COLUMNS

Python

Shared Libraries and Python Packaging, an Experiment

PETER NORTON

Peter works on automating cloud environments. He loves using Python to solve problems. He has contributed to books on Linux and Python, helped with the New York Linux Users Group, and helped to organize past DevOpsDays NYC events. Even though he is a native New Yorker, he is currently living in and working from home in the northeast of Brazil. In addition to Python, Peter is slowly improving his knowledge of Rust, Clojure, and maybe other fun things. pcnorton@rbox.co.

I've been thinking about sharing some thoughts and experiments with the weird science experiment that is `memfd_create()`. It's a system call in somewhat recent versions of the Linux kernel—3.17 and later.

First let's take a trip back in time, and then we'll return to this system call with what I think is a really fun idea that could be used to explore and maybe improve an inconvenient aspect of Python packaging.

Shared Libraries

To get started, I want to talk about shared libraries.

When I was first exposed to UNIX systems in college, there was a tremendous amount of work being done to make the servers of the day more efficient. What computers of the day did was to act as time-sharing systems, allowing shell, compilation, mail, gopher, talk, netnews, and many other activities for multiple users. Like today, most users relied on software that the system administrator either compiled or installed as a package, which would benefit from the use of dynamically linked binaries.

These would help memory usage because by being dynamically linked, they were being linked at runtime to shared libraries. "Shared" in this case had more than one meaning. It meant both that they provide shared code—different programs could benefit from not having to write the same functions over and over—but by a neat trick it also meant that the read-only library codes that were used in *N* programs would all be mapped by the kernel into the same real set of bytes of memory, so each mapping of the library into a program only required a little memory overhead. This meant that even if 500 users loaded 100 KB of the same library code, via logging in and running, e.g., `pine` (which was at one point a very common mail reader), each instance of the program would see 100 KB of mappings getting linked in to its local memory, but over 500 invocations. But instead of using 50 MB of memory, an impossibly large amount at the time, all those invocations would use something more like 100 KB *total*, which is pretty cool, via some clever kernel memory mapping.

The involvement of the kernel is very important to bear in mind here. The shared mappings are done by the kernel and the dynamic linker (`ld.so` on Linux) working together to provide shared mappings of the library routines into each process's virtual memory space at an address that is only known when it's loaded. They are then "fixed-up" at runtime to point to newly assigned addresses so the executable can find them. If you've ever wondered why your Python extension modules are always compiled and linked with the `-fpic` or `-fPIC` flag, that's why. (See http://bottomupcs.sourceforge.net/csbu/c3673.htm for more about the mechanisms that are involved here.)

Even back then, you could share actual routines without bringing shared libraries into the picture by statically linking libraries. This is much simpler, but in the era where powerhouse workstations had 8 MB of memory, they didn't tell a good story about memory efficiency.

In modern systems, shared libraries aren't often first and foremost thought of as ways of saving memory by shared mappings between different processes. In fact they're often seen as a waste of effort! With the recent abundance of memory available to systems, and the huge amount of data we're processing with that memory, the savings from the shared memory part of shared libraries that I described above has become a bit of an anachronism.

Especially when using Python, the "shared" part of shared libraries has become more about sharing C code with the Python runtime, making libraries able to be invoked from the interpreter. The benefits of this are so common they're almost a running joke—most answers to questions about making Python faster, for example, usually quickly bring up the answer, "Use `cython`" or "Write it in C and load the faster implementation from the shared library." From a more practical standpoint, a pillar of the Python community is the scientific Python stack built on top of NumPy and SciPy, and it goes one step further where FORTRAN code is built and linked so that it is compatible with C calling conventions, and then Python loads the resulting libraries for fast matrix math. Python obviously has to do more than "just load the library" for this to work, but that's where the rubber meets the road, so to speak.

Packaging

Now, the more common of these libraries are usually packaged up by the operating system maintainer—Debian, Red Hat, etc. if you're a fellow Linux user—or someone who fits into that job if your *nix is a different *nix. But once it's built, a shared lib can be dynamically linked by a Python runtime, whether it's packaged by the operating system maintainer, built yourself, or obtained from a third party like a scientific Python packager.

There was a time when GNU `autoconf` was pretty cutting-edge. It is now considered quite unwieldy. Its heyday was in a world with literally dozens of operating systems that were sort-of-but-not-quite like a POSIX or BSD UNIX, and nothing built for one would compile on any others without inhuman knowledge of different CPU architectures, C compilers, and luck.

That was then, and the world is much simpler now (for UNIXes at least), and that's led to the current generation of popular languages being able to do better than `./configure`. Now instead of just producing a runnable program and maybe making it easy to copy the results to your local file system, modern build toolchains will also package up your work, and often turn them into a tidy single-file image that can just be executed. Golang is arguably the king of this category, where one of its main selling points is that when building your program, you will create a static binary—that's it!

