__` and `__del__` methods; the latter is the destructor that's invoked when the object is deleted.

descriptor that is indexed by the attribute name. If no such descriptor is found, the attribute value is sought in the object's dictionary. If a descriptor *is* found, its `__get__` method is invoked, and the return value is used as the value of the attribute. The specific parameters passed to `__get__` are the object on which the attribute lookup is made, as well as the object's type. Note that the attribute may lack both a descriptor and an entry in the object's dictionary. This means that the attribute doesn't exist, and will result in an `AttributeError` exception being raised, which is indeed a very common run-time error.

Similarly, the default `__setattr__` method looks for a descriptor in the dictionary of the object's type, and invokes its `__set__` method, which defines the effect of the attribute assignment. The parameters passed to `__set__` are the object to which the attribute assignment is made, and the new value of the attribute. If no descriptor is found, the value is assigned directly to the object's dictionary.

The default `__delattr__` method invokes the `__delete__` method of the descriptor, which defines the effect of deleting an attribute. If no descriptor is found, the value is deleted from the object's dictionary.

The implications of attribute descriptors are that types can customize the implementation of their attributes to a great degree, without altering the default attribute lookup methods. This can be done simply by placing customized descriptors in their dictionaries. The default behavior for attributes that are not controlled by descriptors, is to store the attribute values in the object's dictionary. Descriptors may also use the object's dictionary for storing values, but may make all kinds of adaptations to the values that are read and written. For example, a custom descriptor can enforce specific access semantics, such as read-only or write-once, or apply constraints with regard to the possible values of an attribute. Many of the special attributes referred to throughout this thesis, such as the `__class__` and `__bases__` attributes of types, described in Section 2.4.1, are implemented by means of special descriptors.

2.5.4 Wrapper descriptors

As described in Section 2.4, type objects contain method slots that define the effect of various generic operations when applied to the type's instances. Built-in types implement these methods by pointing the method slots at appropriate C functions. However, user-defined types implemented in Python may also define most of these methods, by defining methods that have certain special names. For instance, a built-in type can implement a

string conversion method by pointing the `tp_str` field of the type object to an appropriate C function. A user-defined type can achieve the same effect by defining a method with the special `__str__` name. These two ways of implementing the method are equivalent, and mutually exclusive. Regardless of how the method is implemented, there are always two equivalent ways to invoke it.

- Low-level C code can invoke the method by calling the C function pointed to by the method slot.
- High-level Python code can invoke the method just like any other method, knowing its special name.

For instance, calling the `__str__` method of an object is always equivalent to low-level C code calling the function pointed to by `tp_str`. Python uses a special kind of *attribute descriptors*, described in Section 2.5.3, to allow Python code to call a type method implemented in C. The descriptors in question are so-called *wrapper descriptors*, whose purpose is to wrap a low-level C function, making it callable from Python. When a type object is initialized, a wrapper descriptor is added to its dictionary, indexed by the appropriate attribute name, for each of its non-zero method slots. The `__get__` method of wrapper descriptors return callable objects that invoke the low-level C function when called.

Wrapper descriptors ensure that all the methods defined by the method slots of a built-in type, remain callable from Python code as well. However, the inverse problem still exists, when low-level C code wants to invoke a special method defined by a user-defined type implemented in Python. In this case, the actual implementation of the method is found in a Python function, not in a low-level C function. This is solved by assigning special-purpose *dispatcher functions* to the method slots for which Python implementations are available. A dispatcher function has the correct signature for its method slot, and invokes the Python implementation using the generic Python/C API. By keeping the wrapper descriptors and method slots consistent with each other, as described in Section 2.5.5, Python ensures that the two ways of invoking a type method, outlined above, remain equivalent.

2.5.5 Attribute lookup on types

The `type` meta-type overrides the attribute lookup methods described in Section 2.5.2, customizing the attribute lookup algorithm used for types. This is required to implement inheritance, as outlined in Section 2.4.1.

```
>>> class Descriptor(object):
...         def __get__(self, *args):
...             print args
...
>>> class T(object):
...         attr = Descriptor()
...
>>> T.attr
(None, <type '__main__.T'>)
>>> T().attr
(<__main__.T object>, <type '__main__.T'>)
```

Figure 2.11: Interpreter session illustrating the nature of descriptors.

Section 2.4.1 explains how Python supports multiple inheritance, which means that a type may subtype multiple base types. The actual mechanism that implements inheritance are the special attribute lookup rules for type objects. Every type has an inheritance graph similar to the one in figure 2.6. In a statically typed language, the compiler traverses the inheritance graph of a type to resolve attribute accesses such as method calls or field references. Python essentially does the same, except that attributes are resolved at run-time. The `type` meta-type implements a specialized attribute lookup algorithm for type objects, by overriding the special `__getattribute__` method described in Section 2.5.2. The algorithm traverses the type's inheritance graph, examining each type in turn. When a type is encountered that has a value for the given attribute, the search terminates. The order in which the inheritance graph is traversed is called the *method resolution order*[5], and is calculated when the type object is created. It is stored as a list of types in the special `__mro__` attribute of the type.

The algorithm used for calculating the MRO is the C3 algorithm described in [1], and is quite complicated, since it tries to generate an optimal order for potentially complex inheritance graphs. The important point, however, is that a subtype always precedes its base types in the MRO, which means that its attribute values override those of the base types.

Aside from the fact that an attribute lookup on a type will traverse the type's inheritance graph attempting to find a value for the attribute, there are differences in how types make use of the descriptors found in their dictionaries. The default lookup algorithm described in Section 2.5.3 examines the dictionary of the object's type. Similarly, an attribute lookup

[5] Since the order is used for resolution of attributes in general, rather than methods in particular, a more accurate term would be the *attribute resolution order*.

on a type object will examine the dictionary of the *meta-type*, looking for a descriptor. However, in the event that no descriptor for the attribute is found in the meta-type, type objects do not unconditionally return the value found in their dictionary. If the value found in the type's dictionary is a descriptor, the value of the attribute is determined by calling the descriptor's `__get__` method. Consequently, the `__get__` method of a descriptor gets called in response to attribute lookups on both the type itself and its instances. The two cases are distinguished by the parameters passed to the descriptor.

- As described in Section 2.5.3, when an attribute lookup is made on the type's instance, the parameters to the `__get__` method are the instance in question, as well as the type.
- When an attribute lookup is made on the type itself, the first parameter to the `__get__` method is `None`, and the second is the type. In other words, the instance in question is `None` in this case.

Figure 2.11 shows an example interpreter session that might help clarify the semantics of descriptors. The session defines a trivial descriptor type, whose `__get__` method simply prints its arguments to the console. The definition of the `T` type assigns a descriptor object to an attribute, which amounts to placing the descriptor in its dictionary. The output shows the parameters passed to `__get__` as a result of an attribute lookup on the type, and its instance, respectively.

The ability to distinguish between attribute lookups on a type and its instances, using the same descriptor, is crucial to the implementation of methods. When the methods of a type are defined in the body of a class statement, they are initially regular function objects. However, the `type` meta-type wraps them in special *method descriptors* that implement the semantics of *bound* and *unbound methods*. A regular method call on an object actually consist of two separate operations. Firstly, the name of the method is looked up as an attribute of the object. Secondly, the resulting attribute value is called. Method descriptors implement such method calls by returning a *bound method* from the attribute lookup. This means that the method is bound to an object, and calling it will implicitly pass that object as the first parameter to the method, by convention named `self`. *Unbound methods*, on the other hand, are not associated with a particular object, and require the `self` parameter to be specified explicitly when called. Unbound method calls are commonly used to invoke a base type's implementation of a method.

Python also uses descriptors to implement so-called `static methods`. The `__get__` method of these descriptors always ignore their first parameter, i.e.

the instance to which the attribute lookup applies. No `self` parameter is passed, regardless of whether the method was invoked on an instance or on the type itself. The effect is that static methods never apply to a specific instance, but rather to the type as a whole, similarly to how static methods behave in languages like C++ and Java.

Although the most drastic changes made to the attribute lookup rules for types are in the context of reading types, some subtle changes have been made to the semantics of attribute assignment, too. This involves the *wrapper descriptors* and *dispatcher functions* described in Section 2.5.4. If a function is assigned to one of the special attributes that correspond to a method slot in the type object, an appropriate dispatcher function is automatically installed in the relevant method slot. This allows Python code to assign the special methods of a type dynamically, while the implementations become available to low-level C code as well. This does not really change the semantics of attribute assignment to types, it is more of an implementation detail to maintain the consistency between the wrapper descriptors and the dispatcher functions.

2.6 Garbage collection

This section provides a brief description of how Python approaches the problem of *garbage collection*[4, 2].

2.6.1 Python reference counting

Starting out as a high-level scripting language, the design of Python included garbage collection as a fundamental premise from the very start. The garbage collection algorithm chosen was *reference counting*, for its simplicity and painless portability. As described in Section 2.3.1, python maintains a reference count for each object, accessible as the `ob_refcnt` field of the `PyObject` structure. This reference count is updated using the `Py_INCREF` and `Py_DECREF` macros, which increment and decrement the count, respectively. Whenever a new reference to an object is introduced, the reference count is incremented, and when a reference is deleted or assigned a new value, the reference count of the object it referred to is decremented. When the reference count reaches 0, the object is deleted. The scheme allows for a simple implementation, although it does put a load on programmers of built-in modules, who must diligently apply the `Py_INCREF` and `Py_DECREF` macros at the correct places. If the reference count of an object is decremented by mistake, the end result will eventually be a core

dump, since the object will be deallocated prematurely. However, a spurious `Py_INCREF` will only lead to memory leaks, which can be harder to detect. As such, correctly updating the reference counts is definitely the most difficult part of programming built-in modules.

2.6.2 Cyclic garbage collection

Although Python code can usually safely rely on garbage collection to dispose of their objects, barring programming errors in built-in modules, reference counting has an inherent disadvantage concerning cyclic references. This occurs when an object directly or indirectly contains a reference to itself, for instance when the first item of a list is the list itself. Even though such a cycle of objects may be unreachable from any other objects, they will never be deallocated, since all of them have nonzero reference counts. Python recently introduced the concept of *cyclic garbage collection*, which attempts to detect such cycles and deallocate them, if possible. If the objects involved in a cycle have custom destructors (`__del__` methods, in other words), there is no well-defined order in which to destruct them. Such a cycle is considered a programming error, and a warning will be issued if it is detected.

Chapter 3

Design and implementation

3.1 Architecture

This section gives a high-level introduction to the design and implementation of POSH, and defines much of the terminology that is used throughout this chapter.

The objective of POSH is to allow regular Python objects to reside in shared memory, where they can be made accessible to multiple processes. Objects allocated in shared memory are referred to as *shared objects*. For convenience, the types of shared objects are referred to as *shared types*, but this does not imply that the *type objects* are allocated in shared memory. Section 3.5.1 explains why type objects cannot reside in shared memory. However, since type objects are essentially read-only data structures, they can easily be made accessible to all processes by creating them prior to any `fork` calls. Section 3.5.1 discusses the implications of using the `fork` system call for process creation.

The process of creating a shared object that is a copy of a given non-shared object, is called *sharing* the object. Objects that are eligible for sharing are called *shareable objects*, and their types are referred to as *shareable types*. POSH maintains a mapping that specifies which shareable types correspond to which shared types. This is the *type map*, described in Section 3.3.1. When an object is shared, its type (which is a shareable type) is mapped to the corresponding shared type, and an instance of the shared type is created, which copies its initial value from the object being shared.

The basic strategy for implementing a shared type is to subtype its equivalent shareable type, overriding its allocation methods. This allows the shared

```
typedef struct {
  SharedRegionHandle my_handle;

  struct { /* ... */ } proctable;
  struct { /* ... */ } regtable;
  struct { /* ... */ } sleeptable;
} Globals;
```

Figure 3.1: Structure of the globally shared data.

type to inherit the existing implementation details of the shareable type, while overriding the crucial aspect of object allocation. The exceptions to this rule are the shared container types, which are reimplemented from scratch by POSH, using the concept of *memory handles* for storing references to other shared objects.

POSH wraps all shared objects in special *proxy objects*, that shield the shared objects from direct access.

3.2 Shared memory management

A basic service required by POSH is the ability to create shared memory regions and attach them to the process' address space. Shared memory is used by POSH to maintain global data structures accessible to all processes. Furthermore, fine-grained management of shared memory is required to manage the relatively small memory chunks needed for shared objects' main and auxilliary data structures. This section describes how POSH manages shared memory.

3.2.1 Globally shared data

POSH uses shared memory to implement global control and globally shared data.

Global control is a requirement in POSH. For example, garbage collection of shared objects, as described in section 3.7, requires the cooperation and coordination of all processes to determine when an object should be reclaimed. To implement global control, some form of inter-process communication is required. The IPC mechanism used by POSH for this purpose is shared memory. The data structures needed for the communication are allocated

in a shared memory region. These data structures are referred to as *globally shared data*, since they are accessible to all processes.

There is a single shared memory region that contains all the globally shared data. This memory region is created and attached at module initialization time, when the module is first imported. The interface described in Section 3.2.2 is used for this purpose. The attached region is accessed through a pointer named `globals`. Since all processes using POSH are descendants of the process that first imported the module, the shared memory region is attached at the same address in all processes, and the `globals` pointer is valid for all processes. The structure of the globally shared data to which `globals` points is shown in Figure 3.1. The `my_handle` member is a handle for the shared memory region in which the globally shared data is allocated. The handle is needed to remove the region as part of the cleanup performed by the last process to exit, as described in Section 3.10.3. A high-level description of the remaining members is given here — the details are provided in the relevant sections.

proctable This is the *process table*, which is used to assign each process a numeric ID in the range 0–`MAX_PROCESSES`. This makes it easy to implement a set of process IDs as a bitmap, as described in Section 3.10.1.

regtable This is the *region table*, which assigns a numeric ID in the range 0–`MAX_REGIONS` to each shared memory region. The region table is used for global cleanup purposes and by the implementation of *memory handles* described in section 3.5.1.

sleeptable This is the *sleep table*, which contains one semaphore per process. It is used by the implementation of *shared locks* described in section 3.9, to allow processes to sleep while waiting for an event.

When shared memory is used for IPC, it must be complemented with a synchronization mechanism. The primitives used for synchronizing the inter-process communication in POSH are *shared locks*, which are described in section 3.9.

3.2.2 Management of shared memory regions

POSH defines a low-level and a higher-level interface for management of shared memory regions, which is intended to improve the portability of the code.

```
typedef /*...Platform-specific...*/ SharedRegionHandle;

SharedRegionHandle _SharedRegion_New(size_t* size);
void _SharedRegion_Destroy(SharedRegionHandle h);
void* _SharedRegion_Attach(SharedRegionHandle h);
int _SharedRegion_Detach(void* addr);
```

Figure 3.2: Low-level interface for management of shared memory regions.

```
typedef struct {
    struct {
        Lock lock;
        int count;
        int freepos;
        SharedRegionHandle regh[MAX_REGIONS];
        size_t regsize[MAX_REGIONS];
    } regtable;

    /* Other members are omitted... */
} Globals;
```

Figure 3.3: Definition of the region table.

Operating systems implement support for shared memory in several different ways. Consequently, their interfaces for management of shared memory regions vary, which is clearly a portability issue for applications using shared memory. As a general strategy, POSH attempts to confine the portions of the code that are non-portable to specific modules with well-defined interfaces, which makes it feasible to completely replace a module's underlying implementation, as long as the same interface is supported.

One of the interfaces defined by POSH is used for the low-level management of shared memory regions, shown in figure 3.2. As a part of the interface, the `SharedRegionHandle` typedef is defined, which is used to uniquely identify a shared memory region. A description of the functions in the interface follows.

`_SharedRegion_New` creates a new shared memory region. The `size` parameter to the `_SharedRegion_New` function is an in/out parameter. On entry, it is the requested size of the new memory region in bytes — on return, it is the actual size of the new region, which may be larger than requested. This will commonly be the case if a size is requested that is not a multiple of the operating system's page size, since operating systems generally implement shared memory by modifying

```
SharedMemHandle SharedRegion_New(size_t *size);
void SharedRegion_Destroy(SharedMemHandle h);
```

Figure 3.4: Higher-level interface for management of shared memory regions.

virtual memory mappings, in which the granularity is restricted to a page. The function returns a handle that uniquely identifies the shared memory region. On failure, the function returns the special `SharedRegionHandle_NULL` value, which can be redefined as a part of the interface.

`_SharedRegion_Destroy` accepts a handle for a shared memory region and destroys it, releasing the operating system resources it consumes.

`_SharedRegion_Attach` accepts a handle for a shared memory region, and attaches the region to the process' address space, returning the address at which it is attached. Note that the same shared memory region may be attached at different virtual addresses in different processes. This has some major implications on how to store recursive data structures in shared memory, as described in Section 3.5.1. On failure, this function returns `NULL`.

`_SharedRegion_Detach` detaches a shared memory region from the address space of the process, undoing the virtual memory mappings established by `_SharedRegion_Attach`. A shared memory region that has been attached must be detached before it is destroyed. This function returns 0 on success, and -1 on failure, in keeping with the general convention in Python.

The current implementation of this interface relies on the System V interface, which offers the abstraction of shared memory regions that persist across process lifetimes. Consequently, they need to be explicitly destroyed. Other (future) implementations may not have this requirement, in which case they could simply implement `_SharedRegion_Destroy` as a null operation, doing nothing. Similarly, some implementations might be able to implement `_SharedRegion_Detach` as a null operation, since explicitly detaching the regions might not be necessary.

As briefly described in Section 3.2.1, a part of the globally shared data is the *region table*, whose definition is shown in Figure 3.3. The region table maintains information about all shared memory regions in existence, with the exception of the region holding the globally shared data itself. A description of the region table's fields follows.

`lock` is a *shared lock*, as described in Section 3.9.3, which is used to synchronize access to the region table.

`count` is the number of shared memory regions registered in the table.

`freepos` is a search finger used to speed up searches for free entries in the `regh` and `regsize` arrays. It holds an index thought to be free, but makes no strong guarantees. Consequently, accessing this field alone can be done without synchronization.

`regh` is an array that contains the handles of all existing shared memory regions. The special `SharedRegionHandle_NULL` value is used to indicate free entries.

`regsize` is an array parallel to `regh`, holding the size of each shared memory region.

Whenever a shared memory region is created, its handle and size should be assigned at a free index in the `regh` and `regsize` arrays. Conversely, the corresponding entries should be freed when a shared memory region is destroyed. The purpose of maintaining this global state is two-fold:

- Firstly, it enables the last terminating process to remove shared memory regions that haven't been explicitly removed. This form of cleanup is important, since the current implementation relies on the System V interface[3], which allows shared memory regions to persist across process lifespans. Thus, some cleanup strategy is required to avoid left-behind memory regions that persist only to waste system resources.
- Secondly, the region table maps the handles for shared memory regions to the set of integers in the range 0–`MAX_REGIONS`. This facilitates the implementation of constant-time mapping from *memory handles* to pointers, as desribed in section 3.5.1.

Since the region table must be updated whenever a shared memory region is created or destroyed, there is a higher-level interface that manages these tasks, shown in Figure 3.4. `SharedRegion_New` creates a new shared memory region using `_SharedRegion_New`, and finds a free index for it in the region table. It returns a *memory handle*, described in Section 3.5.1, for the region's starting address. The `size` parameter is an in/out parameter, with the same behaviour as in `_SharedRegion_New`. `SharedRegion_Destroy` frees the region's entries in the region table, and destroys it using `_SharedRegion_Destroy`.

```
h.alloc(size) -> address, size
h.realloc(address, size) -> address, size
h.free(address) -> None
```

Figure 3.5: Interface supported by a heap object `h`.

```
PyObject* Address_FromVoidPtr(void* ptr);
void* Address_AsVoidPtr(PyObject* obj);
```

Figure 3.6: Interface for creating and accessing `Address` objects.

The higher-level interface is implemented entirely using the lower-level interface (prefixed by underscores), so only the latter needs to be reimplemented in order to use a different underlying implementation of shared memory.

3.2.3 Fine-grained shared memory management

POSH implements fine-grained management of shared memory by delegating the task to special *heap objects* that support a well-defined interface.

Section 3.2.2 describes the interface POSH provides for creating relatively large regions of shared memory. However, placing objects in shared memory requires more fine-grained memory management. POSH delegates the task of fine-grained management of shared memory to special *heap objects* that are associated with shared types. Consequently, the low-level C code implemented by the heap objects can be easily extended by Python code through subtyping or delegation. This is a great advantage when higher-level facilities such as logging are to be added to the memory management. Figure 3.5 shows the interface that heap objects are required to implement. A brief description of the methods follows.

`alloc` accepts a `size` argument, which specifies the number of bytes to allocate on the heap. The method returns two values (as a tuple); the address of the allocated chunk of memory, and the actual size of the chunk, which may be larger than requested. Errors are indicated by raising exceptions.

`realloc` accepts the address of an existing memory chunk, and reallocates the chunk to make room for the requested number of bytes. The (possibly) new address of the chunk is returned, along with its actual size. Exceptions are raised in case of errors.

```
void* SharedAlloc(PyObject* self, size_t* size);
void* SharedRealloc(PyObject* self, void* ptr, size_t* size);
void SharedFree(PyObject* self, void* ptr);
```

Figure 3.7: Low-level interface used by shared objects to allocate memory from their type's data heap.

`free` accepts the address of an existing memory chunk, and releases it to the heap's pool of memory. It returns None on success, and raises an exception on failure.

The `size` arguments and return values are passed as integer objects, and the addresses are passed as objects of type `Address`. The `Address` type is is a simple built-in type provided by POSH, that essentially wraps a C variable of type `void*`. POSH also defines an interface for easy manipulation of `Address` objects, listed in Figure 3.6.

Each shared type has two special attributes that refer to heap objects.

`__instanceheap__` refers to the *instance heap* of the type, on which the instances of the shared type are allocated. Specifically, the heap contains the object data structures described in Section 2.3. Being shared objects, their data structures also include the `SharedObject` header described in Section 3.4.1.

`__dataheap__` refers to the *data heap* of the type, which is used for the auxiliary data structures of the type's instances, such as the vectors maintained by the shared lists and tuples described in Section 3.5.2.

In order to use a heap object for memory allocation, low-level C code needs to create the `int` and/or `Address` objects needed as parameters, call the appropriate method using one of the generic functions defined by the Python/C API, and unwrap the return values to store them in regular C variables. This is essentially an application of the general technique of *data marshalling*, and does incur a small performance penalty compared to a pure low-level C interface. However, at the time the design choice was made, the advantages gained in extensibility were considered to outweigh the drawback of a small performance penalty.

As a concrete example that might serve to justify the design choice, the verbose mode of POSH wraps all heap objects in special `VerboseHeap` objects. These *verbose heaps* implement the heap interface by delegating

```
typedef struct
{
  SharedMemHandle head[NOF_ALLOC_SIZES];
  Lock lock[NOF_ALLOC_SIZES];
} root_t;

typedef struct
{
  PyObject_HEAD
  root_t *root;
} SharedHeapObject;
```

Figure 3.8: Definition of the object structure for `SharedHeap` objects, and the root data structure.

all calls to the wrapped heap object, while printing verbose debugging information to the console. `VerboseHeap` is a user-defined type implemented entirely in Python, that provides very valuable debugging information, without touching the low-level C code that implements the actual heap object. As a consequence, the development time needed to implement the feature was vastly shortened, and no recompilation of the C code is required to activate or deactive the verbose mode. An example of the output produced by the verbose heaps can be seen in Figure 3.31.

For convenience, POSH defines a low-level C interface, shown in Figure 3.7, which is designed for shared types implemented in C. The interface allows a shared object to allocate memory from the data heap of its type, without dealing with the data marshaling described above. It resembles the well-known memory management interface provided by the standard C library (`malloc`, `realloc` and `free`), but makes some adaptations. Firstly, each function accepts a `self` parameter, which is the object requesting the allocation. The heap object used for the allocation is found by looking up the `__dataheap__` attribute of `self`'s type. Secondly, `size` is an in/out parameter, which contains the actual size of the allocated chunk when the function returns. Since the actual size of the chunk may be larger than requested, making this information available to the caller may allow it to optimize memory utilization. For instance, shared lists and dictionaries, described in Section 3.5, make use of this information when resizing their auxiliary data structures.

```
typedef unsigned int word_t;

typedef struct
{
  SharedMemHandle next;
  Lock lock;
  word_t nof_units;
  word_t nof_free_units;
  word_t unit_size;
  word_t unit_mask;
  word_t head;
  unsigned char data[1];
} page_t;
```

Figure 3.9: Definition of the page data structure used by `SharedHeap` objects.

3.2.4 The `SharedHeap` type

POSH provides a built-in type that implements the heap interface described in Section 3.2.3 in a relatively straightforward way.

As described in Section 3.2.3, fine-grained implementation of shared memory is delegated to special heap objects. POSH implements a built-in type named `SharedHeap` that supports the required interface. It is intended as a minimal implementation, designed to be simple, yet reasonably efficient. As explained in Section 3.2.3, the heap objets used for shared memory allocation are determined by special attributes of the shared types. Thus, modifying POSH to use an alternate implementation of shared heaps is extremely easy, and doesn't even require a recompilation of the C code.

The `SharedHeap` type uses a simple algorithm with fixed-size allocation units that increase exponentially in size. For each unit size, a linked list of *pages* is maintained. Each page is a shared memory region that holds a linked list of allocation units of a given size. New pages are created and linked into their respective lists as needed.

As shown in Figure 3.8, the object structure of `SharedHeap` objects only contains a single pointer (in addition to the mandatory fields defined by `PyObject`). The pointer points to a *root data structure*, defined by the C structure `root_t`, which is allocated in a shared memory region. The root data structure contains the actual implementation of the heap. This technique in effect makes the `SharedHeap` object shared, since a fork will only

copy the object, which simply contains a memory handle, and not the actual implementation, which is located in shared memory. The `SharedHeap` type could have been implemented by starting out with a `NonSharedHeap` and using the regular object sharing mechanism to allow sharing. However, this would require it to implement copy semantics, as described in Section 3.3.2, introducing unnecessary complications, since only shared instances would be created anyway.

Figure 3.9 shows the definition of a page. Each page is a node in a linked list of pages, and contains a linked list of free allocation units. A description of the individual fields in a page follows.

`next` is a memory handle that refers to the next page in the linked list of pages. The `SharedMemHandle_NULL` value is used to terminate the list.

`lock` is a *shared lock*, described in Section 3.9.3, used for synchronizing access to this individual page. The lock protects the consistency of the linked list of free units.

`nof_units` is the total number of allocation units contained in the page.

`nof_free_units` is the current number of free allocation units in the page — equal to the length of the linked list of free units.

`unit_size` is the size of an allocation unit in this page. All the allocation units of a page are of the same size.

`unit_mask` equals the allocation unit size minus one. This value can be used as a bitmask to align a byte offset to the boundary of two allocation units (by performing a logical *and*).

`head` is the byte offset, from the start of the page, of the first free allocation unit. The first word of each free allocation unit contains the byte offset of the next, effectively implementing a linked list. An offset of 0 is used to terminate the list.

`data` marks the start of the data area of the page, where the allocation units are actually stored. The allocation units are aligned according to their size, meaning that there might be a gap of unused memory preceding the very first allocation unit.

Note that the linked lists of pages use memory handles to implement the "links", while the linked lists of allocation units use the byte offsets from the start of the page for this purpose. Since they are located in shared memory, neither of them uses pointers, for the reasons explained in Section 3.5.1.

`SharedHeap` implements the `alloc` method by first selecting the smallest allocation unit size that is large enough to fulfill the request. Subsequently, the corresponding list of pages is traversed until a non-empty page is found. If all the pages are full, a new page is created and linked into the list. Finally, one allocation unit is unlinked from the free list of the page, and the address of the unit is returned. The actual size of the allocation unit is returned as well, allowing the caller to make full use of the allocated memory block. If the requested size is larger than the maximum size of an allocation unit, an entire shared memory region is created to satisfy the request.

The `free` method releases an allocated unit by first determining what page it is allocated in. This is done by performing a reverse mapping from the address to a memory handle, using the `SharedMemHandle_FromVoidPtr` function described in Section 3.5.1. The memory handle contains the offset of the address from the start of the shared memory region. By subtracting this offset from the address, the start of the page is found. Since a typical implementation would have stored the allocation unit's offset as a separate word in the unit, this approach in effect saves a word in every allocated unit. Once the start of page is found, the allocation unit can be linked into the free list of the page, effectively releasing the memory.

3.3 Shareable and shared types

This section describes the requirements imposed on shareable types, and the mechanisms used to create the equivalent shared types.

3.3.1 Declaration of shareable types

POSH maintains a so-called *type map* that maps all shareable types to their equivalent shared types and proxy types. Shareable types are registered with POSH using the `allow_sharing` function described in Section 3.11.1. This triggers the creation of a corresponding shared type, using the `SharedType` meta-type described in Section 3.3.3. Also, a new custom-tailored proxy type is created, as described in Section 3.6.1. The relationship between the shareable, shared and proxy types is stored in the proxy map, which is a dictionary that maps the shareable type to a tuple containing the other two.

3.3.2 Creation of shared objects

The process of sharing an object requires the allocation and initialization of a shared object. As explained in Section 2.2.3, all values in Python are objects, so the user needs to specify the initial value of the shared object by passing an object. Consequently, this operation essentially amounts to copying an existing object, placing the copy in shared memory.

A shareable type is required to support *copy semantics*, meaning that it can be used to create copies of existing instances. The convention is that a type, when called with an existing instance of the type as the only parameter, returns a copy of the instance. This is the equivalent of providing a *copy constructor* in the C++ language. For instance, `int(2)` returns a copy of the integer 2, and `tuple((1,2))` returns a copy of the given tuple. If the user wants to share instances of a user-defined type, the type should implement an `__init__` method that follows the same convention.

POSH creates shared objects through the following sequence of steps.

- Firstly, the type of the object to be shared is examined, and its corresponding shared type is retrieved from the type map described in Section 3.3.1. If the type map contains no mapping for the type, the object in question is not eligible for sharing, and an exception is raised.
- Secondly, a new instance of the shared type is created, by calling the shared type, passing the object to be shared as the only argument. Since the shared type inherits the `__init__` method of its shareable base type, copy semantics will ensure that the new shared object is equal to the original shareable one. However, since the shared type overrides the allocation methods of its base type, the shared object is allocated in shared memory.
- Finally, the shared object is wrapped in a proxy object before returned to the caller.

POSHalso defines a built-in function named `ShareObject` that shares an object without wrapping the result in a proxy object. This function is for internal use, to enable shared containers to share the items that are assigned to them, for immediate storage in the form of memory handles.

```
class SharedType(type):
    __instanceheap__ = _core.SharedHeap()
    __dataheap__ = _core.SharedHeap()

    if VERBOSE:
        __instanceheap__ = VerboseHeap("Instance heap", __instanceheap__)
        __dataheap__ = VerboseHeap("Data heap", __dataheap__)

    __synch__ = MONITOR

    def __new__(tp, name, bases, dct):
        def wrap_built_in_func(func):
            return lambda *args, **kwargs: func(*args, **kwargs)

        if len(bases) > 1:
            raise ValueError, "this meta-type only supports single inheritance"

        dct["__getattribute__"] = wrap_built_in_func(_core.shared_getattribute)
        dct["__setattr__"] = wrap_built_in_func(_core.shared_setattr)
        dct["__delattr__"] = wrap_built_in_func(_core.shared_delattr)

        newtype = type.__new__(tp, name, bases, dct)

        _core.override_allocation(newtype)
        return newtype
```

Figure 3.10: Implementation of the `SharedType` meta-type.

3.3.3 The `SharedType` meta-type

As noted in Section 3.1, the general strategy for creating shared types is to subtype the corresponding shareable types, overriding their allocation methods and attribute lookup methods. In addition, shared types should have certain attributes that control the behaviour of their instances. As explained in Section 3.2.3, the `__instanceheap__` and `__dataheap__` attributes refer to heap objects that control the allocation of the shared type's instances and their auxiliary data structures. Furthermore, the `__synch__` attribute defines the synchronization policy used for the shared objects, as described in Section 3.8.1.

Section 2.4.4 describes how meta-types implement the creation of new types.

The implementation of `SharedType`, stripped of docstrings and comments,

is short enough to fit in Figure 3.10. The type inherits from the standard `type` meta-type, only extending the special `__new__` method. As described in Section 2.4.4, the effect of the class statement is a call to the specified meta-type. The effect of the call is defined by the `type` meta-type's special `__call__` method, which `SharedType` inherits. The `__call__` method in turn invokes the `__new__` method defined by `SharedType`. This method is mainly implemented by invoking `type`'s implementation, which creates a new type. In addition, three adaptations to the new type are made.

- Single inheritance is enforced, meaning that the new type can only have one base type. This restriction is not really necessary, but as the intended usage of the meta-type only involves single inheritance, it was added as a safeguard against programming errors.
- The attribute lookup methods of the new type are overridden. This is done by modifying the dictionary before passing it to `type`'s implementation. The allocation methods, which are described in detail in Section 3.4.2, are implemented as built-in functions. This requires them to be "wrapped" in Python functions to achieve the correct binding behaviour. This workaround is needed due to an apparent shortcoming in the Python language, that will hopefully be addressed in future versions. The crux of the matter is that Python functions implement the `__get__` method described in Section 2.5.3, allowing them to function as attribute descriptors, whereas built-in functions, for no apparent reason, lack this method. This is arguably a small matter, but since the wrapping incurs a performance penalty, not a trivial one.
- The allocation methods of the new type are overridden. This is done using a special purpose built-in function, as described in Section 3.4.2.

The `SharedType` meta-type also assigns values to its special `__instanceheap__`, `__dataheap__` and `__synch__` attributes. Since the attributes of a meta-type serve as default values for the attributes of its instances (which are types), these are the default values used for shared types that refrain from specifying their own values. None of the shared types defined by POSH define the special heap attributes, which means that they all use the heaps specified by `SharedType`. However, specifying a different synchronization policy is common. Since the default policy enforces monitor semantics, all immutable shared types defined by POSH assign `None` to the `__synch__` attribute, specifying that no synchronization is required.

```
typedef struct {
  Lock lock;
  SharedMemHandle dicth;
  Spinlock reflock;
  ProcessBitmap proxybmp;
  unsigned int srefcnt : sizeof(int)*8 - 2;
  unsigned int is_corrupt : 1;
  unsigned int no_synch : 1;
  /* Start of normal PyObject structure */
  PyObject pyobj;                                                    10
} __attribute__((packed)) SharedObject;

#define SharedObject_FROM_PYOBJECT(ob) \
  ((SharedObject*) (((void*) (ob)) - offsetof(SharedObject, pyobj)))

#define SharedObject_AS_PYOBJECT(shob) \
  (&(((SharedObject*) (shob))->pyobj))

#define SharedObject_AS_PYVAROBJECT(shob) \
  ((PyVarObject*) &(((SharedObject*) (shob))->pyobj))                20

#define SharedObject_VAR_SIZE(type, nitems) \
  (_PyObject_VAR_SIZE(type, nitems) + offsetof(SharedObject, pyobj))

void SharedObject_Init(SharedObject* obj, PyTypeObject* type, int nitems);
```

Figure 3.11: Definition of the `SharedObject` structure, and related macros and functions.

3.4 Shared objects

This section describes the implementation of shared objects, which are objects located in shared memory. Section 3.4.1 describes how shared objects are represented in memory. Section 3.4.2 explains how shared objects are allocated in shared memory. Section 3.4.3 describes how attributes are implemented for shared objects.

3.4.1 Representation of shared objects

POSH associates additional data with each shared object by prepending it to the object's data structure.

