系統程式設計
Systems Programming

鄭卜王教授
臺灣大學資訊工程系

Tei-Wei Kuo, Chi-Sheng Shih, Hao-Hua Chu, and Pu-Jen Cheng©2007
Department of Computer Science and Information Engineering
Graduate institute of Multimedia and Networking, National Taiwan University

Advanced I/O

- ✓ Nonblocking (Ch14.2)
- ✓ I/O Multiplexing (Ch14.5)
- ✓ Comparison between I/O Models
- ✓ Record/Byte-range Locking (Ch14.3)
- ✓ Network Programming (Ch16)

Blocking vs Nonblocking

- **Blocking**
  - Process is suspended until all bytes in the count field are read or written
- **Nonblocking**
  - The OS only reads or writes as many bytes as possible without suspending the process
  - "Slow system calls" are those that can block forever
    - Reads or writes on pipes, terminal devices, and network devices
  - Nonblocking I/O lets us issue an I/O operation, such as an open, read, or write, and not have it block forever.
  - Two ways to specify nonblocking I/O
    - open() or fcntl() with the O_NONBLOCK flag

Turn on one or more flags with fcntl()

val | flags: turn on flags
val &= ~flags: clear the flags

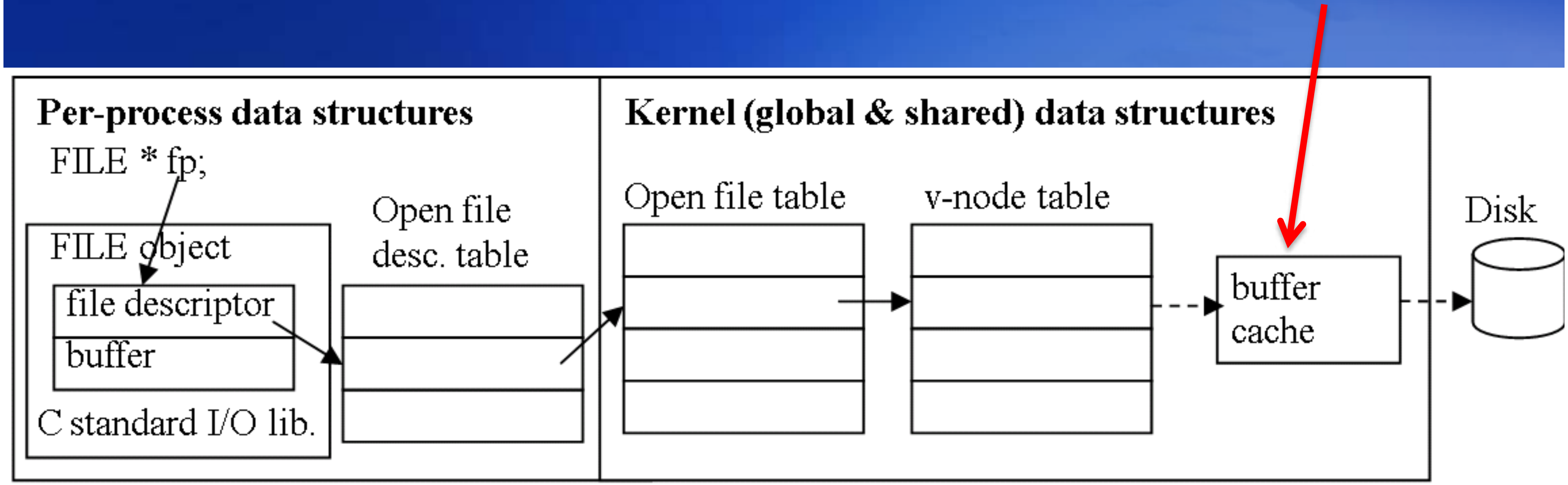
```
#include        <fcntl.h>
#include        "ourhdr.h"

void
set_fl(int fd, int flags) /* flags are file status flags to turn on */
{
        int             val;

        if ( (val = fcntl(fd, F_GETFL, 0)) < 0)
                err_sys("fcntl F_GETFL error");

        val |= flags;           /* turn on flags */

        if (fcntl(fd, F_SETFL, val) < 0)
                err_sys("fcntl F_SETFL error");
}
```

Example of Nonblocking

(char   buf[500000];)

```
int
main(void)
{
    int     ntowrite, nwrite;
    char    *ptr;

    ntowrite = read(STDIN_FILENO, buf, sizeof(buf));
    fprintf(stderr, "read %d bytes\n", ntowrite);

    set_fl(STDOUT_FILENO, O_NONBLOCK); /* set nonblocking */

    ptr = buf;
    while (ntowrite > 0) {
        errno = 0;
        nwrite = write(STDOUT_FILENO, ptr, ntowrite);
        fprintf(stderr, "nwrite = %d, errno = %d\n", nwrite, errno);

        if (nwrite > 0) {
            ptr += nwrite;
            ntowrite -= nwrite;
        }
    }

    clr_fl(STDOUT_FILENO, O_NONBLOCK); /* clear nonblocking */

    exit(0);
}
```

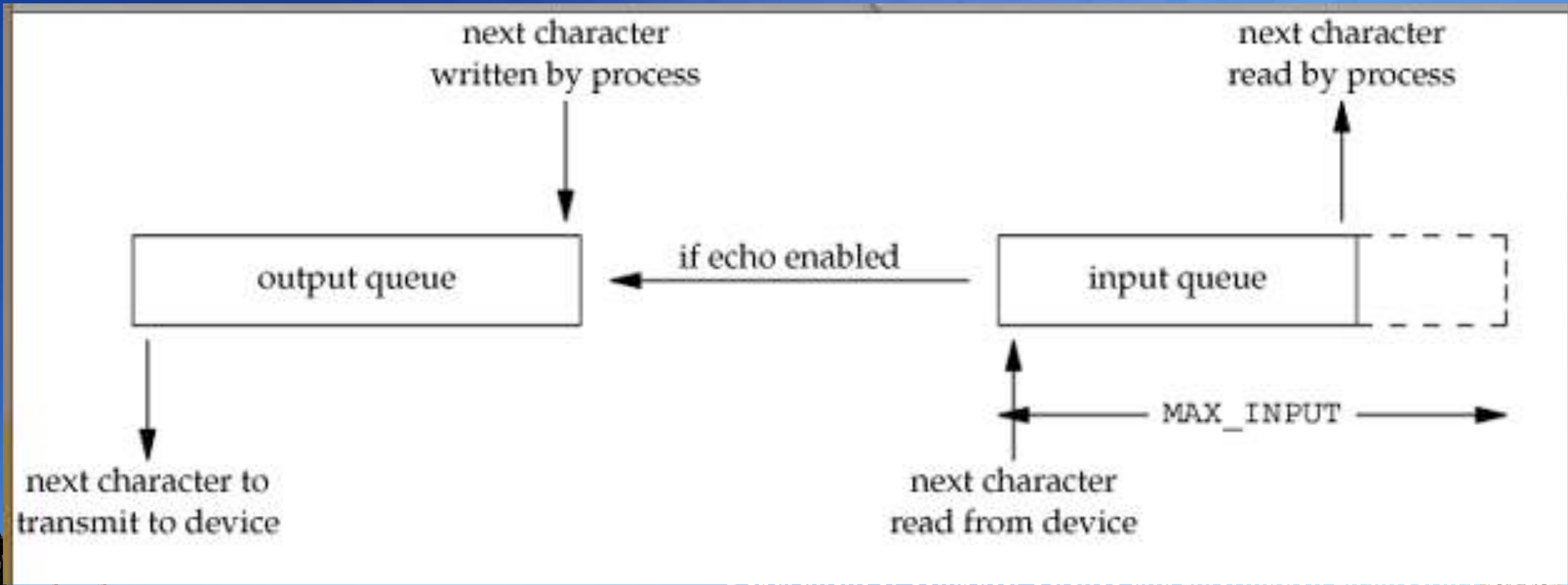
```
$ ls -l /etc/termcap                                        print file size
-rw-r--r-- 1 root      702559 Feb 23  2002 /etc/termcap
$ ./a.out < /etc/termcap > temp.file                        try a regular file first
read 500000 bytes
nwrite = 500000, errno = 0                                  a single write
$ ls -l temp.file                                           verify size of output file
-rw-rw-r-- 1 sar       500000 Jul   8 04:19 temp.file

$ ./a.out < /etc/termcap 2>stderr.out                           output to terminal
                                                                lots of output to terminal ...
$ cat stderr.out
read 500000 bytes
nwrite = 216041, errno = 0
nwrite = -1, errno = 11                                         1,497 of these errors
...
nwrite = 16015, errno = 0
nwrite = -1, errno = 11                                         1,856 of these errors
...
nwrite = 32081, errno = 0
nwrite = -1, errno = 11                                         1,654 of these errors
...
nwrite = 48002, errno = 0
nwrite = -1, errno = 11                                         1,460 of these errors
...
                                                                and so on ...
nwrite = 7949, errno = 0
```



