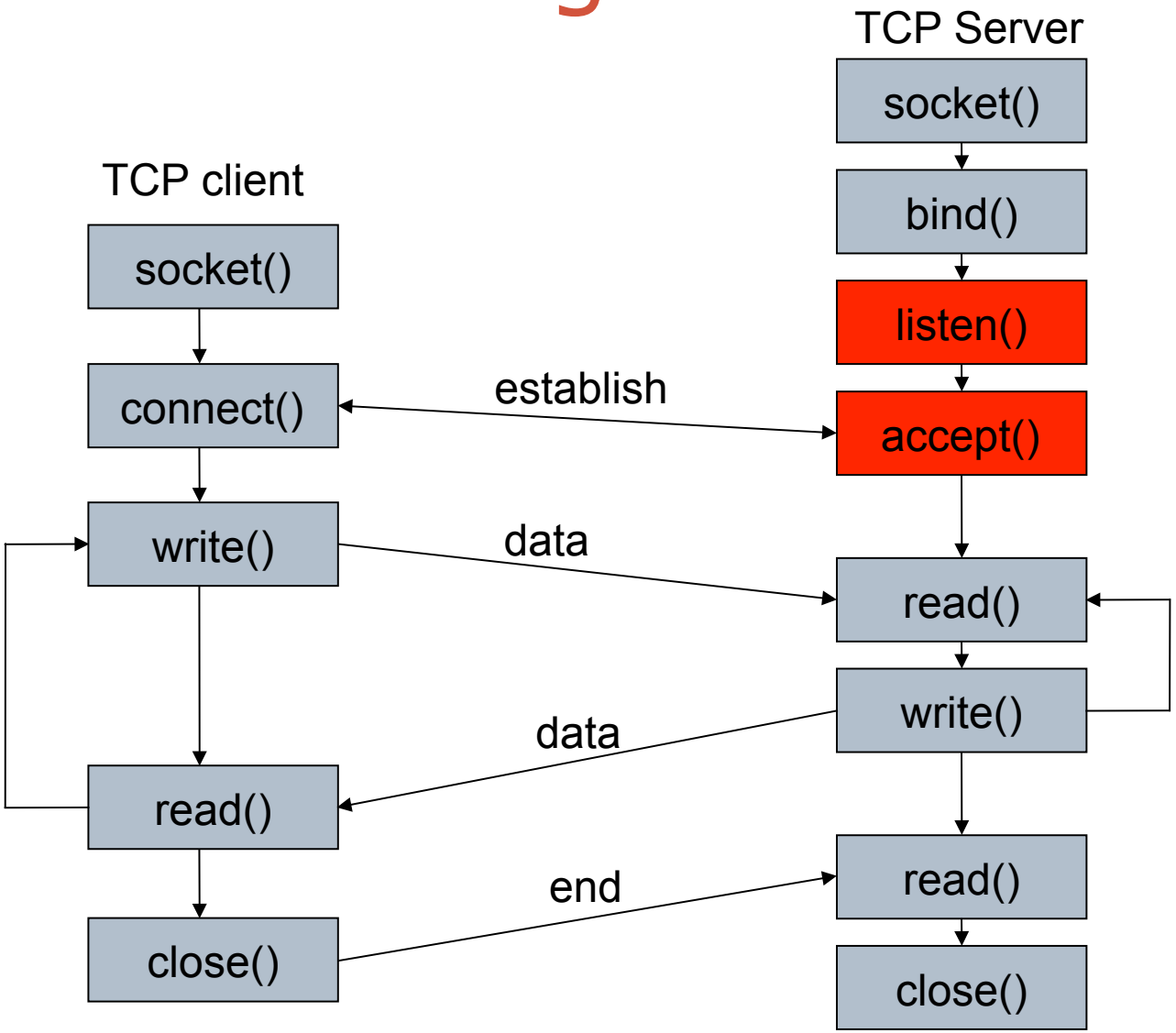
MULTI-THREAD TCP SERVER

Content

- Forking Server
- Socket I/O Models
- select()

Server for a single client



Question

- How could server accept another client after a client close its session?
- What happened if there are two clients want to connect to Server ?

Socket Mode

- Types of server sockets
  - *Iterating server*: Only one socket is opened at a time.
  - *Forking server*: After an accept, a child process is forked off to handle the connection.
  - *Concurrent single server*: use “select” to simultaneously wait on all open socketIds, and waking up the process only when new data arrives

Iterating server

- Simple server
- When a client request takes longer to service, we can’t handle other clients
  - →Need a *concurrent server*
- The simplest way to write a concurrent server under Unix is to *fork* a child process to handle each client

Fork process

- **fork** is an operation where a process creates a copy of itself
- Happen in multitask operating system when a process launch another process → child process
- The parent process makes a copy of its memory and gives to the child process

- Fork system call is first introduced in UNIX.
- First process in Linux is “init”

fork

```
#include <unistd.h>

pid_t fork(void);
```

- This function (and its variants) is the only way in Unix-like OS to create a new process.
- It returns
  - Once in the calling process (called the parent) with a return value that is the process ID of the newly created process (the child).
  - Once in the child, with a return value of 0 → *distinguish the child and the parent process*

