Networking without an OS

Josh Triplett
josh@joshtriplett.org

PyCon 2016

PYCON 2016

ROSE

CITY

PORTLAND, OREGON

MAY 28TH – JUNE 5TH

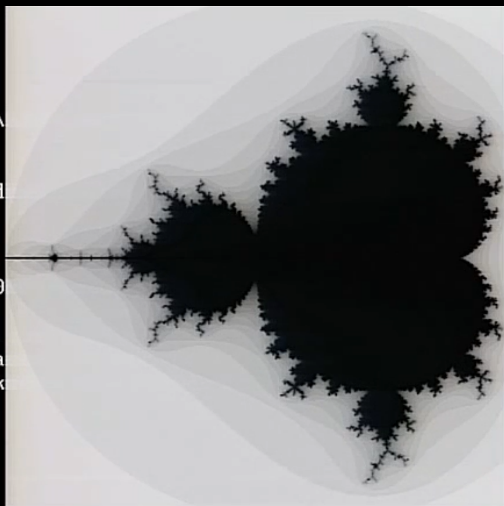
PyCon 2015
Montréal • April 8-16

Porting Python to run without an OS

PyCon 2015

- BIOS Implementation Test Suite (BITS)
- Python in GRUB and EFI, without an OS
- Explore and test hardware and firmware

```
0L
>>> out.OutputString(out, "Hello world!\
Hello world!
0L
>>> fb = (c_uint32 * 800 * 600).from_add
meBufferBase)
>>> for y in range(400):
...     for x in range(400):
...         zx,zy = cx,cy = -2+2.5*x/400
...         for i in range(25):
...             zx,zy = zx*zx-zy*zy+cx,
...             if zx*zx+zy*zy > 4: brea
...         fb[y][x+400] = (250-10*i)*0x
...
>>> _
```



```
Starting the Python interactive interpreter. Press Ctrl-D or Esc to exit.
>>> fb = (c_uint32 * 800 * 600).from_address(bits.present.gop.Mode.contents.Fra
meBufferBase)
>>> import time
>>> def clear():
...   out.ClearScreen(out)
...
>>> def game():
...   k = EFI_KEY_DATA()
...   x = y = 10; xv = 1; yv = 0
...   while True:
...     _ = t.ReadKeyStrokeEx(t, k)
...     if k.scan == 23: break
...     elif k.scan >= 1 and k.scan <= 4:
...       xv = (0,0,1,-1)[k.scan-1]
...       yv = (-1,1,0,0)[k.scan-1]
...     x += xv ; y += yv
...     t1 = time.time()
...     while time.time() - t1 < 0.02: pass
...     if fb[y][x] == 0x0000ff00: break
...     fb[y][x] = 0x0000ff00
...
>>> game()
>>> _
```

BITS Capabilities

- Interactive Python interpreter (with line editing and tab completion)

BITS Capabilities

- Interactive Python interpreter (with line editing and tab completion)
- Direct access to hardware and physical memory

BITS Capabilities

- Interactive Python interpreter (with line editing and tab completion)
- Direct access to hardware and physical memory
- Python can call EFI firmware protocols via ctypes

BITS Capabilities

- Interactive Python interpreter (with line editing and tab completion)
- Direct access to hardware and physical memory
- Python can call EFI firmware protocols via ctypes
- Most of the Python standard library

Most of the Python standard library

Most of the Python standard library

Some modules don't make sense without an OS

os.execve

os.fork

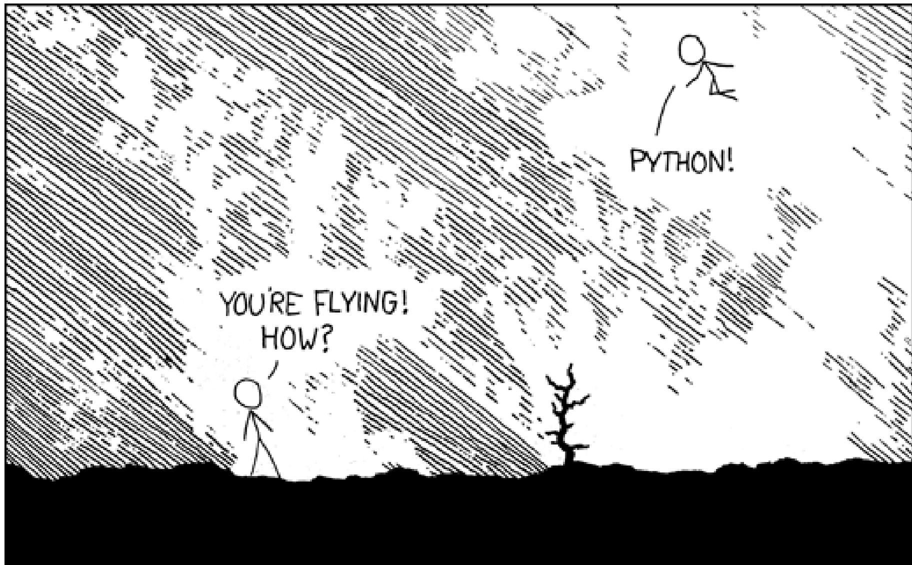
os.execve
os.fork

multiprocessing
popen2
subprocess

os.execve
os.fork

multiprocessing
popen2
subprocess

webbrowser



import antigravity

socket
select

socket
select

urllib2
httplib
SocketServer
BaseHTTPServer

EFI networking protocols

- EFI_IP4_CONFIG2_PROTOCOL
- EFI_TCP4_PROTOCOL

Why networking in firmware?

- Send scripts or test data into the machine

Why networking in firmware?

- Send scripts or test data into the machine
- Read test data or logs from the machine

Why networking in firmware?