Per-process data structures
FILE * fp;
FILE object
file descriptor
buffer
C standard I/O lib.
Open file
desc. table
Kernel (global & shared) data structures
Open file table
v-node table
buffer
cache
Disk

Terminal Device (Ch18.2)

- Each terminal device has an input queue and an output queue
- The shell redirects standard input to the terminal (in canonical mode), and each read returns at most one line.
- When the output queue starts to fill up
  - Blocking: put process to sleep until room is available.
  - Non-blocking: *polling* (busy waiting; a waste of CPU time on a multiuser system)

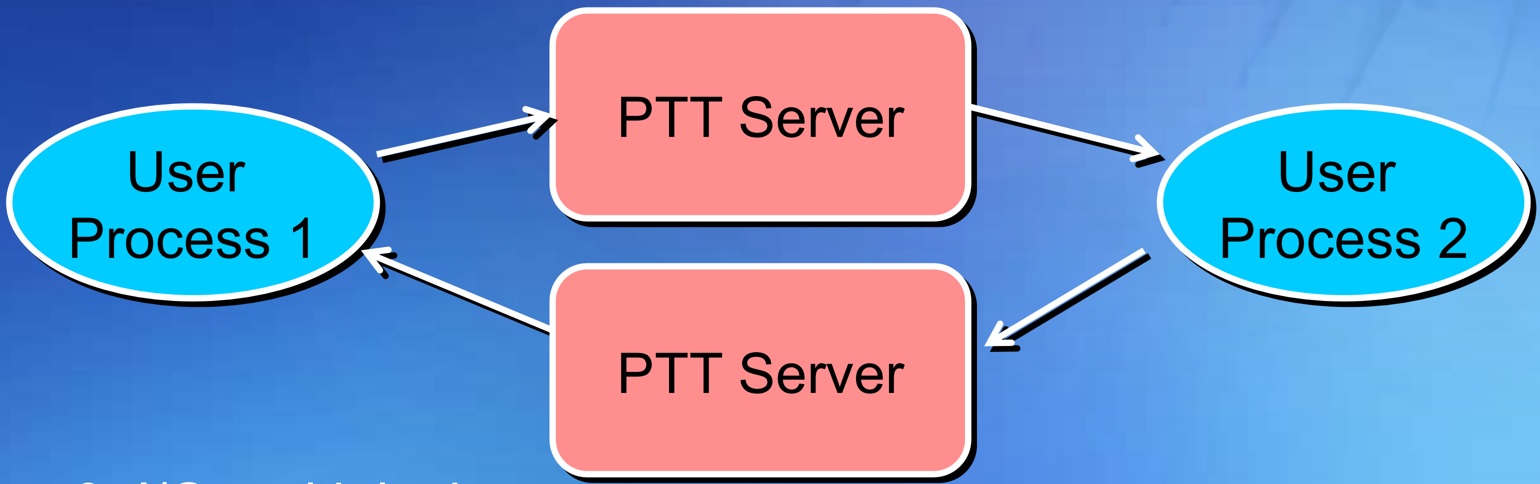


I/O Multiplexing

Problem of blocking I/O

Solution 1: non-blocking I/O, huge waste of cycles if data is not ready

Solution 2: multi-processes, too many processes, IPC overhead



Solution 3: I/O multiplexing

I/O Multiplexing

- When an application needs to handle multiple I/O descriptors at the same time
  - **E.g. file and socket descriptors, multiple socket descriptors**
- When I/O on any one descriptor can result in blocking

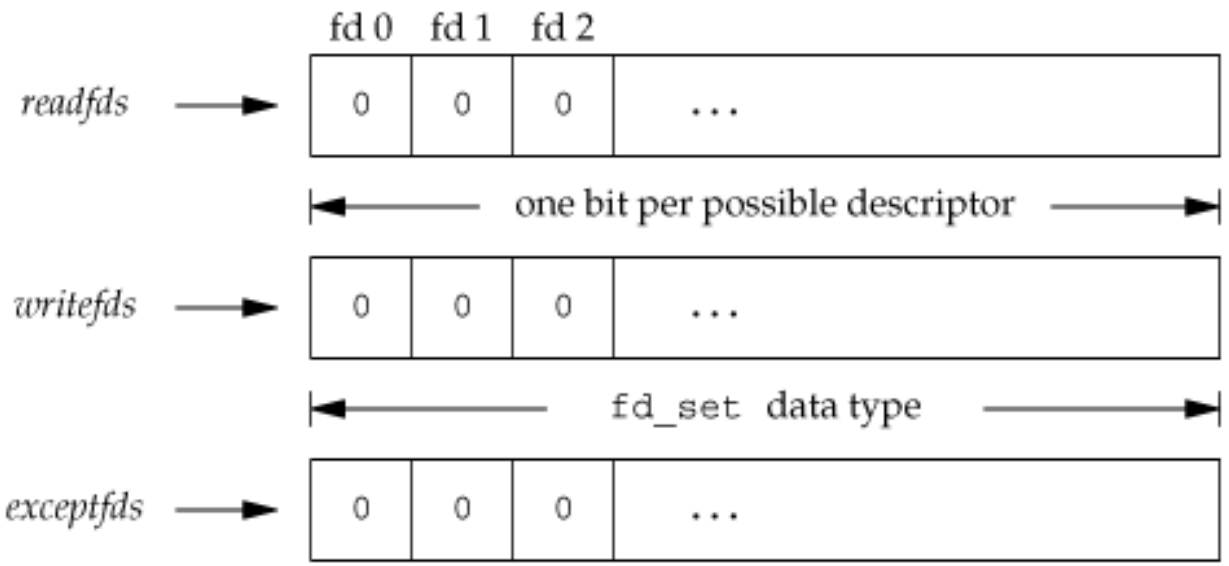
select

- **totalFds =**
  select(nfds, readfds, writefds, errorfds, timeout)
  - nfds: the range (0.. nfds-1) of file descriptors to check
    getdtablesize(): get file descriptor table size
  - readfds: bit map of filedes. Set the bit X to ask the kernel to check if filedes X is ready for reading. Kernel returns 1 if data can be read on the filedes.
  - writefds: bit map if filedes Y is ready for writing (same as read bitmap).
  - errorfds: bit map to check for errors.
  - timeout: how long to wait before un-suspending the process (microseconds)

    ```
    select ( 0, NULL, NULL, NULL, &timeval)
    ```

    Usual sleep(…) call has resolution of seconds.
  - totalFds = number of ready descriptors, 0 on timeout, negative for error