Use fork

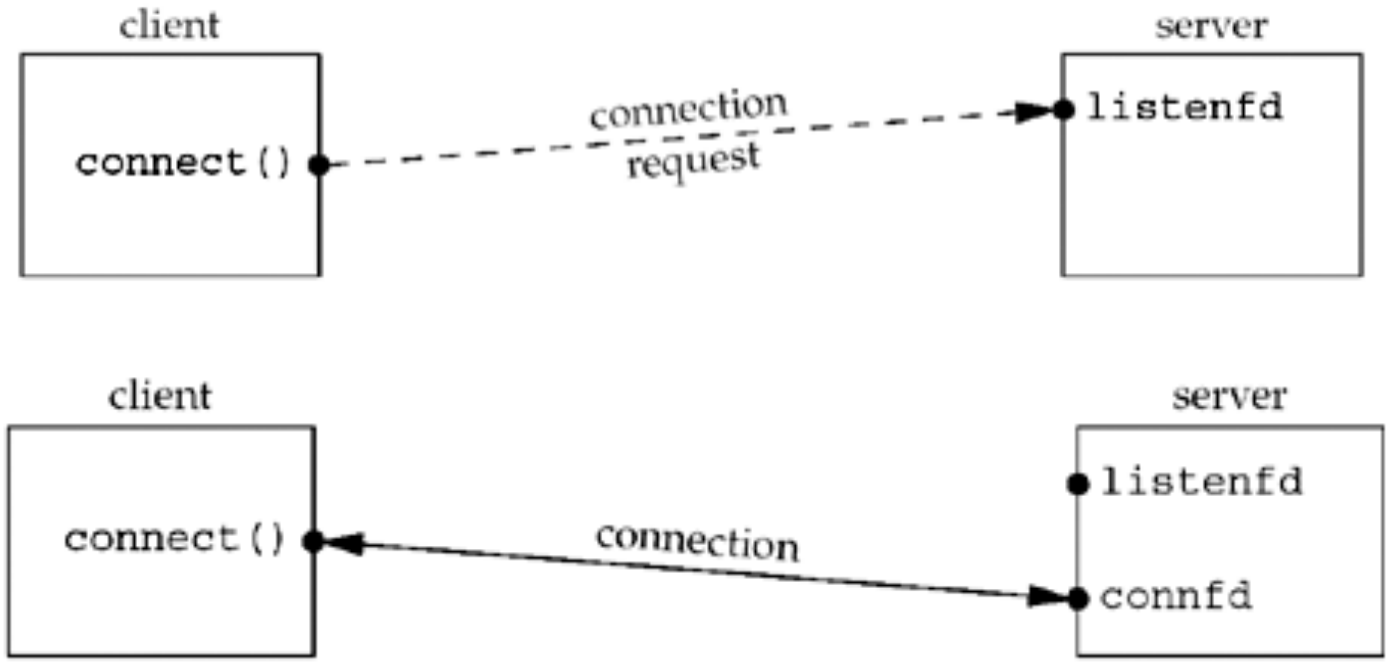
```
pid_t pid;
int listenfd, connfd;
listenfd = socket( ... );
/* fill in sockaddr_in{} with server's well-known port */
bind(listenfd, ... );
listen(listenfd, 5);
for ( ; ; ) {
    connfd = accept (listenfd, ... ); /* probably blocks */

    if( (pid = fork()) == 0) {

        close(listenfd); /* child closes listening socket */
        doit(connfd); /* process the request */
        close(connfd); /* done with this client */
        exit(0); /* child terminates */
    }
    close(connfd); /* parent closes connected socket */
}
```

