I/O Multiplexing

Insu Yun

Approaches for Writing Concurrent Servers

Allow server to handle multiple clients concurrently

1. Process-based
   - Kernel automatically interleaves multiple logical flows
   - Each flow has its own private address space
2. Event-based
   - Programmer manually interleaves multiple logical flows
   - All flows share the same address space
   - Uses technique called *I/O multiplexing*
3. Thread-based
   - Kernel automatically interleaves multiple logical flows
   - Each flow shares the same address space
   - Hybrid of of process-based and event-based

Motivation

- The server must respond to two independent I/O requests
  (1) A network client making a connection request
  (2) A user typing a command line at the keyboard

- Which event do we wait for first?
  - Neither option is ideal!

- One solution to this dilemma is I/O multiplexing

Overview

- Use the `select` function to ask the kernel to suspend process, returning control to the application only after one or more I/O events have occurred

- Example:
  - Return when any descriptor in the set {0, 4} is ready for reading
  - Return when any descriptor in the set {1, 2, 7} is ready for writing.
  - Time out if 152.13 seconds have elapsed waiting for an I/O event to occur

select

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

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

void FD_CLR(int fd, fd_set *set);   /* Clear all bits in fdset */
int  FD_ISSET(int fd, fd_set *set); /* Clear bit fd in fdset */
void FD_SET(int fd, fd_set *set);   /* Turn on bit fd in fdset */
void FD_ZERO(fd_set *set);          /* Is bit fd is fdset on */
```

More on `select`

- `select` function manipulates set of type `fd_set`, which are known as *descriptor sets.*
- A descriptor set: a bit vector of size n

$$b_{n-1}b_{n-2}\ldots b_1 b_0$$

  - Each bit b_k corresponds to a descriptor k

More on `select`

- The `select f`unction takes five inputs:
  (1) n: A cardinality of descriptor sets
  (2) readset: A descriptor set for checking readable
  (3) writeset: A descriptor set for checking writable
  (4) exceptset: A descriptor set for checking exception conditions
    - Arrival of out-of-band data for a socket
    - The presence of control status information to be read from the master side of a pseudo terminal
  (5) Timeout

`select` function Descriptor Arguments

- Array of integers : each bit in each integer correspond to a descriptor (fd_set)

- 4 macros
  - void FD_ZERO(fd_set *fdset); /* clear all bits in fdset */
  - void FD_SET(int fd, fd_set *fdset); /* turn on the bit for fd in fdset */
  - void FD_CLR(int fd, fd_set *fdset); /* turn off the bit for fd in fdset*/
  - int FD_ISSET(int fd, fd_set *fdset);/* is the bit for fd on in fdset ? */

`nfds` argument to select function

- Specifies the number of descriptors to be tested.
- Its value is the maximum descriptor to be tested, plus one. (hence maxfd + 1)
  - Descriptors 0, 1, 2, up through and including `nfds`-1 are tested
  - Example: interested in fds 1,2, and 5 -> maxfdp1 = 6
  - Your code has to calculate the maxfdp1 value

- Constant FD_SETSIZE defined by including <sys/select.h>
  - is the number of descriptors in the fd_set datatype. (often = 1024)

Example