Conceptually, all shared types can be thought of as subtypes of a common

`SharedObject` base type, that contains all the low-level implementation details on which shared objects rely. However, this conceptual relationship can not be directly reflected in the actual implementation. The reason is that Python disallows multiple inheritance involving more than one built-in base type, as explained in Section 2.4.1. However, shared types are generally subtypes of their shareable counterparts, as described in Section 3.1. Consequently, many of them already have a built-in base type, which prevents them from subtyping another built-in type such as `SharedObject` as well.

The fact remains that shared objects have many implementation details in common, and somehow additional data structures must be associated with each shared object. Section 2.4.2 explains how subtypes extend the object structures of their base types by appending their data to the end. For shared objects, the technique of appending data to the object's structure is infeasible, since the size and lay-out of the object structures vary from object to object. The solution chosen by POSH is instead to *prepend* the additional data to the shared object's structure, adding a common header to the representation of all shared objects. This is possible since the allocation of shared objects needs to be overridden in any case. Thus, reserving space for a header when the object is allocated is straightforward. Figure 3.11 shows the definition of the `SharedObject` header. A brief description of its fields is given here.

`lock` is a *shared lock*, as described in section 3.9.3, which is used for protecting access to the shared object when implicit synchronization is enabled.

`dicth` is a *memory handle*, as described in Section 3.5.1, that refers to a shared dictionary. The shared dictionary is used to implement attribute lookup on shared objects, as described in Section 3.4.3.

`reflock` is a *spin lock*, as described in Section 3.9.1, which is used to synchronize the access to the `proxybmp` and `srefcnt` fields.

`proxybmp` **and** `srefcnt` are used by the implementation of garbage collection for shared objects, described in section 3.7. The `srefcnt` lends a few of its bits to the flag fields as a space optimization.

`is_corrupt` is a flag used to signify that the shared object may be corrupted. This can occur if a process terminates abnormally while accessing the object, as described in section 3.10.4.

`no_synch` is a flag used to optimize the synchronization protocol described in section 3.8.1 in the case where implicit synchronization is disabled.

```
PyObject* SharedObject_Alloc(PyTypeObject* type, int nitems);
void SharedObject_Free(PyObject* obj);
```

Figure 3.12: Allocation functions for the `tp_alloc` and `tp_free` method slots of shared types.

`pyobj` marks the start of the shared object's regular object structure. What data actually follows at this point is dependent on the actual type of the shared object.

The approach of prepending a header to shared objects provides a non-intrusive way to effectively extend their object structure, regardless of its original length. The technique is non-intrusive, since shared objects retain the capability of acting as regular Python objects, while code that is aware of their shared nature can make use of the data structures in their headers.

Given a pointer to a `SharedObject`, it can be viewed as a regular `PyObject` by taking the address of its `pyobj` member. Conversely, if an object pointed to by a `PyObject` pointer is known to be shared, it can be viewed as a shared object by subtracting the proper number of bytes from the pointer and casting it to a `SharedObject` pointer. For convenience, these conversions are performed using the `SharedObject_AS_PYOBJECT` and `SharedObject_FROM_PYOBJECT` macros, listed in Figure 3.11. The figure also shows the definiton of the `SharedObject_Init` function, which is used to initialize the `SharedObject` header as well as the object structure of a shared object. The `SharedObject_VAR_SIZE` macro calculates the number of bytes needed for a shared object, including the header, and is used by the allocation functions described in Section 3.4.2. The macro makes sure that extra space is reserved to accomodate the shared objects' headers.

3.4.2 Overriding allocation of shared objects

POSH implements allocation functions for shared objects that delegate the allocation to the *heap object* bound to the special `__instanceheap__` attribute of the object's type.

As noted in Section 2.4, Python type objects contain method slots that define the behaviour of their instances, including their allocation and deallocation. The specific method slots that control allocation and deallocation are named `tp_alloc` and `tp_free`. POSH implements two functions, listed in Figure 3.12, that have the proper signature for these method slots.

`SharedObject_Alloc` allocates a new instance of the given type. The `nitems` parameter is 0 for fixed-size objects, and denotes the initial value of the `ob_size` field in variable-size objects. The function calculates the number of bytes needed for the object, using the `SharedObject_VAR_SIZE` macro described in Section 3.4.1. It proceeds by looking up the `__instanceheap__` attribute of the type, delegating the allocation to the heap object, just like the `SharedAlloc` function described in Section 3.2.3. Finally, the shared object is initialized by calling the `SharedObject_Init` function described in Section 3.4.1.

`SharedObject_Free` frees the memory occupied by an object, by delegating the call to the instance heap associated with the object's type, just like the `SharedFree` function described in Section 3.2.3.

Unlike most of the other method slots contained in a type object, the `tp_alloc` and `tp_free` methods can not be implemented by Python code. The only way to override the methods is to explicitly point the method slots at appropriate functions, a task that must be performed by C code. Therefore, POSH implements a small built-in function named `override_allocation`, which installs the `SharedObject_Alloc` and `SharedObject_Free` functions in the proper method slots of a type object. This function is invoked by the `SharedType` meta-type on creation of a shared type, as described in Section 3.3.3.

3.4.3 Attribute lookup for shared objects

POSH supports attributes for shared objects by associating a memory handle for a shared dictionary with each shared object.

As described in Section 2.5.1, a dictionary may be associated with every Python object, to provide storage for the object's attributes. However, the attributes of a shared object cannot be stored in a regular dictionary; for this purpose, a shared dictionary is required. Hence, POSH associates a memory handle that refers to a shared dictionary with every shared object. This is the `dicth` field of the `SharedObject` header described in Section 3.4.1.

When shared types are created by the `SharedType` meta-type described in Section 3.3.3, the attribute lookup methods described in Section 2.5.2 are overridden. The alternative implementation provided by POSH uses the exact same algorithm described in Section 2.5.3, except that the object's shared dictionary is used instead of the regular dictionary.

Since attributes of shared objects are stored in a shared dictionary, the semantics of shared dictionaries apply to attributes as well. This implies that

```
typedef struct
{
  int regndx;
  unsigned int offset;
} SharedMemHandle;

void* SharedMemHandle_AsVoidPtr(SharedMemHandle handle);
SharedMemHandle SharedMemHandle_FromVoidPtr(void* ptr);
```

Figure 3.13: Definition of the `SharedMemHandle` structure, and the related mapping functions.

the values assigned to an attribute are shared implicitly, and the attribute values are wrapped in proxy objects when read. Section 3.8.1 describes how different synchronization policies may be specified for each shared type. Even if no synchronization is specified for a user-defined type, the attributes of the shared object may be accessed safely, since the synchronization policy specified for shared dictionaries enforces monitor access semantics.

3.5 Shared container objects

Section 3.5.1 examines the general problems related to storing recursive data structures in shared memory. It concludes that shared container objects cannot rely on the implementation of the regular container objects, since the latter reference other objects using direct pointers. As a consequence, the shared container objects must be reimplemented using an alternative mechansim for referencing other objects. Section 3.5.2 describes how this is done for shared lists and tuples. Section 3.5.3 describes the implementation of shared dictionaries.

When objects are assigned to a shared container, they are implicitly shared. When shared objects are read from the container, they are implicitly wrapped in *proxy objects*, as described in Section 3.6.

3.5.1 Memory handles

POSH uses *memory handles* to refer to locations in shared memory across processes.

A well-known complication with using shared memory is that recursive data structures allocated in shared memory cannot be represented using pointers.

```
static int root = −1;

typedef struct {
  void* start; /* First byte in the region */
  void* end; /* Last byte in the region */
  int left, right; /* These are indexes in the table */
} AttachedRegion;

static AttachedRegion at_map[MAX_REGIONS];
```

Figure 3.14: Definition of the attachment map.

This is because shared memory regions may be attached at different addresses in different processes. Therefore, a pointer cannot be stored in shared memory unless it is guaranteed that what it points to is located at the same address in all processes.

The implications of this are two-fold when it comes to allocating Python objects in shared memory:

- Firstly, every object contains a type pointer, and there is no way to change its meaning without rewriting Python from the ground up. Thus, POSH must guarantee that the type objects pointed to by shared objects are indeed located at the same address in all processes. Furthermore, the method slots of type objects point to C functions. In the presence of dynamically linked code, care must also be taken to ensure that these functions have the same addresses in all processes. The type object also contains pointers to its base classes and a pointer to its docstring. All of these pointers must be valid for all the participating processes. A simple way to fulfill these requirements is to use the `fork` system call provided by all POSIX-compliant platforms to create new processes. This duplicates the entire memory lay-out of the calling process, including all the type objects and functions in question. As a consequence, however, types that should be shareable must be registered prior to any `fork` calls. In practice, this is a small limitation.

- Secondly, shared objects are allocated on heaps that grow dynamically, creating new shared memory regions as needed. Thus, there is no guarantee that they have the same address in all processes. This implies that shared container objects (lists, tuples and dictionaries) cannot store their references to other objects as plain `PyObject` pointers. Some other mechanism is needed for referring to shared objects across processes.

POSH tackles the general problem of referring to locations in shared memory across processes by introducing the concept of *memory handles*. A memory handle consists of an identifier for a shared memory region plus an offset in this region. This uniquely identifies a memory location to every process, regardless of where the region is attached. The actual data structure that implements a memory handle is shown in Figure 3.13. The `regndx` field of the `SharedMemHandle` structure is the shared memory region's index in the *region table* described in Section 3.2.1. The `offset` field is the offset in bytes from the start of the memory region to the location in question.

Whenever a process needs to access a memory location referred to by a memory handle, the handle must be converted to a pointer. Conversely, a process may wish to store a handle to a memory location for which it only has a pointer; this requires converting the pointer to a handle. These operations are implemented by the `SharedMemHandle_AsVoidPtr` and `SharedMemHandle_FromVoidPtr` functions, also listed in Figure 3.13. Their implementation relies on a data structure called the *attachment map*, listed in Figure 3.14. Every process has its own private copy of the attachment map (it is *not* a part of the globally share data described in Section 3.2.1). The data structure records which shared memory regions are attached in the process' address space, and at what addresses. It is implemented as an array, in which every shared memory region has the same index as in the *region table* described in section 3.2.1. Each array entry also contains `left` and `right` fields, which hold the indexes of other entries. This allows the array to be viewed as a binary search tree as well. The root entry's index is stored in the `root` variable.

This particular data structure was designed to allow very efficient (linear-time) mapping from memory handles to pointers, as well as efficient (logarithmic-time) reverse mappings from pointers to memory handles.

`SharedMemHandle_AsVoidPtr` uses the attachment map to implement constant-time handle-to-pointer mappings, by simply retrieving the starting address of the shared memory region from `at_map[handle.regndx]` , and adding the appropriate offset.

`SharedMemHandle_FromVoidPtr` implements the reverse mapping by viewing the attachment map as a binary search tree, searching it for an attached region that contains the given address. Being a binary search, this operation is performed in logarithmic time on average. POSH ensures that this is also the worst-case performance, by regenerating an optimal search tree every time a new shared memory region is attached. This is considered worthwhile, since the attachment of a shared memory region is a very rare event compared to the reverse

```
typedef struct
{
  PyObject_VAR_HEAD
  int capacity;
  SharedMemHandle vectorh;
} SharedListBaseObject;

typedef struct
{
  PyObject_VAR_HEAD
  SharedMemHandle vector[1];
} SharedTupleBaseObject;
```

Figure 3.15: Object structures of the `SharedListBase` and `SharedTupleBase` types.

> mapping from a pointer to a memory handle. The optimal search tree is generated by sorting the shared memory regions according to the addresses at which they're attached, and recursively picking the median as the root entry. This algorithm executes in $O(n \log n)$, as a consequence of the requirement to sort the regions.

As described in Sections 3.5.2 and 3.5.3, POSH implements shared versions of the standard container types using memory handles instead of pointers to store the references to other objects.

3.5.2 Shared lists and tuples

Section 3.1 explains how shared types normally subtype their shareable counterparts, using a special meta-type that adapts the new type to enable sharing of its instances. As explained in Section 3.5.1, this is not possible for shared container objects, since they cannot rely on the implementation provided by the standard container types. However, the approach of producing shared types through inheritance is attractive for its simplicity. As a consequence, the strategy employed by POSH to implement shared container objects, is to implement suitable base types for the actual shared types. To this end, POSH implements the built-in `SharedTupleBase` and `SharedListBase` types, that implement the most basic operations of shared tuples and lists. These are subsequently subtyped from Python code to produce the `SharedTuple` and `SharedList` types. This allows some of the methods to be implemented in Python code, simplifying the implementation.

```
static PyObject*
vector_item(SharedMemHandle* vector, int size, int index)
{
    SharedObject* obj;

    if(index < 0)
        index += size;
    if(index < 0 || index >= size) {
        PyErr_SetString(PyExc_IndexError, "index out of range");
        return NULL;
    }
    obj = (SharedObject*) SharedMemHandle_AsVoidPtr(vector[index]);
    assert(obj != NULL);
    return MakeProxy(obj);
}
```

Figure 3.16: The `vector_item` function.

There are many similarities between lists and tuples in Python. Both types implement a sequence of objects, and support reading operations such as iteration, indexing and slicing. Moreover, they support repetition (using the `*` operator), concatenation (using the `+` operator) and comparisons. The main difference is that lists are mutable objects, supporting self-modifying operations such as insertion and sorting, while tuples are immutable, meaning that they remain in their initial state throughout their lifetime. In recognition of their similarities, POSH implements shared lists and tuples using a common code base, sharing implementation details wherever appropriate.

Section 2.3 explains the basics of how Python objects are represented. The main abstraction used in the representation of shared lists and tuples is the *vector*. A vector is simply an array of memory handles, in which each memory handle refers to another shared object. Given that they consist of memory handles, vectors can be stored unambiguously in shared memory. Shared lists and tuples both maintain a vector, which holds all their references to other objects. However, shared lists require the ability to resize their vector, since the number of items in a list may change. Therefore, the vector isn't stored in the object structure itself, but as an auxiliary data structure that can be resized as needed. A single memory handle that refers to the vector is stored in the object structure. Consequently, shared lists are fixed-size objects, just like the standard `list` type. Tuples, on the other hand, retain their initial size throughout their lifetime, so they can embed the vector directly in their object structure. Like the standard `tuple` type, shared tuples are implemented as variable-size objects. The difference is that the

```
static int
vector_ass_item(SharedMemHandle* vector, int size, int index, PyObject* value)
{
    SharedObject* olditem;
    SharedObject* newitem;

    if(index < 0)
        index += size;
    if(index < 0 || index >= size) {
        PyErr_SetString(&PyExc_IndexError, "index out of range");
        return -1;
    }
    newitem = ShareObject(value);
    if(newitem == NULL)
        return -1;
    SharedObject_IncRef(newitem);
    olditem = (SharedObject*) SharedMemHandle_AsVoidPtr(vector[index]);
    SharedObject_DecRef(olditem);
    vector[index] = SharedMemHandle_FromVoidPtr(newitem);
    return 0;
}
```

Figure 3.17: The `vector_ass_item` function.

```
class SharedTuple(_core.SharedTupleBase):
    __metaclass__ = SharedType
    __slots__ = []

    def __getslice__(self, i, j):
        indices = range(len(self))[i:j]
        return tuple([self[i] for i in indices])

    def __str__(self):
        items = map(repr, self)
        return "("+", ".join(items)+")"
```

Figure 3.18: Excerpt of the `SharedTuple` type's implementation.

items of regular tuples are stored as pointers, while memory handles serve the same purpose in shared tuples. Figure 3.15 shows the object structures of the `SharedListBase` and `SharedTupleBase` types.

As an examle of the low-level code, 3.16 shows the `vector_item` function. It operates on a vector, which could either belong to a shared list or a shared tuple. It is used to implement the special `__getitem__` method that

```
class SharedList(_core.SharedListBase):
    __metaclass__ = SharedType
    __slots__ = []

    def __getslice__(self, i, j):
        indices = range(len(self))[i:j]
        return [self[i] for i in indices]

    def extend(self, seq):
        if seq is not self:
            # Default implementation, uses iterator
            for item in seq:
                self.append(item)
        else:
            # Extension by self, cannot use iterator
            for i in range(len(self)):
                self.append(self[i])
```

Figure 3.19: Excerpt of the `SharedList` type's implementation.

specifies the effect of indexing an object (using brackets). The function accepts a pointer to the vector, its size, and the index of the item to retrieve. After checking for bounds errors, a reference to the shared object is retrieved by mapping the corresponding memory handle to a pointer, using the `SharedMemHandle_AsVoidPtr` function described in Section 3.5.1. As explained in Section 3.6, all shared objects must be wrapped in *proxy objects* before they are made accessible to Python code. Therefore, the function wraps the return value in a proxy object using the `MakeProxy` function described in Section 3.6.

A similar function, `vector_ass_item`, implements assignment into a vector. Figure 3.17 shows its implementation, which is used by shared lists to implement the special `__setitem__` method. This method specifies the effect of assigning an item to an index in the object. (Shared tuples have no such method, since they are immutable). Assigning an item to a shared container will implicitly share the item. The `vector_ass_item` function calls the `ShareObject` function described in Section 3.3.2 for this purpose. It proceeds to store a reference to the shared object, by mapping its address to a memory handle using the `SharedMemHandle_FromVoidPtr` function described in Section 3.5.1. As a part of that operation, it has to increase the reference count of the shared object using the `SharedObject_IncRef` function described in Section 3.7. Conversely, it needs to decrease the reference count of the old item, whose reference it overwrites. This is done

```
class SharedDict(_core.SharedDictBase):
    """Dictionary type whose instances live in shared memory."""
    __metaclass__ = SharedType
    __slots__ = []
```

Figure 3.20: Implementation of the `SharedDict` type.

```
typedef struct {
  PyObject_HEAD
  int fill;
  int used;
  int mask;
  SharedMemHandle tableh;
} SharedDictBaseObject;

typedef struct {
  int state;
  long hash;
  SharedMemHandle keyh;
  SharedMemHandle valueh;
} Entry;
```

Figure 3.21: Object structure of the `SharedDictBase` type, and definiton of a hash table entry.

using the `SharedObject_DecRef` function, which may trigger the destruction of the old value if there are no other references to it.

Figure 3.18 shows an excerpt of the `SharedTuple` type's implementation, which illustrates how some operations are implemented in Python code. In the figure, the `__getslice__` and `__str__` methods are implemented by relying on the `__getitem__` method implemented by the base type. Note the special `__metaclass__` attribute, which causes the class statement to use the `SharedType` meta-type described in Section 3.3.3 for the creation of the shared type. Figure 3.19 shows an excerpt of the `SharedList` type's implementation, in which the `__getslice__` and `extend` methods are implemented. The latter relies on the `append` method, which is implemented by the base type.

3.5.3 Shared dictionaries

In the implementation of shared dictionaries, a strategy similar to the one used by shared lists and tuples was chosen. The strategy involves the implementation of a built-in base type, which is subtyped with Python code to add the functionality that can be implemented at a higher level. However, when mirroring the implementation of regular dictionaries, it turned out that the number of inter-dependencies largely dictated that all of the functionality was implemented in C. Consequently, the built-in `SharedDictBase` type essentially provides the entire implementation of shared dictionaries, while the `SharedDict` type is created merely to apply the adaptations made by the `SharedType` meta-type. Figure 3.20 lists the implementation of the `SharedDict` type in its entirety.

Python dictionaries are associative containers implemented using open addressing hashing. The keys of a dictionary must be hashable, meaning that they implement a special `__hash__` method that calculates a hash code for the object. In addition, the key must support equality comparisons in a way that is consistent with the hash codes. The central data structure in a dictionary is a hash table, which has a size that is some power of two. The entry to which a specific key maps is found by using the appropriate number of bits from its hash code, starting with the least significant bits. Collisions are resolved by taking more of the leading bits into account, calculating a new index until a free entry is found. The use of open addressing hashing implies that deleted entries cannot simply be cleared; this could make other key lookups fail erroneously. Thus, deleted entries are marked as deleted, but still occupy their entries. This means that the number of occupied entries in the hash table may increase, even as the number of keys in the dictionary decreases. The percentage of occupied entries in the hash table is known as its *fill factor*, and when it reaches a certain threshold, all the keys in the table are rehashed, building a completely new table. This is known as *resizing* the hash table, even though its size may very well remain the same, or even decrease, during such an operation.

POSH implements shared dictionaries using the exact same algorithms as the implementation of regular dictionaries. This implies that the same behaviour can be expected for operations whose effect is not formally defined, such as the order in which an iteration visits the keys in the dictionary. It should be emphasized, however, that such properties of dictionaries should never be relied on in any case, since they may change as a result of a modified implementation. Shared dictionaries rely on shared objects to have the same hash functions as their shareable equivalents, which is a reasonable expectations, since shared types will generally inherit the special `__hash__` method from the shareable type. The only exceptions are shared tuples,

whose implementation does take care to implement the same hash function as regular tuples.

Figure 3.21 shows the object structure for shared dictionaries, as well as the definiton of a single hash table entry. A description of the fields in the object structure follows.

`fill` counts the number of entries in the hash table that are either occupied or deleted. When this count reaches a certain threshold, the hash table is resized.

`used` counts the number of entries in the hash table that are occupied, which equals the number of keys in the dictionary.

`mask` is the current size of the hash table *minus one*. This value is more useful than the actual size, since it can be used as a bitmask to extract the appropriate number of least significant bits from a hash code.

`tableh` is a memory handle that refers to an array of hash table entries, which forms the hash table of the dictionary. The `state` field of a hash table entry indicates the current state of that entry, which may be either free, occupied or deleted. In the case of an occupied entry, the `hash` field contains the cached hash code of the key, and the `keyh` and `valueh` fields are memory handles for the entry's key and value, respectively.

The hash table of the shared dictionary is allocated as an auxiliary data structure on the type's *data heap*, using the interface described in Section 3.2.3.

As is the case for shared lists, the items assigned to a shared dictionary are implicitly shared. Likewise, values read from the dictionary are always wrapped in proxy objects to protect them from direct access. For the specific details of the `SharedDictBase` implementation, the reader is referred to the "SharedDictBase.c" file in the source code listings.

3.6 Proxy objects

This section explains how proxy objects control the access to shared objects in order to enforce proper synchronization and garbage collection.

posh protects shared objects from direct access by wrapping them in *proxy objects*. The primary purpose of this is to enable satisfactory garbage

```
PyObject* MakeProxy(SharedObject* obj);
```

Figure 3.22: Definition of the `MakeProxy` function.

collection of shared objects, as described in section 3.7. It also provides the opportunity to apply implicit synchronization of the operations performed on shared objects. POSH never allows Python code to operate directly on shared objects, with the one exception of the `self` argument, as described in Section 3.6.2.

3.6.1 Creation of proxy objects

As explained in Section 3.7, a requirement of the multi-process garbage collection algorithm is that a given shared object has at most one proxy object in any process.

POSH defines a factory function for the creation of proxy objects, `MakeProxy`, which ensures that this requirement is met. The `MakeProxy` function, whose definition is shown in Figure 3.22, maintains a weak-valued dictionary known as the *proxy map*. This is a dictionary in which the values are referred to by weak references. The dictionary maps the addresses of shared objects to their corresponding proxy objects, with an entry for every shared object that has a proxy object in the process. The keys in the dictionary are stored as instances of the `Address` type described in Section 3.2.3, while the values are weak references to proxy objects. When `MakeProxy` is called, it checks the proxy map to see if it has an entry for the address of the given shared object. If so, that proxy object is returned, and the creation of another proxy object is avoided. If the proxy map has no entry for the shared object's address, a new proxy object is created and inserted into the proxy map. Since the values in the proxy map are weak references, the presence of the proxy map does not prevent the normal deletion of proxy objects. When a proxy object is deleted, the semantics of weak references ensure that it is removed from the proxy map.

The actual creation of proxy objects is done by looking up the proxy type that corresponds to the shared object's type in the *type map* described in Section 3.3.1. Subsequently, the proxy type is instantiated, passing a reference to the shared object, which creates a proxy objects that wraps the shared object.

```
typedef struct {
  PyObject_HEAD
  SharedObject* referent;
  PyObject* weakreflist;
} ProxyObject;
```

Figure 3.23: Object structure of `Proxy` objects.

```
>>> import posh
>>> lst = posh.share([1,2])
>>> type(lst)
<type 'posh._proxy.SharedListProxy'>

>>> lst._call_method('append', (3,), {})
>>> lst.append(4)
>>> lst
[1, 2, 3, 4]
```

Figure 3.24: Interpreter session illustrating usage of the `_call_method` method.

3.6.2 Creation of custom-tailored proxy types

Section 3.6 describes proxy objects, which are used to control the access to shared objects. Proxy objects should implement the same interface as the shared objects they wrap, making their presence transparent to the user. This requires one distinct proxy type for each shared type in existence. POSH relies on the dynamic nature of Python to generate new proxy types as needed.

POSH defines a built-in type named `Proxy`, which implements the basic functionality needed for proxy objects, and is designed to serve as a base type for all proxy types. To tailor the interface of a proxy type to mimic that of a specific shared type, special *attribute descriptors* are used. Section 2.5.3 describes the basics of attribute descriptors.

Figure 3.23 shows the object structure defined for proxy objects. Each proxy object has a pointer to its *referent*, which is the shared object that the proxy object wraps. The `weakreflist` is used by the Python runtime to support *weak references* to the proxy object, which is required by the `MakeProxy` function described in Section 3.6.1.

Proxy objects implement a single access point to their referent, which

```
>>> type(lst).__dict__['append']
<posh._proxy.ProxyMethodDescriptor object>

>>> type(lst.append)
<type 'posh._proxy.ProxyMethod'>

>>> a = lst.append
>>> a(5)
>>> lst
[1, 2, 3, 4, 5]
```

Figure 3.25: Continuation the interpreter session in Figure 3.24.

provides the only way for Python code to access a shared object. This is a method named `_call_method`, which dispatches a method call to the shared object. The caller passes the name of the method to be invoked, along with the arguments to the method. The positional arguments to the method are passed as a tuple, and the keyword arguments as a dictionary. `_call_method` thus takes three arguments, regardless of the arguments required by the method call it dispatches. Figure 3.24 shows an interpreter session that invokes the `append` method of a shared list in two different ways. The first one is an explicit call to `_call_method`, which is cumbersome and in no way meets the transparency requirements for proxy objects. The second invocation of `append`, however, uses the regular method call syntax, and would work on a regular list object as well. POSH implements support for the latter by means of customized attribute descriptors.

As explained in Section 2.5.3, attribute descriptors provide great flexibility in customizing the attributes of objects. POSH tailors the interface of a proxy type by adding special `ProxyMethodDescriptor` descriptors to it at creation. One such descriptor is added for each of the methods supported by the shared object. The descriptors return special *proxy methods* when their attribute is read, which are callable objects that actually dispatch the call to the `_call_method` method. This allows the normal method call syntax to be used, while preserving `_call_method` as the only access point to the shared object. Figure 3.25 shows a continuation of the interpreter session in Figure 3.24, which illustrates how the `append` method is implemented as an instance of the `ProxyMethod` type. Figure 3.26 shows a slightly abbreviated implementation of the `ProxyMethodDescriptor` and `ProxyMethod` types. Some error checks have been omitted, as well as the `__str__` and `__repr__` methods of the types.

With the availability of proxy methods and their descriptors, producing a proxy type that supports a given interface is a simple matter. The

```
class ProxyMethod(object):
        def __init__(self, inst, cls, mname):
                self.proxy_inst = inst
                self.proxy_cls = cls
                self.mname = mname
                self.cname = cls.__name__

        def __call__(self, *args, **kwargs):
                if self.proxy_inst is None:
                        # Call to unbound proxy method                    10
                        return args[0]._call_method(self.mname, args[1:], kwargs);
                else:
                        # Call to bound proxy method
                        return self.proxy_inst._call_method(self.mname, args, kwargs);


class ProxyMethodDescriptor(object):
        def __init__(self, mname):
                self.mname = mname
                                                                          20
        def __get__(self, inst, cls):
                return ProxyMethod(inst, cls, self.mname)

        def __set__(self, inst, value):
                raise TypeError, "read-only attribute"
```

Figure 3.26: Implementation of the `ProxyMethod` and `ProxyMethodDescriptor` types.

required steps are to subtype the built-in `Proxy` type, and add one `ProxyMethodDescriptor` object to the new type for each of the methods in the referent's type. Figure 3.27 shows the implementation of the `MakeProxyType` function, which uses this approach to create a proxy type that mimics the interface of a given shared type.

Since the purpose of proxy objects is to shield shared objects from direct references, forcing all references to go to the proxy object instead, some restrictions must be imposed on shared objects regarding their `self` argument.[1]While the other arguments passed to shared objects are either non-shared or wrapped in proxy objects, this would contradict the language itself with regard to the `self` argument, since it is supposed to be an instance of the type being called. This implies that the `self` argument must be a direct reference to the shared object. Consequently, a general restriction imposed on shared objects is that they cannot store their `self` argument in a way

[1] `self` is by convention the name of the first argument of an instance method.

```
method_desc_types = (type(list.__add__), type(list.append),
                                        types.UnboundMethodType)


def MakeProxyType(reftype):
        d = {"__slots__": []}
        for attrname in dir(reftype):
                if not hasattr(_core.Proxy, attrname):
                        attr = getattr(reftype, attrname)
                        if type(attr) in method_desc_types:
                                d[attrname] = ProxyMethodDescriptor(attrname);
        name = reftype.__name__+"Proxy"
        return type(name, (_core.Proxy,), d)
```

Figure 3.27: Implementation of the `MakeProxyType` function.

```
void SharedObject_IncRef(SharedObject* obj);
void SharedObject_DecRef(SharedObject* obj);
void SharedObject_SetProxyBit(SharedObject* obj);
void SharedObject_ClearProxyBit(SharedObject* obj);
```

Figure 3.28: Low-level interface used for garbage collection of shared objects.

that makes it persist beyond the lifetime of the method call, since that would make it available to Python code, without the protection of a proxy object. For example, storing `self` in a global variable is not allowed for shared objects. This restriction is not enforced in any way, since that would be nearly impossible, but types that violate the rules may get undefined results. If a shared object needs to store a reference to itself, it should store a reference to its own proxy object, which can be obtained by calling `posh.share(self)`. Another issue is that some methods by convention return `self`, for instance the special method to implement in-place addition, named `__iadd__`. This is allowed, but the implementation of the `_call_method` method will detect it and automatically wrap the return value in a proxy object.

3.7 Garbage collection of shared objects

POSH implements garbage collection of shared objects using an alternative garbage collection algorithm that associates a *process bitmap* and a counter with each shared object.

As described in Section 2.6.1, python implements garbage collection through reference counting, maintaining a simple reference count for each object. This reference count is updated using the `Py_INCREF` and `Py_DECREF` macros, which increment and decrement the count without any form of synchronization. Consequently, it will not work correctly for shared objects, which can be accessed concurrently by multiple processes. Furthermore, using a single reference count per object makes it impossible to avoid memory leaks when processes terminate abnormally, since no knowledge is maintained about the number of references any particular process holds to an object. A garbage collection algorithm for shared objects must be able to account for references from all live processes, and should be able to correct the reference counts of shared objects when processes terminate abnormally. POSH implements an alternative garbage collection algorithm for shared objects that meets these requirements. It builds on the observation that references to shared objects fall within one of two categories.

- References stored in a process' address space. This may be thought of as a reference leading from a process to the shared object.
- References stored (as a memory handle) in another shared object. This may be thought of as a reference leading from one shared object to another.

If the issue were simply to count the number of references in either of these categories, a single counter would suffice. In order to maintain knowledge about which specific processes hold references to an object, one counter per process is required. This seems to be quite expensive in terms of space. However, references from a process to a shared object are always wrapped in *proxy objects*, as described in Section 3.6. Furthermore, a caching system ensures that each process holds at most one proxy object for any given shared object. A process may have many references to the proxy object of a shared object, but only one such proxy object. In effect, the proxy object multiplexes the references to the shared object. This means that a shared object only needs to know the set of processes holding a proxy object for it. This set of processes can be implemented as a bitmap, as explained in section 3.10.1. From another viewpoint, the bitmap represents a binary reference count per process.

Maintaining such a bitmap consequently accounts for all references in the first category listed above. To account for references in the second category, leading from one shared object to another, a separate count is maintained. Together, the bitmap and the counter constitute the state that must be maintained for each shared object. When all the bits in the bitmap are

```
synch.enter(obj, opname) -> rv
synch.leave(obj, rv) -> None
```

Figure 3.29: Interface supported by a synchronization policy object `synch`.

cleared (indicating that no process has a proxy object for it), and the counter is 0, the shared object can (and should) be deleted.

The state required by this algorithm is maintained as the `proxybmp` and `srefcnt` fields of the `SharedObject` structure described in Section 3.4.1. Access to the fields is synchronized using the `reflock` field, which is an instance of the *spin locks* described in Section 3.9.1. Figure 3.28 lists the low-level garbage collection interface defined by POSH. The `SharedObject_IncRef` and `SharedObject_DecRef` functions are used by shared container objects described in Section 3.5 to update the `srefcnt` field, as items are assigned to or deleted from a shared container. The `SharedObject_SetProxyBit` and `SharedObject_ClearProxyBit` functions are used exclusively by the implementation of proxy objects. On creation, a proxy object sets the bit in the shared object that it refers to, and when the proxy object is deleted, the bit is cleared. The *process index*, described in Section 3.10.1, determines which bit in the bitmap that should be modified.

The normal reference count, `ob_refcnt`, has no meaning for shared objects, and is initialized to 2^{30}. This value is chosen so that C code that views shared objects as regular `PyObject`s, can use the `Py_INCREF` and `Py_DECREF` macros without harm, as long as the operations performed are neutral with regard to the reference count. The unsynchronized access to the `ob_refcnt` leads to the theoretical possibility of *lost updates*, which could be a problem if the reference count reached 0 or overflowed. However, the probability of a lost update incorrectly modifying the reference count *in the same direction* a total of 2^{30} times is so extremely low that the possibility can be disregarded.

3.8 Synchronizing access to shared objects

POSH allows for implicit synchronization of the access to shared objects, by associating a *synchronization policy* with each shared type.

```
static PyObject*
enter(PyObject* self, PyObject* args)
{
  PyObject* obj;
  PyObject* opname = NULL;
  SharedObject* shobj;

  if (!PyArg_ParseTuple(args, "O|S", &obj, &opname))
        return NULL;
  shobj = SharedObject_FROM_PYOBJECT(obj);                    10

  if (Lock_Acquire(&shobj->lock))
        return NULL;

  Py_INCREF(Py_None);
  return Py_None;
}

static PyObject*
leave(PyObject* self, PyObject* args)                         20
{
  PyObject* obj;
  PyObject* ignored = NULL;
  SharedObject* shobj;

  if (!PyArg_ParseTuple(args, "O|O", &obj, &ignored))
      return NULL;
  shobj = SharedObject_FROM_PYOBJECT(obj);

  Lock_Release(&shobj->lock);                                 30

  Py_INCREF(Py_None);
  return Py_None;
}
```

Figure 3.30: Implementation of the built-in `Monitor` type's `enter` and `leave` methods.

3.8.1 Synchronization policies

The synchronization requirements for a shared object may vary greatly. Mutable objects such as shared lists and dictionaries usually require synchronization to protect the consistency of their data structures, given that multiple processes may access the objects concurrently. There are still cases, however, where the user is able to guarantee that the shared object's consistency is maintained, for example by applying synchronization at a

```
>>> import posh
>>> i = posh.share(3)
[7060] Instance heap: Allocating 72 bytes at address 0x5004c080 (size 128).

>>> j = posh.share(7)
[7060] Instance heap: Allocating 72 bytes at address 0x5004c100 (size 128).

>>> i*j
[7060] Monitor: Entering __mul__ on SharedInt object at 0x5004c0bc
[7060] Monitor: Entering __mul__ on SharedInt object at 0x5004c13c
[7060] Monitor: Leaving __mul__ on SharedInt object at 0x5004c13c
[7060] Monitor: Leaving __mul__ on SharedInt object at 0x5004c0bc
21
```

Figure 3.31: Interpreter session using POSH in verbose mode.

higher level, with larger granularity. Immutable objects such as integers, strings and tuples clearly don't require any synchronization to protect their data structures, since these are never modified. User-defined types classify as mutable objects if they support attribute assignment.