- Send scripts or test data into the machine
- Read test data or logs from the machine
- Avoid relying on a writable filesystem

Why networking in firmware?

- Send scripts or test data into the machine
- Read test data or logs from the machine
- Avoid relying on a writable filesystem
- Speed up edit/compile/boot/run cycle

Demo

Bridging EFI networking and Python sockets

- Could call EFI network protocols directly
- Want compatibility with existing Python networking code
- Python modules import `socket` and `select`
- `socket` (Python) imports `_socket` (C)

Sockets overview

- “Berkeley” sockets

Sockets overview

- "Berkeley" sockets
- Standard on UNIX/POSIX systems, and on Windows via WinSock

Sockets overview

- "Berkeley" sockets
- Standard on UNIX/POSIX systems, and on Windows via WinSock
- Focusing exclusively on TCP/IP connections

Creating a socket

- `int s = socket(AF_INET, SOCK_STREAM, 0);`

Creating a socket

- `int s = socket(AF_INET, SOCK_STREAM, 0);`
- `AF_INET` - IP

Creating a socket

- `int s = socket(AF_INET, SOCK_STREAM, 0);`
- `AF_INET` - IP
- `SOCK_STREAM` - TCP

Creating a socket

- `int s = socket(AF_INET, SOCK_STREAM, 0);`
- `AF_INET` - IP
- `SOCK_STREAM` - TCP
- Can use `s` as either client or server socket

Client socket

- `socket`

Client socket

- `socket`
- `connect`

Client socket

- `socket`
- `connect`
  - `struct sockaddr`

Client socket

- `socket`
- `connect`
  - `struct sockaddr`
- `send/recv`

Client socket

- `socket`
- `connect`
  - `struct sockaddr`
- `send/recv`
- `close`

Server socket

- `socket`

Server socket

- `socket`
- `bind`

Server socket

- `socket`
- `bind`
  - `struct sockaddr`

Server socket

- `socket`
- `bind`
  - `struct sockaddr`
- `listen`

Server socket

- `socket`
- `bind`
  - `struct sockaddr`
- `listen`
- `accept` - returns a new connected socket

Server socket

- `socket`
- `bind`
  - `struct sockaddr`
- `listen`
- `accept` - returns a new connected socket
  - `send/recv`

Server socket

- socket
- bind
  - struct sockaddr
- listen
- accept - returns a new connected socket
  - send/recv
  - close

select - waiting for activity

- Pass sets of file descriptors to monitor for reading and for writing

select - waiting for activity

- Pass sets of file descriptors to monitor for reading and for writing
  - And for exceptions, but ignoring that here

select - waiting for activity

- Pass sets of file descriptors to monitor for reading and for writing
  - And for exceptions, but ignoring that here
- Pass a timeout

select - waiting for activity

- Pass sets of file descriptors to monitor for reading and for writing
  - And for exceptions, but ignoring that here
- Pass a timeout
- Waits for a connected socket to have data to `recv` or `send`

select - waiting for activity

- Pass sets of file descriptors to monitor for reading and for writing
  - And for exceptions, but ignoring that here
- Pass a timeout
- Waits for a connected socket to have data to `recv` or `send`
- Waits for a listening socket to have a connection

select - waiting for activity

- Pass sets of file descriptors to monitor for reading and for writing
  - And for exceptions, but ignoring that here
- Pass a timeout
- Waits for a connected socket to have data to `recv` or `send`
- Waits for a listening socket to have a connection
- Core of the main loop in most network servers

select - waiting for activity

- Pass sets of file descriptors to monitor for reading and for writing
    - And for exceptions, but ignoring that here
- Pass a timeout
- Waits for a connected socket to have data to `recv` or `send`
- Waits for a listening socket to have a connection
- Core of the main loop in most network servers
- Many OS-specific replacements for scalability and performance

Python bindings

- `import socket, select`

Python bindings

- `import socket, select`
- `s = socket.socket()` - defaults to TCP/IP

Python bindings

- `import socket, select`
- `s = socket.socket()` - defaults to TCP/IP
- `s.connect`

Python bindings

- `import socket, select`
- `s = socket.socket()` - defaults to TCP/IP
- `s.connect`
- `s.bind`, `s.listen`, `s.accept`

Python bindings

- `import socket, select`
- `s = socket.socket()` - defaults to TCP/IP
- `s.connect`
- `s.bind`, `s.listen`, `s.accept`
- `s.sendall`, `s.recv`

