csdesign Documentation

Release 0.1

Ruslan Spivak

Sep 27, 2017

Contents

Contents:

CHAPTER 1

Simple Examples of Concurrent Server Design in Python

The client.py can be used with all the servers.

1. TCP Concurrent Server, One Child per Client (fork)

   server01.py

2. TCP Concurrent Server, I/O Multiplexing (select)

   server02.py

3. TCP Preforked Server, Children Call 'accept'

   Calling *'fork'* by the server for every client connection can be costly in terms of resources. One possible solution is to prefork some number of children when the server starts and use the pool of preforked processes to handle incoming connections.

   server03.py

   This design is a little bit unusual in a sense that it's not the server that calls *'accept'* to accept a new connection, but all children are blocked in the call to *'accept'* on the **same listening socket** passed from the parent process.

   The way it works is that all child processes are blocked waiting for an event on the same listening socket. When a new connection arrives **all** children are awakened. The first child process to run will make a call to *'accept'* and the rest of the processes will be put to sleep on the same call. Rinse repeat. The behavior is called Thundering herd problem

   To see the distribution of connections to the children (i.e. how many times every child succeeded in calling *'accept'* and getting a new connection socket) run the following server in the foreground and press Ctrl-C when the client is done.

   server03a.py

   On Linux the connection distribution is uniform. With the following client parameters on my Fedora box

   ```
   $ python client.py -i localhost -p 2000 -c 15 -t 100 -b 4096
   ```

   this is how the distribution looks like:

```
child 0 : 127 times
child 1 : 144 times
child 2 : 138 times
child 3 : 147 times
child 4 : 160 times
child 5 : 161 times
child 6 : 161 times
child 7 : 130 times
child 8 : 181 times
child 9 : 133 times
```

4. TCP Preforked Server, Passing Descriptor to Child

   To avoid the Thundering herd problem described in the "TCP Preforked Server, Children Call 'accept"' section we can make our parent process to handle *'accept'* and pass a connected socket to one of its preforked children for further handling.

   This example requires **Python 3.3.x** to run. Tested on Linux with Python 3.3.a4.

   server04.py

CHAPTER 2

Miscellanea

RST Packet Generation

The iterative server rstserver.py binds to *localhost*, port *2000* and serves incoming requests. After accepting a new connection it immediately sends an RST packet over that connection. To generate an RST packet it uses SO_LINGER socket option.

rstserver.py

The Nature of SIGPIPE

At some point when writing to a socket you might get an exception *"socket.error: [Errno 32] Broken pipe"*.

The rule is that when a process tries to write to a socket that has already received an RST packet, the SIGPIPE signal is sent to that process which causes the exception.

The code in *sigpipe.py* shows how to simulate *SIGPIPE*.

sigpipe.py

First you need to start rstserver.py

Then run *sigpipe.py* which in turn connects to the *rstserver*, gets an RST as a response, ignores it and tries to write to the socket:

```
$ python sigpipe.py
[Errno 104] Connection reset by peer

Traceback (most recent call last):
  File "sigpipe.py", line 43, in <module>
    s.send('hello')
socket.error: [Errno 32] Broken pipe
```

SELF-PIPE Trick

If a process needs to monitor several descriptors for I/O and wait for the delivery of a signal then a race condition can happen.

Consider the following code excerpt:

```
GOT_SIGNAL = False

def handler(signum, frame):
    global GOT_SIGNAL
    GOT_SIGNAL = True

...

signal.signal(signal.SIGUSR1, handler)

# What if the signal arrives at this point?

try:
    readables, writables, exceptions = select.select(rlist, wlist, elist)
except select.error as e:
    code, msg = e.args
    if code == errno.EINTR:
        if GOT_SIGNAL:
            print 'Got signal'
    else:
        raise
```

The problem here is that if the *SIGUSR1* is delivered after setting the signal handler but before call to *select* then the *select* call **will block** and we won't execute our application logic in response to the event thus effectively *"missing"* the signal (our application logic in this case is printing the message: *Got signal*).

That's an example of possible nasty racing. Let's simulate that with selsigrace.py

Start the program

```
$ python selsigrace.py
PID: 32324
Sleep for 10 secs
```

and send the *USR1* signal to the PID(it's different on every run) within the 10 second interval while the process is still sleeping:

```
$ kill -USR1 32324
```

You should see the program produce additional line of output 'Wake up and block in "select"' and **block without exiting**, no message "Got signal":

```
$ python selsigrace.py
PID: 32324
Sleep for 10 secs
Wake up and block in "select"
```

If you send yet another *USR1* signal at this point then the *select* will be interrupted and the program will terminate with a message:

```
$ kill -USR1 32324
```

```
$ python selsigrace.py
PID: 32324
Sleep for 10 secs
Wake up and block in "select"
Got signal
```

Self-Pipe Trick is used to avoid race conditions when waiting for signals and calling *select* on a set of descriptors.

The following steps describe how to implement it:

1. Create a pipe and change its read and write ends to be nonblocking
2. Add the read end of the pipe to the read list of descriptors given to *select*
3. Install a signal handler for the signal we're concerned with. When the signal arrives the signal handler writes a byte of data to the pipe.

   Because the write end of the pipe is nonblocking we prevent the situation when signals flood the process, the pipe becomes full and the process blocks itself in the signal handler.
4. When *select* successfully returns check if the read end of the pipe is in the *readables* list and if it is then our signal has arrived.
5. When the signal arrives read **all** bytes that are in the pipe and execute any actions that have to be done in response to the signal delivery.

Start the selfpipe.py and send a *USR1* signal to it. You should see it output a message *Got signal* and exit.

Alternatives: **pselect** (not in Python standard library), **signalfd** (Linux only, not in Python standard library)

CHAPTER 3

Need for Speed - SENDFILE System Call

A very common operation of Web servers these days is transferring files to clients. They do that by reading the files' contents from a disk and writing it back to the clients' sockets.

The copying of the file on Linux/UNIX, for example, could be done in a loop using *read/write* system calls:

```
import os
...
while True:
    data = os.read(filefd, 4096)
    if not data:
        break
    os.write(socketfd, data)
```

On the surface this looks perfectly fine, but for transferring large files, when pre-processing of the file contents isn't necessary, this is pretty inefficient.

The reason is that *read* and *write* system calls involve copying data from kernel space to user space and vice versa and all that happens in a loop:

That's where sendfile system call comes in handy. It provides a nice optimization for this particular use case by doing all the copying from the file descriptor to the socket descriptor completely in the kernel space:

Python 3.3 provides os.sendfile as part of the standard library.

CHAPTER 4

Roadmap

- TCP Concurrent Server, One Thread per Client
- TCP Concurrent Server, I/O Multiplexing (poll)
- TCP Concurrent Server, I/O Multiplexing (epoll)
- TCP Prethreaded Server
- Miscellanea, TCP_CORK socket option
- Documentation for every example

CHAPTER 5

Acknowledgments

- The book "Unix Network Programming, Volume 1: The Sockets Networking API (3rd Edition)" by W. Richard Stevens, Bill Fenner, Andrew M. Rudoff

  It's the best book on the subject. I took and use many techniques from that book.

- "The Linux Programming Interface: A Linux and UNIX System Programming Handbook" by Michael Kerrisk

  Also the best book on the subject.

CHAPTER 6

Indices and tables

- genindex
- modindex
- search