In response to the varying requirements, POSH allows synchronization policies to be associated with each shared type. A synchronization policy is specified by assigning a special *policy object* to the shared type's `__synch__` attribute. `None` is a legal value for the attribute, in which case no synchronization is applied. If the attribute has another value, the object it refers to must support the interface listed in Figure 3.29.

To gain access to a shared object, the `enter` method must be invoked, passing the object to which access is desired, along with a string identifying the operation to be performed. For method calls, the string equals the name of the method. For other generic operations, invoked by low-level C code, the name corresponds to the special attribute name associated with that operation. For instance, the name used to specify comparison is `__cmp__`. When the `enter` method returns, the caller is free to access the shared object in question. When done, the `leave` method is invoked, passing the shared object, along with the return value from the `enter` call.

This protocol was designed to allow policy objects the flexibility to implement a wide range of synchronization policies, which may or may not take into account the actual operations the caller wishes to perform. The requirement that the return value from `enter` to is passed to `leave` effectively allows the policy object to have the callers maintain its state, which may simplify its implementation. Since POSH currently provides just

a single implementation of policy objects, it is hard to make any qualified statements about the appropriateness of the protocol. More experience with the implementation of policy objects is required to reveal any design flaws in the protocol.

As an example of the extensibility that policy objects provide, POSH implements a type named `VerboseSynch` that is used when POSH executes in verbose mode. `VerboseSynch` instances are wrapper objects that implement the synchronization protocol by delegating all calls to another synchronization policy object. The `VerboseSynch` only adds verbose output to the console, and does not change the access semantics of the object in any way. The output provided by the verbose policy object provides very valuable insights in the flow of the program. Figure 3.31 shows an interpreter session in which posh operates in verbose mode. The output reveals the nonobvious fact that a multiplication of two shared objects causes the synchronization protocol to be carried out for both of them, in turn. The figure also shows the output produced by the *verbose heaps* described in Section 3.2.3.

The general approach of using a synchronization protocol for every access to a shared object clearly incurs a performance penalty for shared objects that don't require synchronization. To minimize this performance penalty, the `nosynch` flag is reserved in the `SharedObject` header described in Section 3.4.1. If the flag is set, it indicates that the shared object's type has a `__synch__` attribute equal to `None`. Checking this flag is obviously considerably faster than performing the attribute lookup on each access.

Note that the state maintained for all shared objects, stored in the `SharedObject` header described in Section 3.4.1, may also be accessed concurrently by multiple processes, even for immutable objects. However, access to these fields is synchronized by separate synchronization mechanisms, as described in the relevant sections.

3.8.2 The `Monitor` type.

POSH provides one implementation of policy objects, apart from the `VerboseSynch` type described in Section 3.8.1. This is the built-in `Monitor` type, which enforces monitor access semantics for shared objects.

`Monitor` objects make use of the *shared lock*, described in Section 3.9, that is associated with every shared object. It is accessible as the `lock` field of the `SharedObject` header described in Section 3.4.1. Note that other synchronization policies may very well ignore the `lock` field, as is the case when the `__synch__` attribute is `None`. Thus, the memory occupied

```
PyObject* SharedObject_Enter(SharedObject* obj, PyObject* opname);

PyObject* SharedObject_EnterString(SharedObject* obj, char* opname,
                                   PyObject** opname_cache);

void SharedObject_Leave(SharedObject* obj, PyObject* state);
```

Figure 3.32: Low-level interface for adhering to the synchronization protocol.

```
PyObject* SharedObject_Repr(SharedObject* obj);
PyObject* SharedObject_Str(SharedObject* obj);
long SharedObject_Hash(SharedObject* obj);
int SharedObject_Print(SharedObject* obj, FILE* fp, int flags);
int SharedObject_Compare(SharedObject* a, PyObject* b);
PyObject* SharedObject_RichCompare(SharedObject* a, PyObject* b, int op);
int SharedObject_RichCompareBool(SharedObject* a, PyObject* b, int op);
```

Figure 3.33: Utility functions that mirror the Python/C API, while adhering to the synchronization protocol.

by the locks of some shared objects is essentially wasted. However, the `SharedObject` header provides the easiest way of associating data structures with shared objects, and allowing the header to vary in size would complicate the implementation nontrivially. Therefore, a design choice was made to tolerate this space wastage in favour of a simple implementation.

The `Monitor` type implements a synchronization policy that enforces monitor access semantics for shared objects. Figure 3.30 lists its implementation of the `enter` and `leave` methods. Apart from the standard code to parse arguments and build a return value, the methods are almost trivial. They respectively acquire and release the shared object's lock, using the interface described in Section 3.9.3, and simply ignore their second arguments.

3.8.3 Applying synchronization to shared objects

The first step in synchronizing access to shared objects is to identify their potential access points. As explained in Section 3.6.2, the only way for Python code to access a shared object is through the `_call_method` method of its proxy object. This was a deliberate design choice to provide a single access point at which to apply synchronization. Another issue is the potential for low-level C code to access shared objects.

POSHdefines a low-level C interface for using the synchronization protocol described in Section 3.8.1. The interface is listed in Figure 3.32.

`SharedObject_Enter` accepts a pointer to a shared object, and a Python string object containing the name of the operation. The latter is passed on the `enter` method of the given object's policy object, which is found by looking up the special `__synch__` attribute.

`SharedObject_EnterString` is a slightly more complicated version of `SharedObject_Enter` that accepts a null-terminated C string as the name of operation. The last argument points to a statically declared pointer, and is used to cache a Python string object that contains the C string. This is a performance optimization to avoid the frequent construction and destruction of Python string objects.

`SharedObject_Leave` calls the `leave` method of the given shared object's policy object. For convenience, this function will execute even if an exception has been raised, re-raising the exception when done. This allows shared objects to raise exceptions without disrupting the synchronization protocol.

Synchronization of the accesses made by Python code is straightforward, given the one and only access point provided by the `Proxy` type. Its implementation of the `_call_method` method simply calls the `SharedObject_Enter` and `SharedObject_Leave` before and after dispatching the call.

As for low-level C code, it needs to follow the synchronization protocol whenever it accesses a shared object. For convenience, a set of functions is defined that mirrors some common functions in the Python/C API, while adhering to the synchronization protocol. Figure 3.33 lists these functions. By replacing the `SharedObject` prefix with `PyObject`, the names of the corresponding Python/C API functions are obtained. These functions are mainly used by the low-level implementations of shared lists, tuples and dictionaries, which are described in Section 3.5.

In general, references to shared objects should not be passed to code that is unaware of their shared nature, unless access to the object has already been obtained. If access has not been obtained, the only safe approach is to pass a reference to the shared object's proxy object, which can be obtained by means of the `MakeProxy` function described in Section 3.6.1. This is generally how shared objects should be passed to third-party C code.

```
static inline void acquire(int *mutex)
{
    __asm__ volatile (" movl %0,%%eax           \n"
                      "1: lock                  \n"
                      "  btsl $0, 0(%%eax)      \n"
                      "  jc 1b                  \n"
                      :
                      :"m"(mutex)
                      :"eax");
}

static inline void release(int *mutex)
{
    __asm__ volatile (" movl %0,%%eax           \n"
                      "1: lock                  \n"
                      "  andl $0, 0(%%eax)      \n"
                      :
                      :"m"(mutex)
                      :"eax");
}
```

Figure 3.34: The `acquire` and `release` functions used to implement spin locks.

3.9 Synchronization primitives

The basic synchronization primitives employed by POSH are reentrant locks located in shared memory. The current implementation of such *shared locks* requires two basic services.

- An implementation of *spin locks*, which are light-weight locks that are obtained through busy waiting. Section 3.9.1 describes how POSH implements spin locks.
- A way for processes to sleep, and a way to wake other sleeping processes. Section 3.9.2 describes the *sleep table*, which is used to implement this service.

3.9.1 Spin locks

POSH defines the low-level interface listed in Figure 3.35, which is used to access spin locks.

```
typedef int Spinlock;

void Spinlock_Init(Spinlock* spinlock);
void Spinlock_Destroy(Spinlock* spinlock);
void Spinlock_Acquire(Spinlock* spinlock);
void Spinlock_Release(Spinlock* spinlock);
```

Figure 3.35: Low-level interface for accessing spin locks.

```
typedef struct {
  struct {
        SemSet semset;
        SharedMemHandle addr[MAX_PROCESSES];
  } sleeptable;

  /* Other members are omitted... */
} Globals;
```

Figure 3.36: Definition of the sleep table.

Figure 3.34 shows the assembly functions on which the current implementation of spin locks are based. The implementation is targeted for the Intel IA-32 processor family. The `acquire` function uses the *bit-test-and-set* instruction to atomically set the least significant bit of an integer, while reading the old value of the bit. The instruction is repeated until the old value of the bit is seen to be 0. The bit-test-and-set instruction is prefixed by the `lock` instruction, which locks the processor-memory bus for the duration of the instruction. This makes it globally atomic on multiprocessor architectures. The `release` function clears all the bits of the integer, effectively releasing the spinlock.

3.9.2 The sleep table

As noted in Section 3.2.1, the *sleep table* is a part of the globally shared data. Its definition is shown in Figure 3.36. The sleep table contains a set of `MAX_PROCESSES` distinct semaphores, one for each process that may exist. Its purpose is to allow a process to block while waiting for an event. This is achieved by executing a `down` operation on the semaphore that corresponds to the index of the process. To wake a sleeping process, a process executes an `up` operation on the same semaphore.

```
typedef struct {
  Spinlock spinlock;
  int owner_pid;
  int nest_count;
  ProcessBitmap waiting;
} Lock;

void Lock_Init(Lock* lock);
void Lock_Destroy(Lock* lock);
int Lock_Acquire(Lock* lock);
int Lock_TryAcquire(Lock* lock);
int Lock_Release(Lock* lock);
int Lock_OwnedBy(int pid);
```

Figure 3.37: Low-level interface for accessing shared locks.

POSH relies on the System V IPC interface for the implementation of the semaphores.

3.9.3 Shared locks

POSH uses reentrant locks located in shared memory for synchronization between processes.

The low-level interface for shared locks is listed in Figure 3.37. A shared lock is represented by the `Lock` structure defined in the figure. A brief description of its fields follows.

`spinlock` is a spin lock, as described in Section 3.9.1, which must be acquired to gain access to the remaining fields.

`owner_pid` is the process ID of the current owner of the lock.

`nest_count` is the nest count for the lock, which is maintained to allow for reentrant locking.

`waiting` is a *process bitmap*, as described in Section 3.10.1, which represents the set of processes that are waiting for access to the lock.

A brief description of the functions defined in Figure 3.37 is given here.

`Lock_Init` initializes the lock structure.

`Lock_Destroy` deinitializes the lock structure.

`Lock_Acquire` acquires the lock, blocking if necessary. The operation first acquires the spin lock, then checks the `owner_pid` to see if the lock is already owned by another process. If so, the process sets a bit in the `waiting` bitmap, and goes to sleep by blocking on the appropriate semaphore in the sleep table. If the lock is available, the process assigns its process ID to the `owner_pid` field and proceeds.

`Lock_TryAcquire` is a non-blocking version that follows the same procedure as `Lock_Acquire`, but returns with an error if if fails to acquire the lock, rather than blocking.

`Lock_Release` releases a lock by clearing the `owner_pid` field. It then examines the `waiting` bitmap, which indicates which processes are waiting for the lock. If any processes are waiting, one of them is woken by performing an `up` operation on the corresponding semaphore in the sleep table.

`Lock_OwnedBy` checks the ownership of a lock to see if it is owned by a specific process.

The current implementation of shared locks has not been given the attention it deserves from a performance-based point of view, since the primary requirement was simply to have a working implementation. However, with a well-defined interface, the entire implementation could painlessly be replaced if performance considerations were to deem it necessary.

3.10 Process Management

The creation and destruction of processes are important events to which POSH must respond in order to keep the globally shared data structures in a consistent state. This section describes the globally shared process table, and how POSH handles the creation and termination of processes.

3.10.1 The process table

As noted in Section 3.2.1, a part of the globally shared data is the *process table*. Its purpose is to map the process IDs of all processes to the set of integers in the range 0–`MAX_PROCESSES`. Figure 3.38 shows the definition of the process table.

```
typedef struct {
    struct {
        Lock lock;
        int pid[MAX_PROCESSES];
    } proctable;

    /* Other members are omitted... */
} Globals;
```

Figure 3.38: Definition of the process table.

```
#define PROCESS_BITMAP_INTS 4
#define INT_BITS (sizeof(int)*8)
#define MAX_PROCESSES (INT_BITS*PROCESS_BITMAP_INTS)

typedef struct {
  int i[PROCESS_BITMAP_INTS];
} ProcessBitmap;

#define ProcessBitmap_SET(bmp, ndx) \
  (bmp).i[(ndx)/INT_BITS] |= (1 << (ndx)%INT_BITS)

#define ProcessBitmap_IS_SET(bmp, ndx) \
  ((bmp).i[(ndx)/INT_BITS] & (1 << (ndx)%INT_BITS) != 0)

#define ProcessBitmap_CLEAR(bmp, ndx) \
  (bmp).i[(ndx)/INT_BITS] &= ~(1 << (ndx)%INT_BITS)
```

Figure 3.39: Definition of a process bitmap, and macros to modify it.

Access to the process table is synchronized by a *shared lock*, described in Section 3.9, which is stored in the field named `lock`. The table itself is a simple array of process IDs. A value of -1 indicates an unused entry. On creation, all processes insert their process ID into an unused entry in the table. The index at which the process ID is stored is known as the *process index* of that process.

Since all processes have indexes in the range 0–`MAX_PROCESSES`, a set of processes may be implemented as a bitmap of `MAX_PROCESSES` bits, where a set bit indicates that the process with the corresponding process index is contained in the set. The definition of such a bitmap, along with the macros to operate on it, is shown in Figure 3.39. This compact representation of a set of processes is used both by the multi-process garbage collection algorithm described in Section 3.7, and by the implementation of *shared locks* described

in Section 3.9.

3.10.2 Process creation

POSH uses a modified version of the standard `os.fork` function for process creation.

As explained in Section 3.5.1, pointers can only be stored safely in shared memory under certain circumstances. Notably, the data to which they point must be allocated at the same virtual address in all processes accessing the pointer. In the context of shared objects, which contain a pointer to their type object, this requires all type objects to be allocated at identical addresses in all processes. POSH relies on the semantics of the POSIX `fork` system call to resolve this problem, since it guarantees an identical memory lay-out in the parent and child processes. Consequently, applications using POSH must create peer processes by means of the `fork` system call.

The `os.fork` function in Python's standard library implements the `fork` system call. However, POSH needs to perform certain updates to internal data structures when new processes are created. Therefore, POSH provides its own version of the `fork` function, which should be used instead of the one provided by the `os` module. The version provided by POSH invokes the built-in function `init_child` from the new child process. This function takes care of some important initialization tasks.

- Firstly, an unused entry in the `process table` described in Section 3.10.1 is selected, where the process ID of the new process is stored. This provides the process with its process index.
- Secondly, all proxy objects in existence are visited, and the `proxybmp` bitmaps of the shared objects to which they refer are updated using the `SharedObject_SetProxyBit` function described in Section 3.7. This step is required by the multi-process garbage collection algorithm, since the `fork` call duplicates all the proxy objects from the parent process, a fact which must be reflected in the bitmaps of the shared objects.
- Lastly, an exit function is registered, using the `Py_AtExit` function provided by the Python/C API, which will be called when the process terminates. This enables the function to perform the necessary cleanup tasks for the process.

3.10.3 Process termination

When a Python process terminates normally, all of the objects in the process are destructed. The destructors generally release the resources held by their objects, so any furher cleanup tasks are often unnecessary. However, POSH needs to update the globally shared *process table*, described in Section 3.10.1, whenever a process terminates. This is done in the exit function registered by the `init_child` method described in Section 3.10.2. The function clears the entry in the process table that contains the process ID of the terminating process, resetting it to -1.

Additionally, a check is made to see if all other processes have terminated, in which case global cleanup must be performed. The following global cleanup tasks are performed.

- The destruction of all shared memory regions that remain in the *region table* described in Section 3.2.2.
- The destruction of the semaphores in the *sleep table*, described in Section 3.9.2.
- Finally, the destruction of the shared memory region in which the globally shared data is itself allocated. The `my_handle` field of the `Globals` structure contains the handle needed for that purpose.

3.10.4 Abnormal process termination

POSH detects the abnormal termination of processes by handling signals, and attempts to react appropriately.[2]

On POSIX-compliant platforms, the abnormal termination of a process will cause a `SIGCHLD` signal to be sent to its parent process. POSH handles this signal and attempts to take corrective measures. The process handling the `SIGCHLD` signal will perform the regular cleanup tasks on behalf of the child process. This just involves freeing the entry in the process table that belonged to the terminating process. However, there are two particularly difficult problems that arise when a process terminates abnormally.

- Firstly, the terminating process will incorrectly fail to update the proxy bitmaps of the shared objects to which it holds references. This may

[2] Due to time limitations, the algorithms described in this section are not fully implemented.

lead to memory leaks in the form of shared objects. To correct this, all shared objects must be examined, and the appropriate bit in the proxy bitmap cleared.

- Secondly, a process may be accessing a shared object when it terminates, leaving the object in an inconsistent state. To detect this, all shared objects must be examined, to see if their lock is owned by the process that terminated abnormally. If so, the objects are flagged as possibly corrupted, using the `is_corrupt` field of the `SharedObject` structure described in Section 3.4.1. This will trigger an exception on the subsequent access to the shared object, which may be handled by the user. This allows the caller to decide how to handle such cases. It can ignore the exception and try again, with the risk of accessing a corrupted object, or choose some other suitable reaction, like discarding the corrupted object.

To address these two problems, a common requirement is to visit all shared objects in existence. This is why separate heaps are used for the allocation of instances and auxiliary data structures, as described in Section 3.2.3. If heap objects are extended to support a traversal of all their allocated memory chunks, this may in effect be used to visit all shared objects in existence. This is because every allocated memory chunk on the instance heap is known to contain a shared object. Consequently, there is a feasible way of visiting all shared objects, so the corrective measures outlined above can be applied. However, due to time limitations, this has not been implemented.

3.11 The POSH programming interface

This section describes the programming interface that applications using POSH are presented with. Section 3.11.1 gives a brief description on how to create and declare a shareable type. Section 3.11.2 explains how to share objects. Section 3.11.3 describes the functions related to process management. Figure 3.44 shows a larger example of POSH usage — a multi-process program that performs a simple matrix multiplication.

3.11.1 Declaration of shareable types

POSH requires all shareable types to be registered prior to any `fork` calls, for the reasons described in Section 3.5.1. This is done by calling the `allow_sharing` method provided by POSH, passing the shareable type and optionally a synchronization policy for the shared type. As described in

```
import posh

class Person(object):
        def __init__(self, name, age=None):
                if age is None:
                        self.name = name.name
                        self.age = name.age
                else:
                        self.name = name
                        self.age = age

posh.allow_sharing(Person)
```

Figure 3.40: Example of a shareable user-defined type.

Section 3.3.2, user-defined shareable types must implement an `__init__` method that supports copy semantics. This is the most significant constraint on user-defined types. Figure 3.40 shows an example where a user-defined type is created and declared shareable.

The `Person` type's `__init__` method accepts either one or two arguments. (Two or three, counting `self`). Either the name and age of the person is given, or a reference to another person. The type implements copy semantics, since it supports instantiation of copies of existing persons. Note that *any* object with the "name" and "age" attributes would be acceptible to the `__init__` method.

If posh detects that a shareable type violates any of the restrictions imposed on them, for instance by overriding an attribute lookup method, the `allow_sharing` function will raise an exception.

3.11.2 Sharing of objects

Once a type has successfully been declared shareable, its instances can be shared using the `share` method provided by POSH. The method creates a shared copy of the given object, and returns a proxy object for it. The returned object always compares equal to the one passed to the method, even though their types are different. Figure 3.41 shows an interpreter session that imports POSH and shares an integer object. As the session's output shows, the fact that one of the objects is shared is completely transparent. Only an explicit type check can reveals it.

Objects can also be shared implicitly by assigning them to shared container

```
>>> import posh
>>> i = 7
>>> j = posh.share(i)
>>> i == j
1

>>> i+j
14

>>> i*j                                                    10
49

>>> type(i)
<type 'int'>

>>> type(j)
<type 'posh._proxy.SharedIntProxy'>
```

Figure 3.41: Interpreter session involving a shared and non-shared object.

```
>>> import posh
>>> l = posh.share(range(4))
>>> l
[0, 1, 2, 3]

>>> type(l[0])
<type 'posh._proxy.SharedIntProxy'>

>>> l.append('four')
>>> l                                                      10
[0, 1, 2, 3, 'four']

>>> type(l[4])
<type 'posh._proxy.SharedStrProxy'>
```

Figure 3.42: Interpreter session involving shared container objects.

objects. Figure 3.42 shows an interpreter session that illustrates this.

3.11.3 Process management

As outlined in Section 3.10.2, POSH defines its own version of the `os.fork` function, that should be used instead of it. For convenience, several other

```
def forkcall(func, *args, **kwargs):
    pid = fork()
    if not pid:
        exit(func(*args, **kwargs))
    return pid
```

Figure 3.43: Implementation of the `forkcall` method.

process-related functions are imported from the `os` module into the `posh` module as well.

POSH also defines two process-related utility functions on its own. One of them is the `waitall` function, which waits for *all* child processes to terminate by repeatedly calling `wait`. The other is the `forkcall` function, which is a variant of `fork` that allows a function call to be executed in a child process. The `forkcall` method forks a child process, which calls the first argument with the remainder of the arguments. The child process exits when the call returns, passing the return value as its exit status. The parent process returns from the `forkcall` function immediately, with the process ID of the child process as its return value. Figure 3.43 shows its implementation.

3.12 Portability issues

As a general strategy, POSH attempts to confine the portions of the code that are non-portable to specific modules with well-defined interfaces, which makes it feasible to completely replace a module's underlying implementation, as long as the same interface is supported.

So far, POSH has only been tested on the FreeBSD platform. However, it is expected that POSH could be run on any POSIX-compliant platform with little or no modification.

3.13 Summary

POSH attempts to combine the advantages of threads and processes in Python, by implementing memory based sharing of objects between processes. For immutable types, this is achieved by placing objects in shared memory and implementing multi-process garbage collection for shared objects. The modified garbage collection scheme requires that shared objects are wrapped in transparent proxy objects. To allow sharing of mutable

objects, implicit synchronization is applied by the proxy objects. The standard container types are reimplemented, using memory handles instead of pointers to hold references to other objects. This enables sharing of container objects as well.

The end result is that almost any kind of Python object can be shared using POSH, including instances of user-defined types. POSH accomplishes this in a very transparent way, enabling a programming model that greatly resembles multi-threaded programming. At the same time, the advantages of scalability and locality of failure associated with processes are retained.

```
import posh, random
WORKERS = 4
N, M = 12, 8
MIN, MAX = 10, 21

def matrix(N, M):
    """Creates a NxM matrix of only zeroes."""
    return [[0]*M for row in range(N)]

def random_matrix(N, M, min, max):
    """Creates a NxM matrix of random elements from range(min, max)."""
    m = matrix(N, M)
    for row in range(N):
        for col in range(M):
            m[row][col] = random.randrange(min, max)
    return m

def rows(m):
    """Returns the number of rows in a matrix."""
    return len(m)

def columns(m):
    """Returns the number of columns in a matrix."""
    return len(m[0])

def work(A, B, C, W):
    """Worker no. W in the calculation A = B x C.
    Calculates every WORKERS row in A, starting at row W."""
    for row in range(W, rows(A), WORKERS):
        print "Worker %d calculating row %d..." % (W, row)
        for col in range(columns(A)):
            # Calculate A[row][col] by doing a dot product of
            # B's row C's column
            sum = 0
            for x in range(columns(B)):
                sum += B[row][x] * C[x][col]
            A[row][col] = sum
    return 0 # Exit status

if __name__ == "__main__":
    b = random_matrix(M, N, MIN, MAX)
    c = random_matrix(N, M, MIN, MAX)
    a = posh.share(matrix(M, M))
    for w in range(WORKERS):
        posh.forkcall(work, a, b, c, w)
    posh.waitall()
    for name, value in ("B", b), ("C", c), ("A = B x C", a):
        print "Matrix %s:\n%s" % (name, value)
```

Figure 3.44: A multi-process program that performs a matrix multiplication using POSH.

Chapter 4

Discussion and conclusion

4.1 Summary of thesis

The global interpreter lock causes multi-threaded Python programs to scale poorly when running on multiprocessor architectures. A common workaround to achieve parallelism on such architectures is to use multiple processes instead of threads. Multi-process applications typically use ad-hoc shared memory or a messaging API to implement inter-process communication. POSH addresses the problems with the global interpreter lock by allowing regular Python objects to be transparently placed in shared memory. Objects in shared memory are indistinguishable from regular objects, as seen by Python code.

The implementation creates types whose instances are allocated in shared memory, by subtyping regular types and overriding their allocation methods. POSH supports sharing of instances of both user-defined and built-in types. This includes support for attributes of shared objects.

Shared objects are shielded from direct access by wrapping them in custom-tailored proxy objects that mimic their interface. Proxy objects provide a single access point from a process to a shared object, allowing implicit synchronization. Different synchronization policies may be specified for different types of shared objects. This enables use of more efficient locking strategies where applicable. For instance, reader-writer locks may be used in favour of monitor access semantics.

POSH ensures that a maximum of one proxy object exists per process for any given shared object. This largely reduces the problem of multi-process garbage collection to tracking the existence of proxy objects. The

multi-process garbage collection algorithm implemented in POSH maintains information about the specific processes that hold references to a shared object. In case of abnormal process termination, POSH may detect which shared objects are affected by the termination and correct their reference counts. Additionally, by examining lock ownerships, POSH is able to determine whether a process termination has caused a shared object to be left in an inconsistent state. Python exceptions are used to alert processes of potentially corrupted objects. This allows a structured approach to handling of abnormal process terminations. However, this affects the transparency of shared objects.

POSH allows reuse of existing code by making shared objects appear as regular objects to low-level C code as well as Python code. This is achieved by prepending the data structures specific to shared objects to their object structures in a transparent way. For instance, a shared integer may be manipulated by existing C code, which was originally designed to operate on regular integers. No changes to the Python runtime are required to use POSH.

4.2 Evaluation

This section examines the requirements listed in the problem definition in Section 1.2, and evaluates how well POSH meets those requirements.

4.2.1 Making shared objects interchangeable with regular objects

Since shared objects are generally accessed by means of their proxy objects, this requirement concerns the degree of transparency with which proxy objects can be interchanged with regular objects.

With some minor exceptions, a proxy object can be used in place of a regular object in practically every context. Code that performs explicit type checks, only accepting objects of certain specific types, will reject a proxy object, since it has another type than what it expects. However, as argued in Section 2.2.3, explicit type checks are considered bad programming practice in most contexts. Code that requires objects supporting a specific interface will generally also accept proxy objects. This is ensured by the dynamic tailoring of proxy types, described in Section 3.6.2, that creates proxy types with the exact same interface as the corresponding shareable types.

Proxy objects also interoperate transparently with regular objects with regard to evaluation of expressions. For instance, all the operators defined for integer objects will yield the same result in an expression involving two regular integers, as in an expression involving one regular integer and one shared integer. The resulting value will be a regular integer in both cases, and the calculated value will be the same. This is because the shared integer inherits the implementation provided by the regular `int` type, so the exact same code is executed in both cases.

Due to time limitations, not all the operations provided by Python's standard container objects are implemented by their shared counterparts. As a consequence, shared container objects are not fully interchangeable with regular containers, since their interface is only partially supported. Among the operations that remain unimplemented are iterators for shared dictionaries, and slice assignment to shared lists. However, the current implementation clearly demonstrates the feasibility of implementing the full interfaces, so a future version could fulfill this requirement for container objects as well.

4.2.2 Making shared objects accessible to concurrent processes

This requirement concerns the handling of concurrent accesses to shared objects, as well as the mechanisms for making shared objects available to peer processes.

POSH provides two basic ways of making a shared object available to a peer process.

- The object can be shared prior to the `fork` call that creates the peer process. This will make the object available to the peer process as well, since it inherits the reference to the object from the parent process.
- The object can be assigned to an already accessible container object, such as a shared list or dictionary, or to a shared object that supports attributes. This will also implicitly share the object, so it does not have to be shared explicitly in advance.

A combination of these techniques will usually be applied. In particular, a convenient approach is to have the initial process share an object that supports attributes, such as an instance of a user-defined type. This allows all the subsequently created peer processes to make shared objects available, by simply assigning them to the initial shared object.

POSH handles concurrent accesses to shared objects by associating a synchronization policy with each shared type, as described in Section 3.8.1. This allows the synchronization policy to be selected according to the particular requirements of the shared type. Immutable shared objects can be accessed without synchronization, while monitor access semantics are enforced by default for the mutable container objects.

4.2.3 Minimizing constraints on shareable types

POSH places very few constraints on shareable types, and supports sharing of instances of both built-in and user-defined types.

The following list sums up the general restrictions imposed by POSH on shareable types.

- Shareable types may not override the special allocation methods defined by the `tp_alloc` and `tp_free` method slots in the type object. If they do, the effect of the allocation methods will be overridden by POSH. This is an obvious restriction, since the shared type has to override these methods to allocate the object in shared memory.
- Shareable types may not define custom `__getattribute__`, `__setattr__` or `__delattr__` methods. This is because these methods are overridden by the shared type to implement attributes for shared objects, as described in Section 3.4.3. The attribute lookup algorithm of shared objects is identical to the one provided by the `object` type, but operates on a shared dictionary associated with the object, rather than a regular dictionary. Shareable types are allowed to define a custom `__getattr__` method, which is a fall-back method that gets invoked when the normal attribute lookup algorithm fails.
- Shareable types must implement copy semantics, as described in Section 3.3.2. This means that their `__init__` method should support copying of existing instances.
- The methods of shareable types may not store references to the `self` argument in any way that allows it to persist beyond the lifetime of the method call. This is because the `self` argument has to be an instance of the shared type. Consequently, and unlike other arguments, it is not wrapped in a proxy object. Storing the `self` reference would violate the basic premise that all references to shared objects are contained in proxy objects, on which the multi-process garbage collection algorithm described in Section 3.7 relies. It would also allow direct access to

the shared object, without adhering to the synchronization protocol described in Section 3.8.1. Methods of shareable types are allowed to return `self`, in which case the return value is automatically wrapped in a proxy object.

- Shareable types may not define a value for the special `__slots__` attribute other than an empty list. The `__slots__` attribute is a relatively new feature of Python, that allows it restrict the possible names of its attributes. Support for this feature could probably be implemented using special *attribute descriptors*, as described in Section 2.5.3, but this was not attempted due to time limitations.

All the basic built-in types provided by Python, including container objects, adhere to these restrictions. Consequently, they are all shareable. A user-defined type might violate some of the restrictions, but in all cases, there are straightforward workarounds. Designing a shareable user-defined type presents no difficulties if their requirements are kept in mind, and is mostly a question of avoiding certain rare features such as a non-empty `__slots__` attribute. The only significant effort required by the programmer is to implement an `__init__` method that supports copy semantics, and that is usually straightforward as well.

Instances of the `InstanceType` type, also known as *class instances*, may not be shared, since `InstanceType` defines custom attribute lookup methods. As explained in Section 2.4.5, classes and class instances are essentially obsolete features in Python, so this restriction is of little significance.

In addition to the restrictions placed on shareable types, one restriction applies to all objects to be shared, regardless of their types. Since sharing of objects is implemented by relying on the copy semantics provided by the shareable type, objects that refer recursively to themselves, forming reference cycles, cannot be shared. This will cause infinite recursion when attempting to copy the object. However, a shared object may create a reference to itself, forming a reference cycle, as long as it is done after it is shared. The shared dictionaries and lists implemented by POSH do handle reference cycles correctly once they have been created.

4.2.4 Minimizing changes to the Python runtime

POSH makes no changes to the Python runtime, and also reuses the implementation of most existing built-in types by means of inheritance. POSH is implemented as a regular python package, with the bulk of the code compiled as a built-in module. As explained in Sections 3.2.3 and

3.8.1, some major modifications may even be made without recompilation of the low-level C code, as a result of a design that favours extensibility at some expense of performance.

4.3 Future work

POSH is a fully operational system running under Python 2.2. The only major portability issue that has been identified is the dependency on the POSIX `fork` system call. Additionally, POSH contains low-level synchronization code that only executes on Intel IA-32 based systems, and the current implementation relies on the System V interface for management of shared memory regions. Porting of POSH to other platforms is a subject for future work.

The handling of abnormal process termination has not been fully implemented, and the use of more elaborate synchronization policies has not been explored. In addition, the implementations of shared dictionaries and lists are not complete.

Most importantly, performance evaluations of POSH would be very interesting. Specifically, a comparison of multi-process applications using POSH with equivalent applications that employ alternative IPC mechanisms. Measurements of the memory usage overhead incurred by POSH would be valuable as well.

Bibliography

[1] Barrett, K., Cassels, B., Haahr, P., Moon, D. A., Playford, K., and Withington, T. P. A monotonic superclass linearization for Dylan. In *OOPSLA'96* (June 1996).

[2] Deutsch, P. L., and Bobrow, D. G. An efficient, incremental, automatic collector. *Communications of the ACM 9*, 19 (1976), 522–526.

[3] Goodheart, B., and Cox, J. *The magic garden explained*. Prentice Hall, 1994.

[4] Knuth, D. E. *The art of computer programming. Volume 1: Fundamental datastructures*. Addison-Wesley, 1968.

[5] van Rossum, G. PEP 252: Making types look more like classes. On-line at http://www.python.org/peps/pep-0252.html, Apr. 2001.

[6] van Rossum, G. PEP 253: Subtyping built-in types. On-line at http://www.python.org/peps/pep-0253.html, May 2001.