```
fd_set readset, writeset;
FD_ZERO(&readset);
FD_ZERO(&writeset);
FD_SET(0, &readset);
FD_SET(3, &readset);
FD_SET(1, &writeset);
FD_SET(2, &writeset);
select(4, &readset, &writeset, NULL, NULL);
```

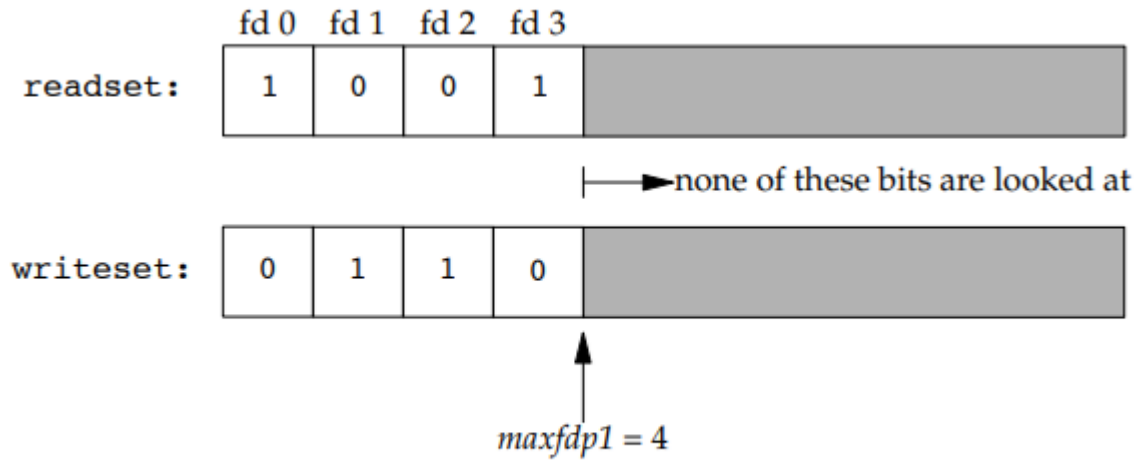


Figure 14.16 Example descriptor sets for `select`

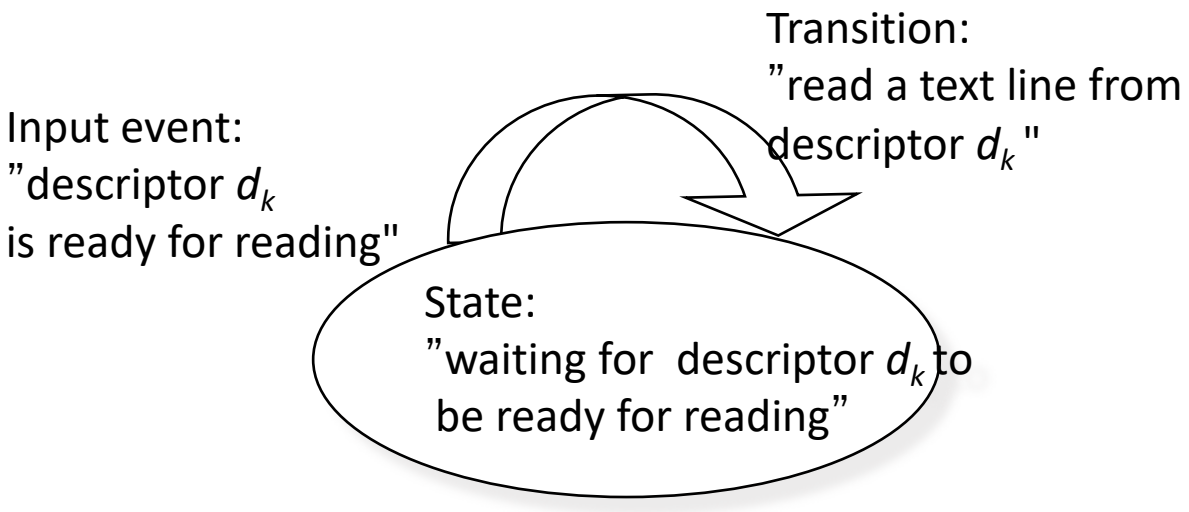
Condition for a socket to be ready for select

Condition	Readable?	Writable?	Exception?
Data to read	•		
Read half of the connection closed	•		
New connection ready for listening socket	•		
Space available for writing		•	
Write half of the connection closed		•	
Pending error	•	•	
TCP out-of-band data			•

More on `select`

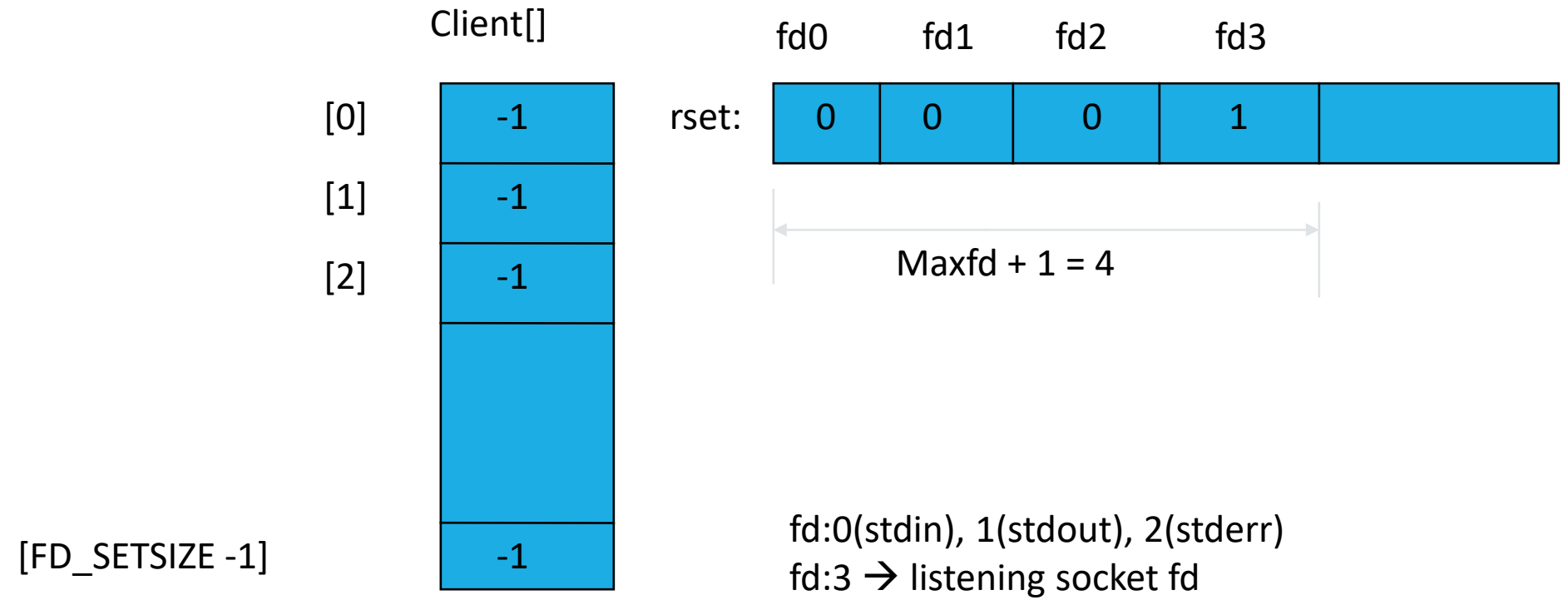
- Three possible return values from `select`
  - A return value of -1 means that an error occurred.