Python bindings

- `import socket, select`
- `s = socket.socket()` - defaults to TCP/IP
- `s.connect`
- `s.bind`, `s.listen`, `s.accept`
- `s.sendall`, `s.recv`
- ```
  rl,wl,xl = select.select([s],[],[])
  if s in rl:
  ```
  ```

CPython implementation

- `socketmodule.c` and `selectmodule.c`

CPython implementation

- `socketmodule.c` and `selectmodule.c`
- Extensive dependencies on POSIX and on C sockets API

CPython implementation

- `socketmodule.c` and `selectmodule.c`
- Extensive dependencies on POSIX and on C sockets API
- Would have to implement those APIs in C

CPython implementation

- `socketmodule.c` and `selectmodule.c`
- Extensive dependencies on POSIX and on C sockets API
- Would have to implement those APIs in C
  - Handle C arguments, addresses, buffer management

CPython implementation

- `socketmodule.c` and `selectmodule.c`
- Extensive dependencies on POSIX and on C sockets API
- Would have to implement those APIs in C
  - Handle C arguments, addresses, buffer management
- Would have to call EFI protocols from C

CPython implementation

- `socketmodule.c` and `selectmodule.c`
- Extensive dependencies on POSIX and on C sockets API
- Would have to implement those APIs in C
  - Handle C arguments, addresses, buffer management
- Would have to call EFI protocols from C
  - Or, have many callbacks from C to Python

BITS implementation

- C helper for safe asynchronous event handling
- Otherwise entirely Python
- Python makes all EFI protocol calls

Calling EFI from Python via ctypes

Behind the scenes

- `efi.system_table` is a data structure

Behind the scenes

- `efi.system_table` is a data structure
- `efi.system_table.ConOut` is a pointer to a protocol

Behind the scenes

- `efi.system_table` is a data structure
- `efi.system_table.ConOut` is a pointer to a protocol
- `.contents` dereferences a ctypes pointer

Behind the scenes

- `efi.system_table` is a data structure
- `efi.system_table.ConOut` is a pointer to a protocol
- `.contents` dereferences a ctypes pointer
- Most EFI calls expect a “this” pointer

Behind the scenes

- `efi.system_table` is a data structure
- `efi.system_table.ConOut` is a pointer to a protocol
- `.contents` dereferences a ctypes pointer
- Most EFI calls expect a "this" pointer
- ctypes converts `"Hello world\r\n"` to a Unicode string

Behind the scenes

- `efi.system_table` is a data structure
- `efi.system_table.ConOut` is a pointer to a protocol
- `.contents` dereferences a ctypes pointer
- Most EFI calls expect a “this” pointer
- ctypes converts `"Hello world\r\n"` to a Unicode string
- ctypes returns the error code from EFI

Resource management

- EFI uses manual memory management

Resource management

- EFI uses manual memory management
- Python uses garbage collection

Resource management

- EFI uses manual memory management
- Python uses garbage collection
- Python GC doesn't know about references from EFI

Resource management

- EFI uses manual memory management
- Python uses garbage collection
- Python GC doesn't know about references from EFI
- Must keep Python object alive as long as EFI references it

Resource management

- EFI uses manual memory management
- Python uses garbage collection
- Python GC doesn't know about references from EFI
- Must keep Python object alive as long as EFI references it
- Must explicitly free EFI resources when no longer referenced from Python

EFI networking protocols

- EFI_IP4_CONFIG2_PROTOCOL

EFI networking protocols

- EFI_IP4_CONFIG2_PROTOCOL
  - Read current IP configuration, or start IP configuration

EFI networking protocols

- EFI_IP4_CONFIG2_PROTOCOL
  - Read current IP configuration, or start IP configuration
- EFI_TCP4_SERVICE_BINDING_PROTOCOL

EFI networking protocols

- EFI_IP4_CONFIG2_PROTOCOL
  - Read current IP configuration, or start IP configuration
- EFI_TCP4_SERVICE_BINDING_PROTOCOL
  - Create a new EFI_TCP4_PROTOCOL, like socket()

EFI networking protocols

- `EFI_IP4_CONFIG2_PROTOCOL`
  - Read current IP configuration, or start IP configuration
- `EFI_TCP4_SERVICE_BINDING_PROTOCOL`
  - Create a new `EFI_TCP4_PROTOCOL`, like `socket()`
- `EFI_TCP4_PROTOCOL`

EFI networking protocols

- EFI_IP4_CONFIG2_PROTOCOL
  - Read current IP configuration, or start IP configuration
- EFI_TCP4_SERVICE_BINDING_PROTOCOL
  - Create a new EFI_TCP4_PROTOCOL, like socket()
- EFI_TCP4_PROTOCOL
  - EFI socket API: Configure, Connect, Accept, Transmit, Receive, Close

EFI networking protocols

- EFI_IP4_CONFIG2_PROTOCOL
  - Read current IP configuration, or start IP configuration
- EFI_TCP4_SERVICE_BINDING_PROTOCOL
  - Create a new EFI_TCP4_PROTOCOL, like socket()
- EFI_TCP4_PROTOCOL
  - EFI socket API: Configure, Connect, Accept, Transmit, Receive, Close

EFI networking protocols

- EFI_IP4_CONFIG2_PROTOCOL
  - Read current IP configuration, or start IP configuration
- EFI_TCP4_SERVICE_BINDING_PROTOCOL
  - Create a new EFI_TCP4_PROTOCOL, like socket()
- EFI_TCP4_PROTOCOL
  - EFI socket API: Configure, Connect, Accept, Transmit, Receive, Close

Glossing over quirks, bugs, error handling, workarounds, and compatibility with older versions

Impedance mismatch: select versus Receive/Accept

- `select` checks for pending data or connections

Impedance mismatch: select versus Receive/Accept

- `select` checks for pending data or connections
  - Does not read data or accept connection

Impedance mismatch: select versus Receive/Accept

- `select` checks for pending data or connections
  - Does not read data or accept connection
- EFI can only check for data by calling `Receive` with a valid buffer

Impedance mismatch: select versus Receive/Accept

- `select` checks for pending data or connections
  - Does not read data or accept connection
- EFI can only check for data by calling `Receive` with a valid buffer
- Solution: buffer received data, call `Receive` when buffer empty

Impedance mismatch: select versus Receive/Accept