Appendix A

Source listing

A.1 Address.h

```
/* Address.h */

#ifndef ADDRESS_H_INCLUDED
#define ADDRESS_H_INCLUDED

#include "_core.h"

#define Address_Check(op) PyObject_TypeCheck(op, &Address_Type)

extern PyTypeObject Address_Type;                                    10

void *
Address_AsVoidPtr(PyObject *obj);

PyObject *
Address_FromVoidPtr(void *ptr);

#endif
```

A.2 Address.c

```
/* Address.c */

#include "Address.h"

typedef struct {
  PyObject_HEAD
  void *ptr;
} AddressObject;

static int tp_compare(PyObject *self_, PyObject *other_)
{
  AddressObject *self = (AddressObject *) self_;
  AddressObject *other = (AddressObject *) other_;

  if(self->ptr < other->ptr)
        return -1;
  if(self->ptr > other->ptr)
        return 1;
  return 0;
}

static long tp_hash(PyObject *self)
{
  return _Py_HashPointer(((AddressObject *) self)->ptr);
}

static PyObject *
tp_str(PyObject *self)
{
  char buf[20];
  PyOS_snprintf(buf, sizeof(buf), "%p", ((AddressObject *) self)->ptr);
  return PyString_FromString(buf);
}

static char tp_doc[] = "Encapsulates a memory address.";

PyTypeObject Address_Type = {
        PyObject_HEAD_INIT(&PyType_Type)
        0,
        "posh._core.Address",
        sizeof(AddressObject),
        0,
        0,              /* tp_dealloc */
        0,              /* tp_print */
        0,              /* tp_getattr */
        0,              /* tp_setattr */
        tp_compare,     /* tp_compare */
        0,              /* tp_repr */
        0,                      /* tp_as_number */
        0,                  /* tp_as_sequence */
        0,                      /* tp_as_mapping */
```

```
        tp_hash,                /* tp_hash */
        0,                      /* tp_call */
        tp_str,                 /* tp_str */
        0,                      /* tp_getattro */
        0,                                  /* tp_setattro */
        0,                                  /* tp_as_buffer */
        Py_TPFLAGS_DEFAULT | Py_TPFLAGS_BASETYPE, /* tp_flags */
        tp_doc,                 /* tp_doc */
        0,                      /* tp_traverse */                          60
        0,                      /* tp_clear */
        0,                      /* tp_richcompare */
        0,                                  /* tp_weaklistoffset */
        0,                                  /* tp_iter */
        0,                                  /* tp_iternext */
        0,                      /* tp_methods */
        0,                                  /* tp_members */
        0,                                  /* tp_getset */
        0,                                  /* tp_base */
        0,                                  /* tp_dict */              70
        0,                                  /* tp_descr_get */
        0,                                  /* tp_descr_set */
        0,                                  /* tp_dictoffset */
        0,                            /* tp_init */
        0,                      /* tp_alloc */
        0,                      /* tp_new */
        0,                      /* tp_free */
};

PyObject *                                                             80
Address_FromVoidPtr(void *ptr)
{
  AddressObject *self = PyObject_New(AddressObject, &Address_Type);
  self->ptr = ptr;
  return (PyObject *) self;
}

void *
Address_AsVoidPtr(PyObject *self)
{                                                                      90
  return ((AddressObject *) self)->ptr;
}
```

A.3 Globals.h

```
/* Globals.h */

/* Global data structures shared between all processes. */

#ifndef GLOBALS_H_INCLUDED
#define GLOBALS_H_INCLUDED

#include "_core.h"
#include "Lock.h"
#include "Process.h"                                                    10
#include "SharedRegion.h"
#include "Handle.h"
#include "SemSet.h"

/* Preferred size of the lock table - one lock per process
   should be enough to avoid contention */
#define LOCK_TABLE_SIZE 5 /*MAX_PROCESSES*/

/* Maximum number of shared memory regions (this limit may of course
   be lower in reality, depending on how _SharedRegion_New() is implemented */  20
#define MAX_REGIONS 500

typedef struct {
  /* The handle for the region this structure is allocated in */
  SharedRegionHandle my_handle;

  /* Table of process ids used to enumerate processes */
  struct {
        Lock lock;
        int pid[MAX_PROCESSES];                                         30
  } proctable;

  /* Table of region handles for all shared memory regions
        that have been created. */
  struct {
        Lock lock;
        int count; /* The current number of regions. */
        int freepos; /* Some position in the table thought to be free, to speed
                                    up searching for a free posititon. */
        SharedRegionHandle regh[MAX_REGIONS];                           40
        size_t regsize[MAX_REGIONS];
  } regtable;

  struct {
        /* The semaphore set contains one semaphore per process, for letting
           the process sleep when needed. There is one handle per process too,
           indicating which address the process is waiting for. */
        SemSet semset;
        SharedMemHandle addr[MAX_PROCESSES];
  } sleeptable;                                                         50
} Globals;
```

```
/* The globals struct is allocated in a shared memory region by the
   first process, prior to any forks. This means that all descendants will
   have this region attached at the same address, so the structure can
   be referenced by a plain pointer. */
extern Globals *globals;

/* Initializes the globals pointer */
int Globals_Init(void);                                                    60

/* Cleans up the global variables */
void Globals_Cleanup(void);

/* Calls _SharedRegion_New() to create a new shared memory region of
   the given size, and stores the handle in the region table, so that
   the region can be destroyed (by Globals_Cleanup()) when the last process
   exits. Returns a SharedMemHandle for the start of the region. */
SharedMemHandle SharedRegion_New(size_t *size);
                                                                           70
/* Calls _SharedRegion_Destroy() to destroy the given shared memory region,
   and also removes it from the region table. */
void SharedRegion_Destroy(SharedMemHandle h);

#endif
```

A.4 Globals.c

```
/* Globals.c */

#include "Globals.h"
#include "SharedRegion.h"
#include "Handle.h"

Globals *globals = NULL;

int
Globals_Init(void)
{
  SharedRegionHandle h;
  size_t size = sizeof(Globals);
  int i;

  LOG();
  assert(globals == NULL);
  /* Allocate the Globals struct in shared memory */
  h = _SharedRegion_New(&size);
  if (SharedRegionHandle_IS_NULL(h))
        return -1;
  globals = (Globals *) _SharedRegion_Attach(h);
  if (globals == NULL) {
        _SharedRegion_Destroy(h);
        return -1;
  }
  globals->my_handle = h;

  /* Initialize the process table. */
  Lock_Init(&globals->proctable.lock);
  for (i = 0; i < MAX_PROCESSES; i++)
        globals->proctable.pid[i] = -1;

  /* Initialize the region table */
  Lock_Init(&globals->regtable.lock);
  for (i = 0; i < MAX_REGIONS; i++)
        globals->regtable.regh[i] = SharedRegionHandle_NULL;
  globals->regtable.count = 0;
  globals->regtable.freepos = 0;

  /* Initialize the sleep table */
  for (i = 0; i < MAX_PROCESSES; i++)
        globals->sleeptable.addr[i] = SharedMemHandle_NULL;
  if (SemSet_Init(&globals->sleeptable.semset, MAX_PROCESSES)) {
        Globals_Cleanup();
        return -1;
  }

  return 0;
}
```

```
void
Globals_Cleanup(void)
{
  int i;

  LOG();
  Lock_Destroy(&globals->proctable.lock);
  Lock_Destroy(&globals->regtable.lock);
  SemSet_Destroy(&globals->sleeptable.semset);
  for (i = 0; i < MAX_REGIONS; i++)
        if (!SharedRegionHandle_IS_NULL(globals->regtable.regh[i]))
          _SharedRegion_Destroy(globals->regtable.regh[i]);
  _SharedRegion_Destroy(globals->my_handle);
  globals = NULL;
}


SharedMemHandle
SharedRegion_New(size_t *size)
{
  SharedRegionHandle regh;
  SharedMemHandle result = SharedMemHandle_NULL;
  int free;

  assert(globals != NULL);
  Lock_Acquire(&globals->regtable.lock);
  if (globals->regtable.count < MAX_REGIONS) {
        /* Find a free index in the table */
        for (free = globals->regtable.freepos;
                !SharedRegionHandle_IS_NULL(globals->regtable.regh[free]);
                free = (free+1) % MAX_REGIONS)
          ;
        regh = _SharedRegion_New(size);
        if (!SharedRegionHandle_IS_NULL(regh)) {
          globals->regtable.count++;
          globals->regtable.regh[free] = regh;
          globals->regtable.regsize[free] = *size;
          globals->regtable.freepos = (free+1) % MAX_REGIONS;
          result.regndx = free;
          result.offset = 0;
        }
  }
  Lock_Release(&globals->regtable.lock);

  return result;
}


void
SharedRegion_Destroy(SharedMemHandle h)
{
  SharedRegionHandle regh;
  void *reg_addr = SharedMemHandle_AsVoidPtr(h);
```

```
  assert(globals != NULL);
  Lock_Acquire(&globals->regtable.lock);
  regh = globals->regtable.regh[h.regndx];
  globals->regtable.regh[h.regndx] = SharedRegionHandle_NULL;                    110
  Lock_Release(&globals->regtable.lock);

  _SharedRegion_Detach(reg_addr);
  _SharedRegion_Destroy(regh);
}
```

A.5 Handle.h

```
/* Handle.h */

#ifndef HANDLE_H_INCLUDED
#define HANDLE_H_INCLUDED

#include "_core.h"

/* The concept of handles for memory chunks is a way to enable
   referring to the same memory location across different
   processes. Since shared memory regions may be attached at
   different addresses in different processes, a pointer does
   not have the same meaning for all processes. A handle
   _does_ have the same meaning for all processes. */

/* Handle for a location in shared memory. */
typedef struct
{
  int regndx; /* This is the region's index in globals->regtable */
  unsigned int offset; /* This is the offset within the region */
} SharedMemHandle;

/* NULL handle value */
#define SharedMemHandle_NULL _shared_mem_handle_null

/* NULL handle value for use in initializers */
#define SharedMemHandle_INIT_NULL { -1, 0 }

/* Macro to test a handle for NULL-ness */
#define SharedMemHandle_IS_NULL(h) (h.regndx == -1)

extern SharedMemHandle _shared_mem_handle_null;

/* Macro to compare two handles for equality */
#define SharedMemHandle_EQUAL(a, b) \
  ((a).regndx == (b).regndx && (a).offset == (b).offset)

/* Maps a handle for a memory location to a pointer to the location.
   This will attach the shared memory region if necessary. */
void *SharedMemHandle_AsVoidPtr(SharedMemHandle handle);

/* Maps a pointer to a memory location to a handle for the location */
SharedMemHandle SharedMemHandle_FromVoidPtr(void *ptr);

#endif
```

A.6 Handle.c

```
/* Handle.c */

#include "Handle.h"
#include "Globals.h"
#include "SharedRegion.h"
#include "SharedObject.h"

/* Table that maps shared memory regions to the addresses at which
   they're attached. The index of a shared memory region is the same
   here as in globals->regtable. The left and right fields are used
   along with the root variable to view the table as a binary search
   tree when performing a reverse mapping.
   On a fork(), the entire data structure will be duplicated, which is
   correct, since the child process will inherit the attachments of its
   parent. */

static int root = -1;

typedef struct {
  void *start; /* First byte in the region */
  void *end; /* Last byte in the region */
  int left, right; /* These are indexes in the table */
} AttachedRegion;

static AttachedRegion at_map[MAX_REGIONS] = {
  {NULL, NULL, 0, 0},
  /* The entire table is zero-filled */
};

/* Recursively builds an optimal search tree of the entries in
   at_map specified by the given indexes. (The indexes must be
   sorted according to where the regions are attached.)
   Returns the index of the root of the tree. */
static int
optimal_tree(int *indexes, int first, int last)
{
  if (last < first)
        return -1;
  else {
        /* The root should be the median of the sorted indexes */
        int mid = (last+first)/2;
        int r = indexes[mid];
        /* Apply this rule recursively */
        at_map[r].left = optimal_tree(indexes, first, mid-1);
        at_map[r].right = optimal_tree(indexes, mid+1, last);
        /* Return the root */
        return r;
  }
}

/* Comparison function used for sorting in build_tree(). */
```

```
static int
compare_indexes(const void *aa, const void *bb)
{
  const int a = *((const int *) aa);
  const int b = *((const int *) bb);

  if (at_map[a].start < at_map[b].start)
        return -1;
  if (at_map[a].start > at_map[b].start)
        return 1;
  if (at_map[a].end < at_map[b].end)
        return -1;
  if (at_map[a].end > at_map[b].end)
        return 1;
  return 0;
}

/* Rebuilds the binary search tree in an optimal way. */
static void
build_tree(void)
{
  int regndx[MAX_REGIONS];
  int i, regcount;

  /* Gather the indexes of all attached regions */
  for (i = regcount = 0; i < MAX_REGIONS; i++)
        if (at_map[i].start != NULL)
          regndx[regcount++] = i;
  /* Sort the indexes according to where the regions are attached */
  qsort(regndx, regcount, sizeof(int), compare_indexes);
  /* Build the optimal search tree from the sorted indexes */
  root = optimal_tree(regndx, 0, regcount-1);
}


/********************/
/* PUBLIC INTERFACE */
/********************/

/* NULL value for a shared memory handle */
SharedMemHandle _shared_mem_handle_null = SharedMemHandle_INIT_NULL;

void *
SharedMemHandle_AsVoidPtr(SharedMemHandle handle)
{
  AttachedRegion *at;

  if (SharedMemHandle_IS_NULL(handle)) {
        /* No error here; SharedMemHandle_NULL maps to NULL */
        return NULL;
  }

  at = &at_map[handle.regndx];
  if (at->start == NULL) {
```

```
        /* Attach the region */
        assert(globals != NULL);
        at->start = _SharedRegion_Attach(globals->regtable.regh[handle.regndx]);
        if (at->start == NULL) {
      /* This is an error */
          return NULL;
    }
        at->end = at->start + globals->regtable.regsize[handle.regndx] - 1;
        build_tree();
  }
  return at->start + handle.offset;
}

SharedMemHandle
SharedMemHandle_FromVoidPtr(void *ptr)
{
  SharedMemHandle result;
  int i;

  if (ptr == NULL) {
        /* No error here; NULL maps to SharedMemHandle_NULL */
    return SharedMemHandle_NULL;
  }

  /* Search at_map viewed as a binary search tree for an attached region which
        contains the address. */
  i = root;
  while (i != -1) {
        if (ptr < at_map[i].start)
          i = at_map[i].left;
        else if (ptr > at_map[i].end)
          i = at_map[i].right;
        else
          break;
  }
  if (i == -1) {
        /* No matching attached region. This is an error. */
        PyErr_SetString(PyExc_RuntimeError,
                  "reverse memory handle mapping failed");
        return SharedMemHandle_NULL;
  }

  /* We've found an attached region that contains the address.
        Construct and return a handle relative to the region. */
  result.regndx = i;
  result.offset = ptr-at_map[i].start;

  return result;
}
```

A.7 Lock.h

```
/* Lock.h */

#ifndef LOCK_H_INCLUDED
#define LOCK_H_INCLUDED

#include "_core.h"
#include "Process.h"
#include "Spinlock.h"

typedef struct {
  Spinlock spinlock;
  int owner_pid;
  int nest_count;
  ProcessBitmap waiting;
} Lock;

/* Initializes the lock. Always succeeds. */
void Lock_Init(Lock *lock);

/* Destroys the lock. Always succeeds. */
void Lock_Destroy(Lock *lock);

/* Acquires the lock, blocking if necessary.
   Returns -1 on failure, 0 on success. */
int Lock_Acquire(Lock *lock);

/* Tries to acquire the lock without blocking.
   Returns -1 on failure, 0 on success. */
int Lock_TryAcquire(Lock *lock);

/* Releases the lock. Returns -1 on failure, 0 on success. */
int Lock_Release(Lock *lock);

/* Returns 1 if the lock is owned by the given process. */
int Lock_OwnedBy(int pid);

#endif
```

A.8 Lock.c

```
/* Lock.c */

#include "Lock.h"
#include "Globals.h"

void
Lock_Init(Lock *lock)
{
  Spinlock_Init(&lock->spinlock);
  lock->owner_pid = -1;
  lock->nest_count = 0;
  ProcessBitmap_CLEAR_ALL(lock->waiting);
}

void
Lock_Destroy(Lock *lock)
{
  Spinlock_Destroy(&lock->spinlock);
  lock->owner_pid = -1;
  lock->nest_count = 0;
  assert(ProcessBitmap_IS_ZERO(lock->waiting));
}

/* Selects which process to wake up when releasing a lock */
static int
select_wakeup(Lock *lock)
{
  static int j = -1;

  if (ProcessBitmap_IS_ZERO(lock->waiting))
    return -1;

  for(;;) {
    /* This loop must terminate, since at least one bit is set */
      j = (j+1) % MAX_PROCESSES;
      if (ProcessBitmap_IS_SET(lock->waiting, j))
        return j;
  }
}

int
Lock_TryAcquire(Lock *lock)
{
  int result = -1;

  Spinlock_Acquire(&lock->spinlock);
  if (lock->owner_pid == -1 || lock->owner_pid == my_pid) {
      /* The lock is free, so grab it. */
      lock->owner_pid = my_pid;
      lock->nest_count++;
      result = 0;
```

```
  }
  Spinlock_Release(&lock->spinlock);
  return result;
}

int
Lock_Acquire(Lock *lock)
{
  for (;;) {                                                          60
        Spinlock_Acquire(&lock->spinlock);
        if (lock->owner_pid == -1 || lock->owner_pid == my_pid) {
          /* The lock is free, so grab it. */
          lock->owner_pid = my_pid;
          lock->nest_count++;
          Spinlock_Release(&lock->spinlock);
          return 0;
        }
        else {
          /* We must wait. Sleep on this process's semaphore in the sleep table,   70
                and retry when woken up. */
          globals->sleeptable.addr[my_pindex] = SharedMemHandle_FromVoidPtr(lock);
          ProcessBitmap_SET(lock->waiting, my_pindex);
          Spinlock_Release(&lock->spinlock);
          if (SemSet_Down(&globals->sleeptable.semset, my_pindex))
                return -1;
        }
  }
}
                                                                      80
int
Lock_Release(Lock *lock)
{
  Spinlock_Acquire(&lock->spinlock);
  if (lock->owner_pid != my_pid) {
        Spinlock_Release(&lock->spinlock);
        return -1;
  }
  if (--lock->nest_count == 0) {
        int wakeup = select_wakeup(lock);                             90
        lock->owner_pid = -1;
        if (wakeup != -1) {
          globals->sleeptable.addr[wakeup] = SharedMemHandle_NULL;
          ProcessBitmap_CLEAR(lock->waiting, wakeup);
          Spinlock_Release(&lock->spinlock);
          return SemSet_Up(&globals->sleeptable.semset, wakeup);
        }
  }
  Spinlock_Release(&lock->spinlock);
  return 0;                                                           100
}
```

A.9 LockObject.h

```
/* LockObject.h */

#ifndef LOCKOBJECT_H_INCLUDED
#define LOCKOBJECT_H_INCLUDED

#include "_core.h"

extern PyTypeObject Lock_Type;

#endif                                                          10
```

A.10 LockObject.c

```
/* LockObject.c */

#include "LockObject.h"
#include "Lock.h"

typedef struct {
  PyObject_HEAD
  Lock lock;
} LockObject;

static int
lock_tp_init(PyObject *self_, PyObject *args, PyObject *kwargs)
{
  LockObject *self = (LockObject *) self_;
  LockObject *copy = NULL;
  static char *kwlist[] = {"copy", NULL};

  if (!PyArg_ParseTupleAndKeywords(args, kwargs, "|O!", kwlist,
                                   &Lock_Type, &copy))
      return -1;
  if (copy != NULL) {
      /* XXX Verify that the lock is unlocked. */
  }
  Lock_Init(&self->lock);
  return 0;
}

static PyObject *
lock_acquire(PyObject *self_, PyObject *noargs)
{
  LockObject *self = (LockObject *) self_;

  if (Lock_Acquire(&self->lock))
      return NULL;
  Py_INCREF(Py_None);
  return Py_None;
}

static PyObject *
lock_try_acquire(PyObject *self_, PyObject *noargs)
{
  LockObject *self = (LockObject *) self_;
  int success;

  success = !Lock_TryAcquire(&self->lock);
  return PyInt_FromLong(success);
}

static PyObject *
lock_release(PyObject *self_, PyObject *noargs)
{
```

```
  LockObject *self = (LockObject *) self_;

  if (Lock_Release(&self->lock))
      return NULL;
  Py_INCREF(Py_None);
  return Py_None;
}

static char lock_acquire_doc[] =                                          60
"lock.acquire() -- acquires the lock; blocks if necessary";

static char lock_try_acquire_doc[] =
"lock.try_acquire() -> bool -- tries to acquire the lock; never blocks";

static char lock_release_doc[] =
"lock.release() -- releases the lock";

static PyMethodDef lock_tp_methods[] = {
      {"acquire", lock_acquire, METH_NOARGS, lock_acquire_doc},           70
      {"try_acquire", lock_try_acquire, METH_NOARGS, lock_try_acquire_doc},
      {"release", lock_release, METH_NOARGS, lock_release_doc},
      {NULL, NULL} /* sentinel */
};

static char lock_tp_doc[] = "Lock([copy]) -> A new reentrant lock.";

PyTypeObject Lock_Type = {
      PyObject_HEAD_INIT(&PyType_Type)
      0,                                                                  80
      "posh._core.Lock",
      sizeof(LockObject),
      0,
      0,                  /* tp_dealloc */
      0,                  /* tp_print */
      0,                  /* tp_getattr */
      0,                  /* tp_setattr */
      0,                  /* tp_compare */
      0,                  /* tp_repr */
      0,                                  /* tp_as_number */              90
      0,                            /* tp_as_sequence */
      0,                                  /* tp_as_mapping */
      0,                      /* tp_hash */
      0,                  /* tp_call */
      0,                      /* tp_str */
      0,                  /* tp_getattro */
      0,                                  /* tp_setattro */
      0,                                  /* tp_as_buffer */
      Py_TPFLAGS_DEFAULT | Py_TPFLAGS_BASETYPE, /* tp_flags */
      lock_tp_doc,        /* tp_doc */                                    100
      0,                  /* tp_traverse */
      0,                  /* tp_clear */
      0,                  /* tp_richcompare */
      0,                                  /* tp_weaklistoffset */
      0,                                  /* tp_iter */
```

```
        0,                                  /* tp_iternext */
        lock_tp_methods,  /* tp_methods */
        0,                                  /* tp_members */
        0,                                  /* tp_getset */
        0,                                  /* tp_base */                    110
        0,                                  /* tp_dict */
        0,                                  /* tp_descr_get */
        0,                                  /* tp_descr_set */
        0,                                  /* tp_dictoffset */
        lock_tp_init,         /* tp_init */
        0,                    /* tp_alloc */
        PyType_GenericNew, /* tp_new */
        0,                    /* tp_free */
};
```

A.11 Monitor.h

```
/* Monitor.h */

#ifndef MONITOR_H_INCLUDED
#define MONITOR_H_INCLUDED

#include "_core.h"

extern PyTypeObject Monitor_Type;

#endif                                                          10
```

A.12 Monitor.c

```
/* Monitor.c */

#include "Monitor.h"
#include "SharedObject.h"
#include "Globals.h"

static PyObject *
enter(PyObject *self, PyObject *args)
{
  PyObject *obj, *opname = NULL;
  SharedObject *shobj;

  /* Parse the arguments, which should be the object and the name of
        the operation to be performed */
  if (!PyArg_ParseTuple(args, "O|S", &obj, &opname))
        return NULL;
  shobj = SharedObject_FROM_PYOBJECT(obj);

  /* Lock the object's lock */
  if (Lock_Acquire(&shobj->lock))
        return NULL;

  Py_INCREF(Py_None);
  return Py_None;
}

static PyObject *
leave(PyObject *self, PyObject *args)
{
  PyObject *obj, *ignored = NULL;
  SharedObject *shobj;

  /* Parse the arguments, which are the object and the return value
        from the enter() call. */
  if (!PyArg_ParseTuple(args, "O|O", &obj, &ignored))
        return NULL;
  shobj = SharedObject_FROM_PYOBJECT(obj);

  /* Unlock the appropriate lock */
  Lock_Release(&shobj->lock);

  Py_INCREF(Py_None);
  return Py_None;
}

static char enter_doc[] =
"M.enter(x) -- Acquires the lock associated with x.";

static char leave_doc[] =
"M.leave(x) -- Releases the lock associated with x.";
```

```
static PyMethodDef tp_methods[] = {
  {"enter", enter, METH_VARARGS, enter_doc},
  {"leave", leave, METH_VARARGS, leave_doc},
  {NULL, NULL} /* sentinel */
};

static char tp_doc[] =
"Synchronization manager that enforces monitor access semantics.";
                                                                        60
PyTypeObject Monitor_Type = {
        PyObject_HEAD_INIT(&PyType_Type)
        0,
        "posh._core.Monitor",
        sizeof(PyObject),
        0,
        0,                  /* tp_dealloc */
        0,                  /* tp_print */
        0,                  /* tp_getattr */
        0,                  /* tp_setattr */                            70
        0,                  /* tp_compare */
        0,                  /* tp_repr */
        0,                                  /* tp_as_number */
        0,                            /* tp_as_sequence */
        0,                                  /* tp_as_mapping */
        0,                  /* tp_hash */
        0,                  /* tp_call */
        0,                  /* tp_str */
        0,                  /* tp_getattro */
        0,                                  /* tp_setattro */           80
        0,                                  /* tp_as_buffer */
        Py_TPFLAGS_DEFAULT | Py_TPFLAGS_BASETYPE, /* tp_flags */
        tp_doc,             /* tp_doc */
        0,                  /* tp_traverse */
        0,                  /* tp_clear */
        0,                  /* tp_richcompare */
        0,                                  /* tp_weaklistoffset */
        0,                                  /* tp_iter */
        0,                                  /* tp_iternext */
        tp_methods,         /* tp_methods */                            90
        0,                                  /* tp_members */
        0,                                  /* tp_getset */
        0,                                  /* tp_base */
        0,                                  /* tp_dict */
        0,                                  /* tp_descr_get */
        0,                                  /* tp_descr_set */
        0,                                  /* tp_dictoffset */
        0,                            /* tp_init */
        0,                  /* tp_alloc */
        PyType_GenericNew, /* tp_new */                                 100
        0,                  /* tp_free */
};
```

A.13 Process.h

```
/* Process.h */

#ifndef PROCESS_H_INCLUDED
#define PROCESS_H_INCLUDED

#include "_core.h"

/* We allow 4 ints for a process bitmap - this limits
   how many processes there can be. */
#define PROCESS_BITMAP_INTS 4
#define INT_BITS (sizeof(int)*8)
#define MAX_PROCESSES (INT_BITS*PROCESS_BITMAP_INTS)

/* A bitmap of processes, and macros for operations on it */
typedef struct {
  int i[PROCESS_BITMAP_INTS];
} ProcessBitmap;

#define ProcessBitmap_SET(bmp, ndx) \
  (bmp).i[(ndx)/INT_BITS] |= (1 << (ndx)%INT_BITS)

#define ProcessBitmap_IS_SET(bmp, ndx) \
  (((bmp).i[(ndx)/INT_BITS] & (1 << (ndx)%INT_BITS)) != 0)

#define ProcessBitmap_CLEAR(bmp, ndx) \
  (bmp).i[(ndx)/INT_BITS] &= ~(1 << (ndx)%INT_BITS)

/* Update these macros in accordance with PROCESS_BITMAP_INTS */
#define ProcessBitmap_IS_ZERO(bmp) \
  ((bmp).i[0] == 0 && (bmp).i[1] == 0 && (bmp).i[2] == 0 && (bmp).i[3] == 0)

#define ProcessBitmap_CLEAR_ALL(bmp) \
  {(bmp).i[0] = 0; (bmp).i[1] = 0; (bmp).i[2] = 0; (bmp).i[3] = 0; }

#define ProcessBitmap_IS_ALL_SET(bmp) \
  ((bmp).i[0] == (~0) && (bmp).i[1] == (~0) \
    && (bmp).i[2] == (~0) && (bmp).i[3] == (~0))

/* Initializes this process. Should be called on startup and after a fork(). */
int Process_Init(void);

/* Called by a process when it terminates (normally). */
void Process_Cleanup(void);

/* Cleans up after a child process that has terminated abnormally. */
void Process_CleanupChild(int pid);

#endif
```

A.14 Process.c

```
/* Process.c */

#include "Process.h"
#include "Globals.h"

int my_pid = -1;
int my_pindex = -1;


/* Get a process's index in the process table. Allocates
   a new index for it if necessary. Returns -1 on error. */
static int
get_pindex(int pid)
{
  int pos, free;

  assert(globals != NULL);
  Lock_Acquire(&globals->proctable.lock);

  for (pos = 0, free = -1; pos < MAX_PROCESSES; pos++)
        if (globals->proctable.pid[pos] == pid)
          break;
        else if (globals->proctable.pid[pos] == -1)
          free = pos;
  if (pos == MAX_PROCESSES) {
        pos = free;
        if (pos > -1)
          globals->proctable.pid[pos] = pid;
  }

  Lock_Release(&globals->proctable.lock);
  return pos;
}


/* Frees a process's index in the process table.
   Returns 1 if this is the last process. */
static int
free_pindex(int pid)
{
  int pos, used;

  assert(globals != NULL);
  Lock_Acquire(&globals->proctable.lock);

  for (pos = 0, used = 0; pos < MAX_PROCESSES; pos++)
        if (globals->proctable.pid[pos] == pid)
          break;
        else if (globals->proctable.pid[pos] != -1)
          used++;
  if (pos < MAX_PROCESSES)
```

```
        globals->proctable.pid[pos] = -1;

  Lock_Release(&globals->proctable.lock);
  return (used == 0);
}


int
Process_Init()                                                          60
{
  static int first_call = 1;

  if (first_call) {
        first_call = 0;
        /* This is the first process to be initialized. We should initialize
           the global data structures. */
        if (Globals_Init())
          return -1;
  }                                                                     70
  my_pid = getpid();
  my_pindex = get_pindex(my_pid);
  if (my_pindex == -1)
        return -1;
  if (Py_AtExit(Process_Cleanup))
        return -1;
  LOGF("pid=%d pindex=%d", my_pid, my_pindex);
  return 0;
}
                                                                        80

void
Process_Cleanup()
{
  LOGF("pid=%d pindex=%d", my_pid, my_pindex);
  if (free_pindex(my_pid)) {
        /* A return value of 1 means that this is the last process. */
        Globals_Cleanup();
  }
}                                                                       90


void
Process_CleanupChild(int pid)
{
  free_pindex(pid);
}
```

A.15 Proxy.h

```
/* Proxy.h */

#ifndef PROXY_H_INCLUDED
#define PROXY_H_INCLUDED

#include "_core.h"
#include "SharedObject.h"

/* C representation of a proxy object */
typedef struct {
  PyObject_HEAD
  SharedObject *referent;
  PyObject *weakreflist;
} ProxyObject;

extern PyTypeObject Proxy_Type;

#define Proxy_Check(op) PyObject_TypeCheck(op, &Proxy_Type)

/* Returns a borrowed reference to a WeakValueDictionary mapping that maps
   the addresses of shared objects to weak references to their local proxy
   objects. The mapping contains all the proxy objects in existence
   (in this process). */
PyObject *GetProxyMap(void);

#endif
```

A.16 Proxy.c

```
/* Proxy.c */

#include "Proxy.h"

PyObject *
GetProxyMap(void)
{
  static PyObject *proxy_map = NULL;

  if (proxy_map != NULL)
        return proxy_map;
  else {
        PyObject *module = NULL, *dict = NULL, *type = NULL;

        /* Import the module "weakref" and instantiate a WeakValueDictionary */
        module = PyImport_ImportModule("weakref");
        if (module != NULL) {
          dict = PyModule_GetDict(module);
          if (dict != NULL) {
                type = PyDict_GetItemString(dict, "WeakValueDictionary");
                if (type != NULL)
                  proxy_map = PyObject_CallFunctionObjArgs(type, NULL);
          }
        }
        Py_XDECREF(module);
        Py_XDECREF(dict);
        Py_XDECREF(type);
        return proxy_map;
  }
}

/*****************************************/
/* Creation and destruction of proxy objects */
/*****************************************/

PyObject *
proxy_tp_new(PyTypeObject *type, PyObject *args, PyObject *kwargs)
{
  ProxyObject *self;
  PyObject *referent;

  if(type == &Proxy_Type) {
        PyErr_Format(PyExc_TypeError, "cannot create '%.100s' instances",
                     type->tp_name);
        return NULL;
  }
  if(!PyArg_ParseTuple(args, "O:Proxy.__new__", &referent))
        return NULL;

  self = (ProxyObject *) type->tp_alloc(type, 0);
  if(self != NULL) {
```

```
        /* Set the proxy bit of the referent */
        self->referent = SharedObject_FROM_PYOBJECT(referent);
        SharedObject_SetProxyBit(self->referent);
        /* tp_alloc initialized all fields to 0 */
        assert(self->weakreflist == NULL);
    }
    return (PyObject *) self;
}

static void
proxy_tp_dealloc(PyObject *self_)
{
    ProxyObject *self = (ProxyObject *) self_;

    /* Notify weak references to the object */
    if (self->weakreflist != NULL)
        PyObject_ClearWeakRefs(self_);
    /* Clear the proxy bit of the referent */
    SharedObject_ClearProxyBit(self->referent);
    /* Free the memory occupied by the object */
    self->ob_type->tp_free(self_);
}

/****************************/
/* Methods for proxy objects */
/****************************/

static int
proxy_tp_print(PyObject *self_, FILE *fp, int flags)
{
    ProxyObject *self = (ProxyObject *) self_;

    return SharedObject_Print(self->referent, fp, flags);
}

static int
proxy_tp_compare(PyObject *self_, PyObject *other)
{
    ProxyObject *self = (ProxyObject *) self_;

    return SharedObject_Compare(self->referent, other);
}

static PyObject *
proxy_tp_repr(PyObject *self_)
{
    ProxyObject *self = (ProxyObject *) self_;

    return SharedObject_Repr(self->referent);
}

static long
proxy_tp_hash(PyObject *self_)
{
```

```
  ProxyObject *self = (ProxyObject *) self_;

  return SharedObject_Hash(self->referent);
}

static PyObject *
proxy_tp_str(PyObject *self_)
{
  ProxyObject *self = (ProxyObject *) self_;

  return SharedObject_Str(self->referent);
}

static PyObject *
proxy_tp_getattro(PyObject *self_, PyObject *name)
{
  static PyObject *opname = NULL;
  ProxyObject *self = (ProxyObject *) self_;
  PyObject *ref = SharedObject_AS_PYOBJECT(self->referent);
  PyObject *state, *result;

  /* Do normal attribute lookup on self, and try again on the referent
        if it fails. */
  result = PyObject_GenericGetAttr(self_, name);
  if (result == NULL) {
        PyErr_Clear();
        state = SharedObject_EnterString(self->referent, "__getattr__", &opname);
        if (state == NULL)
          return NULL;
        result = PyObject_GetAttr(ref, name);
        SharedObject_Leave(self->referent, state);
  }
  return result;
}

static int
proxy_tp_setattro(PyObject *self_, PyObject *name, PyObject *value)
{
  static PyObject *opname = NULL;
  ProxyObject *self = (ProxyObject *) self_;
  PyObject *ref = SharedObject_AS_PYOBJECT(self->referent);
  PyObject *state;
  int result;

  state = SharedObject_EnterString(self->referent, "__setattr__", &opname);
  if (state == NULL)
        return -1;
  result = PyObject_SetAttr(ref, name, value);
  SharedObject_Leave(self->referent, state);
  return result;
}

/* Macro to extract the referent object from a proxy object. */
#define UNWRAP(op) \
```

```
  if (op != NULL && Proxy_Check(op)) \                                    160
    op = SharedObject_AS_PYOBJECT(((ProxyObject *) op)->referent)

/* Maps a tuple of objects to a new tuple where proxy
   objects are substituted with their referents. */
static PyObject *
map_tuple_to_referents(PyObject *tuple)
{
  int i, size = PyTuple_GET_SIZE(tuple);
  PyObject *newtuple, *item;
                                                                          170
  if(size == 0) {
        Py_INCREF(tuple);
        return tuple;
  }
  newtuple = PyTuple_New(size);
  if(newtuple == NULL)
        return NULL;
  for(i = 0; i < size; i++) {
        item = PyTuple_GET_ITEM(tuple, i);
        UNWRAP(item);                                                     180
        /* This may do a normal INCREF on a shared object,
           but this is allowed, and required to counter the
           later normal DECREF when the tuple is destroyed. */
        Py_INCREF(item);
        PyTuple_SET_ITEM(newtuple, i, item);
  }
  return newtuple;
}

/* Maps a dictionary to a new dictionary where proxy object                190
   values (not keys) are substituted with their referents. */
static PyObject *
map_dict_values_to_referents(PyObject *dict)
{
  PyObject *result, *key, *value;
  int pos = 0;

  result = PyDict_New();
  if (result == NULL)
        return NULL;                                                      200
  while (PyDict_Next(dict, &pos, &key, &value)) {
        UNWRAP(value);
        if (PyDict_SetItem(result, key, value)) {
          Py_DECREF(result);
          return NULL;
        }
  }
  return result;
}
                                                                          210
#undef UNWRAP

static char proxy_call_method_doc[] =
```

```
"proxy._call_method(mname, args, kwargs)\n" \
"Calls the named method on the object referred to by the proxy\n" \
"object, using the given positional and keyword arguments.\n" \
"Note that this method takes 3 arguments regardless of how many\n" \
"arguments the named method takes!";

static PyObject *                                                    220
proxy_call_method(PyObject *self_, PyObject *args)
{
  ProxyObject *self = (ProxyObject *) self_;
  PyObject *ref = SharedObject_AS_PYOBJECT(self->referent);
  /* Borrowed references */
  PyObject *mname;
  PyObject *state;
  PyObject *margs, *mkwargs; /* These are borrowed from the args tuple */
  /* New references */
  PyObject *result = NULL;                                           230
  PyObject *meth = NULL;

  /* Parse the arguments */
  if (!PyArg_ParseTuple(args, "SO!O!:call_method", &mname,
                        &PyTuple_Type, &margs, &PyDict_Type, &mkwargs))
      return 0;

  /* Enter the shared object */
  state = SharedObject_Enter(self->referent, mname);
  if (state == NULL)                                                 240
      goto Error;

  /* Do the actual method call */
  meth = PyObject_GetAttr(ref, mname);
  if (meth != NULL)
      result = PyObject_Call(meth, margs, mkwargs);

  /* Leave the shared object */
  SharedObject_Leave(self->referent, state);
                                                                     250
  /* If the method call returned self (the referent), return the proxy
     object instead.
     XXX: Sharing the result in all cases would be another approach,
     but it would prohibit shared objects from returning non-shareable
     objects. This is perhaps better anyway? */
  if (result == ref) {
        Py_DECREF(result);
        result = self_;
        Py_INCREF(result);
  }                                                                  260

 Error:
  Py_XDECREF(meth);
  return result;
}

static char proxy_doc[] = "Abstract base type for proxy types.";
```

```
static PyMethodDef proxy_tp_methods[] = {
  {"_call_method", proxy_call_method, METH_VARARGS, proxy_call_method_doc},   270
  {NULL,              NULL}         /* sentinel */
};

PyTypeObject Proxy_Type = {
        PyObject_HEAD_INIT(&PyType_Type)
        0,
        "posh._core.Proxy",
        sizeof(ProxyObject),
        0,
        proxy_tp_dealloc, /* tp_dealloc */                                      280
        proxy_tp_print,        /* tp_print */
        0,                 /* tp_getattr */
        0,                 /* tp_setattr */
        proxy_tp_compare, /* tp_compare */
        proxy_tp_repr,         /* tp_repr */
        0,                                  /* tp_as_number */
        0,                            /* tp_as_sequence */
        0,                                  /* tp_as_mapping */
        proxy_tp_hash,    /* tp_hash */
        0,                 /* tp_call */                                        290
        proxy_tp_str,      /* tp_str */
        proxy_tp_getattro, /* tp_getattro */
        proxy_tp_setattro, /* tp_setattro */
        0,                                  /* tp_as_buffer */
        Py_TPFLAGS_DEFAULT | Py_TPFLAGS_BASETYPE, /* tp_flags */
        proxy_doc,         /* tp_doc */
        0,                 /* tp_traverse */
        0,                 /* tp_clear */
        0,                 /* tp_richcompare */
        offsetof(ProxyObject, weakreflist),   /* tp_weaklistoffset */          300
        0,                                  /* tp_iter */
        0,                                  /* tp_iternext */
        proxy_tp_methods, /* tp_methods */
        0,                                  /* tp_members */
        0,                                  /* tp_getset */
        0,                                  /* tp_base */
        0,                                  /* tp_dict */
        0,                                  /* tp_descr_get */
        0,                                  /* tp_descr_set */
        0,                                  /* tp_dictoffset */              310
        0,                            /* tp_init */
        0,                 /* tp_alloc */
        proxy_tp_new,      /* tp_new */
        0,                 /* tp_free */
};
```

