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

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

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

epoll register

```
poller.register(fd,mask)
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

```
        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
  - `epoll_wait()` 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
  - `epoll_wait()` 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
  - with blocking `recv()` – call only once per socket in the event processing loop
    - only use with Level Triggered
  - with non-blocking `recv()` – call as much as needed to handle socket events until it returns EAGAIN or EWOULDBLOCK
    - use with Level Triggered or Edge Triggered
- must keep a separate `recv()` cache for each socket, since all sockets are handled by a single thread

Timing Out Idle Sockets

- easy but not accurate
  - set timeout in `epoll_wait()`
  - if `epoll_wait()` 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
  - keep a variable for each socket that tracks the last time it had an I/O event
  - once every t seconds, loop through all sockets and use current time to check if each socket has been idle too long
- timer
  - get current time before calling `epoll_wait()`
  - get current time after calling `epoll_wait()`
  - subtract and see if enough time has passed
  - max time that can pass for each call to `epoll_wait()` is given in timeout parameter

Building a Web Server

Handling Requests

- use non-blocking I/O
- while loop
  - call recv()
  - if returns EAGAIN or EWOULDBLOCK, break from loop
  - 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
  - if messages processed, break from loop
- 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

```
try:
  open(filename)
except IOError as (errno,strerror):
  if errno == 13:
    // 403 Forbidden
  elif errno == 2:
    // 404 Not Found
  else:
    // 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

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

Getting the Time

```
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

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

- takes as input the tieme 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

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

- 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

```
import time
import os
filename = ’/etc/motd’

def get_time(t):
    gmt = time.gmtime(t)
    format = ’%a, %d %b %Y %H:%M:%S GMT’
    time_string = time.strftime(format,gmt)
    return time_string

t = time.time()
current_time = get_time(t)
mt = os.stat(filename).st_mtime
mod_time = get_time(mt)
print current_time
print mod_time
```