    - None of the descriptor sets will be modified
  - A return value of 0 means that no descriptors are ready.
    - All the descriptor sets will be zeroed out
  - A positive return value specifies the number of descriptors that are ready.
    - The only bits left on in the three descriptor sets are the bits corresponding to the descriptors that are ready
    - Use FD_ISSET macro to determine which descriptors are ready for reading

A logical flow in a event-driven server



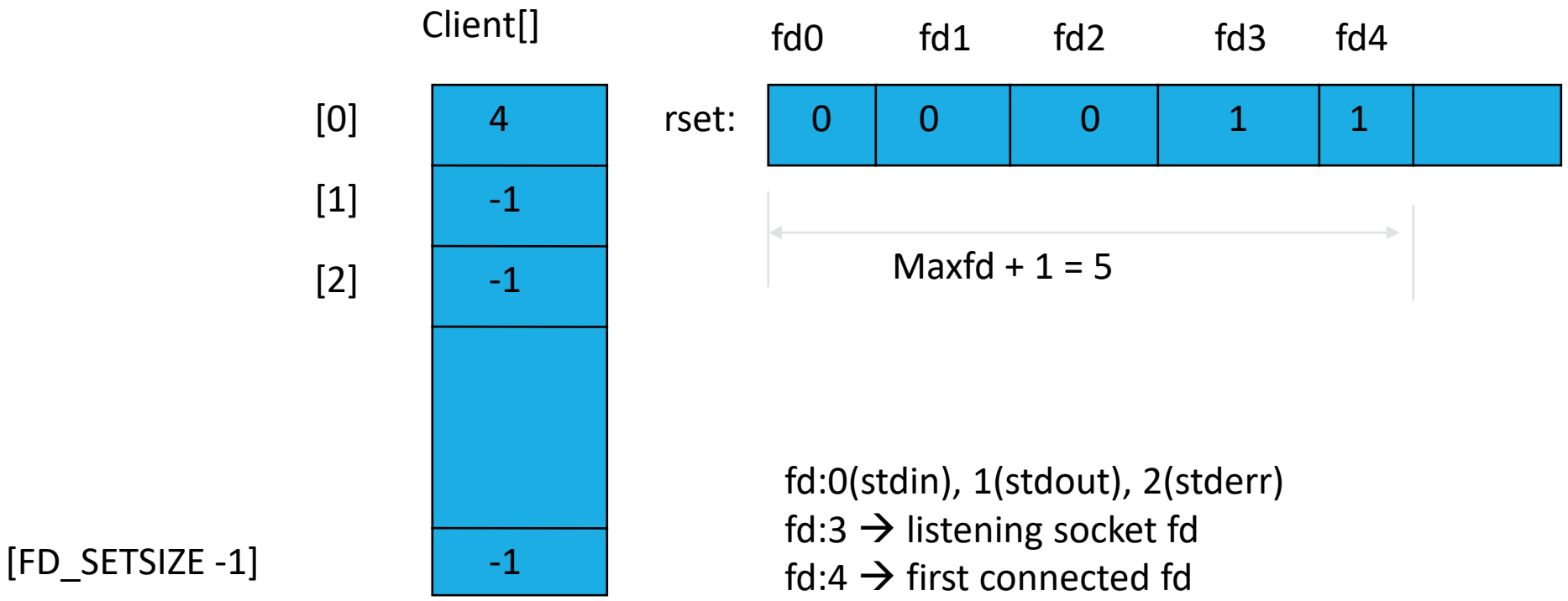
TCP echo server using select 1/5

- Rewrite the server as a single process that uses select to handle any number of clients, instead of forking one child per client.
- Before first client has established a connection



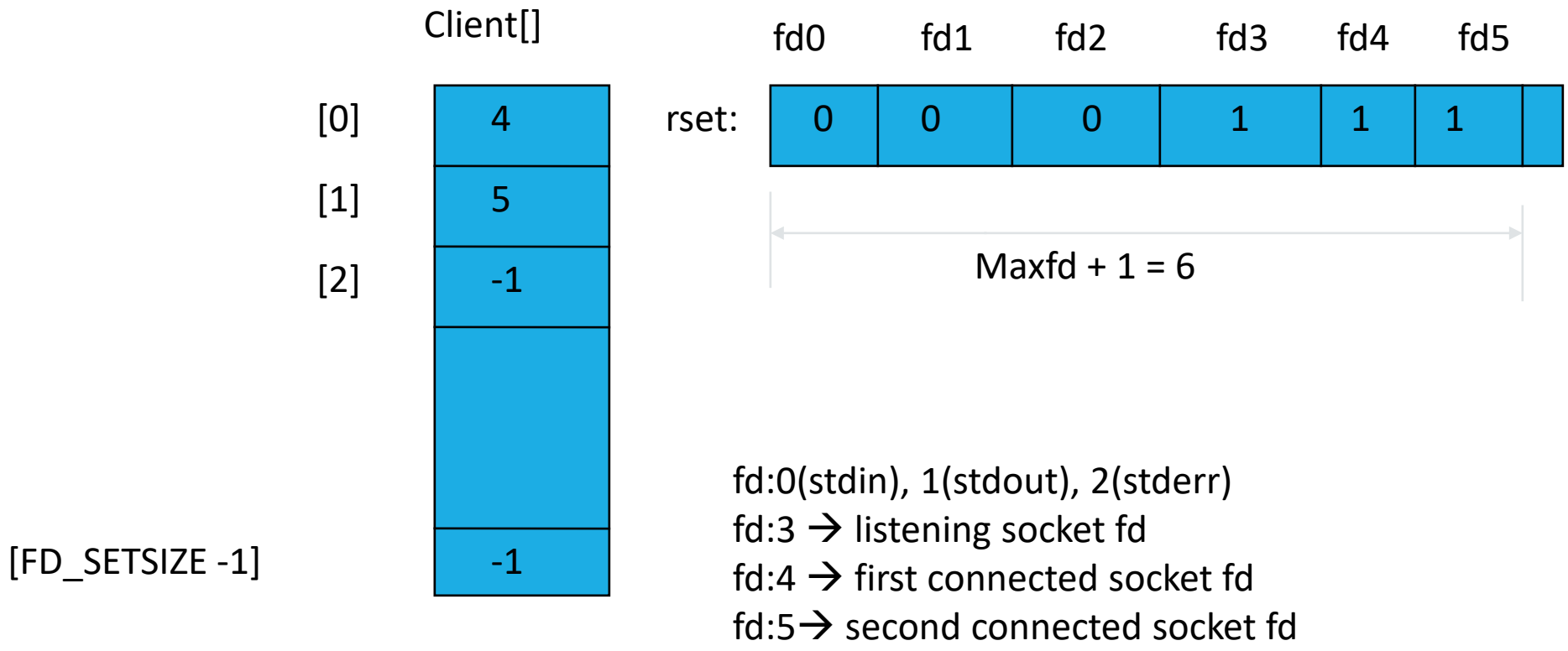
TCP echo server using select 2/5

