Tutorial 02—Implement Scalable Servers with Non-blocking I/O

ECE 459: Programming for Performance

David Xi Cheng
david.cheng@uwaterloo.ca

University of Waterloo

January 21, 2015
Updated: November 14, 2016

References

- Beej's Guide to Network Programming
- The C10K problem
- L. Gammo, T. Brecht, A. Shukla and D. Pariag. Comparing and Evaluating epoll, select and poll Event Mechanisms. In Proceedings of the Ottawa Linux Symposium, 2004
- v. Behren, J. Robert, J. Condit, and E. A. Brewer. Why Events Are a Bad Idea (for High-Concurrency Servers). In HotOS, 2003.
- M. Welsh, D. Culler, and E. Brewer. SEDA: An Architecture for Well-Conditioned, Scalable Internet Services. In SOSP, 2001.

Part I

How to Write a Server

How to Write a Server?

1. create, bind, listen

```
//create a socket file descriptor
int socket(int domain, int type, int protocol);

//bind socket to a particular address:port pair
int bind(int sockfd, struct sockaddr* addr, int addrlen);

//listen for incoming connections
int listen(int sockfd, int backlog);
```

Create, Bind, Listen Example (Init and Create)

```
int create_bind_listen(int port) {
  struct sockaddr_in server_addr;

  //init server addr struct
  memset(&server_addr, 0, sizeof(server_addr));
  server_addr.sin_family = AF_INET;
  server_addr.sin_port = htons(port);
  server_addr.sin_addr.s_addr = htonl(INADDR_ANY);

  //create
  int server_sock = socket(AF_INET, SOCK_STREAM, 0);
  if (server_sock < 0){
    /* something is wrong here */
  }

  // continued on the next page
```

Create, Bind, Listen Example (Bind and Listen)

```
  //bind
  if (bind(server_sock, (struct sockaddr *) &server_addr,
      sizeof(server_addr)) < 0) {
    /* something is wrong here */
  }

  //listen
  if (listen(server_sock, BACK_LOG_SIZE) < 0) {
    /* something is wrong here */
  }

  return server_sock;
}
```

How to Write a Server

2. accept a connection

```
//accept a connection
int accept(int socked, struct sockaddr* addr, socklen_t*
    addrlen);
```

3. communicate with a client

```
//receive messages
int recv(int sockfd, void* buf, int len, int flags);

//send messages
int send(int sockfd, const void* msg, int len, int flags);
```

Trivial Server (Synchronous)

```
int main(){
  int server_sock;
  if ((server_sock = create_bind_listen(PORT_NUMBER)) < 0) {
    /* something is wrong here */
  }

  struct sockaddr_in client_addr;
  socklen_t client_addr_len = sizeof(client_addr);
  int client_fd;

  while (true) {
    client_fd = accept(server_sock, (struct socked *) &
        client_addr, &client_addr_len);
    if (client_fd < 0) {
      /* something is wrong here */
    }

    //communications happen here

    close(client_fd);
  }
}
```

Synchronous Server

Serve one client at a time.

It doesn't work in the presence of,

- concurrent connections
  - connections would timeout if not accepted in time
- persistent connections
  - could serve only one client

Not Scalable!

Part II

How to Write a Scalable Server

How to Write a Scalable Server

What are the options

- Serve one client with each server thread
  - multi-thread/process model
- Serve many clients with each server thread
  - non-blocking multiplexing
    `select, poll, epoll(Linux), kqueue(BSD)`
    We are going to talk about it in this tutorial
  - asynchronous I/O
    `callback upon completion (Linux AIO)`
    Check it out:
    Boost application performance using asynchronous I/O

