SWE2024

SUNGKYUNKWAN UNIVERSITY

Concurrent Programming

Prof. Joonwon Lee (joonwon@skku.edu)
TA – Jaehyung Park (jaeseanpark@gmail.com)
TA – Luke Albano (lukealbano@arcs.skku.edu)
Sungkyunkwan University

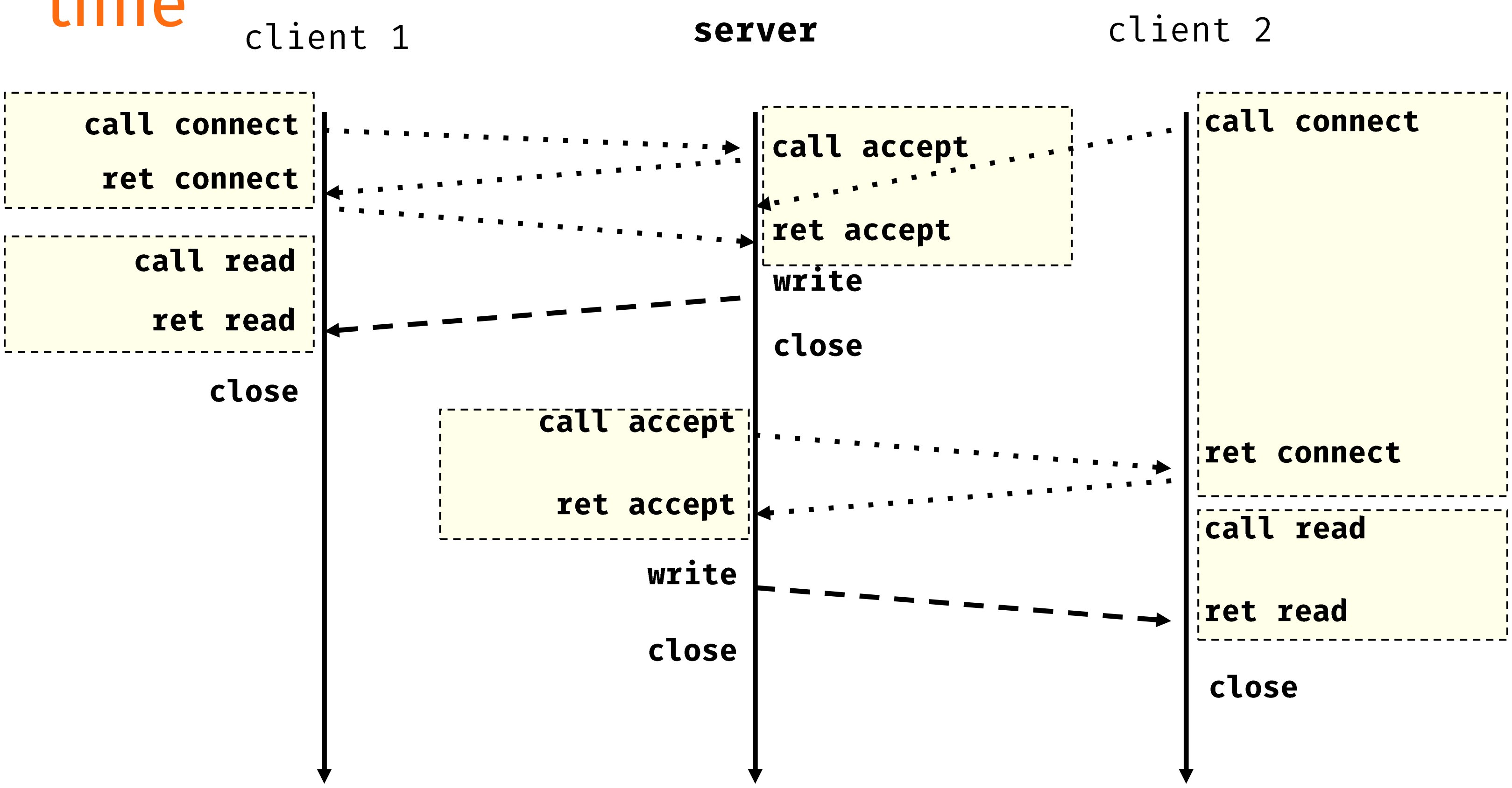
Iterative Server

- Iterative echo server from previous lecture

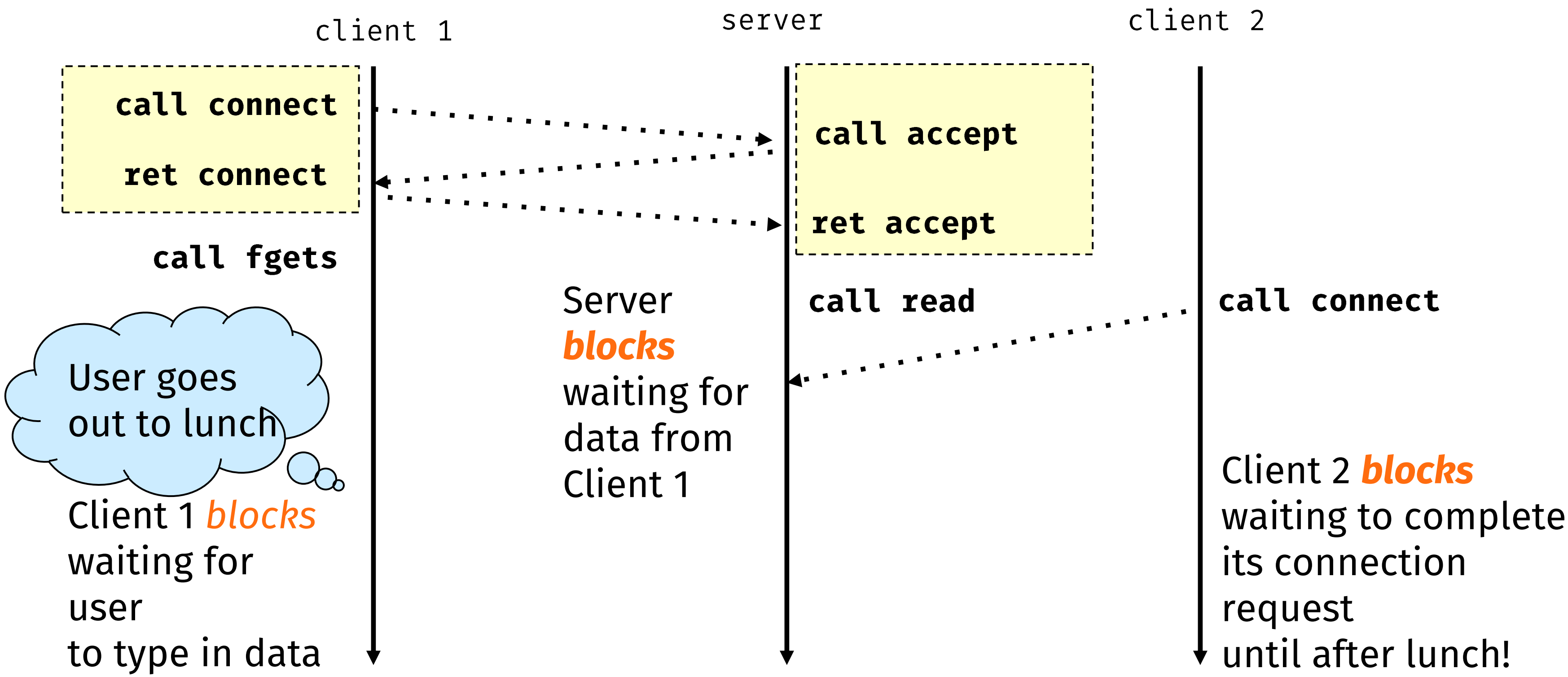
```
int main (int argc, char *argv[]) {
    . . .

    while (1) {
        if ((connfd = accept(listenfd, (struct sockaddr *)&caddr,
                             (socklen_t *)&caddrlen)) < 0) {
            printf("accept() failed\n");
            continue;
        }

        while ((n = read(connfd, buf, MAXLINE)) > 0) {
            printf("got %d bytes from client.\n", n);
            write(connfd, buf, n);
        }

        close(connfd);
    }
}
```