- `select` checks for pending data or connections
  - Does not read data or accept connection
- EFI can only check for data by calling `Receive` with a valid buffer
- Solution: buffer received data, call `Receive` when buffer empty
- Likewise for `Accept`

Impedance mismatch: select versus Receive/Accept

- `select` checks for pending data or connections
  - Does not read data or accept connection
- EFI can only check for data by calling `Receive` with a valid buffer
- Solution: buffer received data, call `Receive` when buffer empty
- Likewise for `Accept`
- Similar to the implementation of sockets in an OS kernel

Impedance mismatch: Poll

- `EFI_TCP4_PROTOCOL` not updated directly from low-level interrupts

Impedance mismatch: Poll

- EFI_TCP4_PROTOCOL not updated directly from low-level interrupts
- Data processed infrequently, even with asynchronous call running

Impedance mismatch: Poll

- `EFI_TCP4_PROTOCOL` not updated directly from low-level interrupts
- Data processed infrequently, even with asynchronous call running
- Caller expected to call `Poll` periodically if waiting

Impedance mismatch: Poll

- EFI_TCP4_PROTOCOL not updated directly from low-level interrupts
- Data processed infrequently, even with asynchronous call running
- Caller expected to call Poll periodically if waiting
  - Improves performance by orders of magnitude

Impedance mismatch: Poll

- EFI_TCP4_PROTOCOL not updated directly from low-level interrupts
- Data processed infrequently, even with asynchronous call running
- Caller expected to call Poll periodically if waiting
  - Improves performance by orders of magnitude
- Solution: call Poll inside helpers for select

Impedance mismatch: asynchronous callbacks

- All EFI socket calls asynchronous

Impedance mismatch: asynchronous callbacks

- All EFI socket calls asynchronous
- Calls take a "completion token" with `EFI_EVENT` to signal when done

Impedance mismatch: asynchronous callbacks

- All EFI socket calls asynchronous
- Calls take a "completion token" with `EFI_EVENT` to signal when done
- For sockets, `EFI_EVENT` must have a callback function

Impedance mismatch: asynchronous callbacks

- All EFI socket calls asynchronous
- Calls take a "completion token" with `EFI_EVENT` to signal when done
- For sockets, `EFI_EVENT` must have a callback function
- Need to handle callback safely from Python

Python concurrency model

- Python expects bytecode ops and C calls to run to completion

Python concurrency model

- Python expects bytecode ops and C calls to run to completion
- Global Interpreter Lock (GIL)

Python concurrency model

- Python expects bytecode ops and C calls to run to completion
- Global Interpreter Lock (GIL)
- Data may have inconsistent state when callback occurs

Python concurrency model

- Python expects bytecode ops and C calls to run to completion
- Global Interpreter Lock (GIL)
- Data may have inconsistent state when callback occurs
- Almost all CPython functions prohibited

Python concurrency model

- Python expects bytecode ops and C calls to run to completion
- Global Interpreter Lock (GIL)
- Data may have inconsistent state when callback occurs
- Almost all CPython functions prohibited
- Same problem arises with Ctrl-C and signals

Py_AddPendingCall

- Register a callback (with context)

Py_AddPendingCall

- Register a callback (with context)
- Python calls it at the next safe point

Py_AddPendingCall

- Register a callback (with context)
- Python calls it at the next safe point
- Can call arbitrary CPython functions from the callback

Handling events

- C module provides event callback function pointer

Handling events

- C module provides event callback function pointer
- Python code creates `EFI_EVENT` with C callback

Handling events

- C module provides event callback function pointer
- Python code creates `EFI_EVENT` with C callback
- C callback invokes universal Python callback

Handling events

- C module provides event callback function pointer
- Python code creates EFI_EVENT with C callback
- C callback invokes universal Python callback
- Python callback dispatches to event-specific callback via dict

Handling events

- C module provides event callback function pointer
- Python code creates `EFI_EVENT` with C callback
- C callback invokes universal Python callback
- Python callback dispatches to event-specific callback via dict
- dict keeps Python objects live while `EFI_EVENT` references them

Implementing `select`

- Takes read and write lists of sockets

Implementing `select`

- Takes read and write lists of sockets
  - Or file descriptors; map to sockets via dict

Implementing `select`

- Takes read and write lists of sockets
    - Or file descriptors; map to sockets via dict
- Loop over sockets until timeout expires

Implementing `select`

- Takes read and write lists of sockets
  - Or file descriptors; map to sockets via dict
- Loop over sockets until timeout expires
- Call `_read_ready` or `_write_ready`

Implementing `select`

- Takes read and write lists of sockets
  - Or file descriptors; map to sockets via dict
- Loop over sockets until timeout expires
- Call `_read_ready` or `_write_ready`
- `_read_ready` checks queue, calls `Receive` or `Accept`

Implementing `select`

- Takes read and write lists of sockets
  - Or file descriptors; map to sockets via dict
- Loop over sockets until timeout expires
- Call `_read_ready` or `_write_ready`
- `_read_ready` checks queue, calls `Receive` or `Accept`
  - If already running from previous call, call `Poll`

Implementing `select`

- Takes read and write lists of sockets
  - Or file descriptors; map to sockets via dict
- Loop over sockets until timeout expires
- Call `_read_ready` or `_write_ready`
- `_read_ready` checks queue, calls `Receive` or `Accept`
  - If already running from previous call, call `Poll`
  - Handle connection closure to provide EOF from `recv`