A.17 SemSet.h

```
/* SemSet.h */

#ifndef SEMSET_H_INCLUDED
#define SEMSET_H_INCLUDED

#include "_core.h"

#define MAX_SYSV_SET_SIZE 60
#define SYSV_SETS_PER_SEMSET 16
                                                                    10
#define MAX_SEMSET_SIZE MAX_SYSV_SET_SIZE * SYSV_SETS_PER_SEMSET

/* A SemSet semaphore set is implemented by up to SYSV_SETS_PER_SEMSET
   system V semaphore sets. */
typedef struct {
  int size;
  int setid[SYSV_SETS_PER_SEMSET];
} SemSet;

/* Initializes a semaphore set with 'size' semaphores initially set to 0. */   20
int SemSet_Init(SemSet *semset, int size);

/* Destroys a semaphore set. */
void SemSet_Destroy(SemSet *semset);

/* Does an 'Up' operation on the specified semaphore in the semaphore set. */
int SemSet_Up(SemSet *semset, int n);

/* Does a 'Down' operation on the specified semaphore in the semaphore set. */
int SemSet_Down(SemSet *semset, int n);                                     30

#endif
```

A.18 SemSet.c

```
/* SemSet.c */

#include "SemSet.h"
#include <sys/sem.h>

#define SETS_NEEDED(size) \
  (((size) + SYSV_SETS_PER_SEMSET - 1) / (SYSV_SETS_PER_SEMSET))
#define SET_NO(x) ((x) / SYSV_SETS_PER_SEMSET)
#define SEM_NO(x) ((x) % SYSV_SETS_PER_SEMSET)

int
SemSet_Init(SemSet *semset, int size)
{
  int sets = SETS_NEEDED(size);
  int i;

  if (sets > SYSV_SETS_PER_SEMSET)
        return -1;
  for (i = 0; i < sets; i++) {
        int id = semget(IPC_PRIVATE, MAX_SYSV_SET_SIZE, IPC_CREAT | SEM_R | SEM_A);
        if (id == -1) {
          /* Destroy the sets that have been created so far */
          int j;
          for (j = 0; j < i; j++)
                semctl(semset->setid[j], 0, IPC_RMID);
          semset->size = 0;
          PyErr_SetString(PyExc_RuntimeError,
                          "semaphore set creation failed");
          return -1;
        }
        semset->setid[i] = id;
  }
  semset->size = size;
  return 0;
}

void
SemSet_Destroy(SemSet *semset)
{
  int sets = SETS_NEEDED(semset->size);
  int i;

  for (i = 0; i < sets; i++)
        semctl(semset->setid[i], 0, IPC_RMID);
}

int
SemSet_Up(SemSet *semset, int n)
{
  static struct sembuf op = {
        0, /* sem_num */
```

```
        1, /* sem_op */
        0, /* sem_flg */
    };
    int setno;

    if (n < 0 || n >= semset->size) {
        PyErr_SetString(PyExc_RuntimeError,
                        "semaphore set index out of range");
        return -1;                                                    60
    }
    setno = SET_NO(n);
    op.sem_num = SEM_NO(n);
    if (-1 == semop(semset->setid[setno], &op, 1)) {
        PyErr_SetString(PyExc_RuntimeError,
                        "semaphore set UP operation failed");
        return -1;
    }
    return 0;
}                                                                    70

int
SemSet_Down(SemSet *semset, int n)
{
    static struct sembuf op = {
        0, /* sem_num */
        -1, /* sem_op */
        0, /* sem_flg */
    };
    int setno;                                                       80

    if (n < 0 || n >= semset->size) {
        PyErr_SetString(PyExc_RuntimeError,
                        "semaphore set index out of range");
        return -1;
    }
    setno = SET_NO(n);
    op.sem_num = SEM_NO(n);
    if (-1 == semop(semset->setid[setno], &op, 1)) {
        PyErr_SetString(PyExc_RuntimeError,                          90
                        "semaphore set DOWN operation failed");
        return -1;
    }
    return 0;
}
```

A.19 Semaphore.h

```
/* Semaphore.h */

#ifndef SEMAPHORE_H_INCLUDED
#define SEMAPHORE_H_INCLUDED

#include "_core.h"

typedef int Semaphore;

/* Semaphore semantics: The usual, but if a process dies, any
   Semaphore_Down() operations by that process that have not been
   followed by a Semaphore_Up() operation are cancelled,
   'releasing' the semaphores. */

int Semaphore_Init(Semaphore *semaphore, int value);
void Semaphore_Destroy(Semaphore *semaphore);
int Semaphore_Up(Semaphore *semaphore);
int Semaphore_Down(Semaphore *semaphore);

#endif
```

A.20 Semaphore.c

```
/* Semaphore.c */

#include "Semaphore.h"
#include <sys/sem.h>

int
Semaphore_Init(Semaphore *semaphore, int value)
{
  union semun semarg;

  semarg.val = value;
  *semaphore = semget(IPC_PRIVATE, 1, IPC_CREAT | SEM_R | SEM_A);
  if (*semaphore == -1)
        return -1;
  if (semctl(*semaphore, 0, SETVAL, semarg) == -1) {
        /* Error - destroy the semaphore again */
        semctl(*semaphore, 0, IPC_RMID);
        return -1;
  }
  return 0;
}

void
Semaphore_Destroy(Semaphore *semaphore)
{
  if (semctl(*semaphore, 0, IPC_RMID) == -1)
        /* Error ignored */;
}

int
Semaphore_Up(Semaphore *semaphore)
{
  static struct sembuf op = {
        0, /* sem_num */
        -1, /* sem_op */
        SEM_UNDO, /* sem_flg */
  };
  if (semop(*semaphore, &op, 1) == -1)
        return -1;
  return 0;
}

int
Semaphore_Down(Semaphore *semaphore)
{
  static struct sembuf op = {
        0, /* sem_num */
        1, /* sem_op */
        SEM_UNDO, /* sem_flg */
  };
  if (semop(*semaphore, &op, 1) == -1)
```

```
        return -1;
  return 0;
}
```

A.21 SharedAlloc.h

```
/* SharedAlloc.h */

#ifndef SHAREDALLOC_H_INCLUDED
#define SHAREDALLOC_H_INCLUDED

#include "_core.h"
#include "SharedObject.h"

/* C interface for allocation of shared memory by objects.

   SharedAlloc(), SharedFree() and SharedRealloc() are used by shared
   objects to allocate their auxilliary data structures.

   SharedObject_Alloc() and SharedObject_Free() are exclusively for use
   in the tp_alloc and tp_free slots of shared objects' types.
*/

/* Allocates shared memory on the data heap of the given object's
   meta-type. May allocate more bytes than requested - on return,
   *size is the number of bytes actually allocated. On error, this
   sets an exception and returns NULL. */
void *SharedAlloc(PyObject *self, size_t *size);

/* Frees shared memory on the data heap of the given object's meta-type.
   If it fails, it fails badly (dumps core). */
void SharedFree(PyObject *self, void *ptr);

/* Rellocates shared memory on the data heap of the given object's
   meta-type. May allocate more bytes than requested - on return,
   *size is the number of bytes actually allocated. On error, this
   sets an exception and returns NULL. */
void *SharedRealloc(PyObject *self, void *ptr, size_t *size);

/* Allocator function for the tp_alloc slot of shared objects' types. */
PyObject *SharedObject_Alloc(PyTypeObject *type, int nitems);

/* Deallocator function for the tp_free slot of shared objects' types. */
void SharedObject_Free(PyObject *obj);

#endif
```

A.22 SharedAlloc.c

```
/* SharedAlloc.c */

#include "SharedAlloc.h"
#include "SharedHeap.h"
#include "SharedObject.h"
#include "Address.h"

/* (Re)allocates shared memory using the (re)alloc() method of
   one of the heaps of the given type. */
static void *                                                          10
shared_alloc(PyTypeObject *type, char *heap_name,
                        void *ptr, size_t *size_arg)
{
  /* New references */
  PyObject *tuple = NULL, *heap = NULL;
  /* Borrowed references */
  PyObject *addr;
  /* Return value */
  void *rv = NULL;
  /* Size as a long */                                                 20
  long size = (long) *size_arg;

  /* Get the heap object from the meta-type */
  heap = PyObject_GetAttrString((PyObject *) type, heap_name);
  if(heap == NULL)
        goto Error;

  if(ptr == NULL) {
        /* Call the alloc() method of the heap */
        tuple = PyObject_CallMethod(heap, S_ALLOC, "l", size);           30
  }
  else {
        /* Call the realloc() method of the heap */
        tuple = PyObject_CallMethod(heap, S_REALLOC, "(Nl)",
                                                    Address_FromVoidPtr(ptr), size);
  }
  if(tuple == NULL)
        goto Error;
  /* The call should return an (address, size) tuple */
  if(!PyArg_Parse(tuple,                                               40
                        "(O!l);(re)alloc() should return an (address, size) tuple",
                        &Address_Type, &addr, &size))
        goto Error;
  /* Extract the allocated address from the object */
  rv = Address_AsVoidPtr(addr);
  if(rv == NULL)
        PyErr_NoMemory();
  /* Return the actual number of bytes allocated */
  *size_arg = (size_t) size;
                                                                       50
 Error:
```

```
  Py_XDECREF(heap);
  Py_XDECREF(tuple);
  return rv;
}

static void
shared_free(PyTypeObject *type, char *heap_name, void *ptr)
{
  PyObject *heap;                                                    60

  heap = PyObject_GetAttrString((PyObject *) type, heap_name);
  if(heap == NULL) {
        /* This sucks, but it really shouldn't happen.
           XXX Swallow it with PyErr_Clear() ?? */
  }
  else {
        PyObject *result;
        result = PyObject_CallMethod(heap, S_FREE, "N", Address_FromVoidPtr(ptr));
        /* We have to DECREF the return value, even if we don't care about it. */   70
        Py_XDECREF(result);
  }
}

void *
SharedAlloc(PyObject *self, size_t *size)
{
  return shared_alloc(self->ob_type, S_DHEAP, NULL, size);
}
                                                                     80
void
SharedFree(PyObject *self, void *ptr)
{
  shared_free(self->ob_type, S_DHEAP, ptr);
}

void *
SharedRealloc(PyObject *self, void *ptr, size_t *size)
{
  return shared_alloc(self->ob_type, S_DHEAP, ptr, size);            90
}


PyObject *
SharedObject_Alloc(PyTypeObject *type, int nitems)
{
  /* The number of bytes required for the new object */
  const size_t req_size = SharedObject_VAR_SIZE(type, nitems);
  /* The number of bytes actually allocated */
  size_t size = req_size;                                           100
  /* The new object */
  SharedObject *new_obj;

  /* Allocate memory for the new object */
  new_obj = (SharedObject *) shared_alloc(type, S_IHEAP, NULL, &size);
```

```
  if (new_obj == NULL)
        return NULL;

  /* Zero out everything */
  memset(new_obj, '\0', req_size);                                    110

  /* Initialize the object */
  SharedObject_Init(new_obj, type, nitems);

  /* Return the new object as a PyObject */
  return SharedObject_AS_PYOBJECT(new_obj);
}


void                                                                  120
SharedObject_Free(PyObject *obj)
{
  shared_free(obj->ob_type, S_IHEAP, SharedObject_FROM_PYOBJECT(obj));
}
```

A.23 SharedDictBase.h

```
/* SharedDictBase.h */

#ifndef SHAREDDICTBASE_H_INCLUDED
#define SHAREDDICTBASE_H_INCLUDED

#include "_core.h"

extern PyTypeObject SharedDictBase_Type;

#endif
```

A.24 SharedDictBase.c

```
/* SharedDictBase.c */

#include "SharedDictBase.h"
#include "SharedAlloc.h"
#include "SharedHeap.h"
#include "share.h"

/* The implementation of SharedDictBase mirrors that of normal dicts,
   found in Objects/dictobject.c. For further explanations of the
   algorithms used, look there. */

/* Constant related to collision resolution in lookup(). See dictobject.c */
#define PERTURB_SHIFT 5

/* Minimum size of the hash table */
#define MIN_SIZE 8

/* Macro to determine whether it is time to resize a dictionary */
#define TIME_TO_RESIZE(d) ((d)->fill*3 >= ((d)->mask+1)*2)

/* SharedDictBase object */
typedef struct {
  PyObject_HEAD
  int fill; /* # Active + # Deleted */
  int used; /* # Active */
  int mask; /* # Slots -1 */
  SharedMemHandle tableh; /* Handle to table of entries */
} SharedDictBaseObject;

/* States for hash table entries */
#define ES_FREE 0
#define ES_INUSE 1
#define ES_DELETED 2
#define ES_ERROR 3

/* Hash table entry */
typedef struct {
  int state;
  long hash;
  SharedMemHandle keyh;
  SharedMemHandle valueh;
} Entry;


/* Entry returned on error */
static Entry error_entry = {
  ES_ERROR, /* state */
  0, /* hash */
  SharedMemHandle_INIT_NULL, /* keyh */
  SharedMemHandle_INIT_NULL, /* valueh */
};
```

```
/* The basic lookup function used by all operations.

   This function must never return NULL; failures are indicated by returning
   an Entry * for which the state field is ES_ERROR.
   Exceptions are never reported by this function, and outstanding
   exceptions are maintained.
*/                                                                    60
static Entry *
dict_lookup(SharedDictBaseObject *d, PyObject *key, long hash)
{
  int i, cmp;
  unsigned int perturb;
  unsigned int mask;
  Entry *freeslot, *ep, *table;
  SharedMemHandle orig_tableh, orig_keyh;
  PyObject *epkey;
  int restore_error;                                                  70
  PyObject *err_type, *err_value, *err_tb;

  table = (Entry *) SharedMemHandle_AsVoidPtr(d->tableh);
  if(table == NULL)
        return &error_entry;

  /* Compute the initial table index */
  mask = d->mask;
  i = hash & mask;
  ep = &table[i];                                                     80
  if (ep->state == ES_FREE)
        return ep;

  /* Save any pending exception */
  restore_error = (PyErr_Occurred() != NULL);
  if (restore_error)
        PyErr_Fetch(&err_type, &err_value, &err_tb);

  /* From here on, all exits should be via 'goto Done' */
                                                                      90
  orig_tableh = d->tableh;
  freeslot = NULL;
  perturb = hash;

  while (1) {
        if (ep->state == ES_FREE) {
          /* When we hit a free entry, it means that the key isn't present.
                If we encountered a deleted entry earlier, that is also a correct
                position for insertion, and a more optimal one. */
          if (freeslot != NULL)                                       100
                ep = freeslot;
          goto Done;
        }
        if (ep->hash == hash && ep->state == ES_INUSE) {
          /* When the hash codes match, the keys are possibly equal. When
```

```
                comparing them, we must be aware that the comparison may mutate
                the dictionary. */
            orig_keyh = ep->keyh;
            epkey = SharedObject_AS_PYOBJECT(SharedMemHandle_AsVoidPtr(orig_keyh));
            cmp = PyObject_RichCompareBool(epkey, key, Py_EQ);                    110
            if (cmp < 1)
                PyErr_Clear(); /* Swallow exceptions during comparison */
            else {
                if (SharedMemHandle_EQUAL(orig_tableh, d->tableh)
                        && SharedMemHandle_EQUAL(orig_keyh, ep->keyh)) {
                  /* The dictionary seems to be intact */
                  if (cmp == 1)
                      /* And the keys are indeed equal */
                      goto Done;
                }                                                                 120
                else {
                  /* The compare did major nasty stuff to the dict: start over. */
                  ep = dict_lookup(d, key, hash);
                  goto Done;
                }
            }
          }
          if (ep->state == ES_DELETED && freeslot == NULL)
            /* This is the first deleted entry we encounter */
            freeslot = ep;                                                        130

          /* Collision - compute the next table index and try again */
          i = (i << 2) + i + perturb + 1;
          perturb >>= PERTURB_SHIFT;
          ep = &table[i & mask];
        }

       Done:
        /* Restore any previously pending exception */
        if (restore_error)                                                        140
            PyErr_Restore(err_type, err_value, err_tb);
        return ep;
      }


      /* Makes the dictionary empty by allocating a new table and clearing it.
         Saves a pointer to the old table. Used by dict_tp_new(),
         dict_resize() and dict_clear(). */
      static int
      dict_empty(SharedDictBaseObject *self, int minused, Entry **oldtable)      150
      {
        int newsize, actsize, twice_newsize;
        size_t bytes;
        Entry *newtable;

        /* Find the smallest table size > minused. */
        for (newsize = MIN_SIZE; newsize <= minused && newsize > 0; newsize <<= 1)
            ;
        if (newsize <= 0) {
```

```
        PyErr_NoMemory();                                              160
        return -1;
    }

    if(oldtable != NULL)
        /* Get a pointer to the old table */
        *oldtable = (Entry *) SharedMemHandle_AsVoidPtr(self->tableh);

    /* Allocate the new table */
    bytes = sizeof(Entry) * newsize;
    newtable = (Entry *) SharedAlloc((PyObject *) self, &bytes);       170
    if (newtable == NULL) {
        PyErr_NoMemory();
        return -1;
    }
    /* SharedAlloc() may actually have allocated more bytes than requested,
       so we take advantage of that if it means we can double the table size */
    actsize = (bytes / sizeof(Entry));
    twice_newsize = newsize << 1;
    while (twice_newsize > 0 && actsize >= twice_newsize) {
        newsize = twice_newsize;                                       180
        twice_newsize <<= 1;
    }
    /* Zero out the table - this makes state == ES_FREE for all entries */
    memset(newtable, 0, sizeof(Entry) * newsize);

    /* Make the dict empty, using the new table */
    self->tableh = SharedMemHandle_FromVoidPtr(newtable);
    self->mask = newsize - 1;
    self->used = 0;
    self->fill = 0;                                                    190
    return 0;
}


/* Resizes the dictionary by reallocating the table and reinserting all
   the items again. When entries have been deleted, the new table may
   actually be smaller than the old one.
*/
static int
dict_resize(SharedDictBaseObject *self, int minused)                   200
{
    int used = self->used; /* Make a copy of this before calling dict_empty() */
    Entry *oldtable, *oldep, *ep;
    PyObject *key;

    /* Make the dictionary empty, with a new table */
    if(dict_empty(self, minused, &oldtable))
        return -1;

    /* Copy the data over from the old table; this is refcount-neutral    210
        for active entries. */
    assert(oldtable != NULL);
    for(oldep = oldtable; used > 0; oldep++) {
```

```
        if(oldep->state == ES_INUSE) {
          /* Active entry */
          used--;
          key = SharedObject_AS_PYOBJECT(SharedMemHandle_AsVoidPtr(oldep->keyh));

          ep = dict_lookup(self, key, oldep->hash);
          if(ep->state == ES_FREE) {                                    220
                ep->keyh = oldep->keyh;
                ep->valueh = oldep->valueh;
                ep->hash = oldep->hash;
                ep->state = ES_INUSE;
                self->fill++;
                self->used++;
          }
          else
                assert(ep->state == ES_ERROR);
        }                                                               230
  }

  /* Free the old table */
  SharedFree((PyObject *) self, oldtable);

  return 0;
}


/* Generic routine for updating the mapping object a with the items    240
   of mapping object b. b's values override those already present in a. */
static int
mapping_update(PyObject *a, PyObject *b)
{
  PyObject *keys, *iter, *key, *value;
  int status;

  /* Get b's keys */
  keys = PyMapping_Keys(b);
  if (keys == NULL)                                                     250
        return -1;

  /* Get an iterator for them */
  iter = PyObject_GetIter(keys);
  Py_DECREF(keys);
  if (iter == NULL)
        return -1;

  /* Iterate over the keys in b and insert the items in a */
  for (key = PyIter_Next(iter); key; key = PyIter_Next(iter)) {         260
        value = PyObject_GetItem(b, key);
        if (value == NULL) {
          Py_DECREF(iter);
          Py_DECREF(key);
          return -1;
        }
        status = PyObject_SetItem(a, key, value);
```

```
        Py_DECREF(key);
        Py_DECREF(value);
        if (status < 0) {                                          270
          Py_DECREF(iter);
          return -1;
        }
  }
  Py_DECREF(iter);
  if (PyErr_Occurred())
        /* Iterator completed, via error */
        return -1;
  return 0;
}                                                                  280


/* Creates a new, empty shared dictionary */
static PyObject *
dict_tp_new(PyTypeObject *type, PyObject *args, PyObject *kwargs)
{
  PyObject *self;

  self = type->tp_alloc(type, 0);
  if (self != NULL)                                                290
        if(dict_empty((SharedDictBaseObject *) self, 1, NULL)) {
          self->ob_type->tp_free(self);
          return NULL;
        }
  return self;
}


/* Initializes a shared dictionary from a dictionary */
static int                                                         300
dict_tp_init(PyObject *self_, PyObject *args, PyObject *kwargs)
{
  PyObject *arg = NULL;

  if(!PyArg_ParseTuple(args, "|O:SharedDictBase.__init__", &arg))
        return -1;
  if(arg != NULL)
        return mapping_update(self_, arg);
  return -1;
}                                                                  310


/* DECREFs the items in a table, and frees the table itself. Used by
   dict_tp_dealloc() and dict_clear().
   CAUTION: This is only safe to use if the items in the table have
   no opportunity to mutate the table when they are destroyed. */
static void
delete_table(PyObject *self_, Entry *table, int used)
{
  Entry *ep;                                                       320
  SharedObject *obj;
```

```
  if(table != NULL) {
        for (ep = table; used > 0; ep++) {
          if (ep->state == ES_INUSE) {
                --used;
                obj = (SharedObject *) SharedMemHandle_AsVoidPtr(ep->keyh);
                SharedObject_DecRef(obj);
                obj = (SharedObject *) SharedMemHandle_AsVoidPtr(ep->valueh);
                SharedObject_DecRef(obj);
          }
        }
        SharedFree(self_, table);
  }
}


/* Deinitializes a shared dictionary */
static void
dict_tp_dealloc(PyObject *self_)
{
  SharedDictBaseObject *self = (SharedDictBaseObject *) self_;
  Entry *table;

  table = (Entry *) SharedMemHandle_AsVoidPtr(self->tableh);
  delete_table(self_, table, self->used);
  self_->ob_type->tp_free(self_);
}


/* Clears a shared dictionary */
static PyObject *
dict_clear(PyObject *self_, PyObject *noargs)
{
  SharedDictBaseObject *self = (SharedDictBaseObject *) self_;
  Entry *oldtable;
  int used = self->used;

  if(dict_empty(self, 1, &oldtable))
        return NULL;
  /* We can now safely delete the items in the old table, because
        dict_empty() allocated a new table for the dict, and we made
        a copy of the number of used slots. This means that the destructors
        of the items can only mutate the new, empty dict */
  delete_table(self_, oldtable, used);
  Py_INCREF(Py_None);
  return Py_None;
}


static PyObject *
dict_tp_repr(PyObject *self_)
{
  SharedDictBaseObject *self = (SharedDictBaseObject *) self_;
  int i;
```

```
PyObject *s, *temp;
SharedObject *key, *value;
PyObject *colon = NULL, *pieces = NULL, *result = NULL;
Entry *table;
                                                                    380
i = Py_ReprEnter(self_);
if (i != 0) {
      /* Recursive data structure */
      return i > 0 ? PyString_FromString("{...}") : NULL;
}
if (self->used == 0) {
      /* Empty dictionary */
      result = PyString_FromString("{}");
      goto Done;
}                                                                   390

/* Allocate an empty list and a ": " string */
pieces = PyList_New(0);
if (pieces == NULL)
      goto Done;
colon = PyString_FromString(": ");
if (colon == NULL)
      goto Done;

table = (Entry *) SharedMemHandle_AsVoidPtr(self->tableh);          400
assert(table != NULL);

/* Do repr() on each key+value pair, and insert ": " between them.
       Note that repr may mutate the dict. */
for (i = 0; i <= self->mask; i++) {
      if (table[i].state == ES_INUSE) {
        int status;

        /* Get the key and value objects from the entry's handles */
        key = (SharedObject *) SharedMemHandle_AsVoidPtr(table[i].keyh);  410
        value = (SharedObject *) SharedMemHandle_AsVoidPtr(table[i].valueh);
        assert(key != NULL && value != NULL);

        /* Prevent repr from deleting value during key format. */
        SharedObject_IncRef(value);
        s = SharedObject_Repr(key);
        PyString_Concat(&s, colon);
        PyString_ConcatAndDel(&s, SharedObject_Repr(value));
        SharedObject_DecRef(value);
        if (s == NULL)                                              420
              goto Done;
        status = PyList_Append(pieces, s);
        Py_DECREF(s); /* append created a new ref */
        if (status < 0)
              goto Done;
      }
}

/* Add "{}" decorations to the first and last items. */
```

```
  assert(PyList_GET_SIZE(pieces) > 0);                                    430
  s = PyString_FromString("{");
  if (s == NULL)
        goto Done;
  temp = PyList_GET_ITEM(pieces, 0);
  PyString_ConcatAndDel(&s, temp);
  PyList_SET_ITEM(pieces, 0, s);
  if (s == NULL)
        goto Done;

  s = PyString_FromString("}");                                           440
  if (s == NULL)
        goto Done;
  temp = PyList_GET_ITEM(pieces, PyList_GET_SIZE(pieces) - 1);
  PyString_ConcatAndDel(&temp, s);
  PyList_SET_ITEM(pieces, PyList_GET_SIZE(pieces) - 1, temp);
  if (temp == NULL)
        goto Done;

  /* Paste them all together with ", " between. */
  s = PyString_FromString(", ");                                          450
  if (s == NULL)
        goto Done;
  result = _PyString_Join(s, pieces);
  Py_DECREF(s);

 Done:
  Py_XDECREF(pieces);
  Py_XDECREF(colon);
  Py_ReprLeave(self_);
  return result;                                                          460
}


/* Raises an exception on attempt to hash a shared dictionary */
static long
dict_tp_nohash(PyObject *self)
{
  PyErr_Format(PyExc_TypeError, "%.100s objects are unhashable",
                        self->ob_type->tp_name);
  return -1;                                                              470
}


static int
dict_mp_length(PyObject *self_)
{
  SharedDictBaseObject *self = (SharedDictBaseObject *) self_;

  return self->used;
}                                                                         480


static PyObject *
```

```
dict_mp_subscript(PyObject *self_, PyObject *key)
{
  SharedDictBaseObject *self = (SharedDictBaseObject *) self_;
  Entry *ep;
  long hash;

  hash = PyObject_Hash(key);                                              490
  if(hash == -1)
        return NULL;
  ep = dict_lookup(self, key, hash);
  if(ep->state == ES_INUSE) {
        SharedObject *obj = (SharedObject *) SharedMemHandle_AsVoidPtr(ep->valueh);
        assert(obj != NULL);
        return MakeProxy(obj);
  }
  PyErr_SetObject(PyExc_KeyError, key);
  return NULL;                                                            500
}


/* Deletes a (key, value) pair from the dictionary. */
static int
dict_delitem(SharedDictBaseObject *self, PyObject *lookupkey, long hash)
{
  Entry *ep;

  ep = dict_lookup(self, lookupkey, hash);                                510
  if(ep->state == ES_INUSE) {
        SharedObject *key, *value;

        key = (SharedObject *) SharedMemHandle_AsVoidPtr(ep->keyh);
        value = (SharedObject *) SharedMemHandle_AsVoidPtr(ep->valueh);
        ep->state = ES_DELETED;
        self->used--;
        SharedObject_DecRef(key);
        SharedObject_DecRef(value);
        return 0;                                                         520
  }
  return -1;
}


static int
dict_mp_ass_sub(PyObject *self_, PyObject *key, PyObject *value)
{
  SharedDictBaseObject *self = (SharedDictBaseObject *) self_;
  SharedObject *shkey, *shvalue;                                          530
  Entry *ep;
  long hash;

  /* Hash the key (we should be able to do this before sharing it,
     since shared objects should have the same hash function as their
     non-shared counterparts) */
  hash = PyObject_Hash(key);
```

```
  if (hash == -1)
        return -1;
                                                                    540
  if (value == NULL) {
        /* Delete an item */
        if (!dict_delitem(self, key, hash))
          return 0;
        PyErr_SetObject(PyExc_KeyError, key);
        return -1;
  }

  /* Share the value object (sharing the key can
         wait - that might not be necessary) */                     550
  shvalue = ShareObject(value);
  if (shvalue == NULL)
        return -1;
  SharedObject_IncRef(shvalue);

  assert(self->fill <= self->mask); /* At least one empty slot */

  ep = dict_lookup(self, key, hash);
  if (ep->state == ES_INUSE) {
        /* Replace the value of an existing key */                  560
        SharedObject *oldvalue;
        oldvalue = (SharedObject *) SharedMemHandle_AsVoidPtr(ep->valueh);
        ep->valueh = SharedMemHandle_FromVoidPtr(shvalue);
        SharedObject_DecRef(oldvalue);
  }
  else {
        assert(ep->state == ES_FREE);
        /* We are inserting a new key, so we must share it */
        shkey = ShareObject(key);
        if (shkey == NULL)                                          570
          return -1;
        SharedObject_IncRef(shkey);

        if(ep->state == ES_FREE)
          self->fill++;

        ep->keyh = SharedMemHandle_FromVoidPtr(shkey);
        ep->valueh = SharedMemHandle_FromVoidPtr(shvalue);
        ep->hash = hash;
        ep->state = ES_INUSE;                                       580
        self->used++;

        /* Possibly resize the dictionary */
        if (TIME_TO_RESIZE(self))
          if (dict_resize(self, self->used*2))
                return -1;
  }
  return 0;
}
                                                                    590
```

```
static int
dict_sq_contains(PyObject *self_, PyObject *key)
{
  SharedDictBaseObject *self = (SharedDictBaseObject *) self_;
  long hash;

  hash = PyObject_Hash(key);
  if (hash == -1)
        return -1;                                                    600
  return (dict_lookup(self, key, hash)->state == ES_INUSE);
}


static PyObject *
dict_has_key(PyObject *self_, PyObject *key)
{
  SharedDictBaseObject *self = (SharedDictBaseObject *) self_;
  long hash;
  long ok;                                                            610

  hash = PyObject_Hash(key);
  if (hash == -1)
        return NULL;
  ok = (dict_lookup(self, key, hash)->state == ES_INUSE);
  return PyInt_FromLong(ok);
}


static PyObject *                                                     620
dict_get(PyObject *self_, PyObject *args)
{
  SharedDictBaseObject *self = (SharedDictBaseObject *) self_;
  PyObject *key;
  PyObject *failobj = Py_None;
  Entry *ep;
  long hash;

  if (!PyArg_ParseTuple(args, "O|O:get", &key, &failobj))
        return NULL;                                                  630

  hash = PyObject_Hash(key);
  if (hash == -1)
        return NULL;

  ep = dict_lookup(self, key, hash);
  if (ep->state == ES_INUSE) {
        SharedObject *obj = (SharedObject *) SharedMemHandle_AsVoidPtr(ep->valueh);
        assert(obj != NULL);
        return MakeProxy(obj);                                        640
  }
  Py_INCREF(failobj);
  return failobj;
}
```

```
static PyObject *
dict_setdefault(PyObject *self_, PyObject *args)
{
  SharedDictBaseObject *self = (SharedDictBaseObject *) self_;                    650
  PyObject *key;
  PyObject *failobj = Py_None;
  Entry *ep;
  long hash;

  if (!PyArg_ParseTuple(args, "O|O:setdefault", &key, &failobj))
        return NULL;

  hash = PyObject_Hash(key);
  if (hash == -1)                                                                 660
        return NULL;

  ep = dict_lookup(self, key, hash);
  if (ep->state == ES_INUSE) {
        SharedObject *obj = (SharedObject *) SharedMemHandle_AsVoidPtr(ep->valueh);
        assert(obj != NULL);
        return MakeProxy(obj);
  }
  if (dict_mp_ass_sub(self_, key, failobj))
        return NULL;                                                              670
  Py_INCREF(failobj);
  return failobj;
}


static PyObject *
dict_keys_or_values(SharedDictBaseObject *self, int keys)
{
  PyObject *list, *proxy;
  SharedObject *obj;                                                              680
  int i, j, used;
  Entry *table;

 again:
  /* Allocate a list to hold the keys or values */
  used = self->used;
  list = PyList_New(used);
  if (list == NULL)
        return NULL;
  if (used != self->used) {                                                       690
        /* The allocation caused the dict to resize - start over. */
        Py_DECREF(list);
        goto again;
  }

  table = SharedMemHandle_AsVoidPtr(self->tableh);
  assert(table != NULL);

  for (i = 0, j = 0; i <= self->mask; i++) {
```

```
        if (table[i].state == ES_INUSE) {                                    700
          /* Get the key/value object from the entry's handle */
          if (keys)
                obj = (SharedObject *) SharedMemHandle_AsVoidPtr(table[i].keyh);
          else
                obj = (SharedObject *) SharedMemHandle_AsVoidPtr(table[i].valueh);
          assert(obj != NULL);

          /* Encapsulate the key/value in a proxy object */
          proxy = MakeProxy(obj);
          if(proxy == NULL) {                                                710
                Py_DECREF(list);
                return NULL;
          }

          /* Insert the object in the list */
          PyList_SET_ITEM(list, j, proxy);
          j++;
        }
  }
  assert(j == used);                                                         720
  return list;
}


static PyObject *
dict_keys(PyObject *self, PyObject *noargs)
{
  return dict_keys_or_values((SharedDictBaseObject *) self, 1);
}
                                                                           730

static PyObject *
dict_values(PyObject *self, PyObject *noargs)
{
  return dict_keys_or_values((SharedDictBaseObject *) self, 0);
}


static PyObject *
dict_items(PyObject *self_, PyObject *noargs)                              740
{
  SharedDictBaseObject *self = (SharedDictBaseObject *) self_;
  PyObject *list;
  int i, j, used;
  PyObject *item, *key, *value;
  SharedObject *shkey, *shvalue;
  Entry *table;

  /* Preallocate the list of tuples, to avoid allocations during
   * the loop over the items, which could trigger GC, which                  750
   * could resize the dict. :-(
   */
 again:
```

```
  used = self->used;
  list = PyList_New(used);
  if (list == NULL)
        return NULL;
  for (i = 0; i < used; i++) {
        item = PyTuple_New(2);
        if (item == NULL) {                                          760
          Py_DECREF(list);
          return NULL;
        }
        PyList_SET_ITEM(list, i, item);
  }
  if (used != self->used) {
        /* The allocations caused the dict to resize - start over. */
        Py_DECREF(list);
        goto again;
  }                                                                  770

  table = (Entry *) SharedMemHandle_AsVoidPtr(self->tableh);
  assert(table != NULL);

  for (i = 0, j = 0; i <= self->mask; i++) {
        if (table[i].state == ES_INUSE) {
          /* Get the key and value objects from the entry's handles */
          shkey = (SharedObject *) SharedMemHandle_AsVoidPtr(table[i].keyh);
          shvalue = (SharedObject *) SharedMemHandle_AsVoidPtr(table[i].valueh);
          assert(key != NULL && value != NULL);                    780

          /* Encapsulate the key and value in proxy objects */
          key = MakeProxy(shkey);
          if(key == NULL) {
                Py_DECREF(list);
                return NULL;
          }
          value = MakeProxy(shvalue);
          if(value == NULL) {
                Py_DECREF(key);                                    790
                Py_DECREF(list);
                return NULL;
          }

          /* Insert the (key, value) tuple in the list */
          item = PyList_GET_ITEM(list, j);
          PyTuple_SET_ITEM(item, 0, key);
          PyTuple_SET_ITEM(item, 1, value);
          j++;
        }                                                          800
  }
  assert(j == used);
  return list;
}


static PyObject *
```

```
dict_update(PyObject *self, PyObject *other)
{
  if(mapping_update(self, other))                                    810
        return NULL;
  Py_INCREF(Py_None);
  return Py_None;
}


static PyObject *
dict_copy(PyObject *self, PyObject *noarg)
{
  PyObject *result = PyDict_New();                                   820

  if(result != NULL) {
        /* Here, result is a plain dict, so we use PyDict_Update(), and not
           the generic mapping_update() */
        if(PyDict_Update(result, self)) {
          Py_DECREF(result);
          result = NULL;
        }
  }
  return result;                                                     830
}


static PyObject *
dict_popitem(PyObject *self_, PyObject *noargs)
{
  SharedDictBaseObject *self = (SharedDictBaseObject *) self_;
  PyObject *res, *proxy;
  SharedObject *key, *value;
  Entry *table, *ep;                                                 840
  int i = 0;

  /* Allocate the result tuple before checking the size. This is because
         of the possible side effects (garbage collection) of the allocation. */
  res = PyTuple_New(2);
  if (res == NULL)
        return NULL;
  if (self->used == 0) {
        PyErr_SetString(PyExc_KeyError, "popitem(): dictionary is empty");
        goto Error;                                                  850
  }

  table = SharedMemHandle_AsVoidPtr(self->tableh);
  assert(table != NULL);

  /* We abuse the hash field of slot 0 to hold a search finger, just like
         the implementation of normal dictionaries. */
  ep = table;
  if (ep->state == ES_INUSE)
        /* Return slot 0 */ ;                                       860
  else {
```

```
        /* Search the remaining slots for a used slot, starting at
           the hash value of slot 0, which may or may not have been
           stored by a previous popitem(). In any case, it works, so
           this is just an optimization to cater to repeated popitem()
           calls. */
        i = (int) ep->hash;

        if (i > self->mask || i < 1)
          i = 1;        /* skip slot 0 */                              870
        while ((ep = &table[i])->state != ES_INUSE) {
          if (++i > self->mask)
                i = 1;
        }
  }

  /* Now put the key and value that ep points to in the result tuple,
         wrapped in proxy objects. */
  key = (SharedObject *) SharedMemHandle_AsVoidPtr(ep->keyh);
  value = (SharedObject *) SharedMemHandle_AsVoidPtr(ep->valueh);          880
  assert(key != NULL && value != NULL);

  proxy = MakeProxy(key);
  if(proxy == NULL)
        goto Error;
  PyTuple_SET_ITEM(res, 0, proxy);
  proxy = MakeProxy(value);
  if(proxy == NULL)
        goto Error; /* This does DECREF the previous proxy, since it is
                                  already stored in the result tuple */   890
  PyTuple_SET_ITEM(res, 1, proxy);

  /* The result tuple is safely constructed - now clear the slot */
  ep->state = ES_DELETED;
  self->used--;

  table[0].hash = i + 1; /* next place to start */

  /* Decref the removed key+value */
  SharedObject_DecRef(key);                                              900
  SharedObject_DecRef(value);
  return res;

 Error:
  Py_DECREF(res);
  return NULL;
}


static PyMappingMethods dict_tp_as_mapping = {                           910
  dict_mp_length, /* mp_length */
  dict_mp_subscript, /* mp_subscript */
  dict_mp_ass_sub, /* mp_ass_subscript */
};
```

```
/* Hack to implement "key in dict" */
static PySequenceMethods dict_tp_as_sequence = {
        0,                                      /* sq_length */
        0,                                      /* sq_concat */                 920
        0,                                      /* sq_repeat */
        0,                                      /* sq_item */
        0,                                      /* sq_slice */
        0,                                      /* sq_ass_item */
        0,                                      /* sq_ass_slice */
        dict_sq_contains,  /* sq_contains */
        0,                                      /* sq_inplace_concat */
        0,                                      /* sq_inplace_repeat */
};
                                                                                930

static char has_key_doc[] =
"D.has_key(k) -> 1 if D has a key k, else 0";

static char get_doc[] =
"D.get(k[,d]) -> D[k] if D.has_key(k), else d. d defaults to None.";

static char setdefault_doc[] =
"D.setdefault(k[,d]) -> D.get(k,d), also set D[k]=d if not D.has_key(k)";
                                                                                940
static char popitem_doc[] =
"D.popitem() -> (k, v), remove and return some (key, value) pair as a\n\
2-tuple; but raise KeyError if D is empty";

static char keys_doc[] =
"D.keys() -> list of D's keys";

static char items_doc[] =
"D.items() -> list of D's (key, value) pairs, as 2-tuples";
                                                                                950
static char values_doc[] =
"D.values() -> list of D's values";

static char update_doc[] =
"D.update(E) -> None. Update D from E: for k in E.keys(): D[k] = E[k]";

static char clear_doc[] =
"D.clear() -> None. Remove all items from D.";

static char copy_doc[] =                                                        960
"D.copy() -> a shallow copy of D";

static char iterkeys_doc[] =
"D.iterkeys() -> an iterator over the keys of D";

static char itervalues_doc[] =
"D.itervalues() -> an iterator over the values of D";

static char iteritems_doc[] =
```

```
"D.iteritems() -> an iterator over the (key, value) items of D";                    970

static PyMethodDef dict_tp_methods[] = {
  {"has_key", dict_has_key, METH_O, has_key_doc},
  {"get", dict_get, METH_VARARGS, get_doc},
  {"setdefault", dict_setdefault, METH_VARARGS, setdefault_doc},
  {"popitem", dict_popitem, METH_NOARGS, popitem_doc},
  {"keys", dict_keys, METH_NOARGS, keys_doc},
  {"items",    dict_items, METH_NOARGS, items_doc},
  {"values", dict_values, METH_NOARGS, values_doc},
  {"update", dict_update, METH_O, update_doc},                                      980
  {"clear",    dict_clear, METH_NOARGS, clear_doc},
  {"copy", dict_copy, METH_NOARGS, copy_doc},
  /* {"iterkeys", dict_iterkeys, METH_NOARGS, iterkeys_doc}, */
  /* {"itervalues", dict_itervalues, METH_NOARGS, itervalues_doc}, */
  /* {"iteritems",     dict_iteritems, METH_NOARGS, iteritems_doc}, */
  {NULL, NULL} /* sentinel */
};


static char dict_tp_doc[] =                                                         990
"Abstract base class for shared dictionaries";

PyTypeObject SharedDictBase_Type = {
        PyObject_HEAD_INIT(&PyType_Type)
        0,
        "posh._core.SharedDictBase",
        sizeof(SharedDictBaseObject),
        0,
        dict_tp_dealloc,    /* tp_dealloc */
        0,                          /* tp_print */                                  1000
        0,                                  /* tp_getattr */
        0,                                  /* tp_setattr */
        0,                                  /* tp_compare */
        dict_tp_repr,       /* tp_repr */
        0,                                  /* tp_as_number */
        &dict_tp_as_sequence, /* tp_as_sequence */
        &dict_tp_as_mapping, /* tp_as_mapping */
        dict_tp_nohash,    /* tp_hash */
        0,                                  /* tp_call */
        dict_tp_repr,       /* tp_str */                                            1010
        0,                      /* tp_getattro */
        0,                                  /* tp_setattro */
        0,                                  /* tp_as_buffer */
        Py_TPFLAGS_DEFAULT | Py_TPFLAGS_BASETYPE, /* tp_flags */
        dict_tp_doc,        /* tp_doc */
        0,                      /* tp_traverse */
        0,                          /* tp_clear */
        0,                          /* tp_richcompare */
        0,                                  /* tp_weaklistoffset */
        0,                                  /* tp_iter */                           1020
        0,                                  /* tp_iternext */
        dict_tp_methods,  /* tp_methods */
        0,                                  /* tp_members */
```

```
        0,                              /* tp_getset */
        0,                              /* tp_base */
        0,                              /* tp_dict */
        0,                              /* tp_descr_get */
        0,                              /* tp_descr_set */
        0,                              /* tp_dictoffset */
        dict_tp_init,       /* tp_init */                                   1030
        0,                              /* tp_alloc */
        dict_tp_new,        /* tp_new */
        0,                              /* tp_free */
};
```