Iterative Server

- Iterative server processes one request at a time



Fundamental Flaw of Iterative Server



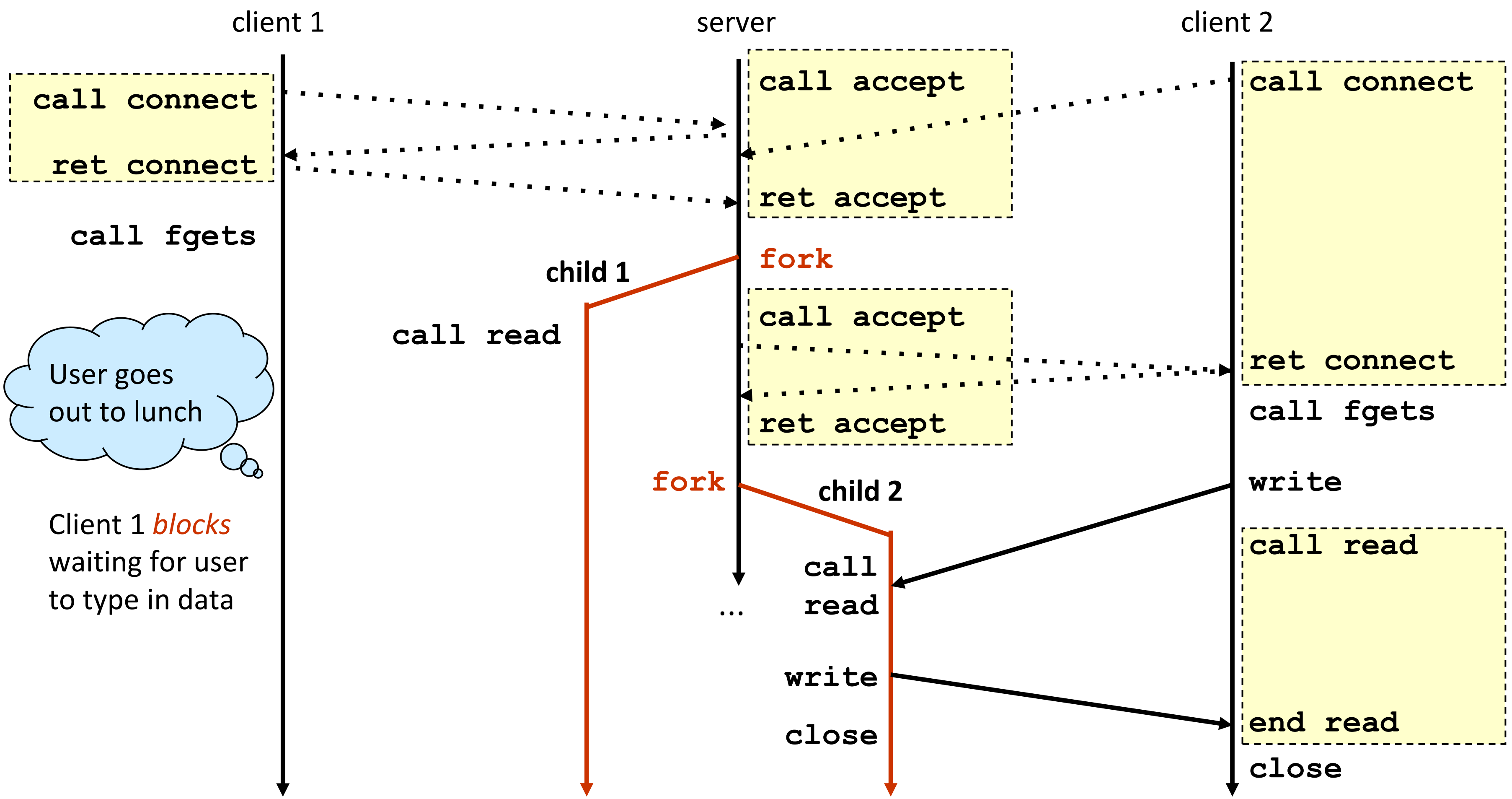
- Solution: use concurrent server
  - Use multiple concurrent flows to serve multiple clients at the same time