Fork a child process

Processing of the child process

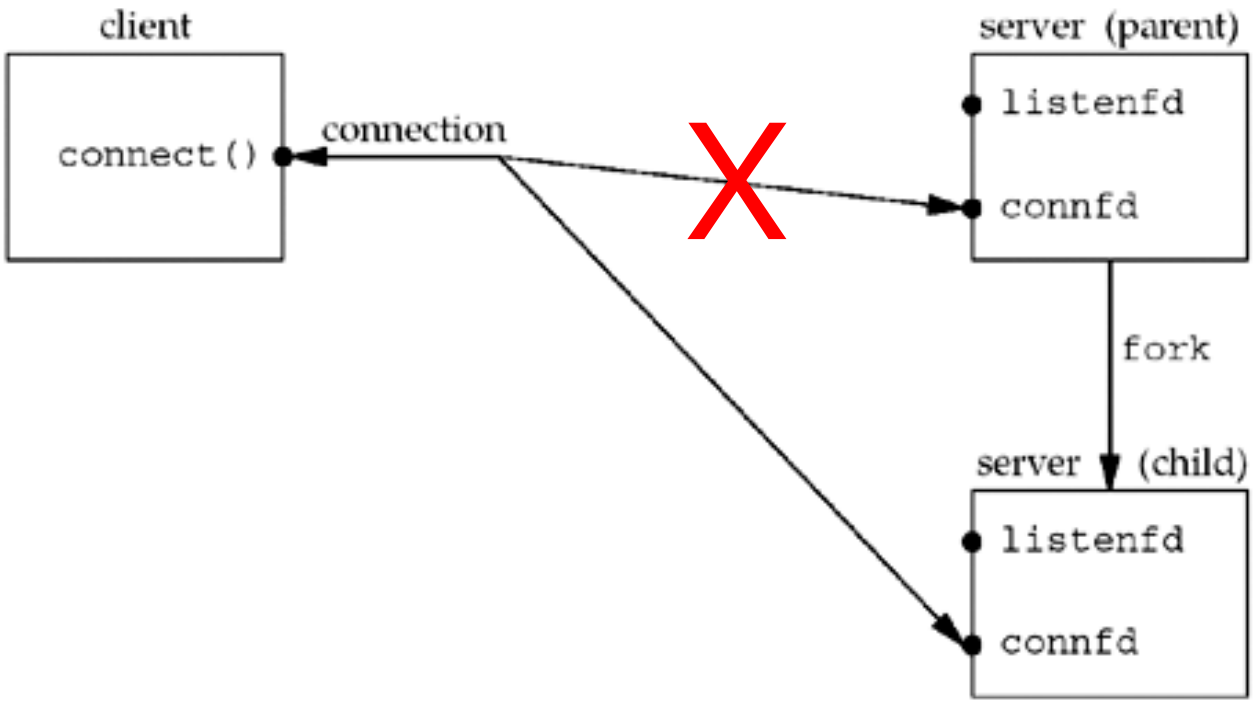
Iterative server vs forking server

- Iterative server



Iterative server vs forking server

- forking server



Normal Termination

- We use forking server, when a child process ends, it sends the *SIGCHLD* signal to the parent
  - Information about the child process is still maintained in “process table” in order to allow its parent to read the child exit status afterward.
- If *SIGCHLD* is ignored
  - Child process will not running but still consumes system resources (in process table)
  - → it is in zombie state
- We need to handle *SIGCHLD* signal
  - Make the child process to wait until the parent read its exit status
  - once the exit status is read via “wait” system call, the zombie's entry is removed from the process table

Signaling

- A signal is a notification to a process that an event has occurred.
- Signals are sometimes called software interrupts.
- Signals usually occur asynchronously.
  - A process doesn't know ahead when a signal will occur.
- Signals can be sent
  - By one process to another process (or to itself)
  - By the kernel to a process

Signaling

- Typing certain key combinations at the controlling terminal of a running process causes the system to send it certain signals:
  - Ctrl-C sends an INT signal ("interrupt", SIGINT)
  - Ctrl-Z sends a TSTP signal ("terminal stop", SIGTSTP)
  - Ctrl-\ sends a QUIT signal (SIGQUIT)
    - by default, this causes the process to terminate and dump core.
- SIGHUP is sent to a process when its controlling terminal is closed (a **hangup**).
- SIGTERM is sent to a process to request its **termination**.
  - Unlike the SIGKILL signal, it can be caught and interpreted or ignored by the process.

Handle a signal using **sigaction**

- We set/change the disposition of a signal by calling the sigaction function
  - **int sigaction(int** *signum*, **const struct sigaction** ***act*, **struct sigaction** ****oldact*);**
  - struct sigaction {

```
        void     (*sa_handler)(int);
        void     (*sa_sigaction)(int, siginfo_t *, void *);
        sigset_t   sa_mask;
        int        sa_flags;
        void     (*sa_restorer)(void);
    };
```

Handle a signal using **sigaction**

- Three choices for the disposition:
  - We can provide in **sigaction** a function that is called whenever a specific signal occurs (sa_handler)

    Prototype: void handler (int signo);
  - We can ignore a signal by setting sa_handler to *SIG_IGN*
  - We can set the default disposition for a signal by setting sa_handler to *SIG_DFL*.

Example

```
#include <signal.h>
#include <stdio.h>

void
termination_handler (int signum)
{
  struct temp_file *p;
  FILE* f=fopen("log.txt","a");
  fprintf(f, "SIGINT:%d\t SIGTSTP:%d\t SIGTERM:%d\n", SIGINT, SIGTSTP,SIGTERM);
  fprintf(f,"In termination handler Signal:%d\n", signum);
  fclose(f);
}

int
main (void)
{
  printf("Begin of program\n");
  struct sigaction new_action, old_action;

  /* Set up the structure to specify the new action. */
  new_action.sa_handler = termination_handler;
  sigemptyset (&new_action.sa_mask);
  new_action.sa_flags = 0;

  sigaction (SIGINT, &new_action, &old_action);  // Ctrl+C for generating the signal
  sigaction (SIGTSTP, &new_action, &old_action); // Ctrl+Z for generating the signal
  sigaction (SIGTERM, &new_action, &old_action); // kill -15 pid for generating the signal
 while (1) {}
}
```

Handle a signal using **signal function**

- An alternative to use sigaction.
- <signal.h>
- void (*signal(int sig, void (*func)(int)))(int)
  - Sig: signal to be handled
  - **func:** a pointer to a function that handle the signal

Example

```c
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <signal.h>

void sighandler(int);

int main()
{
   signal(SIGINT, sighandler);

   while(1)
   {
      printf("Going to sleep for a second...\n");
      sleep(1);
   }
   return(0);
}

void sighandler(int signum)
{
   printf("Caught signal %d, coming out...\n", signum);
   exit(1);
}
```

Handling SIGCHLD Signals

- The purpose of the zombie state is to maintain information about the child in "process table" for the parent to fetch at some later time.
- This information includes the process ID of the child, its termination status, and information on the resource utilization of the child (CPU time, memory, etc.).
- They take up space in the kernel and eventually we can run out of processes

→Whenever we *fork* children, parent must *wait* (read exit status) for them to prevent them from becoming zombies