A.25 SharedHeap.h

```
/* SharedHeap.h */

#ifndef SHAREDHEAP_H_INCLUDED
#define SHAREDHEAP_H_INCLUDED

#include "_core.h"
#include "SharedRegion.h"

/* The type object for SharedHeap objects */
extern PyTypeObject SharedHeap_Type;

/* SharedHeap instances have the following interface:

   alloc(size) -> address, size
   realloc(address, size) -> address, size
   free(address) -> None

   The allocation routines alloc() and realloc() may allocate more bytes
   than requested. The actual size of the allocated block should be returned.

   The addresses are passed as Address objects encapsulating a (void *).
   Out of memory should be signalled by returning an Address object that
   encapsulates NULL. Exceptions raised by the methods will be propagated
   in the normal way.
*/

#endif
```

A.26 SharedHeap.c

```
/* SharedHeap.c */

#include "SharedHeap.h"
#include "Address.h"
#include "Globals.h"
#include "Handle.h"
#include "Lock.h"

typedef unsigned int word_t;
#define WORD_SIZE sizeof(word_t)                                        10

#define MIN_ALLOC_SIZE (WORD_SIZE*16)
#define NOF_ALLOC_SIZES 10
#define MAX_ALLOC_SIZE ((MIN_ALLOC_SIZE << (NOF_ALLOC_SIZES-1)))

#define PAGE_SIZE (MAX_ALLOC_SIZE * 16)

/* Macro for setting a memory word at base+offset */
#define SET_WORD(base, offset, value) \
  *((word_t *) (((void *) base) + (offset))) = value                    20

/* Macro for getting a memory word at base+offset */
#define GET_WORD(base, offset) \
   *((word_t *) (((void *) base) + (offset)))

/* "Root" data structure for a shared heap */
typedef struct
{
  /* Handle to the first page in each of the NOF_ALLOC_SIZES page lists */
  SharedMemHandle head[NOF_ALLOC_SIZES];                                30
  /* One lock per page list, to protect the integrity of the links in
     the list.  Note that each page also has a separate lock to protect
     its list of allocation units. */
  Lock lock[NOF_ALLOC_SIZES];
} root_t;

/* C representation of a SharedHeap object */
typedef struct
{
  PyObject_HEAD                                                         40
  /* Pointer to the (shared) root data structure. We can refer
        to this data structure by a pointer, because a fork() will
        ensure that the region is attached to the same address in
        the child process. */
  root_t *root;
} SharedHeapObject;

/* Page data structure */
typedef struct
{                                                                       50
  /* Handle for the next page */
```

```
  SharedMemHandle next;
  /* Lock to protect the integrity of the linked list of allocation units */
  Lock lock;
  /* The number of allocation units in this page */
  word_t nof_units;
  /* The number of free allocation units in this page */
  word_t nof_free_units;
  /* Allocation unit size, and mask, which is ~(size-1) */
  word_t unit_size;                                                    60
  word_t unit_mask;
  /* Offset in bytes, from the start of the page, of the first free unit */
  word_t head;
  /* Actual data of the page */
  unsigned char data[1];
} page_t;


/* Allocates and initializes a new root data structure */
static root_t *                                                        70
new_root(void)
{
  SharedMemHandle rh;
  root_t *root;
  size_t size = sizeof(root_t);
  int i;

  rh = SharedRegion_New(&size);
  root = (root_t *) SharedMemHandle_AsVoidPtr(rh);
  if(root == NULL)                                                     80
    return NULL;
  for(i = 0; i < NOF_ALLOC_SIZES; i++) {
    root->head[i] = SharedMemHandle_NULL;
    Lock_Init(&root->lock[i]);
  }
  return root;
}

/* Allocates and initializes a page data structure */
static SharedMemHandle                                                 90
new_page(size_t size, word_t unit_size)
{
  word_t offset, nextoffset;
  SharedMemHandle rh;
  page_t *page;

  rh = SharedRegion_New(&size);
  if(SharedMemHandle_IS_NULL(rh))
    return rh;
  page = (page_t *) SharedMemHandle_AsVoidPtr(rh);                    100
  if(page == NULL)
    return SharedMemHandle_NULL;

  /* Initialize fields */
  page->next = SharedMemHandle_NULL;
```

```
  Lock_Init(&page->lock);
  page->unit_size = unit_size;
  page->unit_mask = ~(unit_size-1);
  /* Initialize head, which is the offset of the first free chunk */
  page->head = (offsetof(page_t, data)+unit_size-1) & page->unit_mask;
  /* Initalize the unit counts */
  page->nof_units = (size - page->head)/unit_size;
  page->nof_free_units = page->nof_units;

  /* Create linked list of free blocks */
  offset = page->head;
  nextoffset = offset+unit_size;
  while(nextoffset < size) {
    SET_WORD(page, offset, nextoffset);
    offset = nextoffset;
    nextoffset += unit_size;
  }
  /* Terminate the list with a 0 */
  SET_WORD(page, offset, 0);

  return rh;
}

/* Allocates an allocation unit from the given page.
   Returns NULL on error. */
static void *
alloc_unit(page_t *page)
{
  word_t free;

  /* Unlink the first free unit from the free list */
  Lock_Acquire(&page->lock);
  free = page->head;
  if(free == 0) {
    Lock_Release(&page->lock);
    return NULL;
  }
  page->head = GET_WORD(page, free);
  page->nof_free_units--;
  Lock_Release(&page->lock);

  /* Return a pointer to the unit */
  return ((void *) page) + free;
}

/* Allocates a chunk of memory from the given root structure. */
static void *
alloc_chunk(root_t *root, long *size_arg)
{
  unsigned int ndx;
  page_t *page, *prev_page;
  long size = *size_arg;

  /* Calculate ndx such that 2^(ndx) < size <= 2^(ndx+1) */
```

```
  for (ndx = 0; size > MIN_ALLOC_SIZE; ndx++)                          160
    size >>= 1;
  if (ndx >= NOF_ALLOC_SIZES) {
    /* Allocate very big chunks in shared memory regions of their own. */
    size_t *sz = (size_t *) size_arg;
    return SharedMemHandle_AsVoidPtr(SharedRegion_New(sz));
  }

  /* Find a non-empty page on the root->head[ndx] list */
  prev_page = NULL;
  Lock_Acquire(&root->lock[ndx]);                                      170
  page = (page_t *) SharedMemHandle_AsVoidPtr(root->head[ndx]);
  while (page != NULL && page->nof_free_units == 0) {
    prev_page = page;
    page = (page_t *) SharedMemHandle_AsVoidPtr(page->next);
  }
  /* Create and link in a new page if necessary */
  if (page == NULL) {
    SharedMemHandle pageh = new_page(PAGE_SIZE, MIN_ALLOC_SIZE << ndx);
    if (SharedMemHandle_IS_NULL(pageh)) {
      Lock_Release(&root->lock[ndx]);                                  180
      return NULL;
    }
    if (prev_page == NULL)
      root->head[ndx] = pageh;
    else
      prev_page->next = pageh;
    page = SharedMemHandle_AsVoidPtr(pageh);
  }
  Lock_Release(&root->lock[ndx]);
                                                                       190
  /* Allocate a unit from the page, and return the actual size
     of the allocated chunk, along with the pointer */
  *size_arg = page->unit_size;
  return alloc_unit(page);
}

/* Frees a chunk of memory allocated by alloc_chunk(). */
static void
free_chunk(void *ptr)
{                                                                      200
  page_t *page;

  /* Map the pointer to a handle, so we can find the beginning of
     the shared memory region, which is also the beginning of the page. */
  SharedMemHandle h = SharedMemHandle_FromVoidPtr(ptr);
  if (SharedMemHandle_IS_NULL(h)) {
    /* XXX: This is an (unexpected) error, which is swallowed. */
    return;
  }
  if (h.offset == 0) {                                                 210
    /* The only chunks allocated at offset 0 in a shared memory region
       are big chunks. Small chunks always have a positive offset due
       to the page header. */
```

```
    SharedRegion_Destroy(h);
    return;
  }
  page = (page_t *) (ptr - h.offset);

  /* Link the chunk into the front of the page's free list */
  Lock_Acquire(&page->lock);                                          220
  SET_WORD(ptr, 0, page->head);
  page->head = h.offset;
  page->nof_free_units++;
  Lock_Release(&page->lock);
}

/* Reallocates a chunk of memory in the given root structure.
   On error, this returns NULL, and ptr will still point to a
   valid chunk. */
static void *                                                         230
realloc_chunk(root_t *root, void *ptr, long *size_arg)
{
  page_t *page;
  void *newptr;
  long cur_size;

  /* Map the pointer to a handle, so we can find the beginning of
     the shared memory region, which is also the beginning of the page. */
  SharedMemHandle h = SharedMemHandle_FromVoidPtr(ptr);
  if (SharedMemHandle_IS_NULL(h))                                     240
    return NULL;
  page = (page_t *) (ptr - h.offset);

  /* Get the capacity of the existing allocation unit */
  cur_size = page->unit_size;

  /* Keep the existing unit if it is big enough, and not more
     than twice the size needed. */
  if (cur_size >= *size_arg) {
      /* The existing unit is big enough */                           250
    if (cur_size/4 < MIN_ALLOC_SIZE || cur_size/4 < *size_arg) {
      /* The existing unit is not too big */
      *size_arg = cur_size;
      return ptr;
    }
  }

  /* Allocate a new chunk, copy the contents over, and
       free the old chunk */
  newptr = alloc_chunk(root, size_arg);                               260
  if (newptr == NULL)
      return NULL;
  if (cur_size > *size_arg)
    cur_size = *size_arg;
  memcpy(newptr, ptr, cur_size);
  free_chunk(ptr);
  return newptr;
```

```
}

/**********************/                                              270
/* SharedHeap objects */
/**********************/

/* SharedHeap.__init__() method */
static int
heap_init(PyObject *self_, PyObject *args, PyObject *kwargs)
{
  SharedHeapObject *self = (SharedHeapObject *) self_;
  static char *kwlist[] = {NULL};
                                                                      280
  if (!PyArg_ParseTupleAndKeywords(args, kwargs, "", kwlist))
        return -1;
  self->root = new_root();
  if (self->root == NULL) {
        PyErr_SetString(ErrorObject, "couldn't allocate root data structure");
        return -1;
  }
  return 0;
}
                                                                      290
/* SharedHeap.alloc() method */
static PyObject *
heap_alloc(PyObject *self_, PyObject *args)
{
  SharedHeapObject *self = (SharedHeapObject *) self_;
  long size;
  void *ptr;

  if(!PyArg_ParseTuple(args, "l:alloc", &size))
        return NULL;                                                  300
  ptr = alloc_chunk(self->root, &size);
  return Py_BuildValue("(Nl)", Address_FromVoidPtr(ptr), size);
}

/* SharedHeap.free() method */
static PyObject *
heap_free(PyObject *self, PyObject *args)
{
  PyObject *addr;
  void *ptr;                                                          310

  if(!PyArg_ParseTuple(args, "O!:free", &Address_Type, &addr))
        return NULL;
  ptr = Address_AsVoidPtr(addr);
  if(ptr != NULL)
        free_chunk(ptr);
  Py_INCREF(Py_None);
  return Py_None;
}
                                                                      320
/* SharedHeap.realloc() method */
```

```
static PyObject *
heap_realloc(PyObject *self_, PyObject *args)
{
  SharedHeapObject *self = (SharedHeapObject *) self_;
  PyObject *addr;
  long size;
  void *ptr;

  if(!PyArg_ParseTuple(args, "O!l:realloc", &Address_Type, &addr, &size))
      return NULL;
  ptr = Address_AsVoidPtr(addr);
  ptr = realloc_chunk(self->root, ptr, &size);
  return Py_BuildValue("(Nl)", Address_FromVoidPtr(ptr), size);
}

static char heap_alloc_doc[] =
"heap.alloc(size) -> address, size -- allocate a block of memory";
static char heap_free_doc[] =
"heap.free(address) -- free an allocated block";
static char heap_realloc_doc[] =
"heap.realloc(address, size) -> address, size -- reallocate a block of memory";

static PyMethodDef heap_methods[] = {
      {"alloc", heap_alloc, METH_VARARGS, heap_alloc_doc},
      {"free", heap_free, METH_VARARGS, heap_free_doc},
      {"realloc", heap_realloc, METH_VARARGS, heap_realloc_doc},
      {NULL, NULL} /* sentinel */
};

static char heap_doc[] =
"SharedHeap() -> new shared heap";

PyTypeObject SharedHeap_Type = {
      PyObject_HEAD_INIT(&PyType_Type)
      0,
      "posh._core.SharedHeap",
      sizeof(SharedHeapObject),
      0,
      0,                    /* tp_dealloc */
      0,                          /* tp_print */
      0,                               /* tp_getattr */
      0,                               /* tp_setattr */
      0,                               /* tp_compare */
      0,                          /* tp_repr */
      0,                               /* tp_as_number */
      0,                          /* tp_as_sequence */
      0,                               /* tp_as_mapping */
      0,                          /* tp_hash */
      0,                               /* tp_call */
      0,                               /* tp_str */
      0,                /* tp_getattro */
      0,                               /* tp_setattro */
      0,                               /* tp_as_buffer */
      Py_TPFLAGS_DEFAULT | Py_TPFLAGS_BASETYPE, /* tp_flags */
```

```
        heap_doc,           /* tp_doc */
        0,                  /* tp_traverse */
        0,                  /* tp_clear */
        0,                  /* tp_richcompare */
        0,                                  /* tp_weaklistoffset */                 380
        0,                                  /* tp_iter */
        0,                                  /* tp_iternext */
        heap_methods,       /* tp_methods */
        0,                                  /* tp_members */
        0,                                  /* tp_getset */
        0,                                  /* tp_base */
        0,                                  /* tp_dict */
        0,                                  /* tp_descr_get */
        0,                                  /* tp_descr_set */
        0,                                  /* tp_dictoffset */                     390
        heap_init,                      /* tp_init */
        0,                  /* tp_alloc */
        PyType_GenericNew,  /* tp_new */
        0,                  /* tp_free */
};
```

A.27 SharedListAndTuple.h

```
/* SharedListAndTuple.h */

#ifndef SHAREDLISTANDTUPLE_H_INCLUDED
#define SHAREDLISTANDTUPLE_H_INCLUDED

#include "_core.h"

extern PyTypeObject SharedListBase_Type;
extern PyTypeObject SharedTupleBase_Type;
                                                                        10
#define SharedListBase_Check(ob) PyObject_TypeCheck(ob, &SharedListBase_Type)
#define SharedTupleBase_Check(ob) PyObject_TypeCheck(ob, &SharedTupleBase_Type)

#endif
```

