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
  - `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