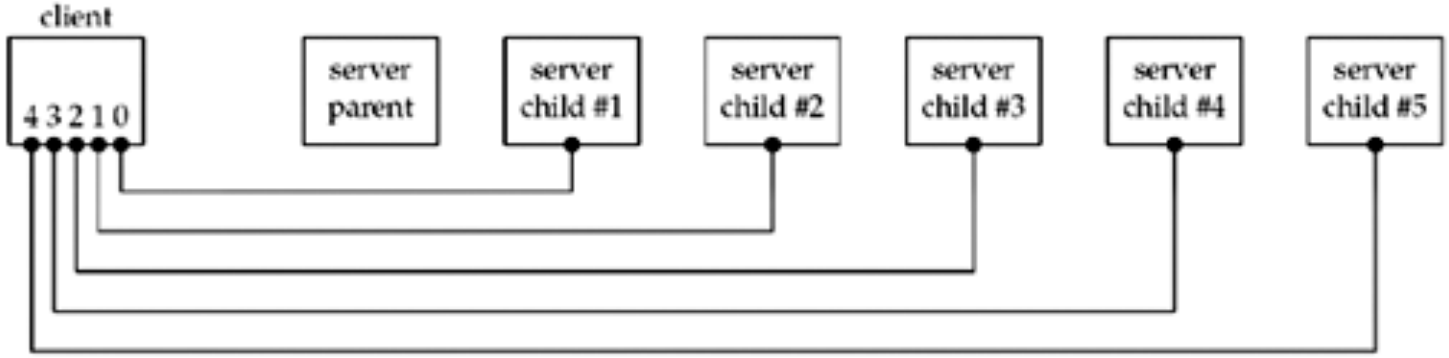
- →establish a signal handler to catch *SIGCHLD*, and within the handler, we call *wait*
- →Add into server: *signal (SIGCHLD, handler);*

wait() and waitpid()

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

pid_t wait (int *statloc);
pid_t waitpid (pid_t pid, int *statloc, int options);
```

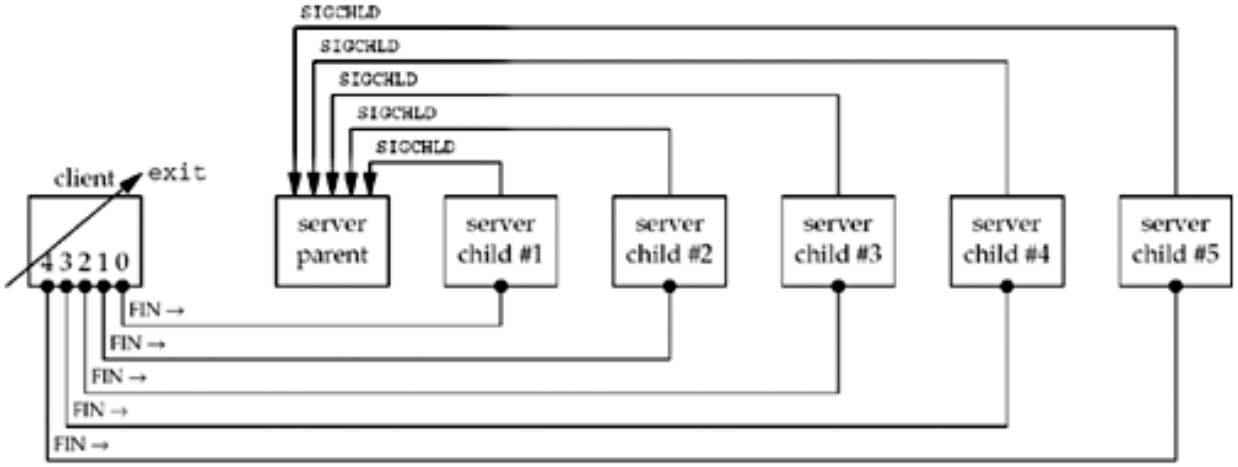
- Wait for the status change of a process.
- Use to handle the terminated child
- Both return two values:
  - The return value of the function
    - is the process ID of the terminated child if OK
    - 0 or -1 if error
  - The termination status of the child (an integer) is returned through the *statloc* pointer.

Difference between wait() and waitpid()



- Create 5 connections from a client to a forking server
- When the client terminates, all open descriptors are closed automatically by the kernel
  - → five connections ended simultaneous

Difference between wait() and waitpid() (2)



- → five SIGCHILD is sent
- The signal is handled once with wait in the server
  - → four children are zombies
- It can happen when many users connect to a server
- → we have to use *waitpid()*

waitpid()

pid_t waitpid (pid_t pid, int *statloc, int options);

- pid < 0: wait for status change of any child process whose process group ID is equal to the absolute value of *pid*.
- **pid = -1**: wait for for status change of any child process.
- pid = 0: wait for any child process whose process group ID is equal to that of the calling process
- pid > 0: wait for the child whose process ID is equal to the value of *pid*
- Without option **WNOHANG, waitpid** blocks until the status change
- With option **WNOHANG, waitpid** returns immediately
- Return
  - Pid of the child whose state has changed
  - with option **WNOHANG**, return 0 if the specified process has not changed status.

Forking server

```
pid_t pid;
int listenfd, connfd;
listenfd = socket( ... );
bind(listenfd, ... );
listen(listenfd, 5);

for ( ; ; ) {
   connfd = accept (listenfd, ... ); /* probably blocks */
   if( (pid = fork()) == 0) {
      close(listenfd); /* child closes listening socket */
      doit(connfd); /* process the request */
      close(connfd); /* done with this client */
      exit(0); /* child terminates */
   }
   signal(SIGCHLD,sig_chld);
   close(connfd); /* parent closes connected socket */
}
```

sig_chld: SIGCHLD handler