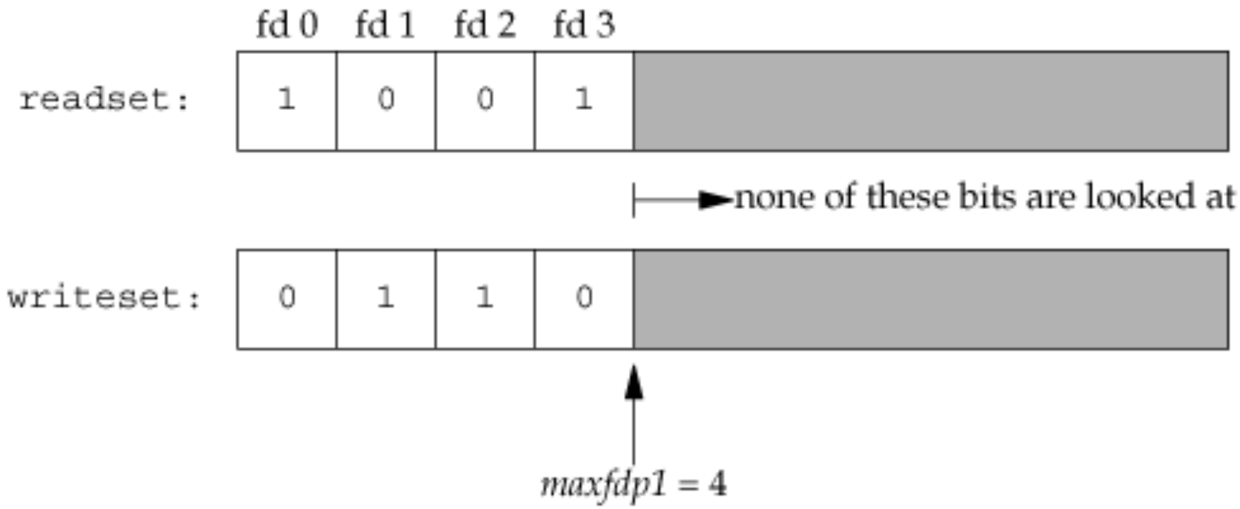
Specifying the read, write, and exception descriptors for `select`



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

int FD_ISSET(int fd, fd_set *fdset);
Returns: nonzero if fd is in set, 0 otherwise

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

```c
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);
```

Example descriptor sets for `select`



```c
int main()
{
        int    i;
        struct timeval  timeout;
        struct fd_set   master_set, working_set;
        char   buf[1024];

        FD_ZERO( &master_set );
        FD_SET( 0, &master_set );
        timeout.tv_sec = 5;
        timeout.tv_usec = 0;
        i = 0;

        while ( 1 )
        {
                memcpy( &working_set, &master_set, sizeof( master_set ) );
                select( 1, &working_set, NULL, NULL, &timeout );
                if ( FD_ISSET( 0, &working_set ) )
                {
                        fgets( buf, sizeof( buf ), stdin );
                        fputs( buf, stdout );
                }
                printf( "iteration: %d\n", i ++ );
        }
        return 0;
}
```

poll

- **totalFds =**
  poll(fdarray[], nfds, timeout)

  - struct pollfd {
        int fd;   // file descriptor to check, or <0 to ignore
        short events; // bit mask: each bit indicates an event of interest on fd
        short revents; // bit mask: each bit indicates an event that occurred on fd
    }
    events: POLLIN, POLLOUT, POLLHUP, POLLNVAL
  - nfds: the number of items in the fds array
  - timeout: how long to wait before un-suspending the process
    -1: wait forever, 0: don't wait: >0: wait (milliseconds)
  - totalFds = number of ready descriptors, 0 on timeout, negative for error

select vs. poll

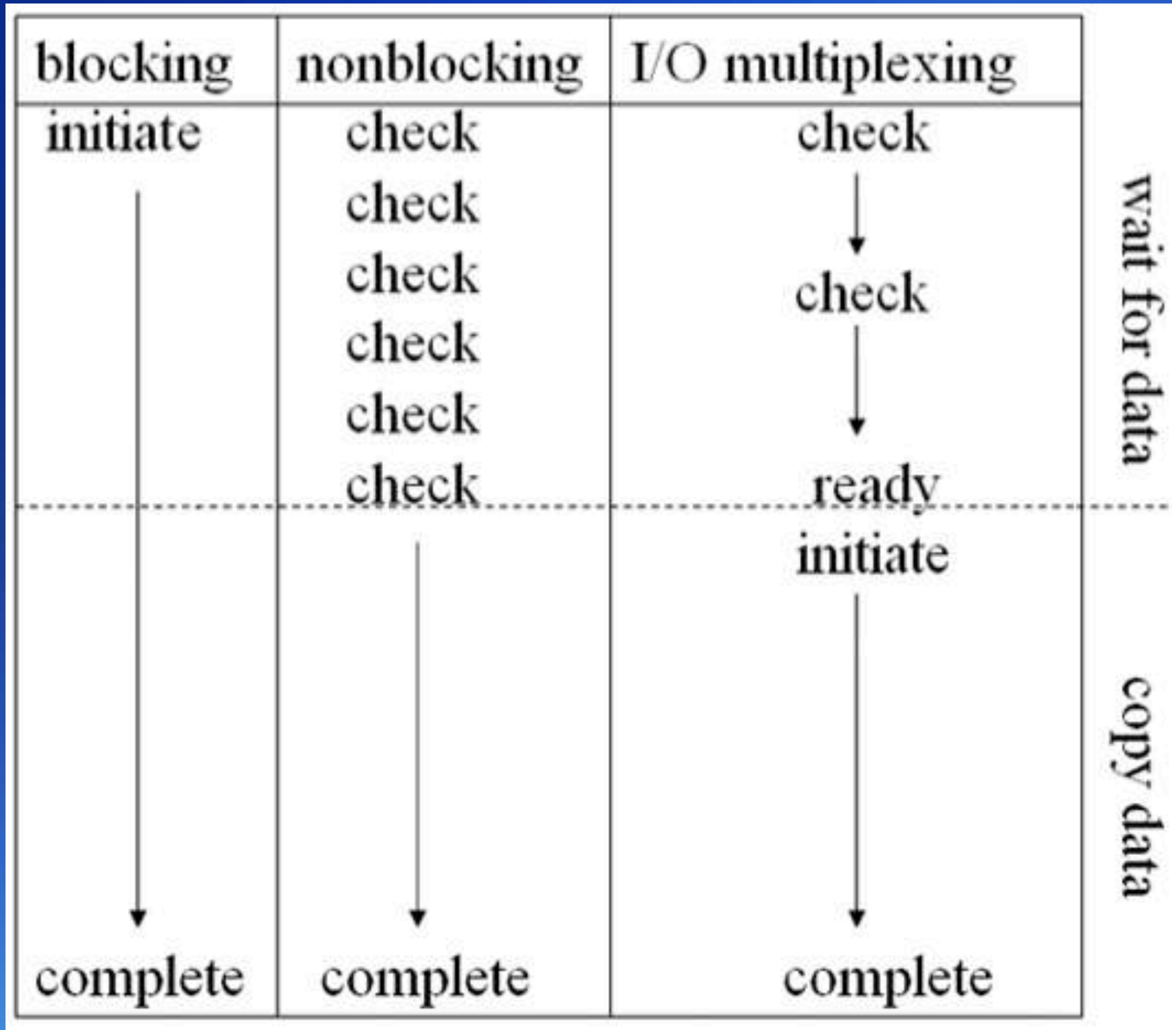
Comparison

- **select:** rewrites a bitmask that is no longer the one you set up
  poll: keeps reusing the same bitmask for input
- **select:** inefficient if only one fd is assigned a high number
  poll: combines multiple bitmasks into one
- **select:** only 3 event types (i.e., read, write, error)
  poll: more event types (e.g., POLLPRI: urgent data like
  out-of-band on TCP socket to read)
- **Both** need to check each fildesc. each time

Alternatives

- **pselect()** (nanoseconds, signal mask),
  epoll() (good for speed & scalability)

Comparison of I/O Models



Why File Lock

- **Exclusively access for avoiding data inconsistency**
  - More than one process may access a file at the same time
  - Without exclusively access, system may overwrite some of the old data while that old data is still being read.

```
lseek ( fd, 100, SEEK_SET);
read( fd, &int_var, 4 );
int_var = int_var – 200;
```

```
lseek( fd, 100, SEEK_SET);
read( fd, &int_var, 4 );
int_var = int_var – 200;
lseek( fd, 100, SEEK_SET);
write( fd, &int_var, 4 );
```

```
lseek( fd, 100, SEEK_SET);
write( fd, &int_var, 4, 100 );
```

  - Think about atomic operation (Ch3)

File Lock in UNIX

- **Exclusively locking a file prevents other processes to read/write the same file**
  - When a lock is granted, the process has the right to read/write the file
  - When a lock is denied, the process has to wait until the lock is released by the lock holder.
- **Three ways to lock**
  - flock(): lock an entire file
  - fcntl(): lock arbitrary byte ranges in a file **(introduced here)**
  - lockf(): built on top of fcntl()