Creating Concurrent Flows

- Process-based
  - Kernel automatically interleaves multiple logical flows
  - Each flow has its own private address space
- Thread-based
  - Kernel automatically interleaves multiple logical flows
  - Each flow shares the same address space
- Event-based
  - User manually interleaves multiple logical flows
  - All flows share the same address space
  - Popular for high-performance server design
  - Using technique called **I/O multiplexing**

Approach #1: Process-based Server

- Concurrent server with multiple processes



Approach #1: Process-based Server

- Server must reap zombie children
  - To avoid fatal memory leak

- Server must close its copy of *connfd*
  - Kernel keeps reference for each socket/open file
  - After **fork()**, **refcnt(connfd) = 2**
  - Connection will not be closed until **refcnt(connfd) = 0**

Process-based Concurrent Echo Server

```c
int main (int argc, char *argv[]) {
    . . .
    // parent process gets SIGCHLD, when child process terminates or stops
    signal(SIGCHLD, handler);
    while (1) {
        if ((connfd = accept(listenfd, (struct sockaddr *)&caddr,
                            (socklen_t *)&caddrlen)) < 0) {
            printf("accept() failed\n");
            continue;
        }
        if (fork() == 0) {
            close(listenfd);
            while ((n = read(connfd, buf, MAXLINE)) > 0) {
                printf("got %d bytes from client.\n", n);
                write(connfd, buf, n);
            }
            close(connfd);
            exit(0);
        }
        close(connfd);
    }
}
```

```c
void handler(int sig) {
    pid_t pid;
    int stat;
    // wait for any child process that has changed state
    while ((pid = waitpid(-1, &stat, WNOHANG)) > 0);
    return;
}
```

Pros and Cons of Process-based Design

- Pros
  - Handling multiple connections concurrently
  - Clean sharing model
    - descriptors (no)
    - open file tables (yes)
    - global variables (no)
  - Simple and straightforward
- Cons
  - Additional overhead for process control
    - Process creation, termination, and switching
  - Nontrivial to share data between processes
    - Require IPC mechanisms

Approach #2: Thread-based Server

- Multiple threads within a process
  - Each thread shares same code, data, virtual address space of the process
  - Each thread has its own logical control flow, stack, thread ID (TID)

- vs. Process
  - Process does not share all code and data
  - Threads are somewhat less expensive than processes
    - Process control (creating and reaping) twice as expensive as thread control

Issues with Thread-based Server

- Must run "detached" to run independently and automatically free memory resources
  - Joinable thread can be reaped and killed by other threads
    - Must be reaped with `pthread_join()`
  - Detached thread cannot be reaped or killed by other threads
    - Resources are automatically reaped on termination
    - Exit state and return value are not saved
  - Default state is joinable
    - Use `pthread_detach(pthread_self())` to make detached

Issues with Thread-based Server

- Must be careful to avoid unintended sharing
  - e.g., what happens if we pass the address of **connfd** to the thread routine?

```
int connfd;
while (1) {
    connfd = accept ( … );
    pthread_create(&tid, NULL, thread_func, &connfd);
}
```

- All functions called by a thread must be thread-safe
  - There is no problem in program execution
    - Even if more than one thread calls function at a time

Thread-based Concurrent Echo Server

```c
int main (int argc, char *argv[]) {
    . . .
    int *connfdp;
    pthread_id tid;
    while (1) {
        connfdp = (int *)malloc(sizeof(int));
        if ((*connfdp = accept(listenfd, (struct sockaddr *)&caddr,
                               (socklen_t *)&caddrlen)) < 0) {
            printf("accept() failed\n");
            free(connfdp);
            continue;
        }
        pthread_create(&tid, NULL, thread_func, connfdp);
    }
}
```

```c
void *thread_func(void *arg) {
    int n;
    char buf[MAXLINE];
    int connfd = *((int*)arg);
    pthread_detach(pthread_self());
    free(arg);
    while ((n = read(connfd, buf, MAXLINE)) > 0)
        write(connfd, buf, n);
    close(connfd);
    return NULL;
}
```