```
void sig_chld(int signo)
{
  pid_t pid;
  int stat;
  while ( (pid = waitpid(-1, &stat, WNOHANG)) > 0)
        printf("child %d terminated\n", pid);
  return;
}
/*
WNOHANG: waitpid does not block
while loop: waitpid repeatedly until there is no child
  process change status, i.e until waitpid returns 0.
*/
```

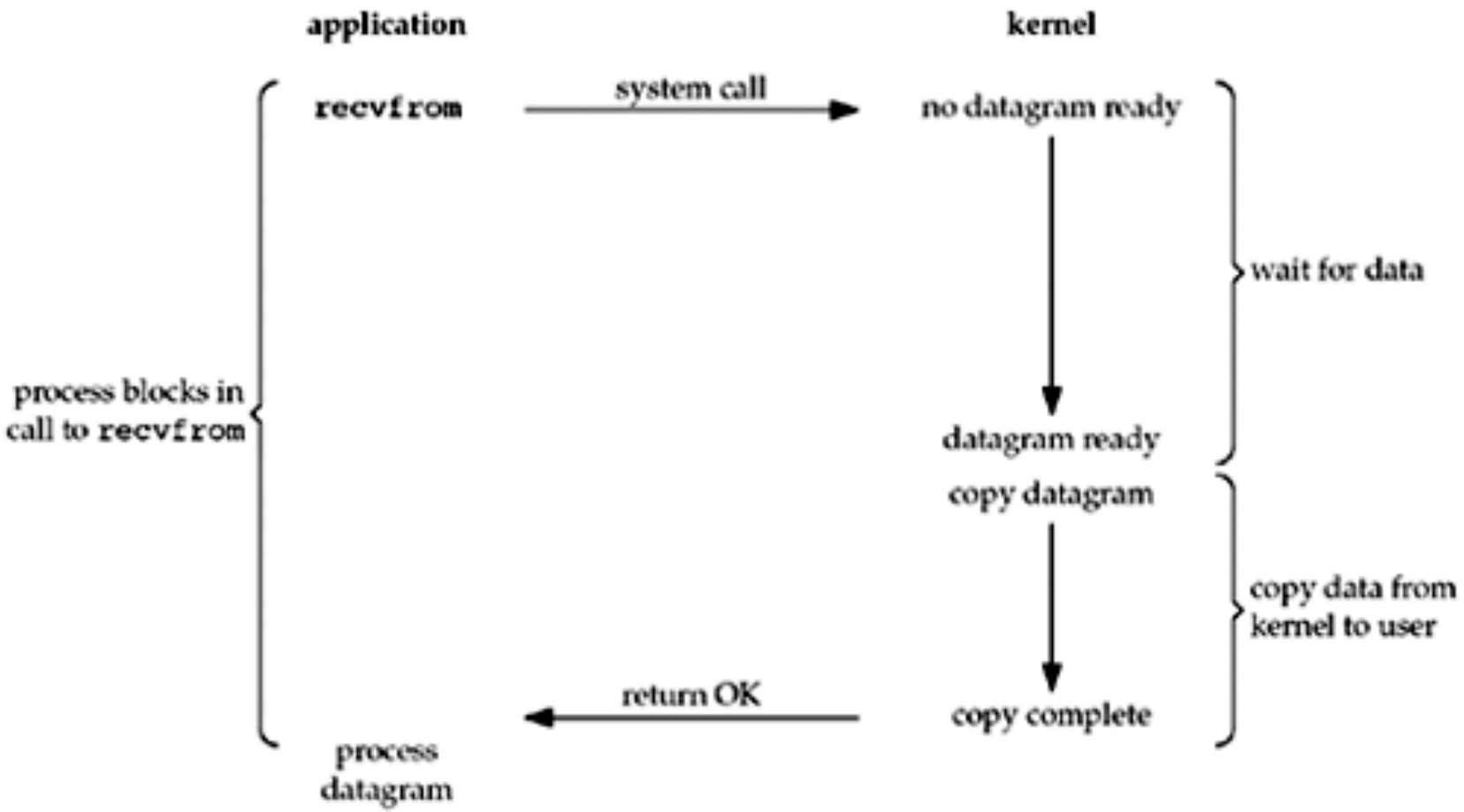
Exercise

- Make your TCP EchoServer be able to work with multiple client in the same time
  - Use forking process
  - Handle the SIGCHLD signal in server.

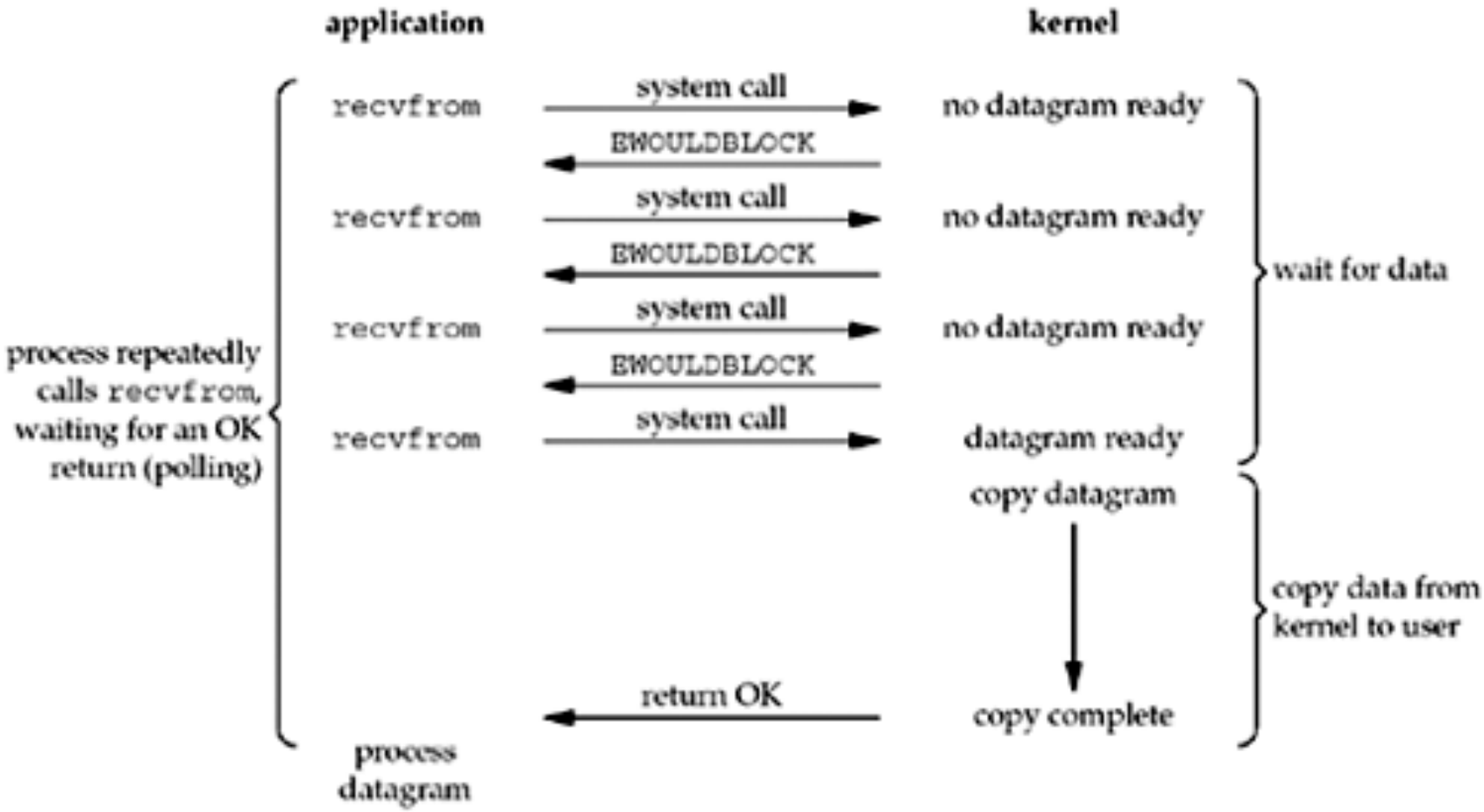
I/O Models

- The basic differences in the five I/O models that are available
  - blocking I/O
  - nonblocking I/O