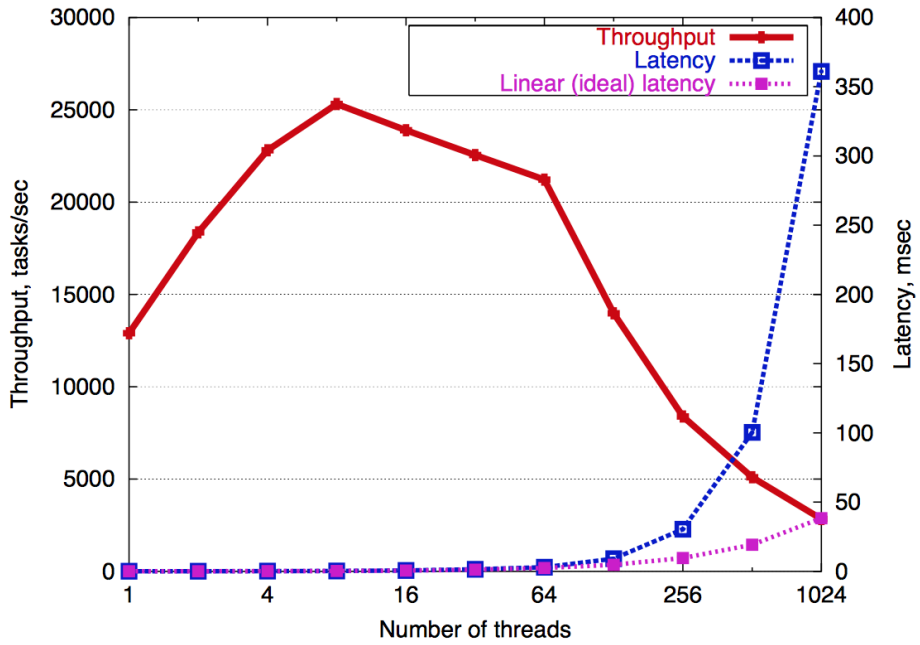
Multi-Thread/Process Model

Each server thread/process serves only one client

- simpler programming style
- handles concurrent connections
- performance degradation after too many threads/processes

  (due to memory and context-switch overhead)

  (although, people would argue to do better with user-level threading)



Threaded server performance degradation on 4wy 500MHz Pentium III with 2 GB memory[1]

[1] Figure 2, SEDA: An Architecture for Well-Conditioned Scalable Internet Services, SOSP 2001

Multi-Thread/Process Model Example

```
int main(){
  int server_sock;
  if ((server_sock = create_bind_listen(PORT_NUMBER)) < 0) {
    /* something is wrong here */
  }

  struct sockaddr_in client_addr;
  socklen_t client_addr_len = sizeof(client_addr);
  int client_fd;

  while (true) {
    client_fd = accept(server_sock, (struct socked *) &
        client_addr, &client_addr_len);
    if (client_fd < 0) {
      /* something is wrong here */
    }

    //create a separate thread/process to
    //communicate with the client
  }
}
```

Non-blocking Multiplexing

Each server thread/process serves multiple clients.

- Multiplexed with notification facilities, i.e., select, poll and epoll.

Generalized Event Loop

1. create/modify set(s) of file descriptors that require readiness notifications.
2. wait on select/poll/epoll until one or more FDs become ready.
3. select/poll/epoll returns with hints of which FDs are ready to read/write or have exceptions.
4. handle those FDs appropriately.
5. repeat from step 1.

Use non-blocking I/O to avoid blocking the event loop!

Blocking v.s. Non-blocking I/O

In previous examples,

all of `accept`,`send`,`recv` operations would block the entire thread if,

- no clients to `accept`.
- no data to `recv`.
- `send` buffer is full.

Blocking v.s. Non-blocking I/O

To avoid blocking the thread in the above conditions, we need to use non-blocking I/O instead.

In the non-blocking mode, -1 is immediately returned with `errno` set to `EWOULDBLOCK`, if the operation would block.