Implementing `select`

- Takes read and write lists of sockets
  - Or file descriptors; map to sockets via dict
- Loop over sockets until timeout expires
- Call `_read_ready` or `_write_ready`
- `_read_ready` checks queue, calls `Receive` or `Accept`
  - If already running from previous call, call `Poll`
  - Handle connection closure to provide EOF from `recv`
  - Queues data or error when callback called

Implementing `select`

- Takes read and write lists of sockets
  - Or file descriptors; map to sockets via dict
- Loop over sockets until timeout expires
- Call `_read_ready` or `_write_ready`
- `_read_ready` checks queue, calls `Receive` or `Accept`
  - If already running from previous call, call `Poll`
  - Handle connection closure to provide EOF from `recv`
  - Queues data or error when callback called
- `_write_ready` returns true if connected

Implementing select

- Takes read and write lists of sockets
  - Or file descriptors; map to sockets via dict
- Loop over sockets until timeout expires
- Call _read_ready or _write_ready
- _read_ready checks queue, calls Receive or Accept
  - If already running from previous call, call Poll
  - Handle connection closure to provide EOF from recv
  - Queues data or error when callback called
- _write_ready returns true if connected
  - Always calls Poll if connected

class socket

- __init__: Create EFI_TCP4_PROTOCOL from binding protocol

class socket

- `__init__`: Create `EFI_TCP4_PROTOCOL` from binding protocol
- `__del__`/`close`: Cancel all outstanding callbacks

class socket

- __init__: Create EFI_TCP4_PROTOCOL from binding protocol
- __del__/close: Cancel all outstanding callbacks
- connect: Call Configure and Connect

class socket

- __init__: Create EFI_TCP4_PROTOCOL from binding protocol
- __del__/close: Cancel all outstanding callbacks
- connect: Call Configure and Connect
- recv: Return data from queue

class socket

- `__init__`: Create `EFI_TCP4_PROTOCOL` from binding protocol
- `__del__`/`close`: Cancel all outstanding callbacks
- `connect`: Call `Configure` and `Connect`
- `recv`: Return data from queue
  - Spin on `_read_ready` if queue empty; this starts a `Receive`

class socket

- `__init__`: Create `EFI_TCP4_PROTOCOL` from binding protocol
- `__del__`/`close`: Cancel all outstanding callbacks
- `connect`: Call `Configure` and `Connect`
- `recv`: Return data from queue
  - Spin on `_read_ready` if queue empty; this starts a `Receive`
- `sendall`: Call `Transmit`; save status for

class socket

- __init__: Create EFI_TCP4_PROTOCOL from binding protocol
- __del__/close: Cancel all outstanding callbacks
- connect: Call Configure and Connect
- recv: Return data from queue
  - Spin on _read_ready if queue empty; this starts a Receive
- sendall: Call Transmit; save status for
- bind: Save provided address and port for later calls

class socket

- __init__: Create EFI_TCP4_PROTOCOL from binding protocol
- __del__/close: Cancel all outstanding callbacks
- connect: Call Configure and Connect
- recv: Return data from queue
  - Spin on _read_ready if queue empty; this starts a Receive
- sendall: Call Transmit; save status for
- bind: Save provided address and port for later calls
- listen: Call Configure for listening socket

class socket

- __init__: Create EFI_TCP4_PROTOCOL from binding protocol
- __del__/close: Cancel all outstanding callbacks
- connect: Call Configure and Connect
- recv: Return data from queue
  - Spin on _read_ready if queue empty; this starts a Receive
- sendall: Call Transmit; save status for
- bind: Save provided address and port for later calls
- listen: Call Configure for listening socket
- accept: Return queued connection if any

class socket

- `__init__`: Create `EFI_TCP4_PROTOCOL` from binding protocol
- `__del__`/`close`: Cancel all outstanding callbacks
- `connect`: Call `Configure` and `Connect`
- `recv`: Return data from queue
  - Spin on `_read_ready` if queue empty; this starts a `Receive`
- `sendall`: Call `Transmit`; save status for
- `bind`: Save provided address and port for later calls
- `listen`: Call `Configure` for listening socket
- `accept`: Return queued connection if any
  - Spin on `_read_ready` if queue empty; this starts an `Accept`

Socket demo and walkthrough

High-level client demo

High-level server demo

Try it out yourself

- BITS: `https://biosbits.org/`

Try it out yourself

- BITS: `https://biosbits.org/`
- QEMU/KVM

Try it out yourself

- BITS: `https://biosbits.org/`
- QEMU/KVM
    - Client works automatically

Try it out yourself

- BITS: `https://biosbits.org/`
- QEMU/KVM
  - Client works automatically
  - For server, use hostfwd option to forward ports inside

Try it out yourself

- BITS: `https://biosbits.org/`
- QEMU/KVM
  - Client works automatically
  - For server, use hostfwd option to forward ports inside
- OVMF: Open Virtual Machine Firmware, EFI for QEMU

Try it out yourself

- BITS: `https://biosbits.org/`
- QEMU/KVM
  - Client works automatically
  - For server, use hostfwd option to forward ports inside
- OVMF: Open Virtual Machine Firmware, EFI for QEMU
- Or try it on physical hardware

Try it out yourself

- BITS: `https://biosbits.org/`
- QEMU/KVM
  - Client works automatically
  - For server, use hostfwd option to forward ports inside
- OVMF: Open Virtual Machine Firmware, EFI for QEMU
- Or try it on physical hardware
  - Check BIOS settings to enable EFI network stack

Try it out yourself

- BITS: `https://biosbits.org/`
- QEMU/KVM
  - Client works automatically
  - For server, use hostfwd option to forward ports inside
- OVMF: Open Virtual Machine Firmware, EFI for QEMU
- Or try it on physical hardware
  - Check BIOS settings to enable EFI network stack
  - Use wired Ethernet

BIOS Implementation Test Suite (BITS)
`http://biosbits.org/`

Questions?

Networking without an OS
Demo Backup

Josh Triplett
josh@joshtriplett.org

PyCon 2016