Pros and Cons of Thread-based Design

- Pros
  - Easy to share data structures between threads
  - Threads are more efficient (cheaper) than processes
- Cons
  - Unintentional sharing can introduce subtle and hard-to-reproduce errors
    - Ease of data sharing is both the greatest strength and the greatest weakness of threads

Approach #3: Event-based Server

- Use I/O multiplexing technique
  - Provide more control with less overhead

- Event-based concurrent servers
  - Maintain pool of connected descriptors
  - Repeat following forever:
    - Use Unix **`select()`** system call to block until:
      - (a) New connection request arrives on the listening descriptor
      - (b) New data arrives on an existing connected descriptor
    - If (a), add new connection to the pool of connections
    - If (b), read any available data from the connection
      - Close connection on EOF and remove if from the pool

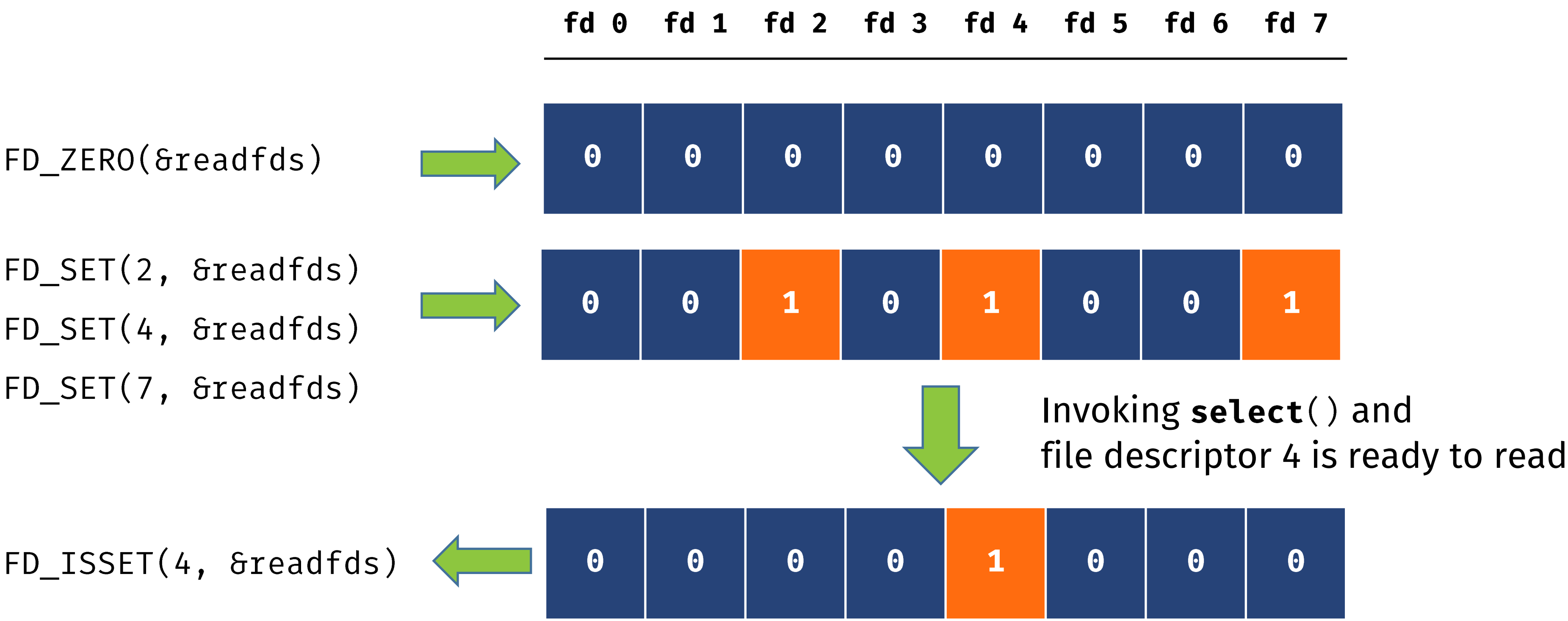
I/O Multiplexing Function

- `int select(int nfds, fd_set* readfds, fd_set* writefds, fd_set*exceptfds, struct timeval* timeout)`
  - **nfds**
    - The highest-numbered fd in any of three sets, plus 1
  - **readfds / writefds / exceptfds**
    - Sets of file descriptors are watched for reading/writing/exception
  - **timeout**
    - Interval that **select()** should block waiting for fds to become ready
  - Return value
    - Return number of ready descriptors
    - And sets each bit of **readfds/writefds/exceptfds** to indicate the ready status of its corresponding descriptor