To set a file descriptor as non-blocking, we need to

```
//retrieve original flags of the FD
int flags = fcntl(fd, F_GETFL, 0);

//add an O_NONBLOCK flag
flags |= O_NONBLOCK

//reset the flag of the FD
fcntl(fd, F_SETFL, flags);
```

Select

```
int select(int nfds, fd_set* readfds, fd_set*
    writefds, fd_set* exceptfds, struct timeval*
    timeout);
```

- first introduced in 4.2 BSD Unix in 1983.
- takes three `fd_set` to monitor read/write/except readiness.
- blocks until one or more file descriptors become ready.
- `fd_set` is essentially a bitmap, fixed in size of `FD_SETSIZE`, 1024 by default. (i.e., support at most 1024 file descriptors)
- the `fd_set` bitmaps are overridden upon return, need to be restored before each call.

You may access `fd_set` with the following macros.

```
void FD_CLR(int fd, fd_set* set);
int  FD_ISSET(int fd, fd_set* set);
void FD_SET(int fd, fd_set* set);
void FD_ZERO(fd_set* set);
```

Select Example (init fd_set)

```
int main() {
  /* create, bind, listen first */

  int fd_max;
  fd_set sock_fds, read_fds;

  FD_ZERO(&sock_fds);
  FD_ZERO(&read_fds);

  //add server_sock to the fd set
  FD_SET(server_sock, &sock_fds);

  //server_sock is the largest fd as of now
  fd_max = server_sock;

  // continued on the next page
```

Select Example (event loop)

```
  while (true) {
    //make a copy of sock_fds
    read_fds = sock_fds;

    if (select(fd_max+1, &read_fds, NULL, NULL, NULL) == -1) {
      /* something is wrong here */
    }

    //read_fds is now overridden with hints of fd readiness.
    for (int i = 0; i <= fd_max; ++i) {
      if (FD_ISSET(i, &read_fds)) {
        //fd i is now ready
        if (i == server_sock) {  //server_sock is ready to accept
          /* accept client(s) here */
          //add client_sock to fd_set
          FD_SET(client_sock, &sock_fds);
          //reset fd_max
          if (client_sock > fd_max) {
            fd_max = client_sock;
          }
        } else {  //client_sock is ready to read
          /* communicate with the client here */
          if( /* done with the client */ ) {
            FD_CLR(i, &sock_fds);
            close(i);
          }
        }
      } //if
    } //for
  } //while
} //main
```

Poll

```
int poll(struct pollfd* fds, nfds_t nfds, int timeout);
```

- introduced to Linux since kernel version 2.1.23 in 1997
- takes an array of `struct pollfd` to monitor fd readiness

  (as opposed to bitmaps in `select`).

  ```
  struct pollfd {
      int   fd;         /* file descriptor */
      short events;     /* requested events */
      short revents;    /* returned events */
  };
  ```

  (fd number, monitored/occurred events are encapsulated as different fields)
- blocks until one or more file descriptors become ready.
- only changes `pollfd.revents` upon return.

  (do not need to recover fd set before each call)
- no upper limit for array size.

Poll Example

```
int main(){
  /* create, bind, listen first */

  //use vector for pollfd array
  std::vector<struct pollfd> sock_fds;

  //create the first pollfd for server_sock
  struct pollfd pfd;
  server_pfd.fd = server_sock;
  server_pfd.events = POLLIN;

  sock_fds.push_back(pfd);

  // continued on the next page
```