- After first client connection is established (assuming connected descriptor returned by accept is 4)



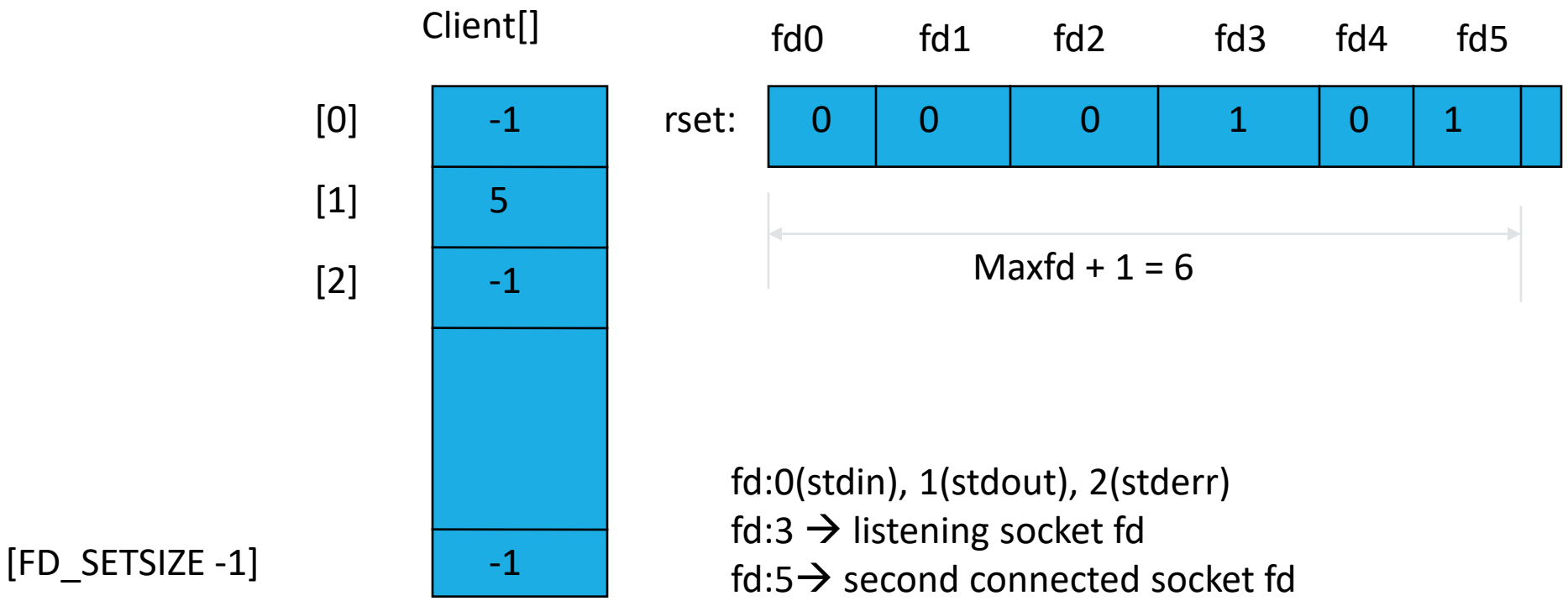
TCP echo server using select 3/5

- After second client connection is established (assuming connected descriptor returned by accept is 5)



TCP echo server using select 4/5

- First client terminates its connection (fd 4 readable and read returns 0 -> the client sends EOF)



TCP echo server using select 5/5

- As clients arrive, record connected socket descriptor in first available entry in client array (first entry = -1)
- Add connected socket to read descriptor set
- Keep track of
  - Highest index in client array that is currently in use
  - maxfd +1
- The limit on number of clients to be served
  - Min (
    FD_SETSIZE,
    Max (Number of descriptors allowed for this process by the kernel))