A.28 SharedListAndTuple.c

```
/* SharedListAndTuple.c */

#include "SharedListAndTuple.h"
#include "Handle.h"
#include "SharedObject.h"
#include "SharedAlloc.h"
#include "share.h"

typedef struct
{
  PyObject_VAR_HEAD
  int capacity;
  SharedMemHandle vectorh;
} SharedListBaseObject;

typedef struct
{
  PyObject_VAR_HEAD
  SharedMemHandle vector[1];
} SharedTupleBaseObject;

/*********************************************/
/* COMMON ROUTINES FOR SHARED LISTS AND TUPLES */
/*********************************************/

/* Parses an argument list consisting of a single sequence argument */
static int
arg_sequence(PyObject *args, char *funcname, PyObject **seq, int *size)
{
  static char fmt[110];
  PyObject *obj;

  PyOS_snprintf(fmt, sizeof(fmt), "O:%.100s", funcname);
  if(!PyArg_ParseTuple(args, fmt, &obj))
        return -1;
  if(!PySequence_Check(obj)) {
        PyErr_Format((PyObject *) PyExc_TypeError,
                              "%.100s expects a sequence", funcname);
        return -1;
  }
  *seq = obj;
  *size = PySequence_Length(obj);
  return 0;
}

static int
common_sq_length(PyObject *self_)
{
  PyVarObject *self = (PyVarObject *) self_;
  return self->ob_size;
}
```

```
/* VECTOR IMPLEMENTATION
   A vector in this context (this file) is an array of handles to
   shared objects. Both SharedListBase and SharedTupleBase objects
   are implemented using a vector. SharedListBase objects contain
   a handle to a vector, while SharedListTuple objects, which are
   immutable, have a vector embedded in them.
 */

/* Deinitializes a vector of shared objects */
static void
vector_deinit(SharedMemHandle *vector, int size)
{
  int i;
  SharedObject *obj;

  for(i = 0; i < size; i++) {
        obj = (SharedObject *) SharedMemHandle_AsVoidPtr(vector[i]);
        SharedObject_DecRef(obj);
  }
}

/* Initializes a vector of shared objects from a sequence */
static int
vector_init(SharedMemHandle *vector, PyObject *seq, int size)
{
  /* New references */
  PyObject *item;
  SharedObject *shared_item;

  int i;
  size_t bytes;

  /* Fill in the vector by sharing the items of the sequence */
  for(i = 0; i < size; i++) {
        item = PySequence_GetItem(seq, i);
        if(item == NULL)
          goto Error;
        shared_item = ShareObject(item);
        Py_DECREF(item);
        if(shared_item == NULL)
          goto Error;
        SharedObject_IncRef(shared_item);
        vector[i] = SharedMemHandle_FromVoidPtr(shared_item);
  }
  return 0;

 Error:
  vector_deinit(vector, i-1);
  return -1;
}

static PyObject *
vector_item(SharedMemHandle *vector, int size, int index)
```

```
{
  SharedObject *obj;

  if(index < 0)
        index += size;                                                    110
  if(index < 0 || index >= size) {
        PyErr_SetString(PyExc_IndexError, "index out of range");
        return NULL;
  }
  obj = (SharedObject *) SharedMemHandle_AsVoidPtr(vector[index]);
  assert(obj != NULL);
  return MakeProxy(obj);
}

static int                                                                120
vector_ass_item(SharedMemHandle *vector, int size, int index, PyObject *value)
{
  SharedObject *olditem, *newitem;

  if(index < 0)
      index += size;
  if(index < 0 || index >= size) {
      PyErr_SetString((PyObject *) &PyExc_IndexError, "index out of range");
      return -1;
  }                                                                       130
  newitem = ShareObject(value);
  if(newitem == NULL)
        return -1;
  SharedObject_IncRef(newitem);
  olditem = (SharedObject*) SharedMemHandle_AsVoidPtr(vector[index]);
  SharedObject_DecRef(olditem);
  vector[index] = SharedMemHandle_FromVoidPtr(newitem);
  return 0;
}
                                                                          140
/*********************************/
/* SharedListBase IMPLEMENTATION */
/*********************************/

/* Resizes the vector of a list, if necessary, to accomodate
   the new size. Accepts a pointer to the list's vector, which
   can be NULL if unknown. Returns a pointer to the list's vector,
   which may be reallocated.
   Returns NULL if resized to 0 or if out of memory.
                                                                          150
   This over-allocates by a factor of 50% to amortize reallocation
   costs over time when growing the list. */
static SharedMemHandle *
list_resize(SharedListBaseObject *self, int newsize, SharedMemHandle *vector)
{
  size_t bytes;
  int newcapacity;
  SharedMemHandle *newvector;
  PyObject *self_ = (PyObject *) self;
```

```
                                                                        160
  if(vector == NULL)
        vector = (SharedMemHandle *) SharedMemHandle_AsVoidPtr(self->vectorh);

  /* XXX: Never downsize for now */
  if(newsize > 0 && newsize <= self->capacity) {
        self->ob_size = newsize;
        return vector;
  }

  if(newsize <= 0) {                                                    170
        /* Free the vector */
        if(vector != NULL)
          SharedFree(self_, vector);
        self->capacity = 0;
        self->ob_size = 0;
        self->vectorh = SharedMemHandle_NULL;
        return NULL;
  }

  /* XXX: Round up this in some way */                                  180
  newcapacity = (newsize*3)/2;

  /* Allocate more memory for the vector */
  bytes = newcapacity * sizeof(SharedMemHandle);
  newvector = (SharedMemHandle *) SharedRealloc(self_, vector, &bytes);
  if(newvector == NULL)
        return NULL;

  /* SharedRealloc() may actually have allocated more bytes than requested */
  self->capacity = bytes / sizeof(SharedMemHandle);                     190
  self->ob_size = newsize;
  self->vectorh = SharedMemHandle_FromVoidPtr(newvector);
  return newvector;
}

/* Creates an empty shared list */
static PyObject *
list_tp_new(PyTypeObject *type, PyObject *args, PyObject *kwds)
{
  SharedListBaseObject *self;                                           200

  if(type == &SharedListBase_Type) {
        PyErr_Format(PyExc_TypeError, "cannot create '%.100s' instances",
                     type->tp_name);
        return NULL;
  }
  self = (SharedListBaseObject *) type->tp_alloc(type, 0);
  if(self != NULL) {
        /* Set the list to be empty (a consistent state) */
        assert(self->ob_size == 0);                                     210
        assert(self->capacity == 0);
        self->vectorh = SharedMemHandle_NULL;
  }
```

```
  return (PyObject *) self;
}

/* Initializes a shared list from a sequence */
static int
list_tp_init(PyObject *self_, PyObject *args, PyObject *kwds)
{                                                                    220
  SharedListBaseObject *self = (SharedListBaseObject *) self_;
  PyObject *seq;
  int size;
  SharedMemHandle *vector;

  /* Parse arguments */
  if(arg_sequence(args, "SharedListBase.__init__", &seq, &size))
        return -1;

  /* Resize the list */                                              230
  vector = list_resize(self, size, NULL);
  if(vector == NULL && size > 0)
        return -1;
  /* Initialize the list's vector */
  if(vector_init(vector, seq, size)) {
        list_resize(self, 0, vector);
        return -1;
  }
  return 0;
}                                                                    240

static void
list_tp_dealloc(PyObject *self_)
{
  SharedListBaseObject *self = (SharedListBaseObject *) self_;
  SharedMemHandle *vector;

  vector = (SharedMemHandle *) SharedMemHandle_AsVoidPtr(self->vectorh);
  if(vector != NULL) {
        vector_deinit(vector, self->ob_size);                        250
        SharedFree(self_, vector);
  }
  self->ob_type->tp_free(self_);
}

static long
list_tp_nohash(PyObject *self)
{
  PyErr_Format(PyExc_TypeError, "%.100s objects are unhashable",
                          self->ob_type->tp_name);                   260
  return -1;
}

static PyObject *
list_tp_repr(PyObject *self_)
{
  SharedListBaseObject *self = (SharedListBaseObject *) self_;
```

```
SharedMemHandle vectorh = SharedMemHandle_NULL;
SharedMemHandle *vector = NULL;
PyObject *s, *temp;                                                    270
PyObject *pieces = NULL, *result = NULL;
int i;

i = Py_ReprEnter(self_);
if (i != 0) {
      /* Recursive data structure */
      return i > 0 ? PyString_FromString("[...]") : NULL;
}

if (self->ob_size == 0) {                                              280
      /* Empty list */
      result = PyString_FromString("[]");
      goto Done;
}

pieces = PyList_New(0);
if (pieces == NULL)
      goto Done;

/* Do repr() on each element. Note that this may mutate the list,       290
       so we must refetch the list size on each iteration. */
for (i = 0; i < self->ob_size; i++) {
      int status;
      SharedObject *item;

      /* Check if the vector handle has changed */
      if(!SharedMemHandle_EQUAL(vectorh, self->vectorh)) {
        vectorh = self->vectorh;
        vector = (SharedMemHandle *) SharedMemHandle_AsVoidPtr(vectorh);
        assert(vector != NULL);                                        300
      }

      /* pieces.append(repr(self[i])) */
      item = (SharedObject *) SharedMemHandle_AsVoidPtr(vector[i]);
      assert(item != NULL);
      s = SharedObject_Repr(item);
      if (s == NULL)
        goto Done;
      status = PyList_Append(pieces, s);
      Py_DECREF(s); /* append created a new ref */                     310
      if (status < 0)
        goto Done;
}

/* Add "[]" decorations to the first and last items. */
assert(PyList_GET_SIZE(pieces) > 0);
s = PyString_FromString("[");
if (s == NULL)
      goto Done;
temp = PyList_GET_ITEM(pieces, 0);                                     320
PyString_ConcatAndDel(&s, temp);
```

```
  PyList_SET_ITEM(pieces, 0, s);
  if (s == NULL)
        goto Done;

  s = PyString_FromString("]");
  if (s == NULL)
        goto Done;
  temp = PyList_GET_ITEM(pieces, PyList_GET_SIZE(pieces) - 1);
  PyString_ConcatAndDel(&temp, s);                                    330
  PyList_SET_ITEM(pieces, PyList_GET_SIZE(pieces) - 1, temp);
  if (temp == NULL)
        goto Done;

  /* Paste them all together with ", " between. */
  s = PyString_FromString(", ");
  if (s == NULL)
        goto Done;
  result = _PyString_Join(s, pieces);
  Py_DECREF(s);                                                       340

 Done:
  Py_XDECREF(pieces);
  Py_ReprLeave(self_);
  return result;
}

static PyObject *
list_sq_item(PyObject *self_, int index)
{                                                                     350
  SharedListBaseObject *self = (SharedListBaseObject *) self_;
  SharedMemHandle *vector;

  vector = (SharedMemHandle *) SharedMemHandle_AsVoidPtr(self->vectorh);
  assert(vector != NULL);
  return vector_item(vector, self->ob_size, index);
}

static int
list_sq_ass_item(PyObject *self_, int index, PyObject *value)         360
{
  SharedListBaseObject *self = (SharedListBaseObject *) self_;
  SharedMemHandle *vector;

  vector = (SharedMemHandle *) SharedMemHandle_AsVoidPtr(self->vectorh);
  assert(vector != NULL);
  return vector_ass_item(vector, self->ob_size, index, value);
}

static int                                                            370
list_sq_ass_slice(PyObject *self_, int ilow, int ihigh, PyObject *v)
{
  SharedListBaseObject *self = (SharedListBaseObject *) self_;
  return -1;
}
```

```
static PyObject *
list_append(PyObject *self_, PyObject *value)
{
  SharedListBaseObject *self = (SharedListBaseObject *) self_;                380
  SharedMemHandle *vector;
  int last = self->ob_size;

  vector = list_resize(self, last+1, NULL);
  if(vector == NULL)
        return NULL;
  if(vector_ass_item(vector, last+1, last, value)) {
        list_resize(self, last, vector);
        return NULL;
  }                                                                           390
  Py_INCREF(Py_None);
  return Py_None;
}

static PyObject *
list_pop(PyObject *self_, PyObject *args)
{
  SharedListBaseObject *self = (SharedListBaseObject *) self_;
  int index = -1;
                                                                              400
  if(!PyArg_ParseTuple(args, "|i:pop", &index))
        return NULL;
  return NULL;
}

static PyObject *
list_insert(PyObject *self_, PyObject *args)
{
  SharedListBaseObject *self = (SharedListBaseObject *) self_;
  int index;                                                                  410
  PyObject *item;

  if(!PyArg_ParseTuple(args, "iO", &index, &item))
        return NULL;
  return NULL;
}

static PyObject *
list_remove(PyObject *self_, PyObject *value)
{                                                                             420
  SharedListBaseObject *self = (SharedListBaseObject *) self_;
  return NULL;
}

static char list_append_doc[] =
"L.append(object) -- append object to end";
static char list_pop_doc[] =
"L.pop([index]) -> item -- remove and return item at index (default last)";
static char list_insert_doc[] =
```

```
"L.insert(index, object) -- insert object before index";
static char list_remove_doc[] =
"L.remove(value) -- remove first occurrence of value";

static PyMethodDef list_tp_methods[] = {
  {"append", list_append, METH_O, list_append_doc},
  {"pop", list_pop, METH_VARARGS, list_pop_doc},
  {"insert", list_insert, METH_VARARGS, list_insert_doc},
  {"remove", list_remove, METH_O, list_remove_doc},
  {NULL, NULL} /* sentinel */
};

static PySequenceMethods list_tp_as_sequence = {
  common_sq_length, /* sq_length */
  0,                /* sq_concat */
  0,                /* sq_repeat */
  list_sq_item,     /* sq_item */
  0,                /* sq_slice */
  list_sq_ass_item, /* sq_ass_item */
  list_sq_ass_slice, /* sq_ass_slice */
  0,                /* sq_contains */
  0,                /* sq_inplace_concat */
  0,                /* sq_inplace_repeat */
};

static char list_tp_doc[] =
"Abstract base class for shared lists";

PyTypeObject SharedListBase_Type = {
        PyObject_HEAD_INIT(&PyType_Type)
        0,
        "posh._core.SharedListBase",
        sizeof(SharedListBaseObject),
        0,
        list_tp_dealloc,    /* tp_dealloc */
        0,                  /* tp_print */
        0,                  /* tp_getattr */
        0,                  /* tp_setattr */
        0,                  /* tp_compare */
        list_tp_repr,       /* tp_repr */
        0,                  /* tp_as_number */
        &list_tp_as_sequence, /* tp_as_sequence */
        0,                  /* tp_as_mapping */
        list_tp_nohash,     /* tp_hash */
        0,                  /* tp_call */
        list_tp_repr,       /* tp_str */
        0,                  /* tp_getattro */
        0,                  /* tp_setattro */
        0,                  /* tp_as_buffer */
        Py_TPFLAGS_DEFAULT | Py_TPFLAGS_BASETYPE, /* tp_flags */
        list_tp_doc,        /* tp_doc */
        0,                  /* tp_traverse */
        0,                  /* tp_clear */
        0,                  /* tp_richcompare */
```

```
        0,                                      /* tp_weaklistoffset */
        0,                                      /* tp_iter */
        0,                                      /* tp_iternext */
        list_tp_methods,   /* tp_methods */
        0,                                      /* tp_members */
        0,                                      /* tp_getset */
        0,                                      /* tp_base */                    490
        0,                                      /* tp_dict */
        0,                                      /* tp_descr_get */
        0,                                      /* tp_descr_set */
        0,                                      /* tp_dictoffset */
        list_tp_init,         /* tp_init */
        0,                              /* tp_alloc */
        list_tp_new,          /* tp_new */
        0,                              /* tp_free */
};
                                                                             500
/*********************************/
/* SharedTupleBase IMPLEMENTATION */
/*********************************/

/* Creates a shared tuple */
static PyObject *
tuple_tp_new(PyTypeObject *type, PyObject *args, PyObject *kwds)
{
  SharedTupleBaseObject *self;
  PyObject *seq;                                                             510
  int size;

  if(type == &SharedTupleBase_Type) {
        PyErr_Format(PyExc_TypeError, "cannot create '%.100s' instances",
                             type->tp_name);
        return NULL;
  }
  if(arg_sequence(args, "SharedTupleBase.__new__", &seq, &size))
        return NULL;
                                                                             520
  self = (SharedTupleBaseObject *) type->tp_alloc(type, size);
  if(self != NULL) {
      if(vector_init(self->vector, seq, size))
          return NULL;
  }
  return (PyObject *) self;
}


static void                                                                  530
tuple_tp_dealloc(PyObject *self_)
{
  SharedTupleBaseObject *self = (SharedTupleBaseObject *) self_;

  vector_deinit(self->vector, self->ob_size);
  self->ob_type->tp_free(self_);
}
```

```
static PyObject *                                                    540
tuple_sq_item(PyObject *self_, int index)
{
  SharedTupleBaseObject *self = (SharedTupleBaseObject *) self_;

  return vector_item(self->vector, self->ob_size, index);
}


/* Hash function for tuples, directly adapted from Objects/tupleobject.c */
static long                                                          550
tuple_tp_hash(PyObject *self_)
{
  SharedTupleBaseObject *self = (SharedTupleBaseObject *) self_;
  long x, y;
  int len = self->ob_size;
  SharedMemHandle *p;
  SharedObject *item;

  x = 0x345678L;
  for(p = self->vector; --len >= 0; p++) {                           560
        item = (SharedObject *) SharedMemHandle_AsVoidPtr(*p);
        assert(item != NULL);
        y = SharedObject_Hash(item);
        if (y == -1)
          return -1;
        x = (1000003*x) ^ y;
  }
  x ^= self->ob_size;
  if (x == -1)
        x = -2;                                                      570
  return x;
}

static PyObject *
tuple_tp_richcompare(PyObject *self_, PyObject *other, int op)
{
  SharedTupleBaseObject *self = (SharedTupleBaseObject *) self_;
  int i, k, selflen, otherlen;
  int other_is_shared;
  SharedObject *a = NULL;                                            580
  PyObject *b = NULL;

  /* For 'other', we accept either a plain tuple or a shared tuple */
  other_is_shared = SharedTupleBase_Check(other);
  if (!other_is_shared && !PyTuple_Check(other)) {
        Py_INCREF(Py_NotImplemented);
        return Py_NotImplemented;
  }

  selflen = self->ob_size;                                           590
  otherlen = ((PyVarObject *) other)->ob_size;
```

```
/* Search for the first index where items are different.
 * Note that because tuples are immutable, it's safe to reuse
 * selflen and otherlen across the comparison calls.
 */
for (i = 0; i < selflen && i < otherlen; i++) {
      /* a = self[i] */
      a = (SharedObject *) SharedMemHandle_AsVoidPtr(self->vector[i]);
      if (a == NULL)                                                    600
        return NULL;

      /* b = other[i] */
      if (other_is_shared) {
        SharedTupleBaseObject *o = (SharedTupleBaseObject *) other;
        SharedObject *shb = (SharedObject *)
              SharedMemHandle_AsVoidPtr(o->vector[i]);
        if (shb == NULL)
              return NULL;
        b = SharedObject_AS_PYOBJECT(shb);                              610
      }
      else
        b = PyTuple_GET_ITEM(other, i);

      /* Compare a and b */
      k = SharedObject_RichCompareBool(a, b, Py_EQ);
      if (k < 0)
        return NULL;
      if (!k)
        break;                                                          620
}

if (i >= selflen || i >= otherlen) {
      /* No more items to compare -- compare sizes */
      int cmp;
      PyObject *res;
      switch (op) {
      case Py_LT: cmp = selflen < otherlen; break;
      case Py_LE: cmp = selflen <= otherlen; break;
      case Py_EQ: cmp = selflen == otherlen; break;                     630
      case Py_NE: cmp = selflen != otherlen; break;
      case Py_GT: cmp = selflen > otherlen; break;
      case Py_GE: cmp = selflen >= otherlen; break;
      default: return NULL; /* cannot happen */
      }
      if (cmp)
        res = Py_True;
      else
        res = Py_False;
      Py_INCREF(res);                                                   640
      return res;
}

/* We have an item that differs -- shortcuts for EQ/NE */
if (op == Py_EQ) {
```

```
        Py_INCREF(Py_False);
        return Py_False;
    }
    if (op == Py_NE) {
        Py_INCREF(Py_True);
        return Py_True;
    }

    /* Compare the final item again using the proper operator */
    return SharedObject_RichCompare(a, b, op);
}


static PySequenceMethods tuple_tp_as_sequence = {
        common_sq_length, /* sq_length */
        0,          /* sq_concat */
        0,          /* sq_repeat */
        tuple_sq_item, /* sq_item */
        0,          /* sq_slice */
        0,          /* sq_ass_item */
        0,          /* sq_ass_slice */
        0,          /* sq_contains */
        0,          /* sq_inplace_concat */
        0,          /* sq_inplace_repeat */
};

static char tuple_tp_doc[] =
"Abstract base class for shared tuples";

PyTypeObject SharedTupleBase_Type = {
        PyObject_HEAD_INIT(&PyType_Type)
        0,
        "posh._core.SharedTupleBase",
        sizeof(SharedTupleBaseObject) - sizeof(SharedMemHandle),
        sizeof(SharedMemHandle),
        tuple_tp_dealloc,  /* tp_dealloc */
        0,                 /* tp_print */
        0,                 /* tp_getattr */
        0,                 /* tp_setattr */
        0,                 /* tp_compare */
        0,                 /* tp_repr */
        0,                 /* tp_as_number */
        &tuple_tp_as_sequence, /* tp_as_sequence */
        0,                 /* tp_as_mapping */
        tuple_tp_hash,     /* tp_hash */
        0,                 /* tp_call */
        0,                 /* tp_str */
        0,                 /* tp_getattro */
        0,                 /* tp_setattro */
        0,                 /* tp_as_buffer */
        Py_TPFLAGS_DEFAULT | Py_TPFLAGS_BASETYPE, /* tp_flags */
        tuple_tp_doc,      /* tp_doc */
        0,                 /* tp_traverse */
        0,                 /* tp_clear */
```

```
    tuple_tp_richcompare, /* tp_richcompare */                    700
    0,                                  /* tp_weaklistoffset */
    0,                                  /* tp_iter */
    0,                                  /* tp_iternext */
    0,                  /* tp_methods */
    0,                                  /* tp_members */
    0,                                  /* tp_getset */
    0,                                  /* tp_base */
    0,                                  /* tp_dict */
    0,                                  /* tp_descr_get */
    0,                                  /* tp_descr_set */      710
    0,                                  /* tp_dictoffset */
    0,                  /* tp_init */
    0,                          /* tp_alloc */
    tuple_tp_new,       /* tp_new */
    0,                          /* tp_free */
};
```

A.29 SharedObject.h

```
/* SharedObject.h */

#ifndef SHAREDOBJECT_H_INCLUDED
#define SHAREDOBJECT_H_INCLUDED

#include "_core.h"
#include "Process.h"
#include "Handle.h"
#include "Spinlock.h"
#include "Lock.h"                                                    10

/* This is the C representation of a shared object, which is basically a
   normal Python object with additional data prepended to it. When shared
   objects are allocated, there is made room for the additional data
   that precedes the normal object structure. */
typedef struct {
  /* Lock for synchronizing access to the object. Note that the object's
        synchronization manager decides if and how to use this lock. */
  Lock lock;
  /* Handle to the object's shared dictionary */                     20
  SharedMemHandle dicth;
  /* Spinlock for protecting the proxybmp and srefcnt fields. */
  Spinlock reflock;
  /* Bitmap showing which processes have a proxy object for the object. */
  ProcessBitmap proxybmp;
  /* Shared reference count; this is the number of references to this
        object from other *shared* objects. References from proxy objects are
        not counted here, but reflected in the proxy bitmap above. */
  unsigned int srefcnt : sizeof(int)*8 - 2;
  /* Flags */                                                        30
  unsigned int is_corrupt : 1; /* True if the object may be corrupted */
  unsigned int no_synch : 1; /* True if no synch. is required on this object,
                                        i.e. type(type(self)).__synch__ is None */
  /* Start of normal PyObject structure */
  PyObject pyobj;
} __attribute__((packed)) SharedObject;

/* "Casting" macros (they do some pointer arithmethics too) */

#define SharedObject_FROM_PYOBJECT(ob) \                             40
  ((SharedObject *) (((void *) (ob)) - offsetof(SharedObject, pyobj)))

#define SharedObject_AS_PYOBJECT(shob) \
  (&(((SharedObject *) (shob))->pyobj))

#define SharedObject_AS_PYVAROBJECT(shob) \
  ((PyVarObject *) &(((SharedObject *) (shob))->pyobj))

/* Macro to calculate the number of bytes needed for a shared object */
#define SharedObject_VAR_SIZE(type, nitems) \                        50
  (_PyObject_VAR_SIZE(type, nitems) + offsetof(SharedObject, pyobj))
```

```
/* Initializes a newly created shared object, including the PyObject part
   of it. The object should be zeroed out already. */
void SharedObject_Init(SharedObject *obj, PyTypeObject *type, int nitems);

/* Increases the shared reference count of a shared object, indicating
   a new reference from another shared object to it. */
void SharedObject_IncRef(SharedObject *obj);

/* Decreases the shared reference count of a shared object, indicating
   that a reference to it from another shared object was destroyed.
   If the shared reference count reaches 0, and the process bitmap
   indicates that there are no proxy objects for the object, it will
   be reclaimed. */
void SharedObject_DecRef(SharedObject *obj);

/* Sets the bit in the shared object that indicates that this process has
   at least 1 proxy object referring to the shared object. */
void SharedObject_SetProxyBit(SharedObject *obj);

/* Clears a bit in the shared object, indicating that this process has no
   more proxy objects referring to the shared object. If all the bits
   in the process bitmap are cleared, and the shared reference count is 0,
   the object will be reclaimed. */
void SharedObject_ClearProxyBit(SharedObject *obj);

/* Calls the enter() method on the __synch__ attribute of a shared object's
   meta-type, acquiring access to the object. */
PyObject *SharedObject_Enter(SharedObject *obj, PyObject *opname);

/* Like SharedObject_Enter(), but opname is a C string, and *opname_cache,
   if non-NULL, will be used to cache the string object from call to call. */
PyObject *SharedObject_EnterString(SharedObject *obj, char *opname,
                                   PyObject **opname_cache);

/* Calls the leave() method on the __synch__ attribute of a shared object's
   meta-type, releasing access to the object. Ignores any exceptions that
   occur. If an exception is set prior to this call, it will be set on return,
   too. The 'state' argument should be the return value from
   SharedObject_Enter(). This function always steals a reference to 'state'. */
void SharedObject_Leave(SharedObject *obj, PyObject *state);

/* Attribute getter function for shared objects that support attributes.
   This is suitable for the tp_getattro slot of shareable types. */
PyObject *SharedObject_GetAttr(PyObject *obj, PyObject *name);

/* Attribute setter function for shared objects that support attributes.
   This is suitable for the tp_setattro slot of shareable types. */
int SharedObject_SetAttr(PyObject *obj, PyObject *name, PyObject *value);

/* The following functions are utility functions that do the same as their
   PyObject_Whatever counterparts, but also call SharedObject_Enter() and
   SharedObject_Leave() for you. */
```

```
/* Like PyObject_Repr() */
PyObject *SharedObject_Repr(SharedObject *obj);

/* Like PyObject_Str() */
PyObject *SharedObject_Str(SharedObject *obj);                    110

/* Like PyObject_Hash() */
long SharedObject_Hash(SharedObject *obj);

/* Like PyObject_Print() */
int SharedObject_Print(SharedObject *obj, FILE *fp, int flags);

/* Like PyObject_Compare() */
int SharedObject_Compare(SharedObject *a, PyObject *b);
                                                                  120
/* Like PyObject_RichCompare() */
PyObject *SharedObject_RichCompare(SharedObject *a, PyObject *b, int op);

/* Like PyObject_RichCompareBool() */
int SharedObject_RichCompareBool(SharedObject *a, PyObject *b, int op);

#endif
```

A.30 SharedObject.c

```
/* SharedObject.c */

#include "SharedObject.h"
#include "Process.h"
#include "Lock.h"
#include "Globals.h"
#include "share.h"

/* 'Neutral' value for normal reference count */
#define NEUTRAL_OB_REFCNT ((int) (1 << ((sizeof(int)*8-2))))


void
SharedObject_Init(SharedObject *obj, PyTypeObject *type, int nitems)
{
  PyObject *synch;

  /* Normal initialization of the PyObject part */
  if (type->tp_flags & Py_TPFLAGS_HEAPTYPE)
        Py_INCREF(type);
  if (type->tp_itemsize == 0)
        (void) PyObject_INIT(SharedObject_AS_PYOBJECT(obj), type);
  else
        (void) PyObject_INIT_VAR(SharedObject_AS_PYVAROBJECT(obj), type, nitems);

  /* Initialization specific to shared objects */

  /* The normal reference count has no meaning for shared objects, so
        we set it to a large number to prevent it from interfering with
        our own reference counting scheme. This makes it safe to pass shared
        objects to functions that are refcount-neutral. */
  obj->pyobj.ob_refcnt = NEUTRAL_OB_REFCNT;

  /* Shared objects start out with a shared reference count of 0. This
        indicates that there are no references to this object from other
        shared objects. The process bitmap also starts out blank, since there
        are no proxy objects referring to the object. Presumably, the
        initialization will be followed by either a SharedObject_IncRef() or a
        SharedObject_SetProxyBit().
  */
  assert(obj->srefcnt == 0);
  assert(ProcessBitmap_IS_ZERO(obj->proxybmp));

  /* Set the flags of the object according to its metatype */
  synch = PyObject_GetAttrString((PyObject *) type->ob_type, S_SYNCH);
  if (synch == NULL) {
        PyErr_Clear();
        obj->no_synch = 1;
  }
  else {
        if (synch == Py_None)
```

```
        obj->no_synch = 1;
      Py_DECREF(synch);
  }

  /* Initialize the remaining nonzero fields */
  Lock_Init(&obj->lock);
  Spinlock_Init(&obj->reflock);
  obj->dicth = SharedMemHandle_NULL;
}                                                                    60


static void
SharedObject_Dealloc(SharedObject *obj)
{
  PyObject *pyobj = SharedObject_AS_PYOBJECT(obj);

  Spinlock_Destroy(&obj->reflock);
  if (!SharedMemHandle_IS_NULL(obj->dicth)) {
      SharedObject *dict = (SharedObject *)                          70
        SharedMemHandle_AsVoidPtr(obj->dicth);
      if (dict != NULL) {
    printf("Decreffing shared dictionary at %p.\n", dict);
        SharedObject_DecRef(dict);
    }
      obj->dicth = SharedMemHandle_NULL;
  }
  pyobj->ob_refcnt = 1;
  Py_DECREF(pyobj);
}                                                                    80


void
SharedObject_IncRef(SharedObject *obj)
{
  Spinlock_Acquire(&obj->reflock);
  obj->srefcnt++;
  Spinlock_Release(&obj->reflock);
}
                                                                     90


void
SharedObject_DecRef(SharedObject *obj)
{
  int dealloc;

  Spinlock_Acquire(&obj->reflock);
  obj->srefcnt--;
  dealloc = (obj->srefcnt == 0 && ProcessBitmap_IS_ZERO(obj->proxybmp));
  Spinlock_Release(&obj->reflock);                                  100
  if (dealloc)
      SharedObject_Dealloc(obj);
}
```

```
void
SharedObject_SetProxyBit(SharedObject *obj)
{
  Spinlock_Acquire(&obj->reflock);
  ProcessBitmap_SET(obj->proxybmp, my_pindex);                              110
  Spinlock_Release(&obj->reflock);
}


void
SharedObject_ClearProxyBit(SharedObject *obj)
{
  int dealloc;

  Spinlock_Acquire(&obj->reflock);                                          120
  ProcessBitmap_CLEAR(obj->proxybmp, my_pindex);
  dealloc = (obj->srefcnt == 0 && ProcessBitmap_IS_ZERO(obj->proxybmp));
  Spinlock_Release(&obj->reflock);
  if (dealloc)
        SharedObject_Dealloc(obj);
}


PyObject *
SharedObject_Enter(SharedObject *obj, PyObject *opname)                     130
{
  /* Disallow calls on the object if it is corrupt */
  if (obj->is_corrupt)
        goto Corrupt;

  if (obj->no_synch) {
        Py_INCREF(Py_None);
        return Py_None;
  }
  else {                                                                    140
        PyObject *pyobj = SharedObject_AS_PYOBJECT(obj);
        PyObject *meta_type = (PyObject *) pyobj->ob_type->ob_type;
        PyObject *synch, *result;

        /* Get the __synch__ attribute of the meta-type */
        synch = PyObject_GetAttrString(meta_type, S_SYNCH);
        if (synch == NULL)
          return NULL;

        /* Call its enter() method */                                       150
        result = PyObject_CallMethod(synch, S_ENTER, "OO", pyobj, opname);

        /* The enter() method may have detected that the object is corrupt */
        if (obj->is_corrupt)
          goto Corrupt;

        return result;
  }
```

```
 Corrupt:                                                              160
  PyErr_SetString(ErrorObject, "shared object may be corrupt");
  return NULL;
}


PyObject *
SharedObject_EnterString(SharedObject *obj, char *opname,
                                              PyObject **opname_cache)
{
  if (opname_cache == NULL) {                                          170
        PyObject *str, *result;

        str = PyString_FromString(opname);
        if (str == NULL)
          return NULL;
        result = SharedObject_Enter(obj, str);
        Py_DECREF(str);
        return result;
  }
  else {                                                               180
        PyObject *str;

        if (*opname_cache == NULL) {
          str = PyString_FromString(opname);
          if (str == NULL)
                return NULL;
          *opname_cache = str;
        }
        else
          str = *opname_cache;                                         190
        return SharedObject_Enter(obj, str);
  }
}


void
SharedObject_Leave(SharedObject *obj, PyObject *state)
{
  if (obj->no_synch) {
        Py_DECREF(state);                                              200
  }
  else {
        PyObject *pyobj = SharedObject_AS_PYOBJECT(obj);
        PyObject *meta_type = (PyObject *) pyobj->ob_type->ob_type;
        PyObject *synch, *result;
        PyObject *err_type, *err_value, *err_tb;
        int restore = 0;

        if (PyErr_Occurred()) {
          PyErr_Fetch(&err_type, &err_value, &err_tb);                 210
          restore = 1;
        }
```

```
        /* Get the __synch__ attribute of the meta-type */
        synch = PyObject_GetAttrString(meta_type, S_SYNCH);
        if (synch == NULL)
          result = NULL;
        else
          /* Call its leave() method */
          result = PyObject_CallMethod(synch, S_LEAVE, "OO", pyobj, state);

        Py_DECREF(state);
        Py_XDECREF(result);
        if (result == NULL)
          PyErr_Clear();
        if (restore)
          PyErr_Restore(err_type, err_value, err_tb);
  }
}


/* Helper that gets a proxy for the shared dictionary of a shared object. */
static int
get_dict(PyObject *obj, PyObject **dictptr)
{
  SharedObject *shobj = SharedObject_FROM_PYOBJECT(obj);
  SharedObject *shdict;
  PyObject *dict;

  if (SharedMemHandle_IS_NULL(shobj->dicth)) {
        *dictptr = NULL;
        return 0;
  }
  shdict = (SharedObject *) SharedMemHandle_AsVoidPtr(shobj->dicth);
  if (shdict == NULL)
        return -1;
  dict = MakeProxy(shdict);
  if (dict == NULL)
        return -1;
  *dictptr = dict;
  return 0;
}


/* Helper that creates a new empty shared dictionary for a shared object
   and returns a proxy for it. */
static PyObject *
set_empty_dict(PyObject *obj)
{
  SharedObject *shobj = SharedObject_FROM_PYOBJECT(obj);
  SharedObject *shdict;
  PyObject *dict;

  assert(SharedMemHandle_IS_NULL(shobj->dicth));
  dict = PyDict_New();
  if (dict == NULL)
        return NULL;
```

```
  shdict = ShareObject(dict);
  if (shdict == NULL)
        return NULL;                                                    270
  dict = MakeProxy(shdict);
  if (dict == NULL)
        return NULL;
  shobj->dicth = SharedMemHandle_FromVoidPtr(shdict);
  return dict;
}

/* It may be useful to read PEP 252 for understanding the implementation
   of SharedObject_GetAttr() and SharedObject_SetAttr(). They mirror the
   implementation of PyObject_GenericGetAttr() and PyObject_GenericSetAttr()   280
   found in Objects/object.c */

PyObject *
SharedObject_GetAttr(PyObject *obj, PyObject *name)
{
  PyTypeObject *tp = obj->ob_type;
  PyObject *descr;
  PyObject *res = NULL;
  descrgetfunc f;
  PyObject *dict;                                                       290

#ifdef Py_USING_UNICODE
  /* The Unicode to string conversion is done here because the
        existing tp_setattro slots expect a string object as name
        and we wouldn't want to break those. */
  if (PyUnicode_Check(name)) {
        name = PyUnicode_AsEncodedString(name, NULL, NULL);
        if (name == NULL)
          return NULL;
  }                                                                     300
  else
#endif
        if (!PyString_Check(name)){
          PyErr_SetString(PyExc_TypeError,
                                    "attribute name must be string");
          return NULL;
        }
        else
          Py_INCREF(name);
                                                                        310
  /* Ready the object's type if needed */
  if (tp->tp_dict == NULL) {
        if (PyType_Ready(tp) < 0)
          goto done;
  }

  /* Look for a descriptor in the type */
  descr = _PyType_Lookup(tp, name);
  f = NULL;
  if (descr != NULL) {                                                  320
        f = descr->ob_type->tp_descr_get;
```

```
        if (f != NULL && PyDescr_IsData(descr)) {
          /* Data descriptors in the type override values
                in the object's dictionary */
          res = f(descr, obj, (PyObject *)obj->ob_type);
          goto done;
        }
  }

  /* Does the object have a dictionary? */                                  330
  if (obj->ob_type->tp_dictoffset) {
        /* Get the object's dictionary */
        if (get_dict(obj, &dict))
          goto done;
        if (dict != NULL) {
          /* Get the value from the object's dictionary */
          res = PyObject_GetItem(dict, name);
          if (res != NULL)
                goto done;
        }                                                                   340
  }

  /* No object dictionary, or missing key -
         use the non-data descriptor, if any */
  if (f != NULL) {
        res = f(descr, obj, (PyObject *)obj->ob_type);
        goto done;
  }

  /* Return the descriptor itself as a last resort */                       350
  if (descr != NULL) {
        Py_INCREF(descr);
        res = descr;
        goto done;
  }

  /* Everything failed - set an AttributeError exception */
  PyErr_Format(PyExc_AttributeError,
                        "'%.50s' object has no attribute '%.400s'",
                        tp->tp_name, PyString_AS_STRING(name));             360
 done:
  Py_DECREF(name);
  return res;
}


int
SharedObject_SetAttr(PyObject *obj, PyObject *name, PyObject *value)
{
  PyTypeObject *tp = obj->ob_type;                                          370
  PyObject *descr;
  descrsetfunc f;
  PyObject *dict;
  int res = -1;
```

```
#ifdef Py_USING_UNICODE
  /* The Unicode to string conversion is done here because the
        existing tp_setattro slots expect a string object as name
        and we wouldn't want to break those. */
  if (PyUnicode_Check(name)) {                                          380
        name = PyUnicode_AsEncodedString(name, NULL, NULL);
        if (name == NULL)
          return -1;
  }
  else
#endif
        if (!PyString_Check(name)){
          PyErr_SetString(PyExc_TypeError,
                                    "attribute name must be string");
          return -1;                                                    390
        }
        else
          Py_INCREF(name);

  /* Ready the object's type if needed */
  if (tp->tp_dict == NULL) {
        if (PyType_Ready(tp) < 0)
          goto done;
  }
                                                                       400
  /* Look for a descriptor in the type */
  descr = _PyType_Lookup(tp, name);
  f = NULL;
  if (descr != NULL) {
        f = descr->ob_type->tp_descr_set;
        if (f != NULL && PyDescr_IsData(descr)) {
          /* Data descriptors in the type override values
                in the object's dictionary */
          res = f(descr, obj, value);
          goto done;                                                   410
        }
  }

  /* Does the object have a dictionary? */
  if (obj->ob_type->tp_dictoffset) {
        /* Get the object's dictionary */
        if (get_dict(obj, &dict))
          goto done;
        /* Create a new one if needed */
        if (dict == NULL && value != NULL) {                           420
          dict = set_empty_dict(obj);
          if (dict == NULL)
                goto done;
        }
        if (dict != NULL) {
          /* Assign/delete the value from the dictionary */
          if (value == NULL)
                res = PyMapping_DelItem(dict, name);
          else
```

```
            res = PyObject_SetItem(dict, name, value);
        if (res < 0 && PyErr_ExceptionMatches(PyExc_KeyError))
            PyErr_SetObject(PyExc_AttributeError, name);
        goto done;
      }
  }

  /* No object dictionary - use the non-data descriptor, if any */
  if (f != NULL) {
      res = f(descr, obj, value);
      goto done;
  }

  /* Everything failed - set an AttributeError exception */
  if (descr == NULL) {
      PyErr_Format(PyExc_AttributeError,
                   "'%.50s' object has no attribute '%.400s'",
                   tp->tp_name, PyString_AS_STRING(name));
      goto done;
  }

  PyErr_Format(PyExc_AttributeError,
               "'%.50s' object attribute '%.400s' is read-only",
               tp->tp_name, PyString_AS_STRING(name));
 done:
  Py_DECREF(name);
  return res;
}


PyObject *
SharedObject_Repr(SharedObject *obj)
{
  static PyObject *opname = NULL;
  PyObject *state, *result;

  state = SharedObject_EnterString(obj, "__repr__", &opname);
  result = PyObject_Repr(SharedObject_AS_PYOBJECT(obj));
  SharedObject_Leave(obj, state);
  return result;
}


PyObject *
SharedObject_Str(SharedObject *obj)
{
  static PyObject *opname = NULL;
  PyObject *state, *result;

  state = SharedObject_EnterString(obj, "__str__", &opname);
  result = PyObject_Str(SharedObject_AS_PYOBJECT(obj));
  SharedObject_Leave(obj, state);
  return result;
}
```

```
long
SharedObject_Hash(SharedObject *obj)
{
  static PyObject *opname = NULL;
  PyObject *state;                                                    490
  long result;

  state = SharedObject_EnterString(obj, "__hash__", &opname);
  if (state == NULL)
        return -1;
  result = PyObject_Hash(SharedObject_AS_PYOBJECT(obj));
  SharedObject_Leave(obj, state);
  return result;
}
                                                                      500

int
SharedObject_Print(SharedObject *obj, FILE *fp, int flags)
{
  static PyObject *opname = NULL;
  PyObject *state;
  int result;

  state = SharedObject_EnterString(obj, "__print__", &opname);
  if (state == NULL)                                                  510
        return -1;
  result = PyObject_Print(SharedObject_AS_PYOBJECT(obj), fp, flags);
  SharedObject_Leave(obj, state);
  return result;
}


int
SharedObject_Compare(SharedObject *a, PyObject *b)
{                                                                     520
  static PyObject *opname = NULL;
  PyObject *state;
  int result;

  state = SharedObject_EnterString(a, "__cmp__", &opname);
  if (state == NULL)
        return -1;
  result = PyObject_Compare(SharedObject_AS_PYOBJECT(a), b);
  SharedObject_Leave(a, state);
  return result;                                                      530
}


PyObject *
SharedObject_RichCompare(SharedObject *a, PyObject *b, int op)
{
  char *opname;
```

```
  PyObject *state, *result;

  switch (op) {                                                    540
  case Py_LT: opname = "__lt__"; break;
  case Py_LE: opname = "__le__"; break;
  case Py_EQ: opname = "__eq__"; break;
  case Py_NE: opname = "__ne__"; break;
  case Py_GT: opname = "__gt__"; break;
  case Py_GE: opname = "__ge__"; break;
  default: return NULL; /* cannot happen */
  }

  state = SharedObject_EnterString(a, opname, NULL);               550
  if (state == NULL)
        return NULL;
  result = PyObject_RichCompare(SharedObject_AS_PYOBJECT(a), b, op);
  SharedObject_Leave(a, state);
  return result;
}


int
SharedObject_RichCompareBool(SharedObject *a, PyObject *b, int op)  560
{
  char *opname;
  PyObject *state;
  int result;

  switch (op) {
  case Py_LT: opname = "__lt__"; break;
  case Py_LE: opname = "__le__"; break;
  case Py_EQ: opname = "__eq__"; break;
  case Py_NE: opname = "__ne__"; break;                            570
  case Py_GT: opname = "__gt__"; break;
  case Py_GE: opname = "__ge__"; break;
  default: return -1; /* cannot happen */
  }

  state = SharedObject_EnterString(a, opname, NULL);
  if (state == NULL)
        return -1;
  result = PyObject_RichCompareBool(SharedObject_AS_PYOBJECT(a), b, op);
  SharedObject_Leave(a, state);                                    580
  return result;
}
```