Poll Example (event loop)

```
  while (true) {
    if (poll(&sock_fds[0], sock_fds.size(), -1) < 0) {
      /* handle poll error here */
    }

    //iterate through sock_fds for readiness
    for (auto it = sock_fds.begin(); it != sock_fds.end(); ++it) {
      if (!(it->revents & POLLIN)){
        continue;
      }
      if (it->fd == server_sock) {  //server_sock is ready to accept
        /* accept a client here */
        //add client_sock to sock_fds
        pfd.fd = client_sock;
        pfd.events = POLLIN;
        sock_fds.push_back(pfd);
      } else {  //client_sock is ready to read
        /* communicate with the client here */
        if (/* done with the client */) {  //remove the client from poll queue
          close(it->fd);
          sock_fds.erase( it );
          --it;
        }
      } //else
    } //for
  } //while
} //main
```

select, poll v.s. epoll

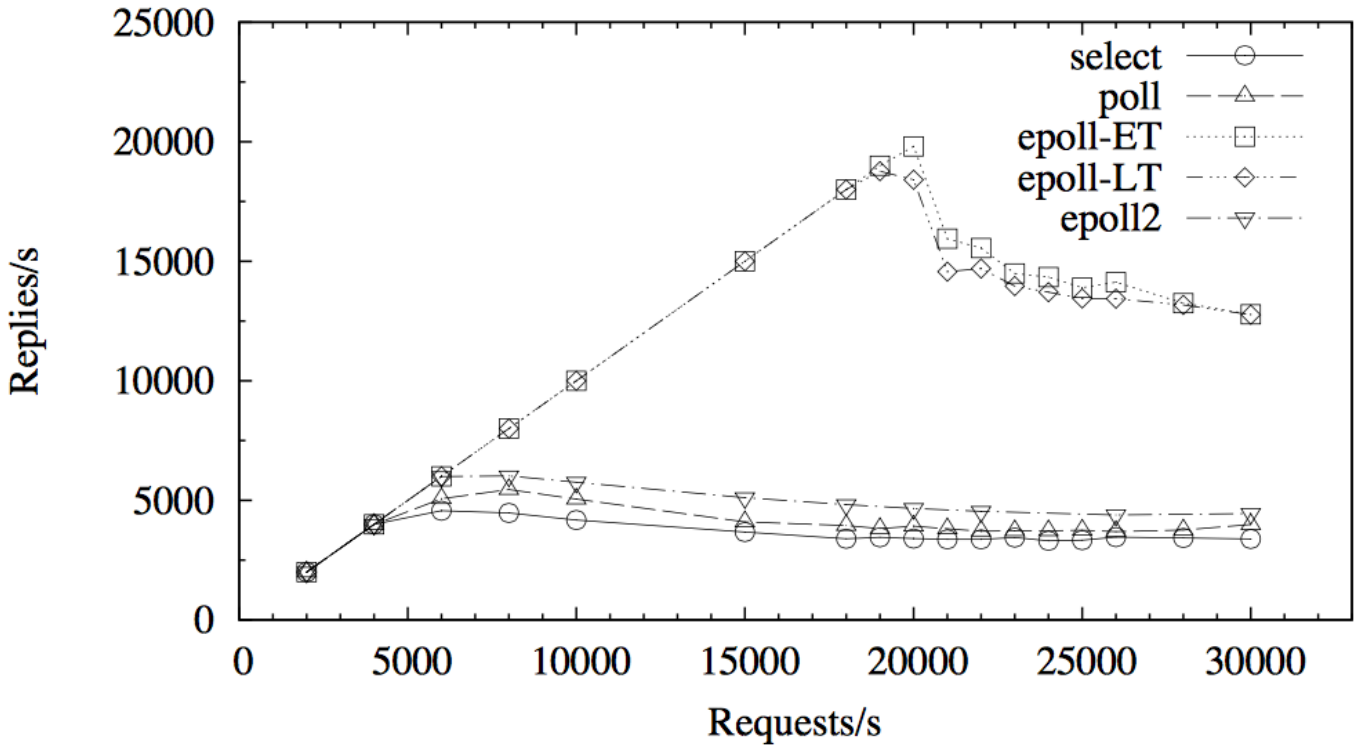
Both select and poll need to scan the entire fd set per call.
Time Complexity: linear to the number of connections.

generally doesn't scale beyond 10,000 connections!