  - I/O multiplexing (select and poll)
  - signal driven I/O (SIGIO)
  - asynchronous I/O (the POSIX aio_functions)

Blocking I/O Model



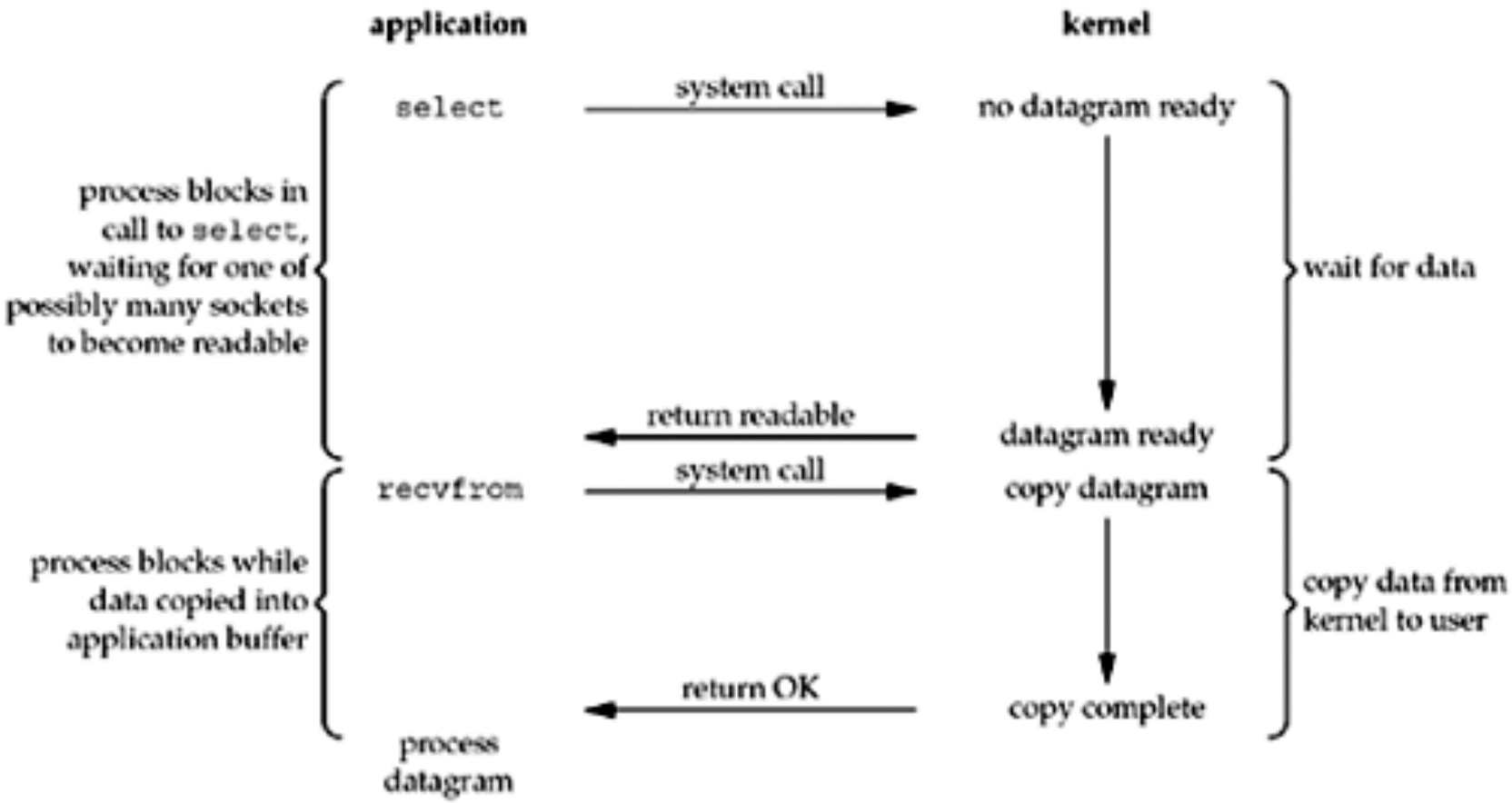
Non-blocking I/O Model



Non-blocking I/O model

- Need to set socket to non-blocking type.
- When we call recvfrom() there is no data and the system return immediately with error EWOULDBLOCK
- Otherwise it returns OK
- We can poll to read data
  - Loop with recv(), recvfrom()

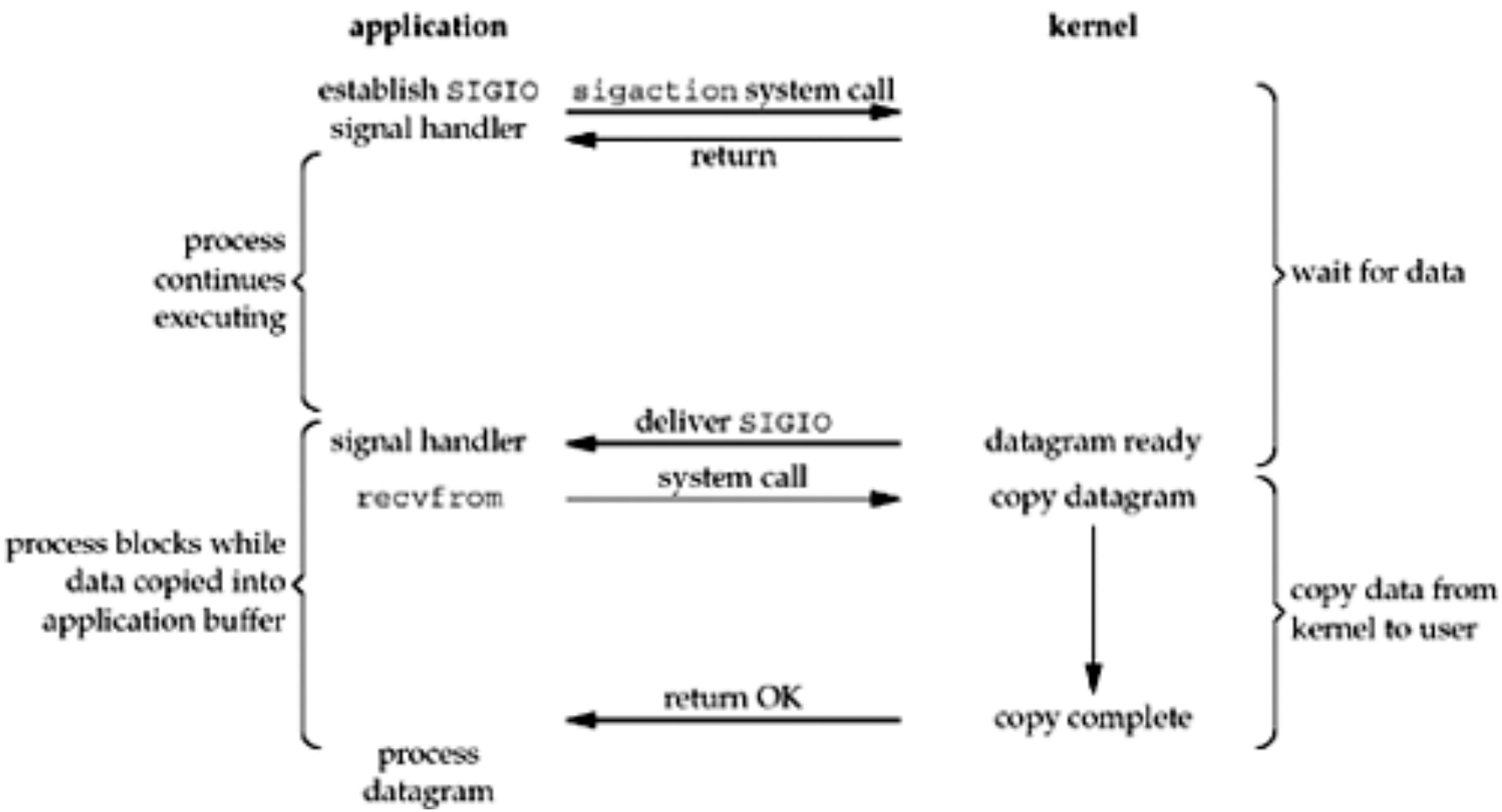
I/O Multiplexing Model



I/O Multiplexing Model

- Use select() to wait for data from several sockets
  - It is a blocking function.
- When data is ready in one socket then select returns
