Asynchronous Sockets

Tran Giang Son, tran-giang.son@usth.edu.vn

ICT Department, USTH

Contents

What & Why

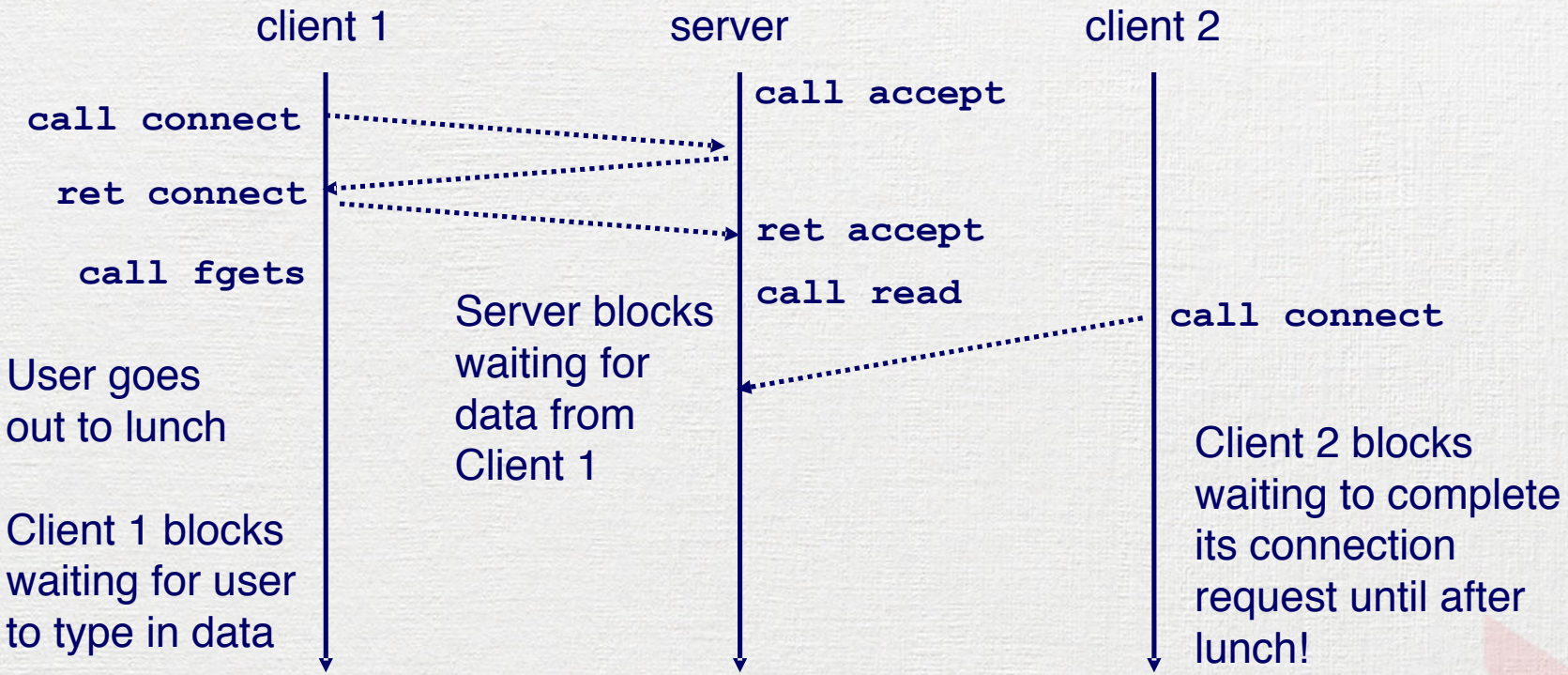
What?

- By default: blocking, e.g.
  - `accept()` only returns when there's an incoming connection
  - `read()`/`recv()` only return when there's some data
  - `write()`/`send()` only return when data is sucessfully sent
- Nonblocking: calls to network functions return to caller immediately

