

Event-Driven Architecture

Daniel Zappala

CS 360 Internet Programming
Brigham Young University

epoll()

Event Driven Server Architecture

- one thread handles all events
- use `epoll()` to multiplex clients
 - provide a list of sockets waiting for I/O events
 - sleeps until an event occurs on one or more sockets
 - can provide a timeout to limit waiting time
- some evidence that event-driven architectures can be more efficient than process or thread architectures
- do not use `select()` or `poll()` – these are older, less efficient

epoll

```
1 poller = select.epoll()
```

- returns a polling object: supports registering and unregistering file descriptors and then polling them for I/O events

epoll register

```
1 poller.register(fd,mask)
2 poller.unregister(fd)
```

- fd is a socket file descriptor
- mask is
 - EPOLLIN: read event
 - EPOLLOUT: write event
 - others (errors, etc.) that are less frequently used
- can OR masks together
- can unregister when socket is closed

polling

```
1      fds = poller.poll(timeout)
```

- returns a list of file descriptors that have had an event occur
- timeout is a floating point value in seconds

Example Code

► Echo Client and Server – Polling Version

Level-Triggered vs Edge-Triggered

- level-triggered interrupts occur whenever the file descriptor is ready for I/O
 - 1000 bytes of data in receive buffer
 - you call `recv()` and extract 500 bytes
 - `poll()` will continue to indicate the fd is ready because there are still 500 bytes in the buffer
- edge-triggered interrupts occur whenever the file descriptor goes from being *not* ready to ready
 - 1000 bytes of data in receive buffer
 - you call `recv()` get extract 500 bytes
 - `poll()` will *not* indicate the fd is ready until the receive buffer goes down to zero and then back up to some positive number
- default in Python is Level Trigger
- use `EPOLLET` mask in register to set Edge Trigger behavior

Coding Practice

Coding Practice

- no shared memory synchronization needed
- must be careful how I/O events are handled
 - Level Triggered – call `recv()` once per socket in the event processing loop
 - Edge Triggered – call `recv()` repeatedly until it returns `EWOULDBLOCK`
 - prefer Level Triggered because one client can't monopolize the server
- must keep a separate `recv()` cache for each socket, since all sockets are handled by a single thread

Timing Out Idle Sockets: The Wrong Way

- set timeout in `poll()`
- if `poll()` returns with a timeout, then any socket still open is closed
- *one idle socket among many active ones will stay open indefinitely*

Timing Out Idle Sockets: Mark and Sweep

- mark
 - keep a variable for each socket that tracks the last time it had an I/O event
- sweep
 - once every t seconds, loop through all sockets and use current time to check if each socket has been idle too long
- sweeping once per t seconds
 - get current time before calling `poll()`
 - get current time after calling `poll()`
 - subtract and see if enough time has passed to do a sweep
 - max time that can pass for each call to `poll()` is given in timeout parameter: you decide the granularity

Building a Web Server

Handling Requests

- use non-blocking I/O
- call `recv()`
 - if returns `EAGAIN` or `EWOULDBLOCK`, return
 - otherwise append to cache for that socket
- check for end of a message (`\r\n\r\n`)
 - process any HTTP messages present
 - leave any remainder in the cache
- handles pipelined requests properly
- prevents a busy client from monopolizing the server

Steps in Handling an HTTP Request

- ① read and parse the HTTP request message
- ② translate the URI to a file name
 - need web server configuration to determine the document root
- ③ determine whether the request is authorized
 - check file permissions or other authorization procedure
- ④ generate and transmit the response
 - error code or file or results of script
 - must be a valid HTTP message with appropriate headers
- ⑤ log request and any errors

Handling Multiple Roots

- use the Host header to find the host name
- configuration file gives the root directory for each host served by the web server
- append the URI path to the root directory to get the complete path

Useful System Calls

Checking File Permissions

- call `open()` to determine whether you can access the file

```
1 try:
2     open(filename)
3 except IOError as (errno, strerror):
4     if errno == 13:
5         // 403 Forbidden
6     elif errno == 2:
7         // 404 Not Found
8     else:
9         // 500 Internal Server Error
```

Accessing File Attributes

- use `os.stat(filename)` to access file size and last modification time
- use in Content-Length and Last-Modified headers

```
1 size = os.stat(filename).st_size
2 mod_time = os.stat(filename).st_mtime
```

Getting the Time

```
1 t = time()
```

- returns the time since the Epoch (00:00:00 UTC, January 1, 1970), measured in seconds, as a floating point number

Converting to GMT

```
1  gmt = time.gmtime(t)
```

- takes as input the time in seconds since the epoch
- returns a structure that uses GMT

Converting to RFC 822, 1123 Time Format

- the recommended date format for HTTP
- used in the Date and Last-Modified headers

```
1 format = '%a, %d %b %Y %H:%M:%S GMT'
2 time_string= time.strftime(format, gmtime())
```

- takes a format string, GMT time struct
- returns a string using RFC 1123 format
- see <http://docs.python.org/library/time.html>

From Time to Time

```
1 import time
2 import os
3 filename = '/etc/motd'
4
5 def get_time(t):
6     gmt = time.gmtime(t)
7     format = '%a, %d %b %Y %H:%M:%S GMT'
8     time_string = time.strftime(format, gmt)
9     return time_string
10
11 t = time.time()
12 current_time = get_time(t)
13 mt = os.stat(filename).st_mtime
14 mod_time = get_time(mt)
15 print current_time
16 print mod_time
```
