Operations that replace other objects may invoke those other objects' `__del__()` method when their reference count reaches zero, and that can affect things. This is especially true for the mass updates to dictionaries and lists. When in doubt, use a mutex!

4.3.5 Can't we get rid of the Global Interpreter Lock?

The *global interpreter lock* (GIL) is often seen as a hindrance to Python's deployment on high-end multiprocessor server machines, because a multi-threaded Python program effectively only uses one CPU, due to the insistence that (almost) all Python code can only run while the GIL is held.

Back in the days of Python 1.5, Greg Stein actually implemented a comprehensive patch set (the "free threading" patches) that removed the GIL and replaced it with fine-grained locking. Adam Olsen recently did a similar experiment in his python-safethread project. Unfortunately, both experiments exhibited a sharp drop in single-thread performance (at least 30% slower), due to the amount of fine-grained locking necessary to compensate for the removal of the GIL.

This doesn't mean that you can't make good use of Python on multi-CPU machines! You just have to be creative with dividing the work up between multiple *processes* rather than multiple *threads*. The `ProcessPoolExecutor` class in the new `concurrent.futures` module provides an easy way of doing so; the `multiprocessing` module provides a lower-level API in case you want more control over dispatching of tasks.

Judicious use of C extensions will also help; if you use a C extension to perform a time-consuming task, the extension can release the GIL while the thread of execution is in the C code and allow other threads to get some work done. Some standard library modules such as `zlib` and `hashlib` already do this.

It has been suggested that the GIL should be a per-interpreter-state lock rather than truly global; interpreters then wouldn't be able to share objects. Unfortunately, this isn't likely to happen either. It would be a tremendous amount of work, because many object implementations currently have global state. For example, small integers and short strings are cached; these caches would have to be moved to the interpreter state. Other object types have their own free list; these free lists would have to be moved to the interpreter state. And so on.

And I doubt that it can even be done in finite time, because the same problem exists for 3rd party extensions. It is likely that 3rd party extensions are being written at a faster rate than you can convert them to store all their global state in the interpreter state.

And finally, once you have multiple interpreters not sharing any state, what have you gained over running each interpreter in a separate process?

4.4 Input and Output

4.4.1 How do I delete a file? (And other file questions...)

Use `os.remove(filename)` or `os.unlink(filename)`; for documentation, see the `os` module. The two functions are identical; `unlink()` is simply the name of the Unix system call for this function.

To remove a directory, use `os.rmdir()`; use `os.mkdir()` to create one. `os.makedirs(path)` will create any intermediate directories in `path` that don't exist. `os.removedirs(path)` will remove intermediate directories as long as they're empty; if you want to delete an entire directory tree and its contents, use `shutil.rmtree()`.

To rename a file, use `os.rename(old_path, new_path)`.

To truncate a file, open it using `f = open(filename, "rb+")`, and use `f.truncate(offset)`; offset defaults to the current seek position. There's also `os.ftruncate(fd, offset)` for files opened with `os.open()`, where *fd* is the file descriptor (a small integer).

The `shutil` module also contains a number of functions to work on files including `copyfile()`, `copytree()`, and `rmtree()`.