```
$ (cd test ; python -m SimpleHTTPServer 8080)
Serving HTTP on 0.0.0.0 port 8080 ...
```

```
>>> import sys
>>> print sys.platform
BITS-EFI
>>> _
```

```
>>> import sys
>>> print sys.platform
BITS-EFI
>>> import urllib2
>>> print urllib2.urlopen("http://10.0.2.2:8080/hello").read()
IP configuration started
IP configuration complete: 10.0.2.15/255.255.255.0
Hello world!

>>> _
```

```
>>> import sys
>>> print sys.platform
BITS-EFI
>>> import urllib2
>>> print urllib2.urlopen("http://10.0.2.2:8080/hello").read()
IP configuration started
IP configuration complete: 10.0.2.15/255.255.255.0
Hello world!

>>> print urllib2.urlopen("http://10.0.2.2:8080/hello").read()
Hello world!

>>> _
```

```
>>> import efi
>>> out = efi.system_table.ConOut.contents
>>> out.OutputString(out, "Hello world\r\n")
Hello world
0L
>>> _
```

```
0L
>>> print efi.system_table
EFI_SYSTEM_TABLE(
  ofs=0 Hdr=TableHeader(
    ofs=0 Signature=0x5453595320494249
    ofs=8 Revision=0x20032
    ofs=12 HeaderSize=0x78
    ofs=16 CRC32=0xc57c8b39
    ofs=20 Reserved=0x0)
  ofs=24 FirmwareVendor=EDK II
  ofs=32 FirmwareRevision=0x10000
  ofs=40 ConsoleInHandle=0x7f637f18
  ofs=48 ConIn=<efi.LP_EFI_SIMPLE_TEXT_INPUT_PROTOCOL object at 0x79fa0a70>
  ofs=56 ConsoleOutHandle=0x7f636bd8
  ofs=64 ConOut=<efi.LP_EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL object at
  0x79fa0a70>
  ofs=72 StandardErrorHandle=0x7f636c58
  ofs=80 StdErr=<efi.LP_EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL object at
  0x79fa0a70>
  ofs=88 RuntimeServices=<efi.LP_EFI_RUNTIME_SERVICES object at 0x79fa0a70>
  ofs=96 BootServices=<efi.LP_EFI_BOOT_SERVICES object at 0x79fa0a70>
  ofs=104 NumberOfTableEntries=0x9
  ofs=112 ConfigurationTablePtr=<efi.LP_ConfigurationTable object at
  0x79fa0a70>)
>>> _
```

```
>>> print out
EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL(
  ofs=0 Reset=<CFunctionType object at 0x7a1f51f0>
  ofs=8 OutputString=<CFunctionType object at 0x7a1f51f0>
  ofs=16 TestString=<CFunctionType object at 0x7a1f51f0>
  ofs=24 QueryMode=<CFunctionType object at 0x7a1f51f0>
  ofs=32 SetMode=<CFunctionType object at 0x7a1f51f0>
  ofs=40 SetAttribute=<CFunctionType object at 0x7a1f51f0>
  ofs=48 ClearScreen=<CFunctionType object at 0x7a1f51f0>
  ofs=56 SetCursorPosition=<CFunctionType object at 0x7a1f51f0>
  ofs=64 EnableCursor=<CFunctionType object at 0x7a1f51f0>
  ofs=72 Mode=<efi.LP_SIMPLE_TEXT_OUTPUT_MODE object at 0x79fa0a70>)
>>> _
```

```
>>> import socket, select
>>> s = socket.socket()
>>> s.connect(("10.0.2.2", 8080))
>>> s.sendall("GET /hello\r\n\r\n")
>>> _
```

```
>>> import socket, select
>>> s = socket.socket()
>>> s.connect(("10.0.2.2", 8080))
>>> s.sendall("GET /hello\r\n\r\n")
>>> print s.recv(4096)
Hello world!

>>> _
```

```
>>> s = socket.socket()
>>> s.connect(("10.0.2.2", 8080))
>>> s.sendall("GET /hello\r\n\r\n")
>>> _
```

```
>>> s = socket.socket()
>>> s.connect(("10.0.2.2", 8080))
>>> s.sendall("GET /hello\r\n\r\n")
>>> print s._sock._recv_queue
[]
>>> _
```

```
>>> s = socket.socket()
>>> s.connect(("10.0.2.2", 8080))
>>> s.sendall("GET /hello\r\n\r\n")
>>> print s._sock._recv_queue
[]
>>> print s.recv(1)
H
>>> _
```

```
>>> s = socket.socket()
>>> s.connect(("10.0.2.2", 8080))
>>> s.sendall("GET /hello\r\n\r\n")
>>> print s._sock._recv_queue
[]
>>> print s.recv(1)
H
>>> print s._sock._recv_queue
[<ctypes.c_char_Array_12 object at 0x79fa3170>]
>>> _
```

```
>>> s = socket.socket()
>>> s.connect(("10.0.2.2", 8080))
>>> s.sendall("GET /hello\r\n\r\n")
>>> print s._sock._recv_queue
[]
>>> print s.recv(1)
H
>>> print s._sock._recv_queue
[<ctypes.c_char_Array_12 object at 0x79fa3170>]
>>> print s._sock._recv_queue[0].raw
ello world!

>>> _
```

```
>>> s = socket.socket()
>>> s.connect(("10.0.2.2", 8080))
>>> s.sendall("GET /hello\r\n\r\n")
>>> print s._sock._recv_queue
[]
>>> print s.recv(1)
H
>>> print s._sock._recv_queue
[<ctypes.c_char_Array_12 object at 0x79fa3170>]
>>> print s._sock._recv_queue[0].raw
ello world!

>>> print s.recv(1)
e
>>> print s._sock._recv_queue
[<ctypes.c_char_Array_11 object at 0x79fa0b90>]
>>> _
```

```
>>> s = socket.socket()
>>> s.connect(("10.0.2.2", 8080))
>>> s.sendall("GET /hello\r\n\r\n")
>>> print s._sock._recv_queue
[]
>>> print s.recv(1)
H
>>> print s._sock._recv_queue
[<ctypes.c_char_Array_12 object at 0x79fa3170>]
>>> print s._sock._recv_queue[0].raw
ello world!