```c
#include "csapp.h"

typedef struct { /* Represents a pool of connected descriptors */
    int maxfd;        /* Largest descriptor in read_set */
    fd_set read_set;  /* Set of all active descriptors */
    fd_set ready_set; /* Subset of descriptors ready for reading  */
    int nready;       /* Number of ready descriptors from select */
    int maxi;         /* Highwater index into client array */
    int clientfd[FD_SETSIZE];    /* Set of active descriptors */
    rio_t clientrio[FD_SETSIZE]; /* Set of active read buffers */
} pool;
```

```c
int byte_cnt = 0; /* Counts total bytes received by server */

int main(int argc, char **argv)
{
  int listenfd, connfd;
  socklen_t clientlen;
  struct sockaddr_storage clientaddr;
  static pool pool;

  if (argc != 2) {
    fprintf(stderr, "usage: %s <port>\n", argv[0]);
    exit(0);
  }
  listenfd = Open_listenfd(argv[1]);
  init_pool(listenfd, &pool);

  while (1) {
    /* Wait for listening/connected descriptor(s) to become ready */
    pool.ready_set = pool.read_set;
    pool.nready = Select(pool.maxfd+1, &pool.ready_set, NULL, NULL, NULL);

    /* If listening descriptor ready, add new client to pool */
    if (FD_ISSET(listenfd, &pool.ready_set)) {
      clientlen = sizeof(struct sockaddr_storage);
      connfd = Accept(listenfd, (SA *)&clientaddr, &clientlen);
      add_client(connfd, &pool);
    }

    /* Echo a text line from each ready connected descriptor */
    check_clients(&pool);
  }
}
```

```c
void init_pool(int listenfd, pool *p)
{
    /* Initially, there are no connected descriptors */
    int i;
    p->maxi = -1;
    for (i=0; i< FD_SETSIZE; i++)
       p->clientfd[i] = -1;