A.31 SharedRegion.h

```
/* SharedRegion.h */
/* Interface for creation and attachment of shared memory regions. */

#ifndef SHAREDREGION_H_INCLUDED
#define SHAREDREGION_H_INCLUDED

#include "_core.h"

/* Handle for a shared memory region. */
typedef int SharedRegionHandle;

/* NULL value for a shared memory region handle */
#define SharedRegionHandle_NULL (-1)

/* Macro to test for NULL-ness of shared memory region handles */
#define SharedRegionHandle_IS_NULL(h) (h == SharedRegionHandle_NULL)

/* Creates a new shared memory region of the given size.
   Returns a handle for the region, or SharedRegionHandle_NULL on error.
   This function should normally not be called directly - use
   SharedRegion_New() instead (defined in Globals.h), which stores
   the handle in a global table for cleanup purposes. */
SharedRegionHandle _SharedRegion_New(size_t *size);

/* Destroys a shared memory region. */
void _SharedRegion_Destroy(SharedRegionHandle h);

/* Attaches a shared memory region. */
void *_SharedRegion_Attach(SharedRegionHandle h);

/* Detaches a shared memory region that is attached at the given address. */
int _SharedRegion_Detach(void *addr);

#endif
```

A.32 SharedRegion.c

```
/* SharedRegion.c */

#include "SharedRegion.h"
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

SharedRegionHandle
_SharedRegion_New(size_t *size)
{
  int id;
  struct shmid_ds ds;

  /* Create the region */
  id = shmget(IPC_PRIVATE, (int) *size, SHM_R | SHM_W);
  if (id == -1)
        return SharedRegionHandle_NULL;

  /* Query it for its actual size */
  if (shmctl(id, IPC_STAT, &ds) == -1) {
        /* Error - remove the region again */
        shmctl(id, IPC_RMID, NULL);
        PyErr_SetString(PyExc_RuntimeError,
                                "creation of shared memory region failed");
        return SharedRegionHandle_NULL;
  }
  *size = (size_t) ds.shm_segsz;

  return id;
}

void
_SharedRegion_Destroy(SharedRegionHandle h)
{
  shmctl(h, IPC_RMID, NULL);
}

void *
_SharedRegion_Attach(SharedRegionHandle handle)
{
  void *addr = shmat(handle, 0, 0);
  if(addr == (void *) -1) {
    /* XXX: Can this really happen? */
        PyErr_SetString(PyExc_RuntimeError,
                                "attachment of shared memory region failed");
    return NULL;
  }
  return addr;
}

int
```

```
_SharedRegion_Detach(void *addr)
{
  return shmdt(addr);
}
```

A.33 Spinlock.h

```
/* Spinlock.h */

#ifndef SPINLOCK_H_INCLUDED
#define SPINLOCK_H_INCLUDED

#include "_core.h"

typedef int Spinlock;

/* All these always succeed (if they return). */                    10
void Spinlock_Init(Spinlock *spinlock);
void Spinlock_Destroy(Spinlock *spinlock);
void Spinlock_Acquire(Spinlock *spinlock);
void Spinlock_Release(Spinlock *spinlock);

#endif
```

A.34 Spinlock.c

```
/* Spinlock.c */

#include "Spinlock.h"

static inline void acquire(int *mutex)
{
    __asm__ volatile (" movl %0,%%eax            \n"
                      "1: lock                   \n"
                      "   btsl $0, 0(%%eax)      \n"
                      "   jc 1b                  \n"
                      :
                      :"m"(mutex)
                      :"eax");


}


static inline void release(int *mutex)
{
    __asm__ volatile (" movl %0,%%eax            \n"
                      "1: lock                   \n"
                      "   andl $0, 0(%%eax)      \n"
                      :
                      :"m"(mutex)
                      :"eax");
}


void
Spinlock_Init(Spinlock *spinlock)
{
  *spinlock = 0;
}

void
Spinlock_Destroy(Spinlock *spinlock)
{
  *spinlock = 0;
}

void
Spinlock_Acquire(Spinlock *spinlock)
{
    acquire(spinlock);
}

void
Spinlock_Release(Spinlock *spinlock)
{
    release(spinlock);
```

```
}
```

A.35 _core.h

```
/* _core.h */
/* Common header file for the _core module. */

#ifndef _CORE_H_INCLUDED
#define _CORE_H_INCLUDED

#include <Python.h>
#include <structmember.h>

#ifdef DEBUG                                                    10
#define LOGF(fmt, args...) printf("%s:%d (%s) " fmt "\n", \
                                __FILE__, __LINE__, __FUNCTION__, args)
#define LOGS(s) LOGF("%s", s)
#define LOG() LOGF("%s", "")
#else
#define LOGF(fmt, args...)
#define LOGS(s)
#define LOG()
#endif
                                                                20
/* String constants */
#define S_IHEAP "__instanceheap__"
#define S_DHEAP "__dataheap__"
#define S_SYNCH "__synch__"
#define S_ENTER "enter"
#define S_LEAVE "leave"
#define S_ALLOC "alloc"
#define S_REALLOC "realloc"
#define S_FREE "free"
                                                                30
extern PyObject *ErrorObject;
extern PyObject *TypeMap;

/* These are updated by Process_Init() each time a new process is forked */
extern int my_pid;
extern int my_pindex;

#endif
```

A.36 init_core.c

```
/* init_core.c */
/* Initialization of the _core module. */

#include "_core.h"
#include "Process.h"
#include "SharedRegion.h"
#include "SharedAlloc.h"
#include "Proxy.h"
#include "SharedHeap.h"
#include "SharedListAndTuple.h"
#include "SharedDictBase.h"
#include "SharedObject.h"
#include "share.h"
#include "Address.h"
#include "Monitor.h"
#include "LockObject.h"

PyObject *ErrorObject = NULL;
PyObject *TypeMap = NULL;

/* Returns the address of the given object as an Address object. */
static PyObject *
address_of(PyObject *null, PyObject *obj)
{
  return Address_FromVoidPtr(obj);
}

/* Maps a regular type to a shared type and a proxy type */
static PyObject *
map_type(PyObject *null, PyObject *args)
{
  PyObject *tp, *stp, *ptp, *tuple;

  /* Parse the arguments, which should be three types */
  if(!PyArg_ParseTuple(args, "O!O!O!:map_type", &PyType_Type, &tp,
                                    &PyType_Type, &stp, &PyType_Type, &ptp))
        return NULL;

  /* Build a tuple of the shared type and proxy type */
  tuple = Py_BuildValue("(OO)", stp, ptp);
  if (tuple == NULL)
        return NULL;

  /* Map the regular type to the tuple */
  if (PyDict_SetItem(TypeMap, tp, tuple)) {
        Py_DECREF(tuple);
        return NULL;
  }
  /* Also map the shared type to the tuple */
  if (PyDict_SetItem(TypeMap, stp, tuple)) {
        Py_DECREF(tuple);
```

```
        return NULL;
  }

  Py_DECREF(tuple);
  Py_INCREF(Py_None);
  return Py_None;
}

/* Shares an object and returns a proxy object for it. */
static PyObject *share(PyObject *null, PyObject *obj)
{
  SharedObject *shobj;

  shobj = ShareObject(obj);
  if(shobj == NULL)
        return NULL;
  return MakeProxy(shobj);
}

/* Initializes a recently forked child process */
static PyObject *
init_child(PyObject *null, PyObject *noargs)
{
  PyObject *map, *values;
  int i, len;

  if (Process_Init()) {
        PyErr_SetString(ErrorObject, "too many processes");
        return NULL;
  }
  map = GetProxyMap();
  if (map == NULL)
        return NULL;
  values = PyMapping_Values(map);
  if (values == NULL)
        return NULL;
  len = PySequence_Length(values);
  for (i = 0; i < len; i++) {
        PyObject *item = PySequence_GetItem(values, i);
        assert(Proxy_Check(item));
        SharedObject_SetProxyBit(((ProxyObject *) item)->referent);
        Py_DECREF(item);
  }
  Py_INCREF(Py_None);
  return Py_None;
}

/* Handles the death of a child process */
static PyObject *
child_died(PyObject *null, PyObject *args)
{
  int pid, killsignal, status, corefile;

  if (!PyArg_ParseTuple(args, "iiii:child_died",
```

```
                                        &pid, &killsignal, &status, &corefile))
        return NULL;
  if (killsignal != 0 || status != 0)
        Process_CleanupChild(pid);
  Py_INCREF(Py_None);                                                    110
  return Py_None;
}

/* Overrides the allocation methods of a type to use
 * SharedObject_Alloc and SharedObject_Free. */
static PyObject *
override_allocation(PyObject *null, PyObject *args)
{
  PyTypeObject *type;
                                                                         120
  if (!PyArg_ParseTuple(args, "O!:override_allocation", &PyType_Type, &type))
        return NULL;

  /* Override tp_alloc and tp_free; there are no descriptors for these, so
        it's only a matter of updating the slots directly. */
  type->tp_alloc = SharedObject_Alloc;
  type->tp_free = SharedObject_Free;
  /* SharedObject_Alloc() and SharedObject_Free() don't support
        cyclic garbage collection, so the new type can't be
        garbage collected */                                             130
  type->tp_flags &= ~Py_TPFLAGS_HAVE_GC;

  Py_INCREF(Py_None);
  return Py_None;
}

/* Returns true if the given type overrides __getattr__,
   __getattribute__, __setattr__ or __delattr__. */
static PyObject *
overrides_attributes(PyObject *null, PyObject *args)                    140
{
  PyTypeObject *type;
  PyObject *desc = NULL;
  PyObject *result = NULL;

  if (!PyArg_ParseTuple(args, "O!:override_allocation", &PyType_Type, &type))
        goto Done;

  if ((type->tp_setattr != 0 &&
       type->tp_setattr != PyBaseObject_Type.tp_setattr) ||                150
      (type->tp_setattro != 0 &&
       type->tp_setattro != PyBaseObject_Type.tp_setattro)) {
    /* Overrides __setattr__ / __delattr__ */
    result = Py_True;
    goto Done;
  }
  if ((type->tp_getattr != 0 &&
       type->tp_getattr != PyBaseObject_Type.tp_getattr)
      || (type->tp_getattr != 0 &&
```

```
        type->tp_getattro != PyBaseObject_Type.tp_getattro)) {
    /* Overrides __getattribute__ and/or __getattr__.
       The latter is allowed, so we have to find out which it is. */
    desc = PyObject_GetAttrString((PyObject *) type, "__getattribute__");
    if (desc==NULL)
      goto Done;
    if (desc->ob_type == &PyWrapperDescr_Type) {
      void *wrapped = ((PyWrapperDescrObject *) desc)->d_wrapped;
      if (wrapped == (void *) PyBaseObject_Type.tp_getattro) {
        /* Only overrides __getattr__, which is fine. */
        result = Py_False;
        goto Done;
      }
    }
    result = Py_True;
    goto Done;
  }
  result = Py_False;

 Done:
  Py_XDECREF(desc);
  Py_XINCREF(result);
  return result;
}

static PyObject *
shared_getattribute(PyObject *null, PyObject *args)
{
  PyObject *self, *name;

  if (!PyArg_ParseTuple(args, "OO", &self, &name))
        return NULL;
  return SharedObject_GetAttr(self, name);
}

static PyObject *
shared_setattr(PyObject *null, PyObject *args)
{
  PyObject *self, *name, *value;

  if (!PyArg_ParseTuple(args, "OOO", &self, &name, &value))
        return NULL;
  if (SharedObject_SetAttr(self, name, value))
        return NULL;
  Py_INCREF(Py_None);
  return Py_None;
}

static PyObject *
shared_delattr(PyObject *null, PyObject *args)
{
  PyObject *self, *name;

  if (!PyArg_ParseTuple(args, "OO", &self, &name))
```

```
        return NULL;
  if (SharedObject_SetAttr(self, name, NULL))
        return NULL;
  Py_INCREF(Py_None);
  return Py_None;
}

static char address_of_doc[] =
"address_of(obj) -> The object's address in memory";
static char map_type_doc[] =
"map_type(tp, stp, ptp)";
static char share_doc[] =
"share(obj) -> Proxy object for a shared object equal to obj";
static char init_child_doc[] =
"init_child()";
static char child_died_doc[] =
"child_died(pid, killsignal, status, corefile)";
static char override_allocation_doc[] =
"override_allocation(type)";
static char overrides_attributes_doc[] =
"overrides_attributes(type) -> True if type overrides __getattribute__, "\
"__setattr__ or __delattr__";
static char shared_getattribute_doc[] =
"__getattribute__ for shared objects";
static char shared_setattr_doc[] =
"__setattr__ for shared objects";
static char shared_delattr_doc[] =
"__delattr__ for shared objects";


/* List of functions defined in the module */
static PyMethodDef functions[] = {
  {"address_of", address_of, METH_O, address_of_doc},
  {"map_type", map_type, METH_VARARGS, map_type_doc},
  {"share", share, METH_O, share_doc},
  {"init_child", init_child, METH_NOARGS, init_child_doc},
  {"child_died", child_died, METH_VARARGS, child_died_doc},
  {"override_allocation", override_allocation, METH_VARARGS,
   override_allocation_doc},
  {"overrides_attributes", overrides_attributes, METH_VARARGS,
   overrides_attributes_doc},
  {"shared_getattribute", shared_getattribute, METH_VARARGS,
   shared_getattribute_doc},
  {"shared_setattr", shared_setattr, METH_VARARGS, shared_setattr_doc},
  {"shared_delattr", shared_delattr, METH_VARARGS, shared_delattr_doc},
  {NULL, NULL} /* sentinel */
};

/* Initialization function for the module (*must* be called init_core) */
DL_EXPORT(void)
init_core(void)
{
  PyObject *m, *d, *nulladdr, *type_map;
  int ok;
```

```
if (Process_Init())
    goto Error;                                                    270

/* Create the error object for the module */
ErrorObject = PyErr_NewException("posh._core.error", NULL, NULL);
if (ErrorObject == NULL)
    goto Error;

/* Create the type map */
TypeMap = PyDict_New();
if (TypeMap == NULL)
    goto Error;                                                    280

/* Ready built-in types */
PyType_Ready(&Address_Type);
PyType_Ready(&SharedHeap_Type);
PyType_Ready(&SharedListBase_Type);
PyType_Ready(&SharedTupleBase_Type);
PyType_Ready(&SharedDictBase_Type);
PyType_Ready(&Proxy_Type);
PyType_Ready(&Monitor_Type);
PyType_Ready(&Lock_Type);                                          290

/* Create the module and get its dictionary */
m = Py_InitModule("_core", functions);
if (m == NULL)
    goto Error;
d = PyModule_GetDict(m);
if (d == NULL)
    goto Error;

/* Add symbols to the module */                                    300
nulladdr = Address_FromVoidPtr(NULL);
if (nulladdr == NULL)
    goto Error;
ok = !PyDict_SetItemString(d, "null", nulladdr);
Py_DECREF(nulladdr);

type_map = PyDictProxy_New(TypeMap);
if (type_map == NULL)
    goto Error;
ok = ok && !PyDict_SetItemString(d, "type_map", type_map);         310
Py_DECREF(type_map);

ok = ok && !PyDict_SetItemString(d, "error", ErrorObject);
ok = ok && !PyDict_SetItemString(d, "SharedHeap",
                                 (PyObject *) &SharedHeap_Type);
ok = ok && !PyDict_SetItemString(d, "SharedListBase",
                                 (PyObject *) &SharedListBase_Type);
ok = ok && !PyDict_SetItemString(d, "SharedTupleBase",
                                 (PyObject *) &SharedTupleBase_Type);
ok = ok && !PyDict_SetItemString(d, "SharedDictBase",              320
                                 (PyObject *) &SharedDictBase_Type);
```

```
  ok = ok && !PyDict_SetItemString(d, "Proxy", (PyObject *) &Proxy_Type);
  ok = ok && !PyDict_SetItemString(d, "Monitor", (PyObject *) &Monitor_Type);
  ok = ok && !PyDict_SetItemString(d, "Lock", (PyObject *) &Lock_Type);

  if (ok)
        return;

 Error:
  Py_XDECREF(ErrorObject);                                                    330
  Py_XDECREF(TypeMap);
  if (PyErr_Occurred() == NULL)
        PyErr_SetString(PyExc_ImportError, "module initialization failed");
}
```

A.37 share.h

```
/* share.h */

#ifndef SHARE_H_INCLUDED
#define SHARE_H_INCLUDED

#include "_core.h"
#include "SharedObject.h"

/* Instantiates a shared object from the given object. */
SharedObject *ShareObject(PyObject *obj);

/* Encapsulates a shared object in a proxy object. */
PyObject *MakeProxy(SharedObject *obj);

#endif
```

A.38 share.c

```
/* share.c */

#include "share.h"
#include "Proxy.h"
#include "SharedObject.h"
#include "Address.h"

SharedObject *
ShareObject(PyObject *obj)
{
  /* Borrowed references */
  PyTypeObject *type = obj->ob_type;
  PyObject *shtype;

  /* Special-case proxy objects */
  if(Proxy_Check(obj))
        return ((ProxyObject *) obj)->referent;

  /* Special-case singletons None, True and False */
  if(obj == Py_None) {
  }
  else if(obj == Py_True) {
  }
  else if(obj == Py_False) {
  }

  /* Get the type's shared type from the type map */
  shtype = PyDict_GetItem(TypeMap, (PyObject *) type);
  if(shtype == NULL) {
        PyErr_Format(ErrorObject, "%.100s instances cannot be shared",
                                   type->tp_name);
        return NULL;
  }
  assert(PyTuple_Check(shtype));
  assert(PyTuple_GET_SIZE(shtype) == 2);
  shtype = PyTuple_GET_ITEM((PyObject *) shtype, 0);

  /* If the type is its own shared type, the object is already shared. */
  if(shtype == (PyObject *) type)
        return SharedObject_FROM_PYOBJECT(obj);

  /* Instantiate the shared type with the object to get a shared object. */
  /* LOGF("Instantiating %.100s object",
          ((PyTypeObject *) shtype)->tp_name); */
  obj = PyObject_CallFunctionObjArgs(shtype, obj, NULL);
  if(obj == NULL)
        return NULL;
  return SharedObject_FROM_PYOBJECT(obj);
}

/* Creates a new proxy object for a shared object */
```

```
static PyObject *
new_proxy(SharedObject *obj)
{
  /* Borrowed references */
  PyObject *pyobj = SharedObject_AS_PYOBJECT(obj);
  PyTypeObject *type = pyobj->ob_type;
  PyObject *ptype;

  /* Get the shared type's proxy type from the type map */                    60
  ptype = PyDict_GetItem(TypeMap, (PyObject *) type);
  if(ptype == NULL) {
        PyErr_Format(ErrorObject, "no proxy type for %.100s", type->tp_name);
        return NULL;
  }
  assert(PyTuple_Check(ptype));
  ptype = PyTuple_GET_ITEM(ptype, 1);

  /* Instantiate the new proxy object */
  return PyObject_CallFunctionObjArgs(ptype, pyobj, NULL);                   70
}

/* Looks in the procy map for a proxy object for the shared object, or
   creates a new proxy object if there is none. */
PyObject *
MakeProxy(SharedObject *obj)
{
  /* New references */
  PyObject *addr = NULL, *proxy = NULL;
  /* Borrowed references */                                                  80
  PyObject *map;

  map = GetProxyMap();
  if (map == NULL)
        goto Done;
  addr = Address_FromVoidPtr(obj);
  if (addr == NULL)
        goto Done;
  proxy = PyObject_GetItem(map, addr);
  if (proxy != NULL)                                                         90
        /* Found an existing proxy object */
        goto Done;

  /* Clear the KeyError from the failed lookup */
  if (!PyErr_ExceptionMatches(PyExc_KeyError))
        goto Done; /* Other exceptions are passed on */
  PyErr_Clear();

  /* Create a new proxy object */
  proxy = new_proxy(obj);                                                   100
  if (proxy != NULL) {
        /* Store the new proxy object in the proxy map */
        if (PyObject_SetItem(map, addr, proxy)) {
          Py_DECREF(proxy);
          proxy = NULL;
```

```
        }
    }
    else {
        /* Creation of the proxy object failed. In case the shared object has
           no other proxy objects, and no references to it from other shared
           objects, we should call SharedObject_ClearProxyBit() to let it be
           reclaimed. */
        SharedObject_ClearProxyBit(obj);
    }

Done:
    Py_XDECREF(addr);
    return proxy;
}
```

A.39 _proxy.py

```
# _proxy.py

# Submodule of the posh package, defining a factory function
# for creating proxy types based on shared types

import types
import _core

class ProxyMethod(object):
        def __init__(self, inst, cls, mname):
                self.proxy_inst = inst
                self.proxy_cls = cls
                self.mname = mname
                self.cname = cls.__name__

        def __str__(self):
                if self.proxy_inst is None:
                        return "<method '%s' of '%s' objects>" % \
                                (self.mname, self.cname)
                return "<method '%s' of '%s' object at %s>" % \
                        (self.mname, self.cname, _core.address_of(self.proxy_inst))

        __repr__ = __str__

        def __call__(self, *args, **kwargs):
                if self.proxy_inst is None:
                        # Call to unbound proxy method
                        if (len(args) < 1) or not isinstance(args[0], self.proxy_cls):
                                fmt = "unbound method %s.%s must be called" + \
                                        "with %s instance as first argument"
                                raise TypeError, fmt % (self.cname, self.mname, self.cname)
                        return args[0]._call_method(self.mname, args[1:], kwargs);
                else:
                        # Call to bound proxy method
                        return self.proxy_inst._call_method(self.mname, args, kwargs);


class ProxyMethodDescriptor(object):
        def __init__(self, mname):
                self.mname = mname

        def __get__(self, inst, cls):
                return ProxyMethod(inst, cls, self.mname)

        def __set__(self, inst, value):
                raise TypeError, "read-only attribute"


method_desc_types = (type(list.__add__), type(list.append),
                                        types.UnboundMethodType)
```

```
def MakeProxyType(reftype):
        d = {"__slots__": []}
        for attrname in dir(reftype):
                if not hasattr(_core.Proxy, attrname):
                        attr = getattr(reftype, attrname)
                        if type(attr) in method_desc_types:
                                d[attrname] = ProxyMethodDescriptor(attrname);
        name = reftype.__name__+"Proxy"                                          60
        return type(name, (_core.Proxy,), d)
```

A.40 _verbose.py

```
# _verbose.py

# Submodule of the posh package, defining classes for verbose output

__all__ = ["PIDWriter", "VerboseHeap", "VerboseSynch"]

from sys import stdout as _stdout
from os import getpid as _getpid
import _core


class PIDWriter(object):
    def __init__(self, output):
        self.output = output
        self.buf = ""

    def write(self, s):
        lines = (self.buf+s).split('\n')
        if lines:
            self.buf = lines[-1]
        else:
            self.buf = ""
        for line in lines[:-1]:
            self.output.write("[%s] %s\n" % (_getpid(), line.strip()))


class VerboseHeap(object):
    def __init__(self, name, heap, output=None):
        object.__init__(self)
        self.name = name
        self.heap = heap
        self.output = output or PIDWriter(_stdout)

    def alloc(self, size):
        self.output.write("%s: Allocating %d bytes" % (self.name, size))
        addr, size = self.heap.alloc(size)
        self.output.write(" at address %s (size %d).\n" % (addr, size))
        return addr, size

    def free(self, addr):
        self.output.write("%s: Freeing memory at address %s.\n" \
                                    % (self.name, addr))
        return self.heap.free(addr)

    def realloc(self, addr, size):
        self.output.write("%s: Reallocating %d bytes from address %s" \
                                    % (self.name, size, addr))
        addr, size = self.heap.realloc(addr, size)
        self.output.write("to %s (size %d).\n" % (addr, size))
        return addr, size
```

```
class VerboseSynch(object):
        def __init__(self, name, synch, output=None):
                object.__init__(self)
                self.name = name
                self.synch = synch
                self.output = output or PIDWriter(_stdout)

        def str(self, x):                                                    60
                return "%s object at %s" % (type(x).__name__, _core.address_of(x))

        def enter(self, x, opname):
                self.output.write("%s: Entering %s on %s\n" \
                                        % (self.name, opname, self.str(x)))
                if self.synch is None:
                        rv = None
                else:
                        rv = self.synch.enter(x, opname)
                return opname, rv                                            70

        def leave(self, x, opname_and_rv):
                opname, rv = opname_and_rv
                self.output.write("%s: Leaving %s on %s\n" \
                                        % (self.name, opname, self.str(x)))
                if self.synch is not None:
                        self.synch.leave(x, rv)
```

A.41 __init__.py

```
# __init__.py

# Main script for the posh package.

# Set this to 1 to get verbose output
VERBOSE = 0

# Names imported by from posh import *
__all__ = """
    share allow_sharing MONITOR wait waitpid _exit getpid exit sleep
    forkcall waitall error Lock
""".split()

import _core
import _proxy
from _verbose import *
import signal as _signal
import types as _types

# Import common process-related symbols into this module.
# We will supply our own version of os.fork().
from os import fork as _os_fork, wait, waitpid, _exit, getpid
from sys import exit, stdout as _stdout
from time import sleep

# Import some names from the _core module
share = _core.share
error = _core.error
Lock = _core.Lock

# The default argument for allow_sharing()
MONITOR = _core.Monitor()

# Argument to allow_sharing() that specifies no synchronization
NOSYNCH = None

if VERBOSE:
    # Wrap MONITOR and NOSYNCH in verbose objects
    MONITOR = VerboseSynch("Monitor", MONITOR)
    NOSYNCH = VerboseSynch("NoSynch", NOSYNCH)


class SharedType(type):
    """Meta-type for shared types.
    """
    # Shared heaps for instances and auxiliary data structures
    __instanceheap__ = _core.SharedHeap()
    __dataheap__ = _core.SharedHeap()

    if VERBOSE:
        # Wrap the heaps in verbose objects
```

```
        __instanceheap__ = VerboseHeap("Instance heap", __instanceheap__)
        __dataheap__ = VerboseHeap("Data heap", __dataheap__)

    # Default synchronization policy
    __synch__ = MONITOR

    # This method gets invoked when the meta-type is called, and
    # returns a new shared type.
    def __new__(tp, name, bases, dct):                                    60
        """Creates a new shared type.
        """
        def wrap_built_in_func(func):
            """Wraps a built-in function in a function object.
            """
            # XXX: This is not quite satisfactory, since we lose the
            # __name__ and __doc__ attributes. Built-in functions
            # really should have binding behaviour!
            return lambda *args, **kwargs: func(*args, **kwargs)
                                                                          70
        if len(bases) > 1:
            raise ValueError, "this meta-type only supports single inheritance"

        # Override attribute access
        dct["__getattribute__"] = wrap_built_in_func(_core.shared_getattribute)
        dct["__setattr__"] = wrap_built_in_func(_core.shared_setattr)
        dct["__delattr__"] = wrap_built_in_func(_core.shared_delattr)

        # Invoke type's implementation of __new__
        newtype = type.__new__(tp, name, bases, dct)                      80

        # Override the allocation methods of the new type
        _core.override_allocation(newtype)
        return newtype


# Simulates an attribute lookup on the given class by traversing its
# superclasses in method resolution order and returning the first class
# whose dictionary contains the attribute. Returns None if not found.
def _type_lookup(tp, name):                                               90
    if hasattr(tp, "__mro__"):
        # Follow the MRO defined by the __mro__ attribute
        for t in tp.__mro__:
            if name in t.__dict__:
                return t
    else:
        # Use the classic left-to-right, depth-first rule
        if name in tp.__dict__:
            return tp
        for t in tp.__bases__:                                            100
            res = _type_lookup(t, name)
            if res:
                return res
    return None
```

```
def allow_sharing(tp, synch=MONITOR):
    """allow_sharing(tp, synch=None) -> None

    Allows sharing of objects of the given type. This must be called prior
    to any fork() calls. The synch parameter may be None for immutable types,
    indicating that no synchronization is needed on these objects, or MONITOR
    for objects that desire monitor access semantics.

    Instances of shareable types should adhere to the following rules:
    * A nonempty __slots__ is not allowed.
    * No custom __getattribute__(), __setattr__() or __delattr__() is allowed.
    * Extension types need not make room for a dictionary in their object
      structure, but they should have a nonzero tp_dictoffset if they want to
      support attributes.
    * No references to other objects should be stored in the object structure
      itself, but rather in the object's dictionary using the generic
      PyObject_SetAttribute() and friends.
    * Extension types should not override the tp_alloc and tp_free slots.
    * References to ``self'' should not be stored in a way that makes them
      persist beyond the lifetime of the call.
    """

    if isinstance(tp, _types.ClassType):
        raise TypeError, "allow_sharing: old-style classes are not supported"
    if not isinstance(tp, type):
        raise TypeError, "allow_sharing: 1st argument (tp) must be a type"

    # Check if we've forked
    if globals().get("_has_forked", 0):
        raise ValueError, "allow_sharing: this call must be made " +\
              "prior to any fork calls"

    # Check if the type is already registered
    if tp in _core.type_map:
        fmt = "allow_sharing: %s objects may already be shared"
        raise ValueError, (fmt % tp.__name__)

    # The given type may not override attribute access
    # except to provide a __getattr__ hook.
    if _core.overrides_attributes(tp):
        fmt = "allow_sharing: %s overrides __getattribute__, " +\
              "__setattr__ or __delattr__"
        raise ValueError, (fmt % tp.__name__)

    # The given type may not have a nonempty __slots__
    tpdir = dir(tp)
    if "__slots__" in tpdir and len(tp.__slots__):
        fmt = "allow_sharing: %s has a nonempty __slots__"
        raise ValueError, (fmt % tp.__name__)

    # If the given type contains no __dict__ descriptor, then the type's
    # instances has no dictionary, so neither should those of the shared
    # type.
```

```
    if not "__dict__" in tpdir:
        d = {'__slots__': []}
    else:
        d = {}

    # Make up a name for the shared type
    name = "Shared"+tp.__name__.capitalize()

    # The shared type is produced by inheriting from the shareable type
    # using the SharedType meta-type
    stp = SharedType(name, (tp,), d)
    # Assign the __synch__ attribute of the new type
    stp.__synch__ = synch

    # We also need a proxy type that looks like the shared type
    ptp = _proxy.MakeProxyType(stp)

    # Register the types with the _core module and we're done
    _core.map_type(tp, stp, ptp)


def init_types():
    # This function initializes the module with some basic shareable
    # types. The function is deleted after it is called.
    # Shared versions of the container types list, tuple and dictionary
    # are implemented from scratch, and are treated specially.

    # Allow the names SharedList, SharedTuple and SharedDict to persist,
    # so that users may subtype them if desired.
    global SharedList, SharedTuple, SharedDict

    def seq_add(self, other):
        return self[:]+other

    def seq_radd(self, other):
        return other+self[:]

    def seq_mul(self, count):
        return self[:]*count

    def seq_contains(self, item):
        for x in self:
            if x == item:
                return 1
        return 0

    class SharedList(_core.SharedListBase):
        """List type whose instances live in shared memory."""
        __metaclass__ = SharedType
        __slots__ = []

        __add__ = seq_add
        __radd__ = seq_radd
        __mul__ = seq_mul
```

```
    __rmul__ = seq_mul
    __contains__ = seq_contains

    def __iadd__(self, other):
        self.extend(other)
        return self
                                                                    220
    def __imul__(self, count):
        lr = range(len(self))
        for i in range(count-1):
            for j in lr:
                self.append(self[j])
        return self

    def __getslice__(self, i, j):
        indices = range(len(self))[i:j]
        return [self[i] for i in indices]                           230

    def count(self, item):
        result = 0
        for x in self:
            if x == item:
                result += 1
        return result

    def extend(self, seq):
        if seq is not self:                                         240
            # Default implementation, uses iterator
            for item in seq:
                self.append(item)
        else:
            # Extension by self, cannot use iterator
            for i in range(len(self)):
                self.append(self[i])

    def index(self, item):
        for i in range(len(self)):                                  250
            if self[i] == item:
                return i
        raise ValueError, "list.index(x): x not in list"

    def reverse(self):
        l = len(self) // 2
        for i in range(l):
            j = -i-1
            # A traditional swap does less work than
            # a, b = b, a - although it is less elegant...          260
            tmp = self[i]
            self[i] = self[j]
            self[j] = tmp

class SharedTuple(_core.SharedTupleBase):
    """Tuple type whose instances live in shared memory."""
    __metaclass__ = SharedType
```

```
        __slots__ = []

        __add__ = seq_add                                                    270
        __radd__ = seq_radd
        __mul__ = seq_mul
        __rmul__ = seq_mul
        __contains__ = seq_contains

        def __getslice__(self, i, j):
            indices = range(len(self))[i:j]
            return tuple([self[i] for i in indices])

        def __str__(self):                                                   280
            # Tuples cannot be recursive, so this is easy
            # to implement using Python code
            items = map(repr, self)
            return "("+", ".join(items)+")"

        __repr__ = __str__

    class SharedDict(_core.SharedDictBase):
        """Dictionary type whose instances live in shared memory."""
        __metaclass__ = SharedType                                           290
        __slots__ = []

    SharedListProxy = _proxy.MakeProxyType(SharedList)
    SharedTupleProxy = _proxy.MakeProxyType(SharedTuple)
    SharedDictProxy = _proxy.MakeProxyType(SharedDict)

    # This maps list to SharedList and so on - this is special, since
    # these shared types are not subtypes of their shareable equivalents,
    # as is normally the case.
    _core.map_type(list, SharedList, SharedListProxy)                        300
    _core.map_type(tuple, SharedTuple, SharedTupleProxy)
    _core.map_type(dict, SharedDict, SharedDictProxy)

    # Produce basic immutable shared types
    for t in int, float, long, complex, str, unicode, Lock:
        allow_sharing(t, synch=None)

init_types()
del init_types
                                                                             310

# Define and set signal handler for SIGCHLD signals
def _on_SIGCHLD(signumber, frame):
    # Reinstall the signal handler
    _signal.signal(_signal.SIGCHLD, _on_SIGCHLD)
    try:
        # Collect the pid and status of the child process that died
        pid, status = wait()
        # Lower 7 bits of status is the signal number that killed it
        killsignal = status & 0x7F                                           320
        # 8th bit is set if a core file was produced
```

```
        corefile = status & 0x80
        # High 8 bits is the exit status (on normal exit)
        status = (status >> 8) & 0xFF
        _core.child_died(pid, killsignal, status, corefile)
    except OSError:
        # We don't know which process died, so we have to assume
        # that it exited normally and left no garbage
        pass
                                                                    330
_signal.signal(_signal.SIGCHLD, _on_SIGCHLD)


# Version of fork() that works with posh
def fork():
    """Posh's version of os.fork().

    Use this instead of os.fork() - it does the same.
    """
    global _has_forked                                              340

    pid = _os_fork()
    _has_forked = 1
    if not pid:
        # Child process
        _core.init_child()
    return pid


def forkcall(func, *args, **kwargs):                                350
    """forkcall(func, *args, **kwargs) -> pid of child process

    Forks off a child process that calls the first argument with
    the remaining arguments. The child process exits when the call
    to func returns, using the return value as its exit status.

    The parent process returns immediately from forkcall(), with
    the pid of the child process as the return value.
    """
    pid = fork()                                                    360
    if not pid:
        exit(func(*args, **kwargs))
    return pid


def waitall(pids=None):
    """waitall([pids]) -> None

    Waits for all the given processes to termintate.
    If called with no arguments, waits for all child processes to terminate.    370
    """
    if pids is None:
        try:
            while 1:
                wait()
```

```
        except OSError:
            pass
    else:
        for pid in pids:
            try:                                                    380
                waitpid(pid, 0)
            except OSError:
                pass
```

Index