>>> print s.recv(1)
e
>>> print s._sock._recv_queue
[<ctypes.c_char_Array_11 object at 0x79fa0b90>]
>>> print s.recv(4096)
llo world!

>>> _
```

```
>>> s = socket.socket()
>>> s.bind(("", 1234))
>>> s.listen(10)
>>> _
```

```
>>> s = socket.socket()
>>> s.bind(("", 1234))
>>> s.listen(10)
>>> print s._sock._accept_queue
[]
>>> _
```

```
>>> s = socket.socket()
>>> s.bind(("", 1234))
>>> s.listen(10)
>>> print s._sock._accept_queue
[]
>>> select.select([s],[],[])

_
```

```
$ telnet localhost 1234
Trying ::1...
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
```

```
>>> s = socket.socket()
>>> s.bind(("", 1234))
>>> s.listen(10)
>>> print s._sock._accept_queue
[]
>>> select.select([s],[],[])
([<socket._socketobject object at 0x7a1d7e50>], [], [])
>>> _
```

```
>>> s = socket.socket()
>>> s.bind(("", 1234))
>>> s.listen(10)
>>> print s._sock._accept_queue
[]
>>> select.select([s],[],[])
([<socket._socketobject object at 0x7a1d7e50>], [], [])
>>> print s._sock._accept_queue
[(True, 2134465624)]
>>> s2, addr = s.accept()
>>> print addr
('10.0.2.2', 57348)
>>> _
```

```
>>> s = socket.socket()
>>> s.bind(("", 1234))
>>> s.listen(10)
>>> print s._sock._accept_queue
[]
>>> select.select([s],[],[])
([<socket._socketobject object at 0x7a1d7e50>], [], [])
>>> print s._sock._accept_queue
[(True, 2134465624)]
>>> s2, addr = s.accept()
>>> print addr
('10.0.2.2', 57348)
>>> print s._sock._accept_queue
[]
>>> _
```

```
>>> s = socket.socket()
>>> s.bind(("", 1234))
>>> s.listen(10)
>>> print s._sock._accept_queue
[]
>>> select.select([s],[],[])
([<socket._socketobject object at 0x7a1d7e50>], [], [])
>>> print s._sock._accept_queue
[(True, 2134465624)]
>>> s2, addr = s.accept()
>>> print addr
('10.0.2.2', 57348)
>>> print s._sock._accept_queue
[]
>>> data = s2.recv(4096)
_
```

```
$ telnet localhost 1234
Trying ::1...
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
Hello from the outside world!
```

```
>>> s = socket.socket()
>>> s.bind(("", 1234))
>>> s.listen(10)
>>> print s._sock._accept_queue
[]
>>> select.select([s],[],[])
([<socket._socketobject object at 0x7a1d7e50>], [], [])
>>> print s._sock._accept_queue
[(True, 2134465624)]
>>> s2, addr = s.accept()
>>> print addr
('10.0.2.2', 57348)
>>> print s._sock._accept_queue
[]
>>> data = s2.recv(4096)
>>> s2.sendall(data.upper())
>>> _
```

```
$ telnet localhost 1234
Trying ::1...
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
Hello from the outside world!
HELLO FROM THE OUTSIDE WORLD!
```

```
>>> print urllib2.urlopen("http://10.0.2.2:8080/hello").read()
Hello world!

>>> _
```

```
>>> print urllib2.urlopen("http://10.0.2.2:8080/hello").read()
Hello world!

>>> post = urllib2.urlopen("http://10.0.2.2:8080/post").read()
>>> _
```

```
>>> print urllib2.urlopen("http://10.0.2.2:8080/hello").read()
Hello world!

>>> post = urllib2.urlopen("http://10.0.2.2:8080/post").read()
>>> print post
#!/usr/bin/python
import BaseHTTPServer

class RequestHandler(BaseHTTPServer.BaseHTTPRequestHandler):
    def do_POST(self):
        l = int(self.headers.get("Content-Length", "-1"))
        data = self.rfile.read(l)
        self.send_response(200)
        print "POST to {}:\n{}".format(self.path, data)

BaseHTTPServer.test(RequestHandler)

>>> _
```

```
$ (cd test ; ./post 8080)
Serving HTTP on 0.0.0.0 port 8080 ...
```

```
>>> print urllib2.urlopen("http://10.0.2.2:8080/hello").read()
Hello world!

>>> post = urllib2.urlopen("http://10.0.2.2:8080/post").read()
>>> print post
#!/usr/bin/python
import BaseHTTPServer

class RequestHandler(BaseHTTPServer.BaseHTTPRequestHandler):
    def do_POST(self):
        l = int(self.headers.get("Content-Length", "-1"))
        data = self.rfile.read(l)
        self.send_response(200)
        print "POST to {}:\n{}".format(self.path, data)

BaseHTTPServer.test(RequestHandler)

>>> print urllib2.urlopen("http://10.0.2.2:8080/post", "Long log data here").re
ad()
>>> _
```

```
$ (cd test ; ./post 8080)
Serving HTTP on 0.0.0.0 port 8080 ...
127.0.0.1 - - [30/May/2016 00:19:52] "POST /post HTTP/1.1" 20
0 -
POST to /post:
Long log data here
```

```
class RequestHandler(BaseHTTPServer.BaseHTTPRequestHandler):
    def do_POST(self):
        l = int(self.headers.get("Content-Length", "-1"))
        data = self.rfile.read(l)
        self.send_response(200)
        print "POST to {}:\n{}".format(self.path, data)

BaseHTTPServer.test(RequestHandler)