Since the modern lifecycle for programs involves multiple deployments per day, there is a lot of appeal to the idea of being able to bundle up a single artifact containing everything a program needs. The prospect of having no external libraries to depend on and no OS packages to install prior—just being able to copy a file and being able to just run the program has become the gold standard of new compiled languages, and once you've done this, it's pretty nice. Golang, Java, and Rust do a great job with having their tooling provide this experience, and they set a standard for other languages to shoot for.

Python has an interesting story in this respect. Python will open a zip file that contains Python code, if the appropriate structure is in place. This is described in PEP 273, and there is some more info in PEP 441. This is the core of some cool stuff that you can get from PyPI, including pre-packaged wheels, eggs, and, outside of PyPI, other less geometrically named things like pexes and pars.

Having all of your dependencies in one place is pretty nice. You don't have to install anything special, you can just point the appropriate Python interpreter at a built zip file and get a really nice experience—both as a developer since the build process is not complicated and as a sysadmin; as long as the version of the Python interpreter is a good match for the application in the archive, you have a pretty good chance at deploying and getting a good night's sleep, too. On the face of it, something as convenient as, perhaps, Java jars. And just about the nicest thing about it is that you simply don't have to worry about installing OS packages or other dependencies.

However, there is one major weak point in using zip archives with Python. Specifically it has to do with shared libraries. If you want to have a shared library in your package, the dynamic linker on your platform, together with the kernel can't map that bit of the archive file into the running program!

That's not the end of the story, though. There is a simple hack that makes these zip archives work: the packaging tool that works with zip files will unzip a shared object from the zip file, write it out to disk, and then use the dynamic loader to make it available to your programs.

Get that? It extracts the library from the zip archive to plant it onto disk. The reason is that neither the Linux kernel nor other UNIX kernels that I'm aware of have special magic to allow portions of a zipfile to be used as a shared memory mapping for a shared library. This means that when you have a zipped-up Python archive for a project that uses common facilities that are best used via shared libraries—like MySQL, gRPC, XML, or what have you—and you want to include them in your bundled artifact in order to guarantee that there aren't dangling, unresolved dependencies, this zip-file format will need to do a few things that you'd prefer not to do:

1. Use space on disk—at least temporarily
2. Use additional disk reads+writes
3. Use additional CPU time at startup and shutdown of the program

None of that seems prohibitive, but in my experience, it can be really demoralizing when you find out that `/tmp` has filled up with detritus from your project, or when you learn that the zip file will get extracted into the running user's home directory, and that user isn't supposed to write there. Or whatever other difficulties your site may discover down in the weedy details of the specific process.

Now, returning to that cool thing I mentioned at the beginning. I heard about a pretty neat new feature in Linux a few months back. It's a system call, `memfd_create()`, that allows us to turn a region in memory into a file in `/proc/<pid>/fd/<the fd number>`. What's really interesting is that it acts like a normal file, which includes being able to be symlinked from other parts of the file system.

So, wanting to reproduce an idea that I'd heard about from some of the super tech companies, I thought it would be fascinating to have the kernel be able to map sections of the zip files—the shared libraries in particular—so that it could be used as a shared library.

This system call doesn't do exactly that, but it seems like it could get us closer to the goal of all-in-one packaging without having to extract to the file system. This works by consuming memory instead of file system space and disk I/O. The question is whether the presence of an appropriate mapping would prevent the dynamic linker from trying to load a library from the system (it should as long as the dependencies are resolved appropriately). If this worked, it would allow `libmysql.so`, for example, to be packaged up and shipped.

And it turns out that as a toy, this seems to work! The core of this is some interesting syscall work that Python lets you do via the ctypes library using the CDDL call, which maps in the library via `dlopen()`. It's pretty nifty—we can load up `libc` in order to get a hold of the syscall we want, then map in the file we have in the archive as bytes, creating the library in memory, and then use CDDL again to load it up.

An outtake of the code, which you can find at https://github.com/pcn/pymyxec, looks like this:

```
# Need to get memfd_create(), which is now in the
# syscall table at 319
# Returns the FD number
def build_a_lib(lib_name, source_bytes):
    memfd_create = 319
    libc = CDLL("libc.so.6")
    print("Lib name is {}".format(lib_name))
    so_file_name = "{}.so".format(lib_name)
    fd = libc.syscall(memfd_create, so_file_name, 0)
    for data in source_bytes:
        os.write(fd, data)
    CDLL("/proc/self/fd/{}".format(fd))
    return fd
```