    /* Initially, listenfd is only member of select read set */
    p->maxfd = listenfd;
    FD_ZERO(&p->read_set);
    FD_SET(listenfd, &p->read_set);
}
```

```c
void add_client(int connfd, pool *p)
{
  int i;
  p->nready--;
  for (i = 0; i < FD_SETSIZE; i++)  /* Find an available slot */
    if (p->clientfd[i] < 0) {
      /* Add connected descriptor to the pool */
      p->clientfd[i] = connfd;
      Rio_readinitb(&p->clientrio[i], connfd);

      /* Add the descriptor to descriptor set */
      FD_SET(connfd, &p->read_set);

      /* Update max descriptor and pool highwater mark */
      if (connfd > p->maxfd)
        p->maxfd = connfd;
      if (i > p->maxi)
        p->maxi = i;
      break;
    }
  if (i == FD_SETSIZE) /* Couldn't find an empty slot */
    app_error("add_client error: Too many clients");
}
```

```c
void check_clients(pool *p)
{
  int i, connfd, n;
  char buf[MAXLINE];
  rio_t rio;

  for (i = 0; (i <= p->maxi) && (p->nready > 0); i++) {
    connfd = p->clientfd[i];
    rio = p->clientrio[i];

    /* If the descriptor is ready, echo a text line from it */
    if ((connfd > 0) && (FD_ISSET(connfd, &p->ready_set))) {
      p->nready--;
      if ((n = Rio_readlineb(&rio, buf, MAXLINE)) != 0) {
        byte_cnt += n;
        printf("Server received %d (%d total) bytes on fd %d\n",
            n, byte_cnt, connfd);
        Rio_writen(connfd, buf, n);
      }

