Porting Python

From OSDev Wiki

This page is a work in progress and may thus be incomplete. Its content may be changed in the near future.

Difficulty level

🟥🟥🟥🟥
Master

> This article is a bit out of date, poll User:Sortie or check his patch to Python 3.4 (https://users-cs.au.dk/sortie/sortix/release/nightly/patches/python.patch) .

Python is a popular scripting language that, if ported, can allow you to do a lot of powerful tasks using simple scripts. I gained a lot from this (http://www.ailis.de/~k/archives/19-ARM-cross-compiling-howto.html) and this (http://whatschrisdoing.com/blog/2006/10/06/howto-cross-compile-python-25/) . Amazing what a simple Google search can do!

The problem with porting Python is that the Python build process actually wants to use the built Python to build the rest of the modules that come with Python (eg, C modules). The build system also does not build build tools with the --build compiler, which means the build can fail quite early if you aren't prepared.

Contents

- 1 Prerequisites
- 2 Freebies
- 3 The Process
- 4 setup.py
- 5 Caveats
- 6 What if my host toolchain uses a different shared object format?
- 7 Works On
- 8 Final Word

Prerequisites

Porting Python is **not** for the faint of heart. You should know a lot about configure scripts and the options of your toolchain.

You will also need a cross toolchain that supports C++ (See OS Specific Toolchain) and has a C++ standard library.

Freebies

I have the patches I use for Python 2.6 and Python 2.7 here:

- Python 2.6 (http://www.pedigree-project.org/projects/pedigree-apps/repository/revisions/master/show/packages/python26/patches)
- Python 2.7 (http://www.pedigree-project.org/projects/pedigree-apps/repository/revisions/master/show/packages/python27/patches)

In the directory above the patches directory each link links to, you can find the scripts I use to automate the Python build.

The Process

Download the Python 2.5 source code. You need to patch it using this patch (http://whatschrisdoing.com/~lambacck/Python2.5_xcompile.patch) in order to make the configure process work properly for the cross-compile.

Basically, this patched configure script now uses two variables: HOSTPYTHON and HOSTPGEN in the later stages of the build. See, Python's build process builds Python itself, then uses the built Python binaries to build the modules. In the case of a cross-compiler, this won't work, so we need to point Python to a working Python binary.

Run:

```
./configure && make python Parser/pgen
```

in the Python-2.5 directory.

This will build a version of Python using your host compiler.

Now we need to make that the host Python and get ready for the real cross compile:

```
mv python hostpython
mv Parser/pgen Parser/hostpgen
make distclean
```

This is where things get complicated. We now have to configure for the new target. Before you do this, export CC, CXX, AR and RANLIB for your target. In my case, I'll use "i686-myos":

```
./configure --host=i686-myos --prefix=/usr/python-cross
```

If things go wrong, you need to make sure your libc and libstdc++ provides the necessary functionality.

At this stage, Python is configured to use your cross toolchain. Now we build Python:

```
make python Parser/pgen
```

Right, that wasn't so hard, was it? So now you have the Python binary ready to run on your OS. However, it's missing libraries and modules. Let's build them now:

```
make HOSTPYTHON=./hostpython HOSTPGEN=./Parser/hostpgen BLDSHARE
```

You will need to edit "setup.py" to modify the default include and libraries directories to those which your compiler uses. See below ("setup.py") for the changes you need to make.

Basically, the Makefile now calls "$HOSTPYTHON ./setup.py" to continue the build. That's why we need to set HOSTPYTHON now. Another important thing to mention is BLDSHARED. If your cross-toolchain can't build shared objects, you need to use one that can. If you're on Linux and you're using ELF in your OS, just set BLDSHARED="gcc -shared". If not, you're out of luck until your cross-toolchain can build shared objects.

Your modules will now build. Keep track of error messages if you can, there will be modules that won't work until you add more functionality to your libc.

To install, run:

```
make install prefix=<install_dir> HOSTPYTHON=./hostpython HOSTPG
```

And you're done!

setup.py

This is where things can get really complicated. setup.py is what Python uses to build the modules for your Python port - however it doesn't work so well for cross-compiling. I'll assume you've already applied the patch specified above, so let's get to it!