Types of Lock

- **Shared read lock**
  - Any number of processes can have a shared read lock on a given byte
  - If there are one or more read locks on a byte, there can't be any write locks on that byte
- **Exclusive write lock**
  - Only one process can have an exclusive write lock on a given byte
  - If there is an exclusive write lock on a byte, there can't be any read locks on that byte

		Request for	
		read lock	write lock
Region currently has	no locks	OK	OK
	one or more read locks	OK	denied
	one write lock	denied	denied

fcntl Record Locking

int fcntl (int *filedes*, int *cmd*, ... /* struct flock *flockptr */);

- **Record locking command**
  - F_GETLK: Get the first lock which blocks the lock description pointed to by the third argument,
  - F_SETLK: Set or clear a file segment lock (like nonblocking)
  - F_SETLKW: This command shall be equivalent to F_SETLK except that if a shared or exclusive lock is blocked by other locks, the caller shall wait until the request can be satisfied (like blocking)
- **Lockptr**

```
struct flock {
    short   l_type;     /* F_RDLCK, F_WRLCK, or F_UNLCK */
    off_t   l_start;    /* offset in bytes, relative to l_whence */
    short   l_whence;   /* SEEK_SET, SEEK_CUR, or SEEK_END */
    off_t   l_len;      /* length, in bytes; 0 means lock to EOF */
    pid_t   l_pid;      /* returned with F_GETLK */
}
```

Parameters to lock files

- **Type of lock**
  - F_RDLCK: a shared read lock
  - F_WRLCK: an exclusive write lock
  - F_UNLCK: unlocking a region
- **The size of the region in bytes, l_len**
  - > 0: the lock extends to l_len bytes from l_whence and can go beyond the current size of the file.
  - = 0: the lock extends to the largest possible offset of the file

Department of Computer Science and Information Engineering
Graduate institute of Multimedia and Networking, National Taiwan University

Remarks

- **File access**
  - To obtain a read lock, the descriptor must be open for reading
  - To obtain a write lock, the descriptor must be open for writing
- **Non-atomic operations**
  - Testing for a lock with F_GETLK and then trying to obtain that lock with F_SETLK or F_SETLKW is not an atomic operation.
- **Implied inheritance and release of locks**
  - When a process terminates, all its locks are released
  - Whenever a descriptor is closed, any locks on the file referenced by that descriptor for that process are released.
  - Locks are never inherited across a fork
  - Locks are inherited across an exec except close_on_exec is set
- **The compatibility rule only applies to different processes**

```
#define read_lock(fd, offset, whence, len) \
            lock_reg((fd), F_SETLK, F_RDLCK, (offset), (whence), (len))

#define write_lock(fd, offset, whence, len) \
            lock_reg((fd), F_SETLK, F_WRLCK, (offset), (whence), (len))
```

```
int
lock_reg(int fd, int cmd, int type, off_t offset, int whence, off_t len)
{
    struct flock lock;

    lock.l_type = type;      /* F_RDLCK, F_WRLCK, F_UNLCK */
    lock.l_start = offset;   /* byte offset, relative to l_whence */
    lock.l_whence = whence;  /* SEEK_SET, SEEK_CUR, SEEK_END */
    lock.l_len = len;        /* #bytes (0 means to EOF) */

    return(fcntl(fd, cmd, &lock));
}
```

What happens to the locks on fd1?

- Example 1

```
fd1 = open(pathname, ...);
read_lock(fd1, ...);
fd2 = dup(fd1);
close(fd2);
```

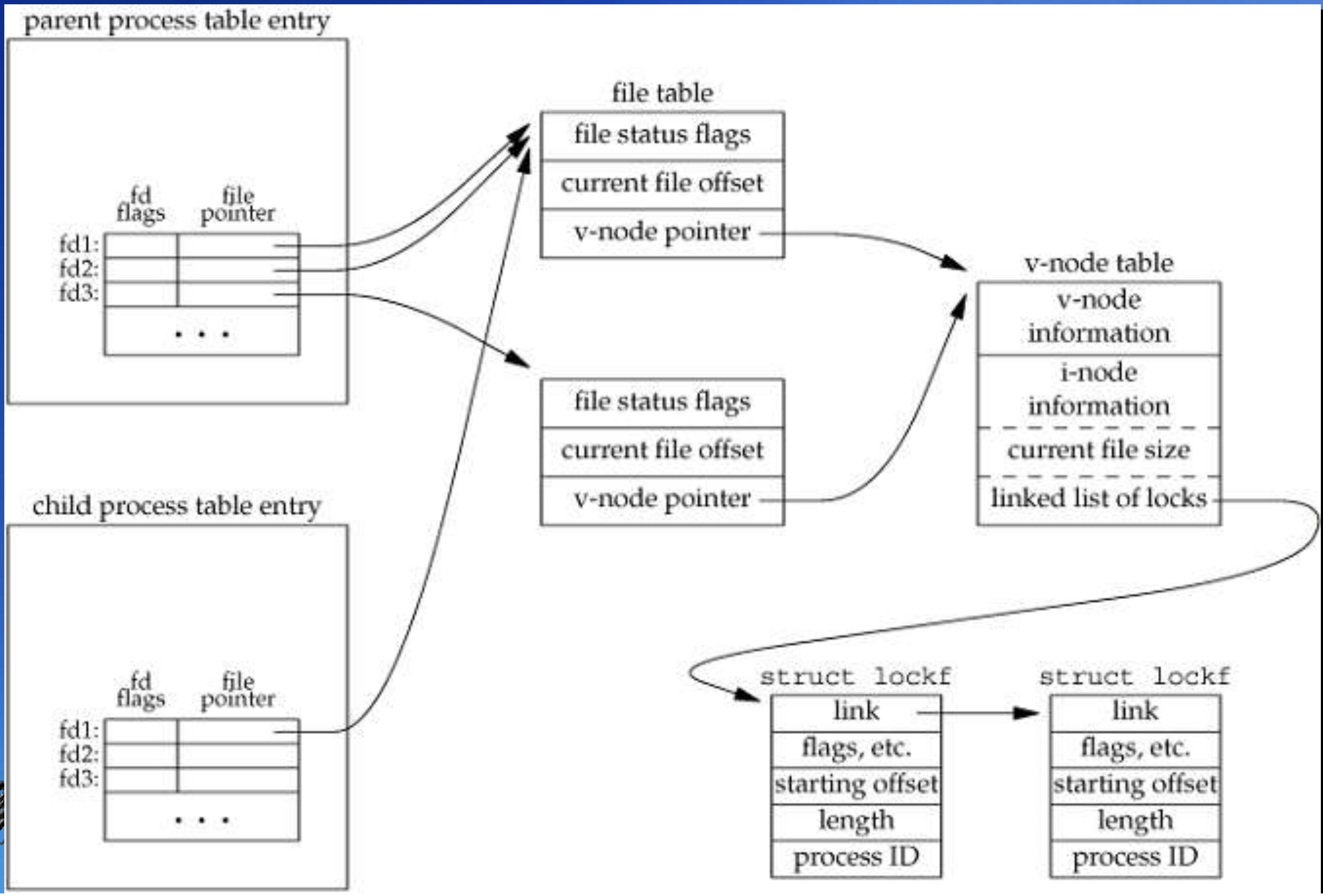
- Example 2

```
fd1 = open(pathname, ...);
read_lock(fd1, ...);
fd2 = open(pathname, ...)
close(fd2);
```

Tei-Wei Kuo, Chi-Sheng Shih, Hao-Hua Chu, and Pu-Jen Cheng©2007
Department of Computer Science and Information Engineering
Graduate institute of Multimedia and Networking, National Taiwan University