      /* EOF detected, remove descriptor from pool */
      else {
        Close(connfd);
        FD_CLR(connfd, &p->read_set);
        p->clientfd[i] = -1;
      }
    }
  }
}
```

Unfortunately, this server is not stable!

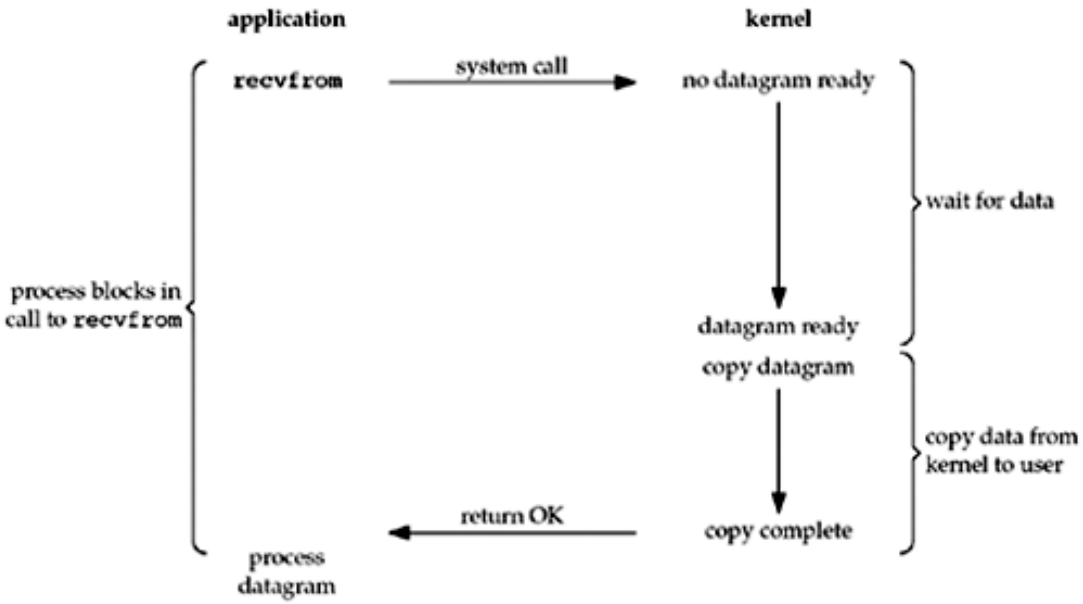
- read() can be blocked!

Under Linux, **select**() may report a socket file descriptor as "ready for reading", while nevertheless a subsequent read blocks. This could for example happen when data has arrived but upon examination has wrong checksum and is discarded. There may be other circumstances in which a file descriptor is spuriously reported as ready. Thus it may be safer to use **O_NONBLOCK** on sockets that should not block.

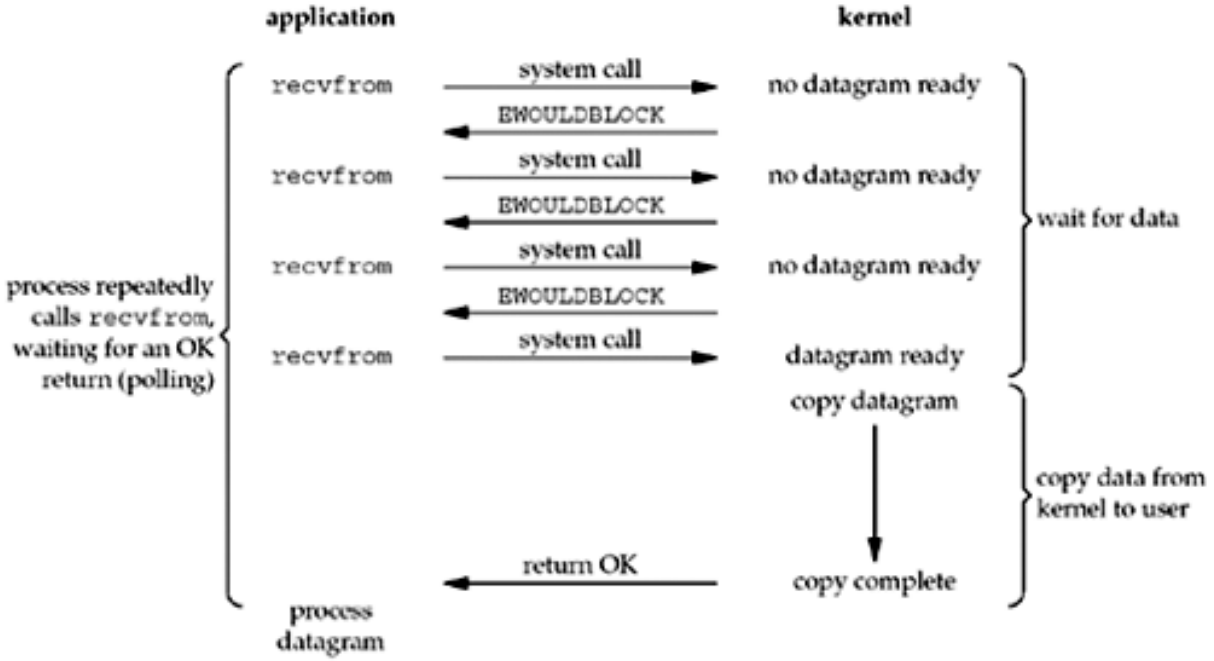
- accept() can be blocked, too!

There may not always be a connection waiting after a **SIGIO** is delivered or ***select****(2)* or ***poll****(2)* return a readability event because the connection might have been removed by an asynchronous network error or another thread before **accept**() is called. If this happens then the call will block waiting for the next connection to arrive. To ensure that **accept**() never blocks, the passed socket *sockfd* needs to have the **O_NONBLOCK** flag set (see ***socket****(7)*).

Non-blocking I/O



Blocking I/O



Non-blocking I/O

Reminder: fcntl( )

- The fcntl function is used for five different purposes:
  - Duplicate an existing descriptor (*cmd* = F_DUPFD or F_DUPFD_CLOEXEC)
  - Get/set file descriptor flags (*cmd* = F_GETFD or F_SETFD)
  - Get/set file status flags (*cmd* = F_GETFL or F_SETFL)
  - Get/set asynchronous I/O ownership (*cmd* = F_GETOWN or F_SETOWN)
  - Get/set record locks (*cmd* = F_GETLK, F_SETLK, or F_SETLKW)

```
#include <fcntl.h>

int fcntl(int fd, int cmd, ... /* int arg */ );

/* Returns: depends on cmd if OK (see following), -1 on error */
```

Make socket non-blocking with fcntl

```
--- echoservers.c
+++ echoserver-non-block.c
@@ -33,6 +33,7 @@
     exit(0);
   }
   listenfd = Open_listenfd(argv[1]);
+  Fcntl(listenfd, F_SETFD, O_NONBLOCK);
   init_pool(listenfd, &pool);

   while (1) {
@@ -44,6 +45,7 @@
     if (FD_ISSET(listenfd, &pool.ready_set)) {
       clientlen = sizeof(struct sockaddr_storage);
       connfd = Accept(listenfd, (SA *)&clientaddr, &clientlen);
+      Fcntl(connfd, F_SETFD, O_NONBLOCK);
       add_client(connfd, &pool);
     }
```

```
--- csapp.c
+++ csapp-non-blocking.c
@@ -582,8 +582,10 @@
 {
     int rc;