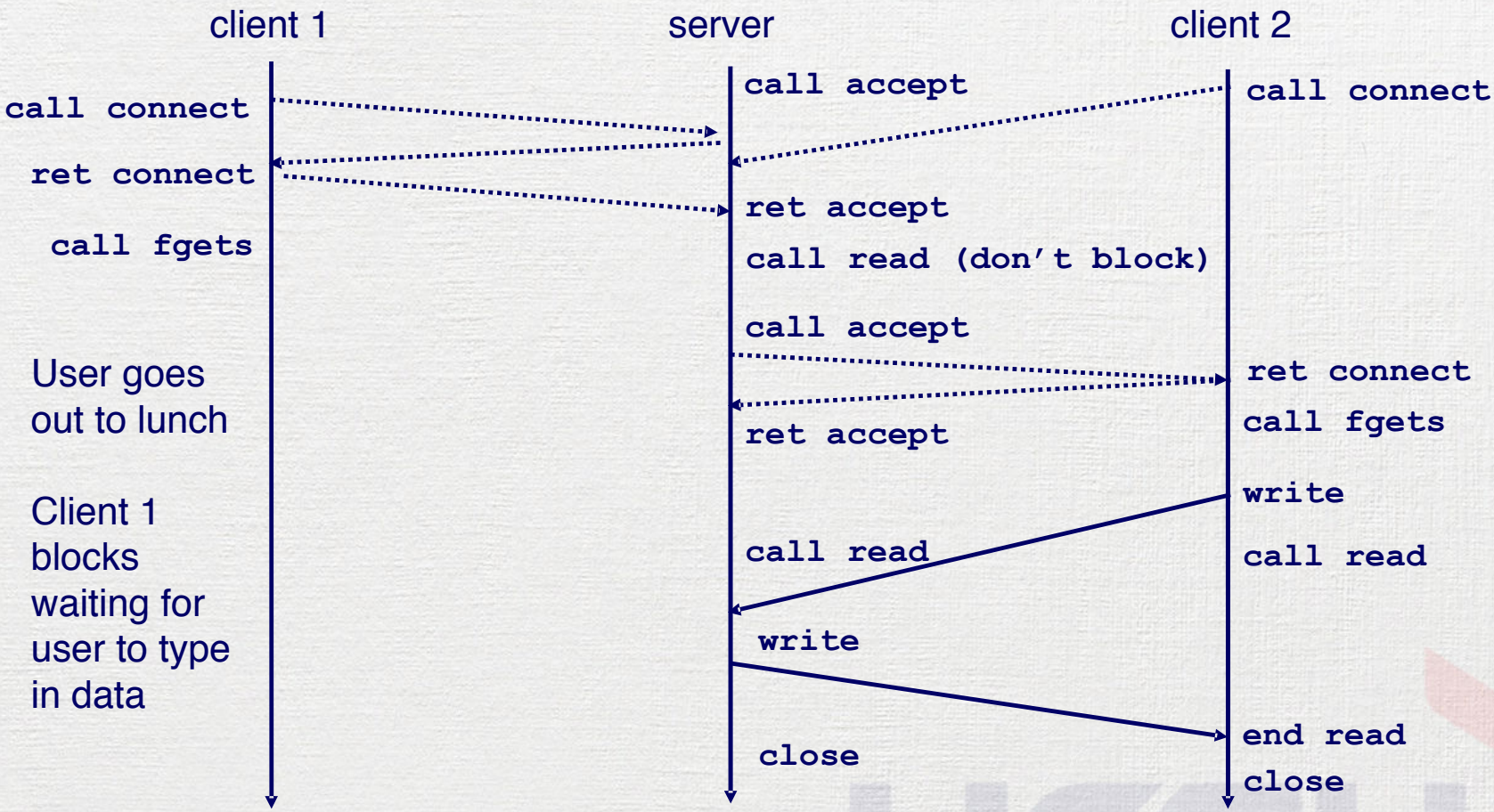
Why?

- With blocking socket
  - Server can only serve 1 client at anytime
  - Needs to take turn
  - No timeout