I/O Multiplexing Macros

- Macros for manipulating set descriptors
  - `void FD_ZERO(fd_set* fdset)`
    - Turn off all bits in **fdset**
  - `void FD_SET(int fd, fd_set* fdset)`
    - Turn on bit **fd** in **fdset**
  - `void FD_CLR(int fd, fd_set* fdset)`
    - Turn off bit **fd** in **fdset**
  - `int FD_ISSET(int fd, fd_set* fdset)`
    - Is bit **fd** in **fdset** turned on?
      - If on, return Non-zero
      - If off, return 0

I/O Multiplexing Macros Example



- *fd_set* does not remember the previous state
  - Value of previous *fd_set* must be stored before invoking **select**()

Event-based Concurrent Echo Server

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

int main (int argc, char *argv[]) {
    . . .
    fd_set readset, copyset;
    FD_ZERO(&readset);
    FD_SET(listenfd, &readset);
    int fdmax = listenfd, fdnum;

    while (1) {
        copyset = readset;
        struct timeval timeout;
        timeout.tv_sec = 5;
        timeout.tv_usec = 0;

        if ((fdnum = select(fdmax + 1, &copyset, NULL, NULL, &timeout)) < 0) {
            printf("select() failed\n");
     exit(0);
        }

        if (fdnum == 0) {
            printf("Time out\n");
     continue;
        }
```

Event-based Concurrent Echo Server