epoll, event-driven notification facility

- new interface introduced since Linux kernel 2.5.44 in 2002.
- tracks the state changes of file descriptors, without iterating through the entire fd set.
- Time Complexity: linear to number of *active* connections.

select, poll v.s. epoll



μserver performance comparison using different notification facilities, with 10k idle connections.[2]

[2] Figure 5, Comparing and Evaluating epoll, select, and poll Event Mechanisms, Linux Symposium Vol.01, 2004

Epoll

```
//deprecated as of kernel version 2.6.8
//the size argument is since ignored,
//in favour of dynamic queue size.
int epoll_create(int size);

//the flags argument could be of,
//0, same as epoll_create() behaviour.
//EPOLL_CLOEXEC, with close-on-exec flag set.
int epoll_create1(int flags);

//op:EPOLL_CTL_ADD to register a new fd
//op:EPOLL_CTL_MOD to change an existing fd
//op:EPOLL_CTL_DEL to delete an existing fd
int epoll_ctl(int epfd, int op, int fd, struct epoll_event* event);

//epoll_event.events is a bit set for monitored events
//epoll_event.data.fd stores the file descriptor to monitor
struct epoll_event {
    uint32_t     events;      /* Epoll events */
    epoll_data_t data;        /* User data variable */
};
```

Epoll

```
int epoll_wait(int epfd, struct poll_event* events, int
    maxevents, int timeout);
```

- `epfd` is the epoll file descriptor that you want to use.
- active FDs will be copied to the `events` array upon return.
- `maxevents` is the max num of active events to copy per call.
- blocks until one or more FDs are ready.

Epoll Example

```
int main(){
  /* create, bind, listen here */

  int epfd = epoll_create1(0);
  if (epfd < 0) {
    /* handle epoll error here */
  }

  //adding server_sock to the epoll queue
  struct epoll_event ev, events[EPOLL_QUEUE_LEN];
  ev.events = EPOLLIN;
  ev.data.fd = server_sock;
  if (epoll_ctl(epfd, EPOLL_CTL_ADD, server_sock, &ev) < 0) {
    /* handle epoll_ctl error here */
  }
  // continued on the next page
```

Epoll Example (event loop)

```
  while (true) {
    num_fds = epoll_wait(epfd, events, EPOLL_QUEUE_LEN, -1);

    if (num_fds < 0) {
      /* handle epoll_wait error here */
    }

    //FDs in events are active
    for (int i = 0; i < num_fds; ++i) {
      if (events[i].data.fd == server_sock) {  //server_sock is ready to accept
        /* accept a client here */
        // you may reuse the ev struct
        ev.events = EPOLLIN;
        ev.data.fd = client_sock;
        if (epoll_ctl(epfd, EPOLL_CTL_ADD, client_sock, &ev) < 0) {
          /* handle epoll_ctl error here */
        }
      } else {  //client_sock is ready to read
        /* communicate with the client here */
        if (/* done with the client */) {  //remove the client from epoll queue
          if (epoll_ctl(epfd, EPOLL_CTL_DEL, events[i].data.fd, NULL) < 0) {
            /* handle epoll_ctl error here */
          }
          close(events[i].data.fd);
        }  //if
      }  //else
    }  //for
  }  //while
}  //main
```

Edge v.s. Level-Triggered

We mentioned that epoll tracks state changes of FDs, epoll could work under two modes,

- Edge-Triggered notify only when an FD changes from not ready to ready

  e.g., won't notify again if you don't empty the buffer first
- Level-Triggered notify whenever an FD is ready

  e.g., will always notify if there is data in the buffer

epoll works under LT mode by default,
this is how you could change it to ET,

```
struct epoll_event ev;
ev.events = EPOLLIN | EPOLLET;
ev.data.fd = sockfd;
if (epoll_ctl(epfd, EPOLL_CTL_ADD, sockfd, &ev) <
    0) {
  /* handle epoll_ctl error here */
}
```