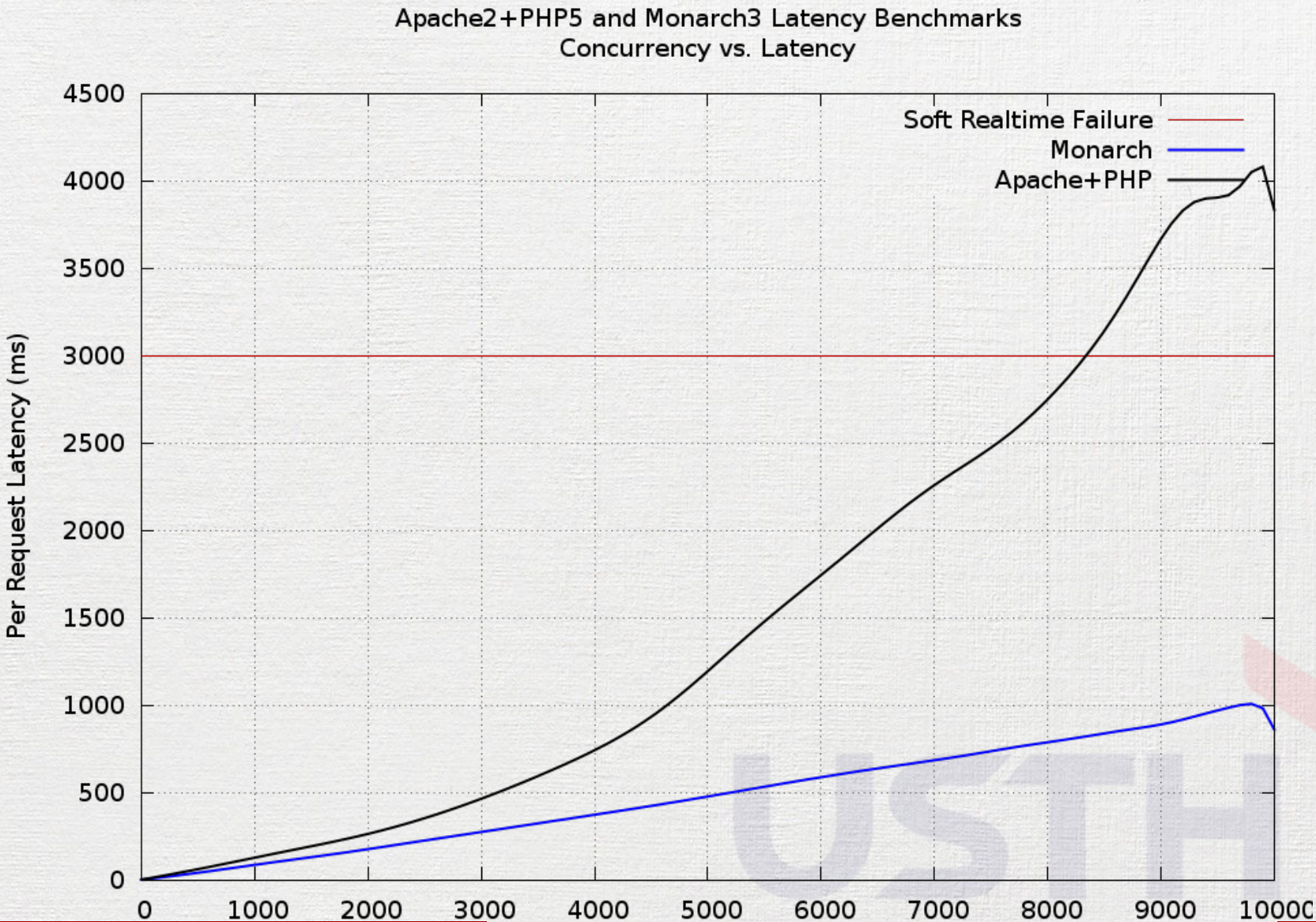
Why?


client 1
server
client 2
call connect
call accept
ret connect
ret accept
call fgets
call read
call connect
User goes out to lunch
Server blocks waiting for data from Client 1
Client 1 blocks waiting for user to type in data
Client 2 blocks waiting to complete its connection request until after lunch!

Why?


client 1
server
client 2
call connect
call accept
call connect
ret connect
ret accept
call fgets
call read (don't block)
call accept
User goes out to lunch
ret connect
ret accept
call fgets
write
Client 1 blocks waiting for user to type in data
call read
call read
write
end read
close
close

Why?


Apache2+PHP5 and Monarch3 Latency Benchmarks
Concurrency vs. Latency
Per Request Latency (ms)
Soft Realtime Failure
Monarch
Apache+PHP
0
500
1000
1500
2000
2500
3000
3500
4000
4500
0
1000
2000
3000
4000
5000
6000
7000
8000
9000
10000

What we need

- A single server that supports multiple client concurrently:
  - accepts multiple connections
  - receives messages from all connected clients
  - sends message from STDIN to all clients

Nonblocking Sockets

What?

- Nonblocking operations
  - `connect()`
  - `accept()`
  - `read()` / `write()`
  - `send()` / `recv()`

What?

- Looks great!
- . . . but. . .
  - Complication
  - Maintenance

How to use it?

- Allow reusing address
- Enable nonblocking option
- Restructure server & client
- Profit.

Reusing address

```
int setsockopt(int socket, int level,
    int option_name,
    const void *option_value, socklen_t option_len);
```

- **Set Sock**et **Opt**ions
- `socket`: the file descriptor returned by `socket()`
- `level`: protocol level
- `option_name`: the option that we need to adjust
- `option_value`: value for that option
- `option_len`: its length

Reusing address

Name	Meaning
SO_DEBUG	recording of debugging information
SO_REUSEADDR	**local address reuse**
SO_REUSEPORT	duplicate address and port bindings
SO_KEEPALIVE	keep connections alive
SO_DONTROUTE	routing bypass for outgoing messages
SO_BROADCAST	permission to transmit broadcast messages
SO_SNDBUF	set buffer size for output
SO_RCVBUF	set buffer size for input
SO_SNDTIMEO	set timeout value for output
SO_RCVTIMEO	set timeout value for input
SO_TYPE	get the type of the socket (get only)

Reusing address

Example:

```
setsockopt(sockfd, SOL_SOCKET,
    SO_REUSEADDR, &(int){ 1 },
    sizeof(int));
```

