Event-Driven Server Architecture

CS 360 Internet Programming

Daniel Zappala

Brigham Young University
Computer Science Department

Event Driven Server Architecture

- one thread handles all events
- must multiplex handling of many clients and their messages
  - use `select()`, `poll()` or `epoll()` to multiplex socket I/O events
  - 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
- `epoll()` is far more efficient than the others
- some evidence that event-driven architectures can be more efficient than process or thread architectures

epoll()

```
1 #include <sys/epoll.h>
2
3 int epoll_create(int size);
4 int epoll_wait(int epfd, struct epoll_event *events,
5                int maxevents, int timeout);
```

- create a poll instance, which returns a poll file descriptor
- then wait for input on a set of events
- returns the number of file descriptors ready for I/O, or zero if the timeout occured, or -1 on error
- more scalable version of `poll()`, used by newer web servers
- *see example code on web site*

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

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
  - with non-blocking `recv()` – call as much as needed to handle socket events until it returns EAGAIN or EWOULDBLOCK
- 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