-    if ((rc = accept(s, addr, addrlen)) < 0)
+    if ((rc = accept(s, addr, addrlen)) < 0) {
+      if (errno != EWOULDBLOCK)
         unix_error("Accept error");
+    }
     return rc;
 }

@@ -754,7 +756,7 @@

     while (nleft > 0) {
        if ((nread = read(fd, bufp, nleft)) < 0) {
-           if (errno == EINTR) /* Interrupted by sig ha
ndler return */
+           if (errno == EINTR || errno == EWOULDBLOCK)
 /* Interrupted by sig handler return */
                nread = 0;      /* and call read() agai
n */
            else
                return -
1;      /* errno set by read() */

…
```

Issues with `select`

- select() can monitor only file descriptors numbers that are less than FD_SETSIZE (1024)

- The implementation of the fd_set arguments as value-result
  - i.e., Need to keep track the original fd_set for the next call

- O(n) for checking which file descriptor is ready for a certain event

epoll

- Linux has a scalable I/O event notification mechanism called epoll that can monitor a set of file descriptors to see whether there is any I/O ready for them. There are three system calls, as described below, that form the api for epoll.

epoll_create1

```
int epoll_create1(int flags);
```

- This function creates an epoll object and returns a file descriptor. The only valid flag is EPOLL_CLOEXEC, which closes the descriptor on exec as you might expect

epoll_wait

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

- This function waits for any of the events being monitored, until there is a timeout. It returns up to maxevents at once and populates the events array with each event that has occurred.

epoll_ctl

```
int epoll_ctl(int epfd, int op, int fd, struct epoll_event *event);
```

- This function configures which descriptors are watched by the object, and op can be EPOLL_CTL_ADD, EPOLL_CTL_MOD, or EPOLL_CTL_DEL. We will investigate struct epoll_event on the next slide.

epoll_event

- The struct epoll_event is defined as follows:

```
struct epoll_event {
 uint32_t events; /* Epoll events */
 epoll_data_t data; /* User data variable */
};
```

- epoll_data_t is a typedef'd union, defined as follows:

```
typedef union epoll_data {
 void *ptr;
 int fd;
 uint32_t u32;
 uint64_t u64;
} epoll_data_t;
```

epoll_event

- The events member is a bit mask, and for our purposes, we care about three values:
  - EPOLLIN : the file is available for reading
  - EPOLLOUT : the file is available for writing
  - EPOLLET : This sets the file descriptor to be "edge triggered", meaning that events are delivered when there is a change on the descriptor (e.g., there is data to be read).

More on edge-triggered

(1) The file descriptor that represents the read side of a pipe (rfd) is registered on the epoll instance.

(2) A pipe writer writes 2 kB of data on the write side of the pipe.

(3) A call to epoll_wait(2) is done that will return rfd as a ready file descriptor.

(4) The pipe reader reads 1 kB of data from rfd

(5) A call to epoll_wait(2) is done.

- epoll with EPOLLET: hang
- epoll without EPOLLET: will return

epoll_echoserver.c

- https://gist.github.com/insuyun/bc6b480f26f50c181a702cd90de558a7

```
    if ((epfd = epoll_create1(0)) < 0) {
        die("epoll_create1");
    }

    listener = setup_socket();

    memset(&ev, 0, sizeof ev);
    ev.events = EPOLLIN;
    ev.data.fd = listener;
    epoll_ctl(epfd, EPOLL_CTL_ADD, listener, &ev);
```

```
    for (;;) {
        int i;
        int nfd = epoll_wait(epfd, events, MAX_EVENTS, -1);

        for (i = 0; i < nfd; i++) {
            if (events[i].data.fd == listener) {
                struct sockaddr_in client_addr;
                socklen_t client_addr_len = sizeof client_addr;

                int client = accept(listener,
                    (struct sockaddr *) &client_addr, &client_addr_len);
                if (client < 0) {
                    if (errno != EWOULDBLOCK)
                            perror("accept");
                    continue;
                }

                setnonblocking(client);
                memset(&ev, 0, sizeof ev);
                ev.events = EPOLLIN | EPOLLET;
                ev.data.fd = client;
                epoll_ctl(epfd, EPOLL_CTL_ADD, client, &ev);
            }
```

```
            } else {
                int client = events[i].data.fd;
                int n = read(client, buffer, sizeof buffer);
                if (n < 0) {
                  if (errno != EWOULDBLOCK) {
                    epoll_ctl(epfd, EPOLL_CTL_DEL, client, &ev);
                    close(client);
                  }
                } else if (n == 0) {
                    epoll_ctl(epfd, EPOLL_CTL_DEL, client, &ev);
                    close(client);
                } else {
                    write(client, buffer, n);
                }
            }
        }
    }
```