Enable nonblocking option

- `fcntl(int fd, int command, int value)`: file control
  - `F_GETFL`
  - `F_SETFL`
- `O_NONBLOCK`
- Example:

```
int fl = fcntl(fd, F_GETFL, 0);
fl |= O_NONBLOCK;
fcntl(fd, F_SETFL, fl);
```

Restructure server & client

Blocking Server

```
socket()...
bind()...
listen()...
while (1) {
  clientfd = accept();
  while (1) {
    read()...
    printf()...
    scanf()...
    write()...
  }
}
close()...
```

Non-blocking Server

```
socket()...
setsockopt()...    // reuse address
fcntl()...         // nonblocking
bind()...
listen()...
while (1) {
  clientfd = accept();
  if (clientfd > 0) {
    fcntl()... // nonblocking client
    while (1) {
      if (read()... > 0) printf()...
      if (poll()...) {
        scanf()...
        write()...
      }
    }
  }
```

Restructure server & client

Blocking Client

```
socket()...
gethostbyname()...
connect()...
while (1) {
    scanf()...
    write()...
    read()...
    printf()...
}
close()...
```

Non-blocking Client

```
socket()...
gethostbyname()...
connect()...
setsockopt()...    // reuse address
fcntl()...         // nonblocking
while (1) {
  if (read()... > 0) printf()...
  if (poll()...) {
    scanf()...
    write()...
  }
}
```

Practical Work 8: Nonblocking System

- Copy your client and server code from 7^{th} practical work to
  - « 08.practical.work.server.nonblock.c »
  - « 08.practical.work.client.nonblock.c »
  - Improve server and client: nonblocking sockets
- Test the system between your laptop and VPS
- Do you see any problem with your server & client?
- Push your C programs to corresponding forked Github repository

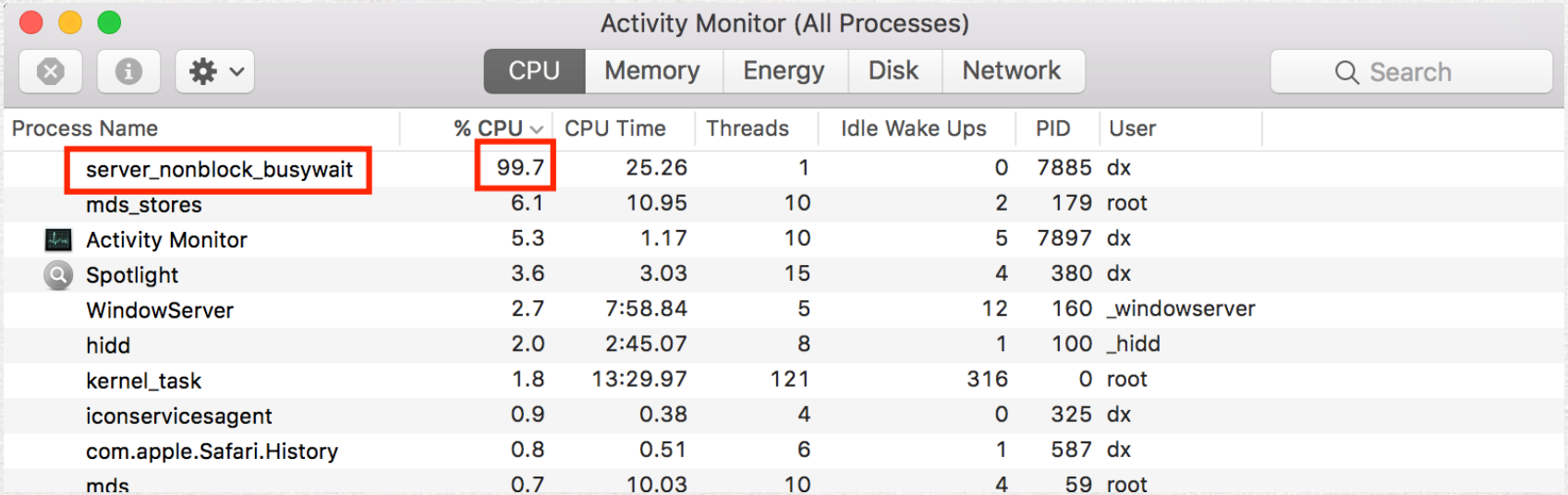
Multiplexing

What?

- Gather all sockets into one common set
- **Non-busy (blocked) wait** for an event from this set
- Handle the event
  - If event is from the `listen()` socket, that's an incoming connection
  - If event is from the `accept()` socket (server) or `connect()` (client), that's data

Why?