FreeBSD Implementation

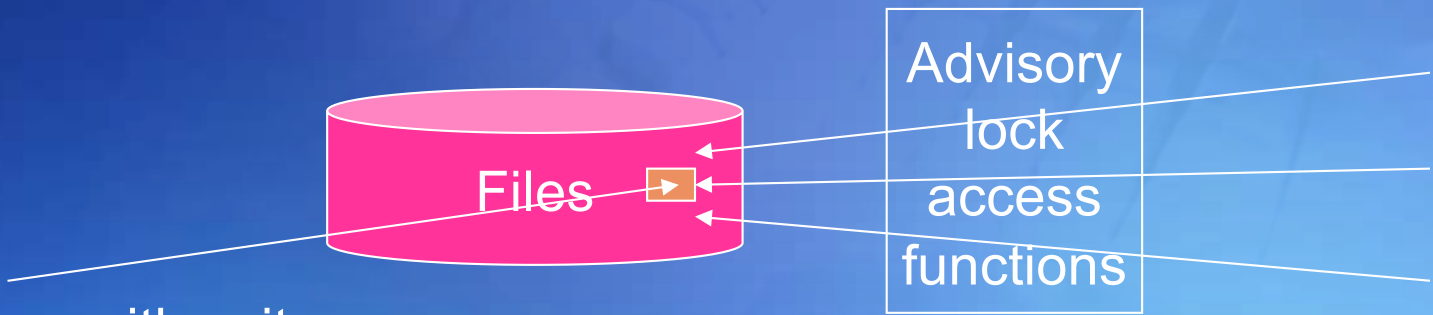
```
fdl = open(pathname, ...);
write_lock(fdl, 0, SEEK_SET, 1);    /* parent write locks byte 0 */
if ((pid = fork()) > 0) {           /* parent */
    fd2 = dup(fdl);
    fd3 = open(pathname, ...);
} else if (pid == 0) {
    read_lock(fdl, 1, SEEK_SET, 1); /* child read locks byte 1 */
}
```



heng©2007
Engineering
n University

Advisory vs. Mandatory lock

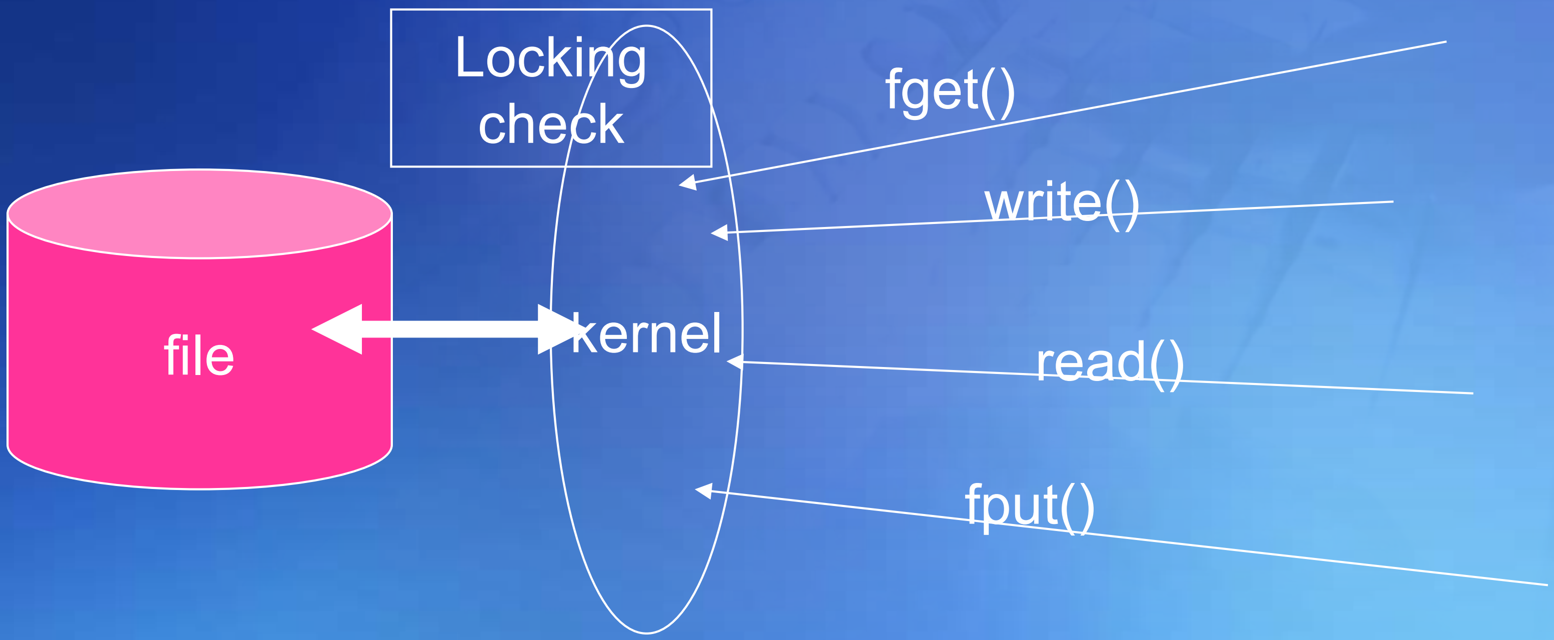
- **Advisory lock**
  - All the processes that access the shared files shall call the file lock function to access file
  - Any process bypasses the file lock functions may lead to inconsistent data
  - fcntl() provides advisory lock



A rude process with write permission of the file

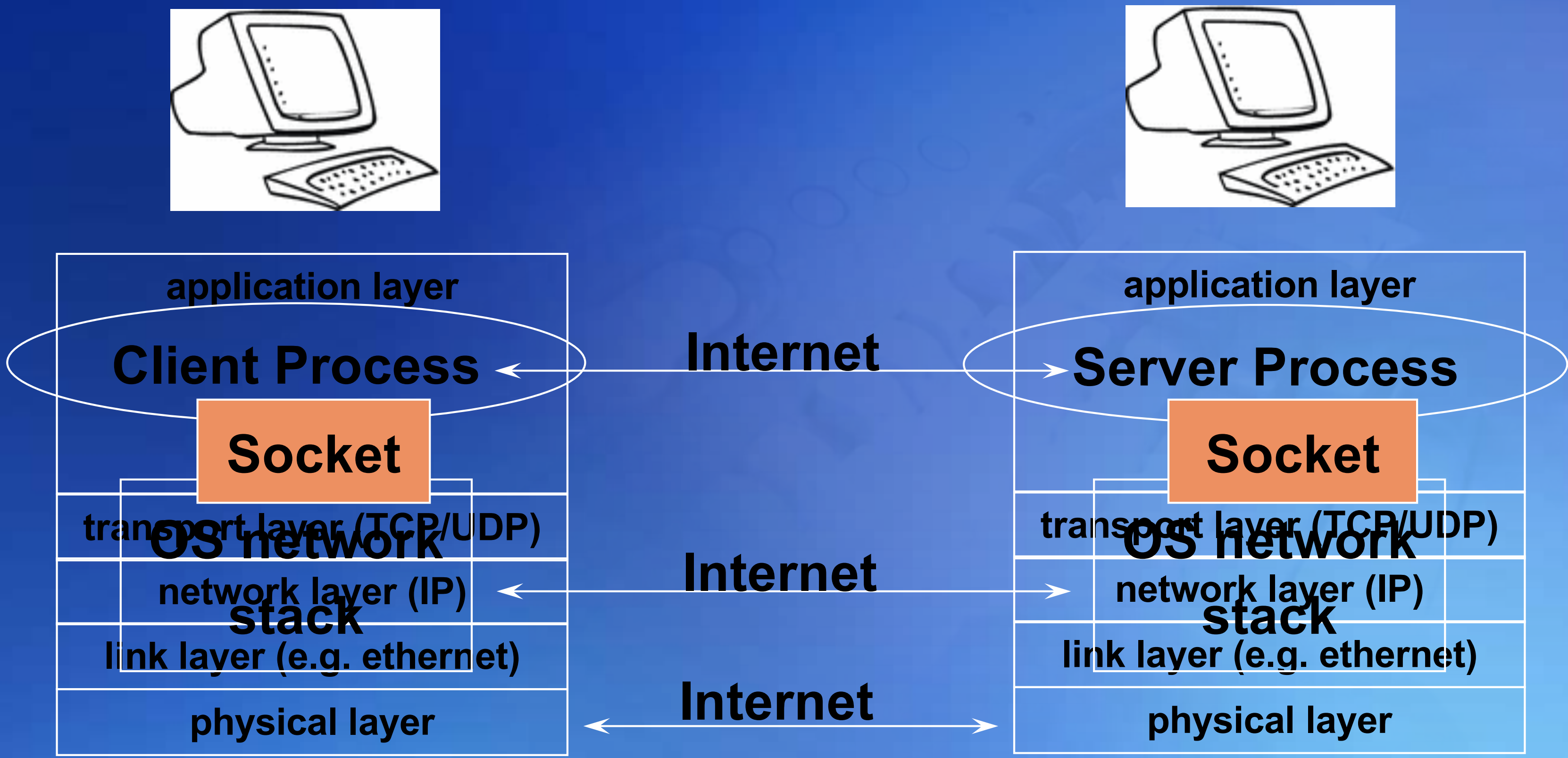
Mandatory lock

- The kernel checks every open, read, and write to verify that the calling process is not violating a lock.
- Mandatory lock is not part of SUS (Single UNIX Specification)
  - Linux 2.4.22 and Solaris 9 provide mandatory record locking
  - FreeBSD 5.2.1 and Mac OS X 10.3 do not.