I'm still smiling and happy at how this has worked. Running this as documented in the README.adoc in the repo shows how:

```
spacey@masonjar:~/dvcs/pcn/pymyxec$ bazel build
        mysql_repl.par; bazel-bin/mysql_repl.par
INFO: Analysed target //:mysql_repl.par (1 packages loaded).
INFO: Found 1 target...
Target //:mysql_repl.par up-to-date:
  bazel-bin/mysql_repl.par
INFO: Elapsed time: 1.389s, Critical Path: 0.84s
INFO: 1 process, linux-sandbox.
INFO: Build completed successfully, 2 total actions
Python 2.7.15rc1 (default, Apr 15 2018, 21:51:34)
[GCC 7.3.0] on linux2
Type "help", "copyright", "credits" or "license"
        for more information.
(InteractiveConsole)
>>> clientinfo = entry("libmysqlclient.so", "__main__
/libmysqlclient.so")
You got it
[u'bazel-bin/mysql_repl.par/pypi__certifi_2018_4_16', u'bazel
-bin/mysql_repl.par/pypi__chardet_3_0_4', u'bazel-bin/mysql
_repl.par/pypi__idna_2_7', u'bazel-bin/mysql_repl.par/pypi
__urllib3_1_23', u'bazel-bin/mysql_repl.par/pypi
__requests_2_19_1', u'bazel-bin/mysql_repl.par/pypi
__docopt_0_6_2', u'bazel-bin/mysql_repl.par/pypi__MySQL
_python_1_2_5', 'bazel-bin/mysql_repl.par', u'bazel-bin/mysql
_repl.par/__main__', '/usr/lib/python2.7', '/usr/lib/python2.7
/plat-x86_64-linux-gnu', '/usr/lib/python2.7/lib-tk', '/usr/lib
/python2.7/lib-old', '/usr/lib/python2.7/lib-dynload', '/home
/spacey/.local/lib/python2.7/site-packages', '/usr/local/lib
/python2.7/dist-packages', '/usr/lib/python2.7/dist-packages',
'/usr/lib/python2.7/dist-packages/gtk-2.0']
<zipfile.ZipExtFile object at 0x7ff59c934190>
Lib name is libmysqlclient.so
this pid is 14200, lib_fd is 14200
>>> modinfo = entry("_mysql",
"pypi__MySQL_python_1_2_5/_mysql.so")
You got it
[u'bazel-bin/mysql_repl.par/pypi__certifi_2018_4_16',
u'bazel-bin/mysql_repl.par/pypi__chardet_3_0_4',
  ...
<zipfile.ZipExtFile object at 0x7ff59c9341d0>
Lib name is _mysql
this pid is 14200, lib_fd is 14200
>>> link_a_lib("_mysql.so", modinfo[0], modinfo[1])
>>> import MySQLdb
```

At the end of that, we can validate that the process is using the library that we've loaded, the shared library in the par file, and not the shared library installed on the file system.

```
 (aws) spacey@masonjar:~/dvcs/pcn/pymyxec$ pmap 14200 |
        grep -i mysql
14200:   python bazel-bin/mysql_repl.par
00007ff59b4d0000     40K r-x-- memfd:_mysql.so (deleted)
00007ff59b4da000   2044K ----- memfd:_mysql.so (deleted)
00007ff59b6d9000      4K r---- memfd:_mysql.so (deleted)
00007ff59b6da000     16K rw--- memfd:_mysql.so (deleted)
00007ff59bc84000   3656K r-x-- memfd:libmysqlclient.so.so
(deleted)
00007ff59c016000   2048K ----- memfd:libmysqlclient.so.so
(deleted)
00007ff59c216000     24K r---- memfd:libmysqlclient.so.so
(deleted)
00007ff59c21c000    456K rw--- memfd:libmysqlclient.so.so
(deleted)
```

This shows that the MySQL libraries that are mapped in are only those that were mapped in via ctypes.CDLL, which is doing the equivalent of a `dlopen()` call and mapping in the library. It also shows that I should update the README on GitHub with one less `.so`. The `(deleted)` is just pmap showing that it thinks the underlying file used to create the mapping was deleted.

It would be nice if `libmysql.so` could be read without having to symlink it into `/tmp` as in the previous example, but using an existing module like this, with a compiled shim library, doesn't give me that flexibility—though someone smarter than I may have an idea about how to do that. Pull requests are welcome!

As a closing thought, one remote possibility would be to see how far we could go with this. For example, could it be possible to store a subset of the required Python support files? Enough of what an interpreter needs from `lib/python<version>/` could be included into the archive, ideally in a way that `memfd_create` could be used to populate, say, a virtualenv with a bunch of symlinks into `/proc/self/fd/<various pids>`, and that virtualenv and the Python interpreter would be entirely spun from the zip file. That way the appropriate Python binary, built and tested as part of the package, would be bootstrapped by the system Python.

I don't know if anyone is interested in that, but if so maybe it'd be a good incentive for me to try to do something with Python 3.

Cheers, and have a great day. I hope this helps you smile a bit.