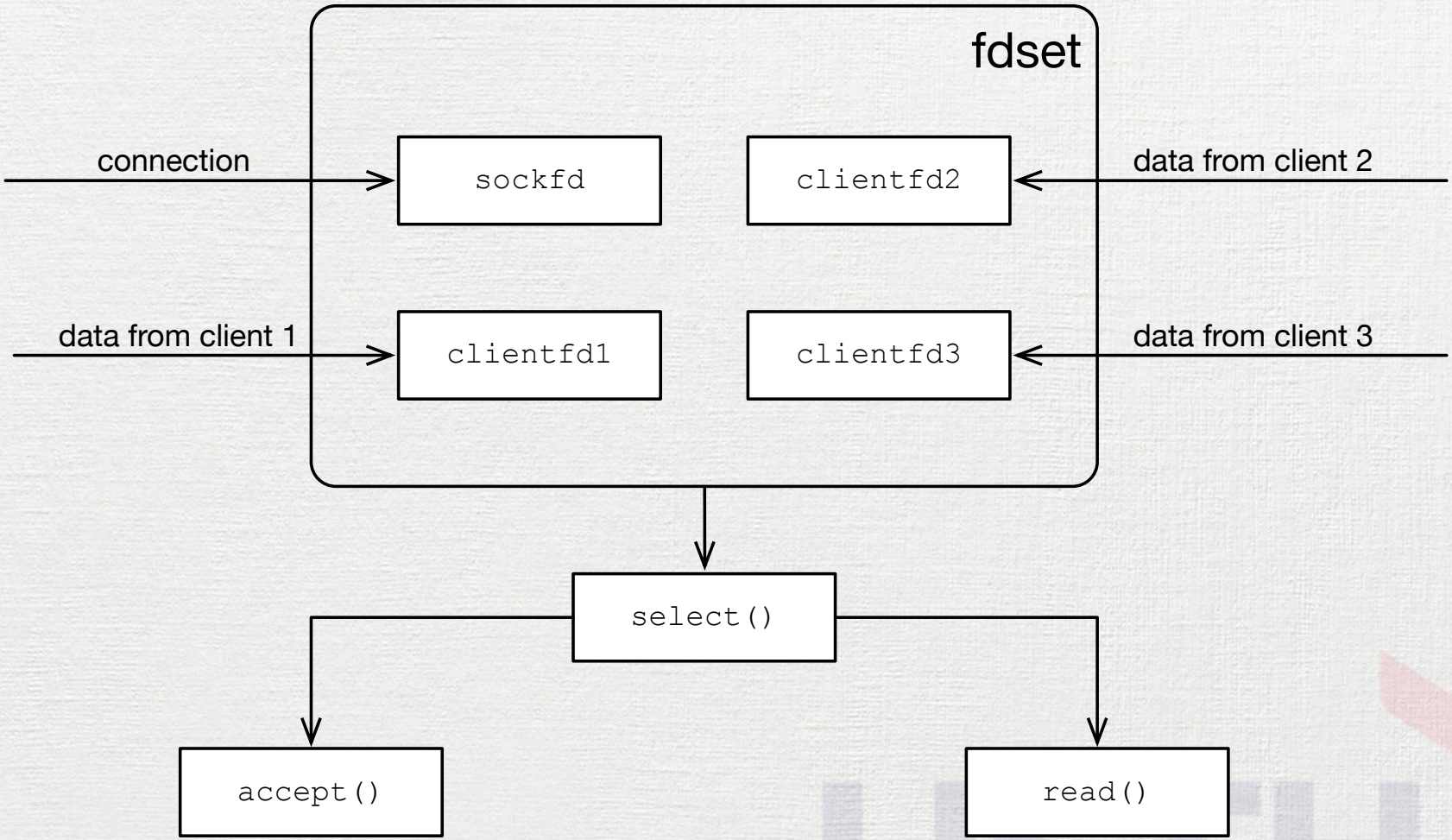
- Nonblocking sockets without multiplexing: busy waiting



Process Name	% CPU	CPU Time	Threads	Idle Wake Ups	PID	User
server_nonblock_busywait	99.7	25.26	1	0	7885	dx
mds_stores	6.1	10.95	10	2	179	root
Activity Monitor	5.3	1.17	10	5	7897	dx
Spotlight	3.6	3.03	15	4	380	dx
WindowServer	2.7	7:58.84	5	12	160	_windowserver
hidd	2.0	2:45.07	8	1	100	_hidd
kernel_task	1.8	13:29.97	121	316	0	root
iconservicesagent	0.9	0.38	4	0	325	dx
com.apple.Safari.History	0.8	0.51	6	1	587	dx
mds	0.7	10.03	10	4	59	root

```
while (1) {
  if (read()... > 0) printf()...
  if (poll()...) {
    scanf()...
    write()...
  }
}
```