- Mandatory locking specified in file permissions
- Mandatory locking involves system overhead

Networking (Ch16)
(Just for Programming Assignment 1)



- Sockets as means for inter-process communication (IPC)

IP

- **Point to point**
  - between machines
  - addressed using IP address
- **Message (packet) based**
- **Unreliable**
  - network failures
  - router buffers fill up
- **Dynamic routing**
  - order may be lost
- **Heterogeneous intermediate networks**
  - fragmentation

TCP & UDP

- **Both**
  - built on top of IP
  - addressed using port numbers
- **process to process (on UNIX platforms)**
- **TCP**
  - connection based, reliable, byte stream
  - used in: FTP, telnet, http, SMTP
- **UDP**
  - connectionless, unreliable, datagram (packet based)
  - used in: NFS, TFTP

Port number

- 16 bit integers
- Unique within a machine
- To connect need IP address + port no
- TCP
  - connection defined by

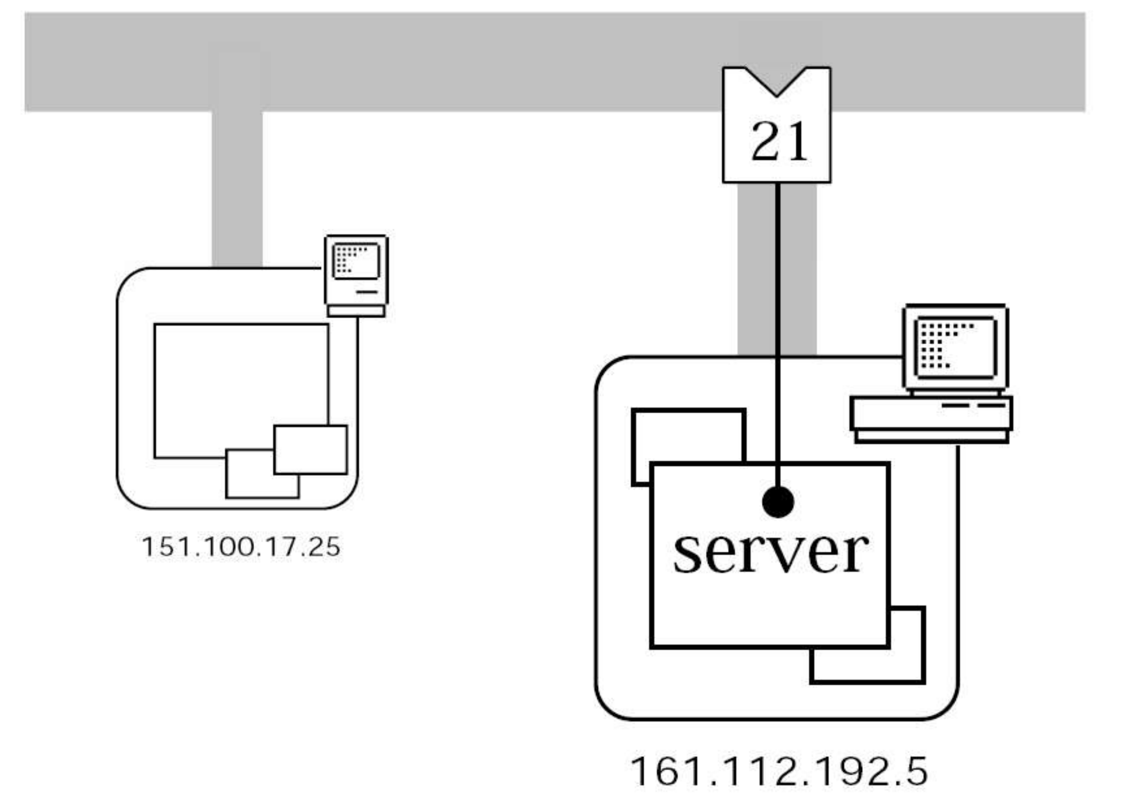
  IP address & port of server + IP address & port of client
- UNIX
  - port < 1023 – root only
  - used for authentication

Internet Connections (TCP/IP)

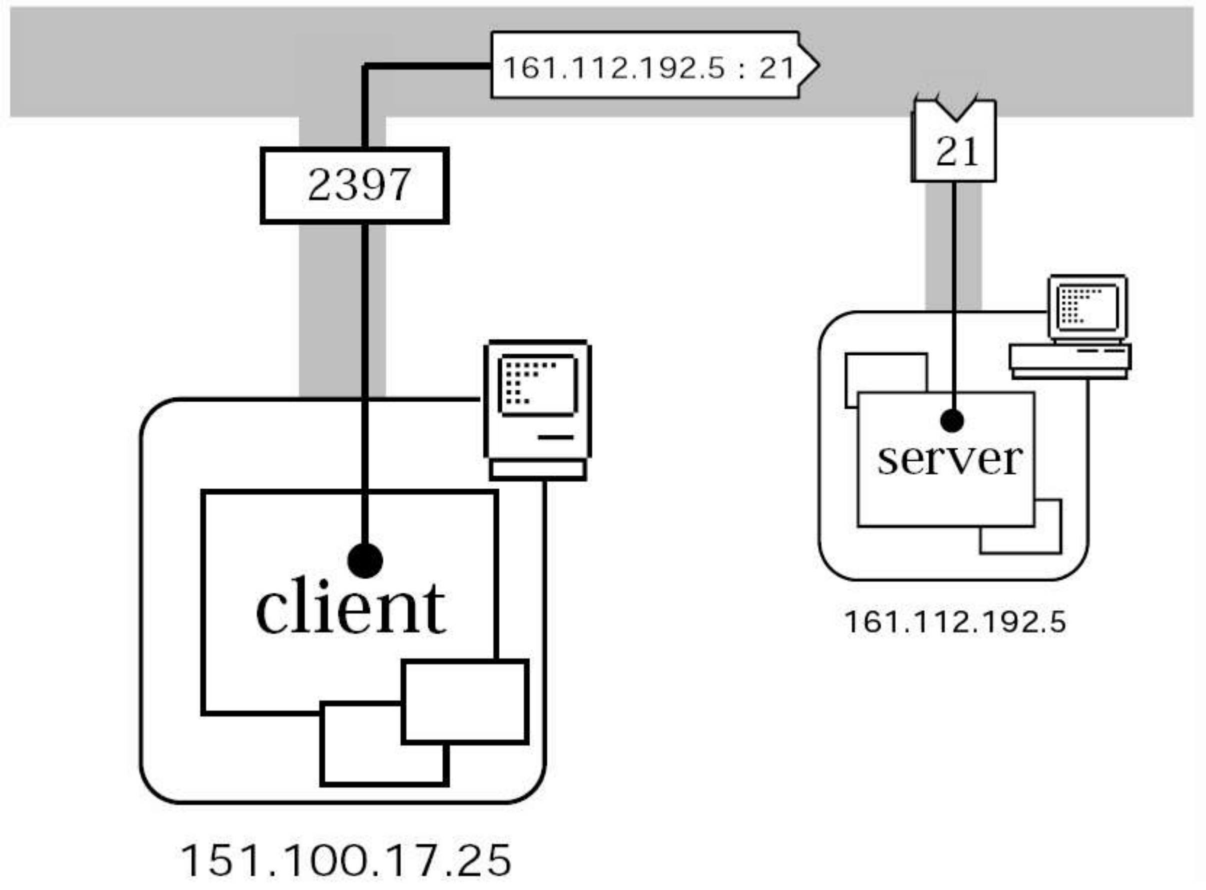
- **Address the machine on the network**
  - By IP address
- **Address the process**
  - By the “port”-number
- **The pair of *IP-address* + *port* – makes up a “*socket-address*”**