```
for (int i = 0; i < fdmax + 1; i++) {
    if (FD_ISSET(i, &copyset)) {
        if (i == listenfd) {
            if ((connfd = accept(listenfd, (struct sockaddr *)&caddr,
                                 (socklen_t *)&caddrlen)) < 0) {
                printf ("accept() failed.\n");
                continue;
            }
            FD_SET(connfd, &readset);
            if (fdmax < connfd) fdmax = connfd;
        }
        else {
            if ((n = read(i, buf, MAXLINE)) > 0) {
                printf ("got %d bytes from client.\n", n);
                write(i, buf, n);
            }
            else {
                FD_CLR(i, &readset);
                printf("connection terminated.\n");
                close(i);
            }
        }
    }
}   . . .
```

Other I/O Multiplexing Function

- *int poll (struct pollfd *fds, nfds_t ndfs, int timeout)*
  - More efficient for large-valued or sparse file descriptors
  - **fds**
    - Set of file descriptors to be monitored is specified
    - Also contain requested events and returned events

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

  - **ndfs**
    - Specify number of items in **fds** array
  - **timeout**
    - Interval that **poll()** should block waiting for **fds** to become ready
  - Return value
    - Number of structures which have nonzero revents fields

Other I/O Multiplexing Function

- `int epoll_wait (int epfd, struct epoll_event *events, int maxevents, int timeout)`
  - O(1) operation
  - `epoll_create()`, `epoll_ctl()`, and wait for events using `epoll_wait()`
  - **`epfd`**
    - epoll file descriptor (return value from `epoll_create()`)
  - **`events`**
    - Return address of `epoll_event` structures that contain ready fd
  - **`maxevents`**
    - Specify that maximum number of events to process at a time
  - **`timeout`**
    - Interval that `epoll_wait()` should block waiting for fd to become ready

Pros and Cons of Event-based Server

- Pros
  - One logical control flow and address space
  - Can single-step with a debugger
  - No process or thread control overhead
    - Design of choice for high-performance servers

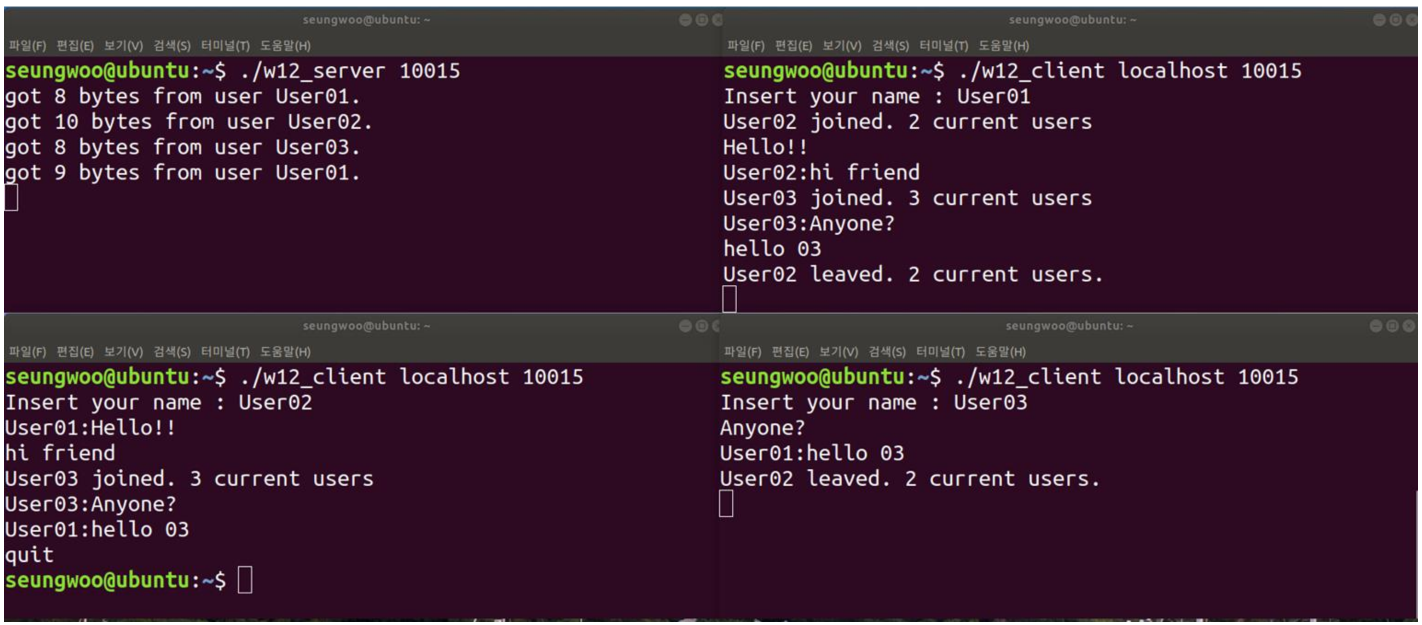
- Cons
  - Significantly more complex to code than process- or thread-
    based designs
  - Can be vulnerable if malicious client sends particular text and halts

Lab Exercise

- Make *chatting room* (Up to 10 users) using select()
  - Client should insert the **name** (Up to 10 bytes) of itself
  - Client should be under an infinite loop with conditional exit ("quit")
  - When client sends message (Up to 80 bytes), **server receives it and sends it to the other clients**
  - When client joins or leaves chatting room, server sends message with number of current clients to other clients
  - Use FD_ISSET() to check if there is input in specific file descriptor
  - **No timeout** is needed
    - e.g., select(nfds, &readset, NULL, NULL, **NULL**);

Lab Exercise

- Skeleton code
  - `cp ~swe2024-41_23s/2023s/p12  ./ -r`
- Example)



Exercise Submission

- Submit your exercise code and Makefile
  - InUiYeJi cluster
  - Submit the folder into p12
  - ~swe2024-41_23s/bin/submit p12 p12
  - Due date: Tonight, 17 May 2023, 23:59
  - We will compile by using command *make*
    - If compilation fails, your points for this exercise will be **zero**

```
./p12

   server.c
   client.c
   Makefile
```

Summary Report

- Summary report about man command result of
  - **`select()`**
  - **`poll()`**
  - **`epoll_wait()`**

- Submission form
  - A4 size PDF format (No page limitation)
  - [SWE2024 Report-10] studentID_name
  - Ex) [SWE2024 Report-10] 2022XXXXXX_홍길동
  - Submit to iCampus
  - Due by Friday, 19 May 2023, 23:59