How: Overview

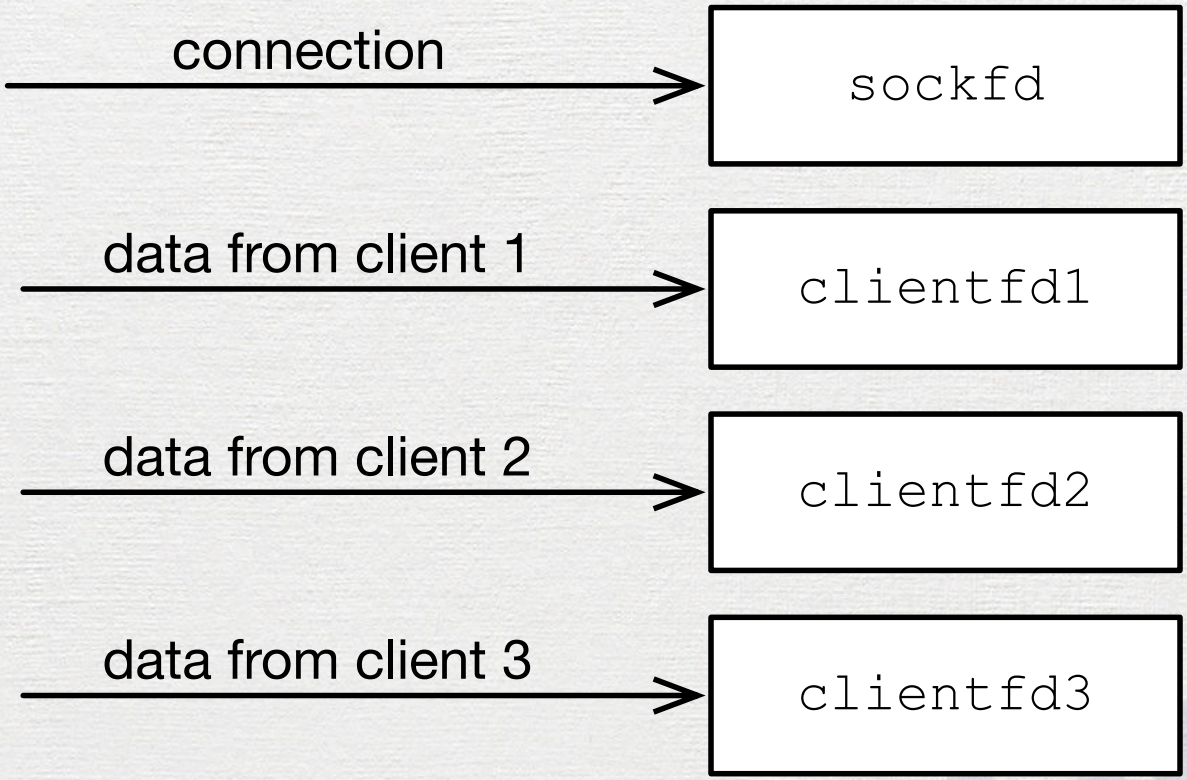


How?

- Group file descriptors into a «set»
- «Poll and **Wait**» the whole set
- Handle the «selected» file descriptors

How: Group file descriptors

What we have



How: Group file descriptors

What we need

How: Group file descriptors

- Include:

  ```
  #include <sys/select.h>
  ```

- Data structure: `fd_set`
- Macros:
  - Clear the set: `FD_ZERO(*set);`
  - Add a FD into the set: `FD_SET(int fd, fd_set *set);`
  - Remove a FD from the set `FD_CLR(int fd, fd_set *set);`
  - Check event for FD: `FD_ISSET(int fd, fd_set *set);`
- For help, `man select`

How: Group file descriptors

Multiplexed Nonblocking Server Example:

```
int clientfds[100];             // list of connected clients, >0 if valid
memset(clientfds, 0, sizeof(clientfds));        // clear the list
int sockfd = socket(AF_INET, SOCK_STREAM, 0);   // TCP, IPv4
setsockopt(sockfd, SOL_SOCKET, SO_REUSEADDR, &(int){1}, sizeof(int));
fcntl(sockfd, F_SETFL, )...   // nonblocking socket
bind(sockfd, (struct sockaddr *)&saddr, sizeof(saddr));
listen(sockfd, 5);              // listen, maximum 5 clients at anytime
while (1) {
    fd_set set;                 // declaration of the set
    FD_ZERO(&set);              // clear the set
    FD_SET(sockfd, &set);       // add listening sockfd to set
    for (int i = 0; i < 100; i++) {
        // add connected client sockets to set
        if (clientfds[i] > 0) FD_SET(clientfds[i], &set);
    }
    // each time we accept, we add client socket to clientfd[] array
    // but that's later
}
```

How: Poll and Wait

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

- `nfds`: maximum value of all FDs in the set, PLUS 1
- `readfds`: FDs to check for ready-to-read
- `writefds`: FDs to check for ready-to-write
- `errorfds`: FDs to check for ready-to-check-error
- `timeout`: as the name says...
  - if NULL, blocks indefinitely