TCP Client/Server Model

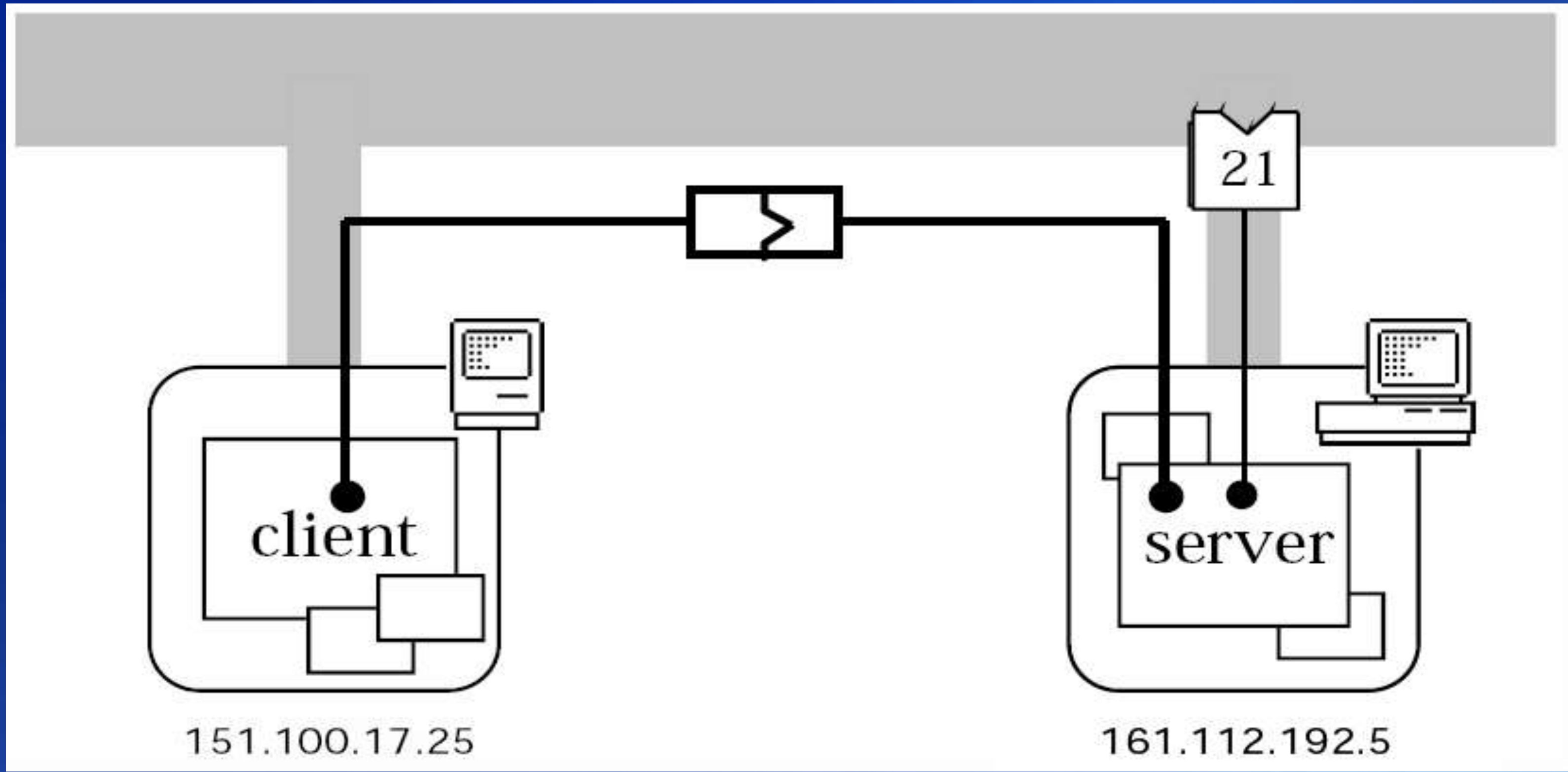
- **Client asks (*request*) – server provides (*response*)**



- **Server process opens on a port & waits for a client to connect**



- Client process usually on a different machine
- Client performs open on the port
  usually automatic (e.g., 2397)
- Network message → server machine
  requests connection



- Server side accepts and TCP connection established
- A bi-directional reliable byte-stream
- Connection identified by both host/port numbers

  e.g. <151.100.17.25:2397/161.112.192.5:2023>
- Server port is not consumed & can stay open for more connections

  like telephone call desk: one number many lines

Socket

- **What is a socket?**
  - To the kernel, a socket is an endpoint of communication.
  - To an application, a socket is a file descriptor that lets the application read/write from/to the network.
    - Remember: All Unix I/O devices, including networks, are modeled as files.
- **Clients and servers communicate by reading from and writing to socket descriptors.**
- **The main distinction between regular file I/O and socket I/O is how the application "opens" the socket descriptors.**

Notes on Socket Programming

- **Network Byte Ordering**
  - Network is big-endian, host may be big- or little-endian
  - Functions work on 16-bit (short) and 32-bit (long) values
  - htons() / htonl() : convert host byte order to network byte order
  - ntohs() / ntohl(): convert network byte order to host byte order
  - Use these to convert network addresses, ports, …

- **Structure Casts**
  - A lot of ‘structure casts’