>>> print urllib2.urlopen("http://10.0.2.2:8080/post", "Long log data here").re
ad()
>>> import bits
>>> data = "\n".join(["{} {}".format(i, bits.cpuid(0,i)) for i in range(10)])
>>> print data
0 cpuid_result(eax=0x0000000d, ebx=0x756e6547, ecx=0x6c65746e, edx=0x49656e69)
1 cpuid_result(eax=0x00000663, ebx=0x00000800, ecx=0x80202001, edx=0x078bfbfd)
2 cpuid_result(eax=0x00000001, ebx=0x00000000, ecx=0x00000000, edx=0x002c307d)
3 cpuid_result(eax=0x00000000, ebx=0x00000000, ecx=0x00000000, edx=0x00000000)
4 cpuid_result(eax=0x00000121, ebx=0x01c0003f, ecx=0x0000003f, edx=0x00000001)
5 cpuid_result(eax=0x00000000, ebx=0x00000000, ecx=0x00000003, edx=0x00000000)
6 cpuid_result(eax=0x00000000, ebx=0x00000000, ecx=0x00000000, edx=0x00000000)
7 cpuid_result(eax=0x00000000, ebx=0x00000000, ecx=0x00000000, edx=0x00000000)
8 cpuid_result(eax=0x00000000, ebx=0x00000000, ecx=0x00000000, edx=0x00000000)
9 cpuid_result(eax=0x00000000, ebx=0x00000000, ecx=0x00000000, edx=0x00000000)
>>> _
```

```
        l = int(self.headers.get("Content-Length", "-1"))
        data = self.rfile.read(l)
        self.send_response(200)
        print "POST to {}:\n{}".format(self.path, data)

BaseHTTPServer.test(RequestHandler)

>>> print urllib2.urlopen("http://10.0.2.2:8080/post", "Long log data here").re
ad()
>>> import bits
>>> data = "\n".join(["{} {}".format(i, bits.cpuid(0,i)) for i in range(10)])
>>> print data
0 cpuid_result(eax=0x0000000d, ebx=0x756e6547, ecx=0x6c65746e, edx=0x49656e69)
1 cpuid_result(eax=0x00000663, ebx=0x00000800, ecx=0x80202001, edx=0x078bfbfd)
2 cpuid_result(eax=0x00000001, ebx=0x00000000, ecx=0x00000000, edx=0x002c307d)
3 cpuid_result(eax=0x00000000, ebx=0x00000000, ecx=0x00000000, edx=0x00000000)
4 cpuid_result(eax=0x00000121, ebx=0x01c0003f, ecx=0x0000003f, edx=0x00000001)
5 cpuid_result(eax=0x00000000, ebx=0x00000000, ecx=0x00000003, edx=0x00000000)
6 cpuid_result(eax=0x00000000, ebx=0x00000000, ecx=0x00000000, edx=0x00000000)
7 cpuid_result(eax=0x00000000, ebx=0x00000000, ecx=0x00000000, edx=0x00000000)
8 cpuid_result(eax=0x00000000, ebx=0x00000000, ecx=0x00000000, edx=0x00000000)
9 cpuid_result(eax=0x00000000, ebx=0x00000000, ecx=0x00000000, edx=0x00000000)
>>> print urllib2.urlopen("http://10.0.2.2:8080/post", data).read()

>>> _
```

```
POST to /post:
0 cpuid_result(eax=0x0000000d, ebx=0x756e6547, ecx=0x6c65746e
, edx=0x49656e69)
1 cpuid_result(eax=0x00000663, ebx=0x00000800, ecx=0x80202001
, edx=0x078bfbfd)
2 cpuid_result(eax=0x00000001, ebx=0x00000000, ecx=0x00000000
, edx=0x002c307d)
3 cpuid_result(eax=0x00000000, ebx=0x00000000, ecx=0x00000000
, edx=0x00000000)
4 cpuid_result(eax=0x00000121, ebx=0x01c0003f, ecx=0x0000003f
, edx=0x00000001)
5 cpuid_result(eax=0x00000000, ebx=0x00000000, ecx=0x00000003
, edx=0x00000000)
6 cpuid_result(eax=0x00000000, ebx=0x00000000, ecx=0x00000000
, edx=0x00000000)
7 cpuid_result(eax=0x00000000, ebx=0x00000000, ecx=0x00000000
, edx=0x00000000)
8 cpuid_result(eax=0x00000000, ebx=0x00000000, ecx=0x00000000
, edx=0x00000000)
9 cpuid_result(eax=0x00000000, ebx=0x00000000, ecx=0x00000000
, edx=0x00000000)
```

```
>>> print post
#!/usr/bin/python
import BaseHTTPServer

class RequestHandler(BaseHTTPServer.BaseHTTPRequestHandler):
    def do_POST(self):
        l = int(self.headers.get("Content-Length", "-1"))
        data = self.rfile.read(l)
        self.send_response(200)
        print "POST to {}:\n{}".format(self.path, data)

BaseHTTPServer.test(RequestHandler)

>>> exec post
Serving HTTP on 10.0.2.15 port 8000 ...
_
```

```
$ curl http://localhost:8000/path -d 'Test data'
$
```

```
>>> print post
#!/usr/bin/python
import BaseHTTPServer

class RequestHandler(BaseHTTPServer.BaseHTTPRequestHandler):
    def do_POST(self):
        l = int(self.headers.get("Content-Length", "-1"))
        data = self.rfile.read(l)
        self.send_response(200)
        print "POST to {}:\n{}".format(self.path, data)

BaseHTTPServer.test(RequestHandler)

>>> exec post
Serving HTTP on 10.0.2.15 port 8000 ...
10.0.2.2 - - [25/May/1970 02:43:33] "POST /path HTTP/1.1" 200 -
POST to /path:
Test data

_
```