How: Poll and Wait

Multiplexed Nonblocking Server Example:

```
while (1) {
    fd_set set;                 // declaration of the set
    FD_ZERO(&set);              // clear the set
    FD_SET(sockfd, &set);       // add listening sockfd to set
    int maxfd = sockfd;         // a required value to pass to select()
    for (int i = 0; i < 100; i++) {
        // add connected client sockets to set
        if (clientfds[i] > 0) FD_SET(clientfds[i], &set);
        if (clientfds[i] > maxfd) maxfd = clientfds[i];
    }
    // poll and wait, blocked indefinitely
    select(maxfd+1, &set, NULL, NULL, NULL);

    // each time we accept, we add client socket to clientfds[] array
    // but that's later
}
```

How: Handle the event

- Use `FD_ISSET()` to check each FDs in the set
  - A "listen'' socket: a new connection
  - A client socket: some data is ready to read / write-check error

How: Handle the event

```
// poll and wait, blocked indefinitely
select(maxfd+1, &set, NULL, NULL, NULL);

// a «listening» socket?
if (FD_ISSET(sockfd, &set)) {
    clientfd = accept(sockfd, (struct sockaddr *) &saddr, &clen);

    // make it nonblocking
    fl = fcntl(clientfd, F_GETFL, 0);
    fl |= O_NONBLOCK;
    fcntl(clientfd, F_SETFL, fl);

    // add it to the clientfds array
    for (int i = 0; i < MAX_CLIENT; i++) {
        if (clientfds[i] == 0) {
            clientfds[i] = clientfd;
            break;
        }
    }
}
```

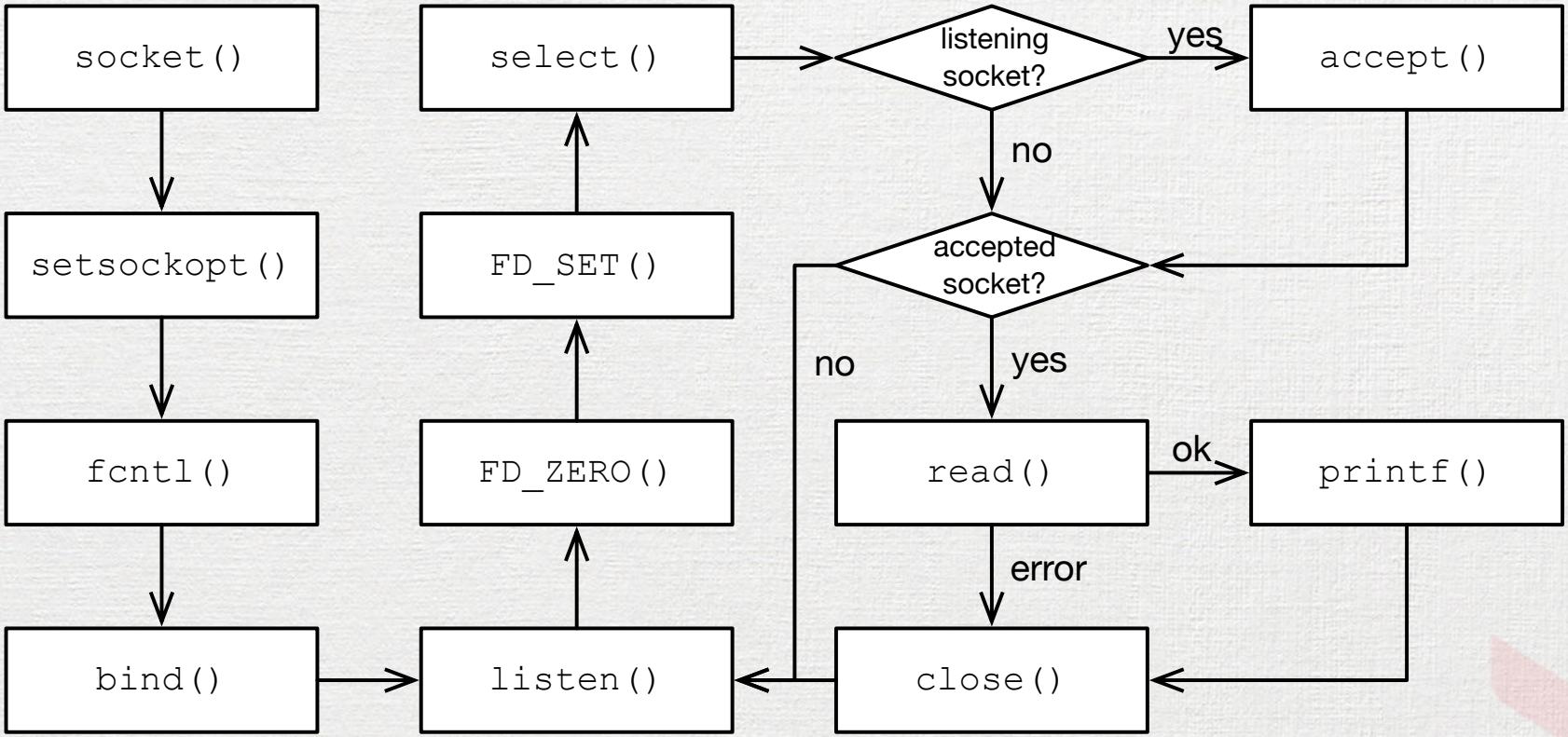
How: Handle the event

```
// poll and wait, blocked indefinitely
select(maxfd+1, &set, NULL, NULL, NULL);

// a «listening» socket?
if (FD_ISSET(sockfd, &set)) { ... }

// is that data from a previously-connect client?
for (i = 0; i < MAX_CLIENT; i++) {
    if (clientfds[i] > 0 && FD_ISSET(clientfds[i], &set)) {
        if (read(clientfds[i], s, sizeof(s)) > 0) {
            printf("client %d says: %s\nserver>", clientfds[i], s);
        }
        else {
            // some error. remove it from the "active" fd array
            printf("client %d has disconnected.\n", clientfds[i]);
            clientfds[i] = 0;
        }
    }
}
```