Variable `x`: `0x01234567`
Address `&x`: `0x100`

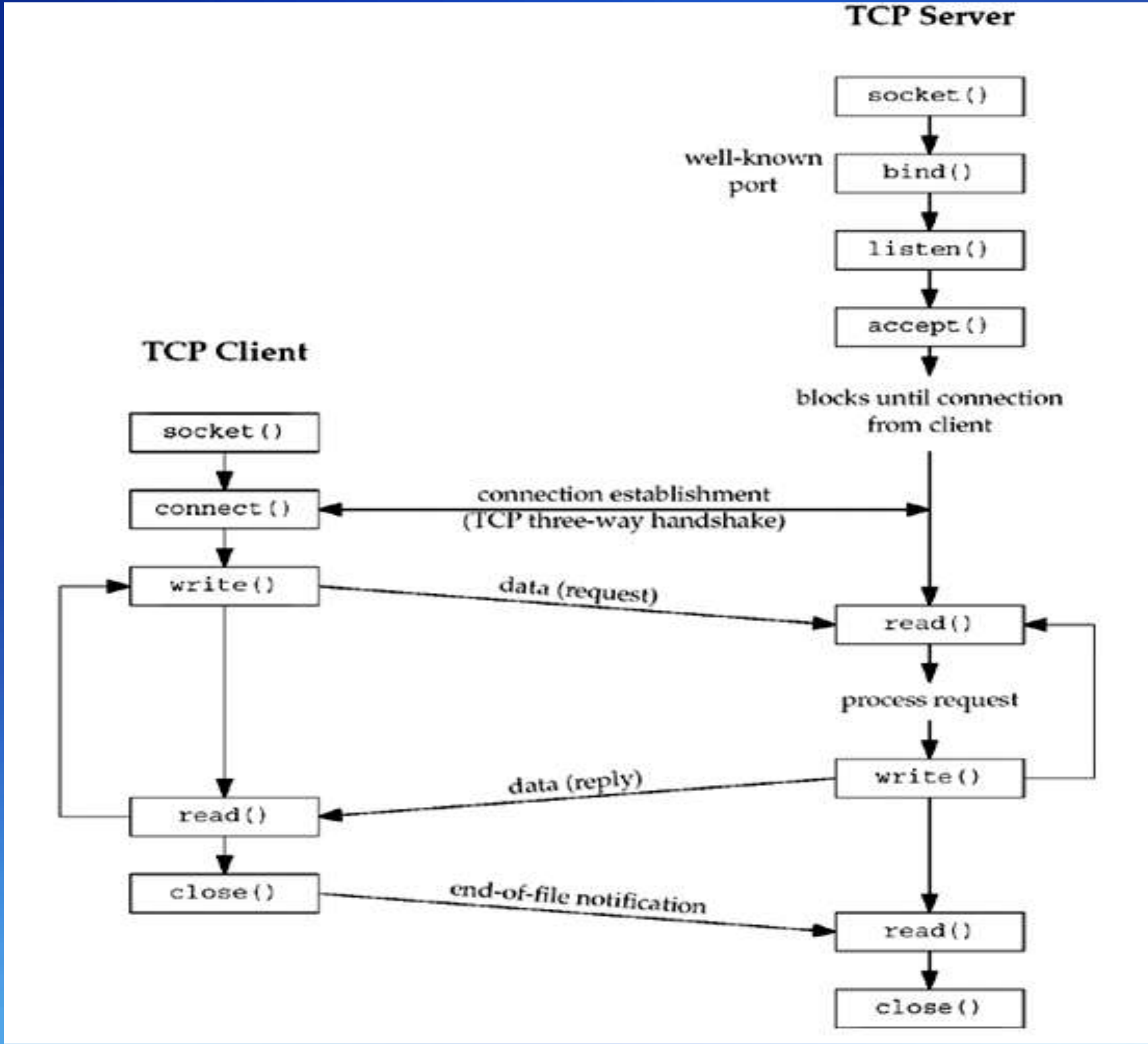
Big Endian

		0x100	0x101	0x102	0x103		
		01	23	45	67		

Little Endian

		0x100	0x101	0x102	0x103		
		67	45	23	01		

Programming for TCP Client/Server Model



Socket Primitives

- **SOCKET: create an endpoint for communication**

  int socket(int *domain*, int *type*, int *protocol*);
  - *domain* := AF_INET (IPv4 protocol)
  - *type* := (SOCK_DGRAM *or* SOCK_STREAM)
  - *protocol* := 0 (IPPROTO_UDP *or* IPPROTO_TCP)
  - *returned*: socket descriptor (*sockfd*), -1 is an error
- **BIND: bind a name/address to a socket**

  int bind(int *sockfd*, struct sockaddr **addr*, int *addrlen*);
  - *sockfd* - socket descriptor (returned from socket())
  - *addr:* socket address, struct sockaddr_in is used
  - *addrlen* := sizeof(struct sockaddr)

```
struct sockaddr_in  {
  unsigned short  sin_family;  /* address family (always AF_INET) */
   unsigned short  sin_port;    /* port num in network byte order */
    struct in_addr  sin_addr;    /* IP addr in network byte order */
   unsigned char   sin_zero[8]; /* pad to sizeof(struct sockaddr) */
};
```

- **LISTEN: listen for connections on a socket**

  int listen(int *sockfd*, int *backlog*);

  - *backlog*: how many connections we want to queue

- **ACCEPT: accept a connection on a socket. It extracts first request on the queue of pending connections for the listening socket, creates a new connected socket, and returns a new file descriptor referring to that socket. The original socket is unaffected. It blocks the caller until a connection is present.**

  int accept(int *sockfd*, void **addr*, int **addrlen*);

  - *addr*: here the socket-address of the caller will be written
  - *returned:* a new socket descriptor (for the temporal socket)

- **CONNECT: connect the socket referred to by the file descriptor *sockfd* to the address specified by *serv_addr***

  int connect(int *sockfd*, struct sockaddr **serv_addr*, int *addrlen*); //used by TCP client

  - parameters are same as for bind()

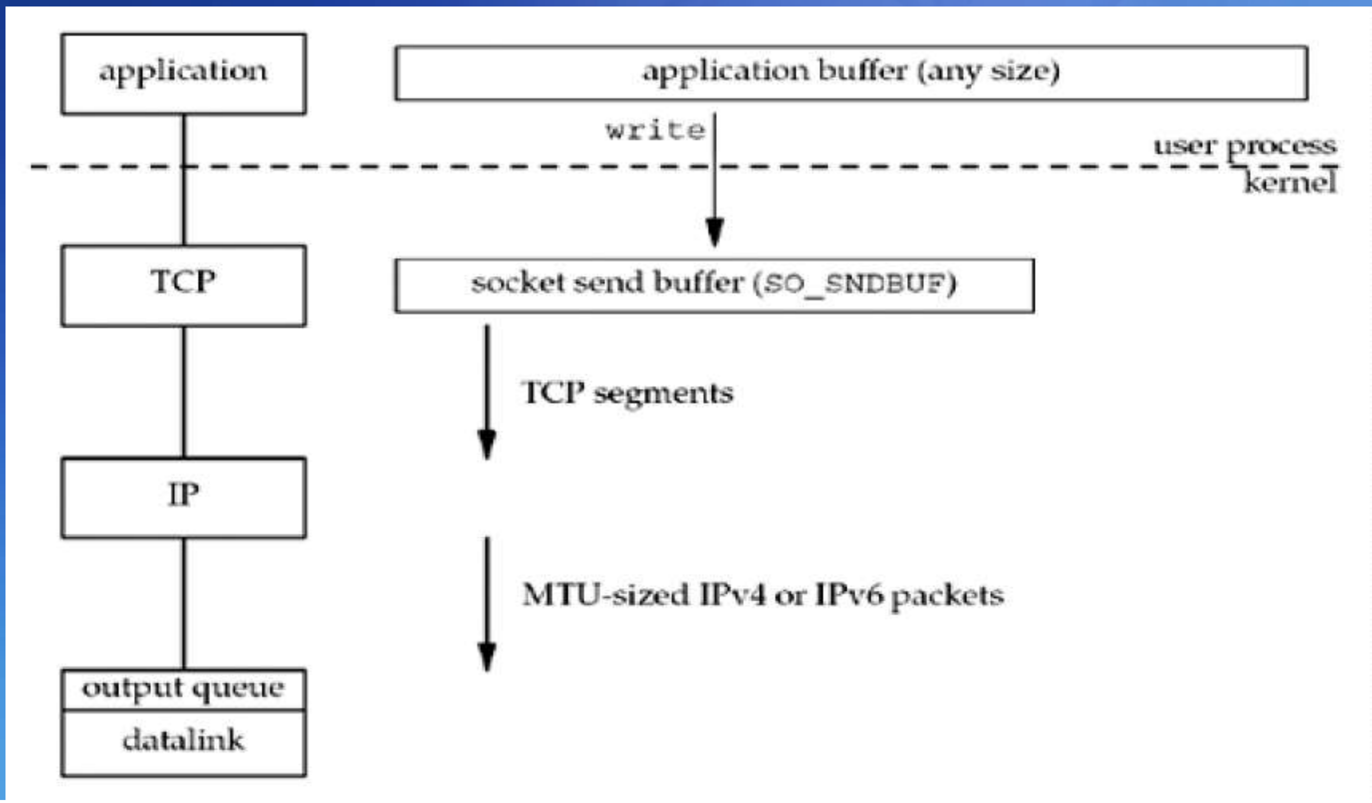
- **CLOSE: close (*socketfd*);**

Read & Write on Sockets

- **Bi-directional byte stream**
  - read and write to same file descriptor
  - possible to close one direction
  - special socket call, e.g., shutdown()
- **Reading may block**
  - reading from a file either:
    - (i) succeeds
    - (ii) gets end of file (ret = 0)
  - Reading from a socket waits until
    - (i) network data received (ret > 0)
    - (ii) connection closed (ret = 0)
    - (iii) network error (ret < 0)

Read & Write on Sockets

- **Writing may block**
  - writing to a socket may
    - (i) send to the network (ret > 0)
    - (ii) find connection is closed (ret = 0)
    - (iii) network error (ret < 0)
  - it may return instantly but may block if buffers are full



Programming Assignment 1

- **Concurrent reading & writing**
- **Client-server model**
- **Network inter-process communication**
  - Do not deal with socket
- **Goal**
  - I/O Multiplexing
  - File Locking