NOTE: Make sure you watch the surrounding indentation when making these changes. Python knowledge would really help in this case. A lot of these changes I'm putting up here without indentation!

The first change we make is at the line "disabled_module_list":

```
disabled_module_list = ['_ctypes', 'syslogmodule', 'nismodule']
```

This is pretty straightforward. If you're going to put anything here, make it '_ctypes'.

Find "def detect_modules(self)". See at the top how it has those two directories? Let's change that...

```
#       add_dir_to_list(self.compiler.library_dirs, '/usr/local/
#       add_dir_to_list(self.compiler.include_dirs, '/usr/local/
        add_dir_to_list(self.compiler.library_dirs, '/usr/pedigr
        add_dir_to_list(self.compiler.library_dirs, '/usr/pedigr
        add_dir_to_list(self.compiler.include_dirs, '/usr/pedigr
        add_dir_to_list(self.compiler.library_dirs, '/usr/pedigr
```

Of course, you'll update these to the correct paths for your cross-toolchain. These will become not only the search paths for libraries but also the command line for the GCC invocation ("-I" and "-L" arguments).

Further down there's a block setting "lib_dirs" to a bunch of folders such as "/lib64", "/usr/lib64", "/lib", and "/usr/lib". Beneath it is a line that sets up the include directories including "/usr/include". Change this to:

```
lib_dirs = self.compiler.library_dirs
inc_dirs = self.compiler.include_dirs
```

And if we keep on heading down just a couple of lines there's a "platform = self.get_platform()". The platform is used to make some assumptions further down, so we should set it to something like our OS name:

```
platform = "pedigree"
```

Halfway there! Find the line "if platform == 'darwin':" and add above it something like this:

```
        if platform == 'pedigree':
          lib_dirs += ['/usr/pedigree-cross/lib']
          inc_dirs += ['/usr/pedigree-cross/i686-pedigree/includ
```

This just adds some directories to the list that we need for the platform. I will admit this is slightly redundant for this example (apart from the ncurses headers) - feel free to skip this if you've already covered it above with the "add_dir_to_list" lines. If you have any extra directories that you want the compile to use you can add them here (eg, unusual libraries locations or include files).

If you have an OpenSSL port (and you can compile against it in your cross-toolchain), skip past this change.

If not, find the lines starting with "ssl_incs = find_file...." and change to:

```
        ssl_incs = None
        #find_file('openssl/ssl.h', inc_dirs,
        #                      search_for_ssl_incs_in
        #                      )
```

This just removes the SSL header from the search path, so you don't get a bunch of undefined references.

Now we're pretty much done. Keep going down until you find the Berkeley DB stuff (starts with "db_inc_paths", comment out all of the paths, keep the []. Comment out all four of the for() loops (they should be completely commented out). comment out the "for dn in inc_dirs" for block, and finally change the "db_inc_paths" variable as below:

```
        db_inc_paths = [] # std_variants + db_inc_paths
```

The last step is to comment out the sqlite include paths (search for sqlite_inc_paths, again, keep the []).

Save, exit your editor, and get compiling!

Caveats

You will need dlopen, dlsym and dlclose to be able to do things like "import socket" which indirectly imports "_socket" (<prefix>/lib/python2.5/lib-dynload/_socket.so). By this stage you should already have shared object support so this should be simple!

What if my host toolchain uses a different shared object format?

Let's say you're on Cygwin, and your toolchain targets ELF. You're a little bit stuck because you can't build any modules, because you can't define BLDSHARED to "gcc -shared" (you'd get a DLL), and your ELF toolchain may not support shared objects. There is, of course, a workaround for this. Your BLDSHARED will look a bit like this:

```
BLDSHARED="i686-myos-gcc -nostdlib -shared -Wl,-shared"
```

That should build a shared object that does not rely on any other libraries.

Works On

I've done this on:

- Python 2.2
- Python 2.5
- Python 2.5.4 (the patch does **not** work - you'll need to read through the patch file and make the changes manually.)
- Python 2.6.2
- Python 2.6.6
- Python 2.7.3

Final Word

Porting Python is really not that complex if you just work around some minor complexities that the build process uses. Good luck!