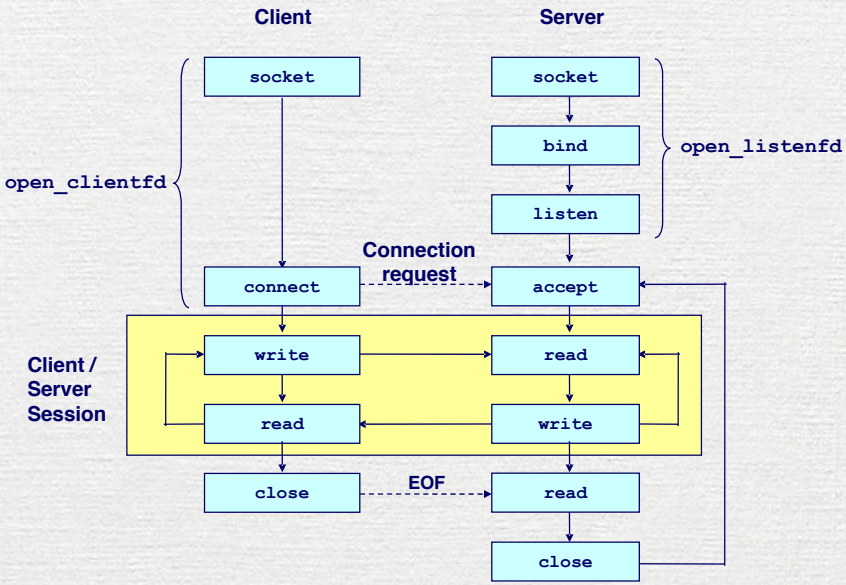
How: Recap

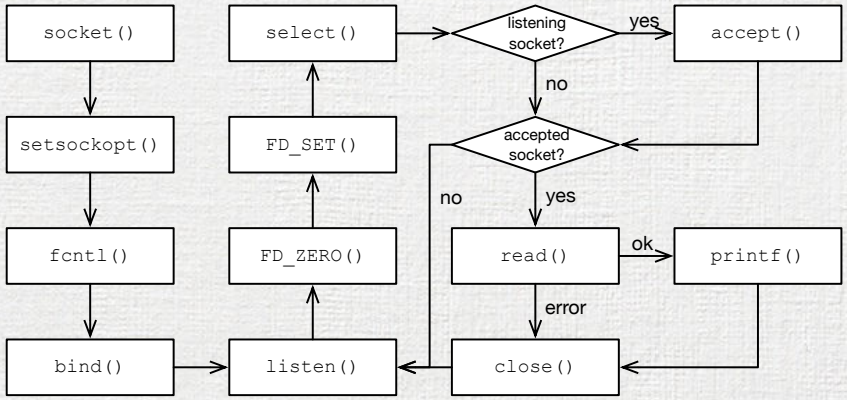

socket()
setsockopt()
fcntl()
bind()
listen()
FD_ZERO()
FD_SET()
select()
listening socket?
yes
accept()
no
accepted socket?
no
yes
read()
ok
printf()
error
close()

How: Recap

```
socket()...
setsockopt()...
fnctl()...
bind()...
listen()...
while (1) {
    FD_ZERO()...
    FD_SET(sockfd, ...)
    for each client:
        FD_SET(clientfd, ...)
    select()...
    if FD_ISSET(sockfd) {
        newclientfd = accept()...
    }
    for each client:
        if FD_ISSET(clientfd) {
            read()...
            printf()...
        }
    }
}
```

Remind: Blocking vs Multiplexed Nonblocking





Practical Work 9: Multiplexed Nonblocking System

- Copy your server code from 8^{th} practical work to
  - « 09.practical.work.server.multiplex.c »
  - Improve server: add multiplexing
- Test the system between your laptop and VPS
- Push your C program to corresponding forked Github repository