- We can then use recvfrom() to read from the chosen socket.

Signal-Driven I/O Model



Signal-Driven I/O Model

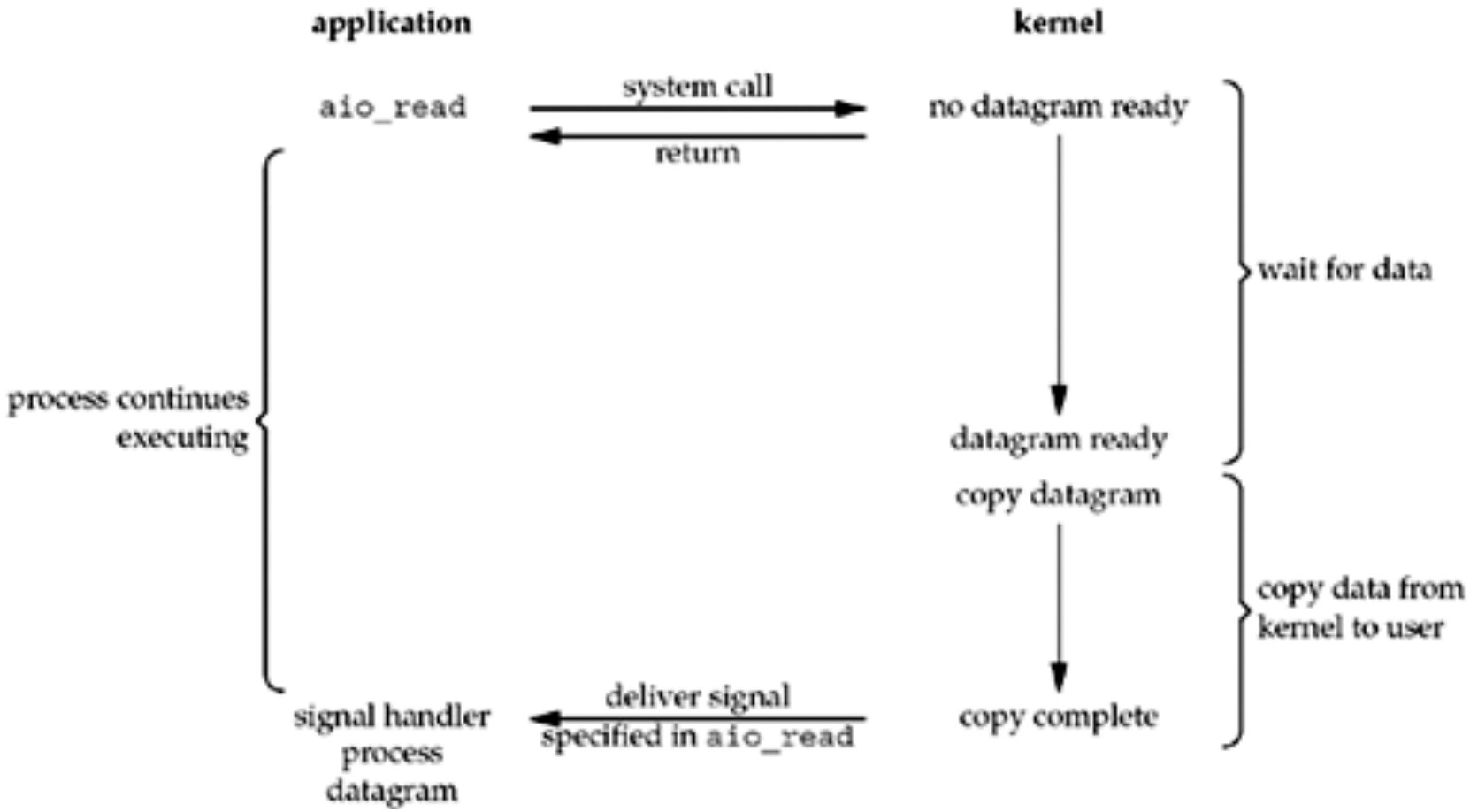
- Set socket to Signal-Driven I/O mode.
- When data arrive, a SIGIO occurs.
- Solution:
  - Associate SIGIO with a signal handler
  - When SIGIO occurs read data by recvfrom().
  - → No blocking

Exercise

- Revise echoServer and the echoClient so that when there are only 2 clients:
  - When client 1 sends something to server, it repeats it to client 2
  - When client 2 sends something to server, it repeats it to client 1
  - One client can send a string to server anytime independently with reception. (similar to chat)

- Hint:
  - On the client side, associate SIGIO with a function which calls recvfrom() to receiving data uniquely when data arrives.

Asynchronous I/O Model



Asynchronous I/O Model (2)

- Call aio_read
  - POSIX asynchronous I/O functions begin with aio_
- Function asks kernel to start waiting for data and notifies when data is ready in buffer.
- Pass the kernel
  - the descriptor
  - buffer pointer
  - buffer size (the same three arguments for read)
  - buffer offset (similar to lseek)
  - how to notify us when the entire operation is complete
- This system call returns immediately
  - No-blocking while waiting for the I/O to complete.

Comparison of the I/O Models