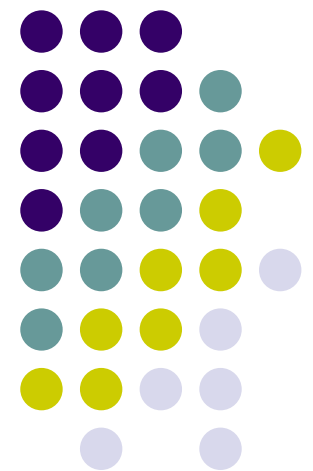


# CS241

# System Programming

---

Discussion Section 7  
March 13 – March 16





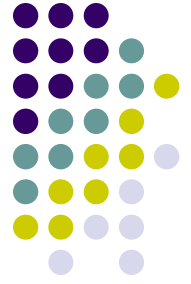
# Outline

- UNIX I/O
  - Basic Operations
    - `open, close, read, write`
    - `select, poll`
    - File Representation
- Serialization

# Terminology



- Peripheral device
  - Hardware accessed by a computer system
- Device Driver
  - OS modules that enable other programs to interact with a device through system calls
- UNIX provides uniform access to most devices
  - 5 functions: `open`, `close`, `read`, `write`, `ioctl`



# Review from class

- Read

```
ssize_t read(int fildes, void *buf, size_t nbyte);
```

- Write

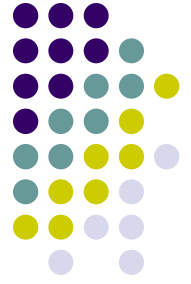
```
ssize_t write(int fildes, const void *buf, size_t nbyte);
```

- Open

```
int open(const char *path, int oflag, ...);
```

- Close

```
int close(int fildes);
```

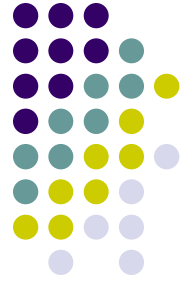


# Example

- Reading a specific number of bytes (Program 4.7)

```
ssize_t readblock(int fd, void *buf, size_t size) {
    char *bufp;
    size_t bytestoread;
    ssize_t bytesread;
    size_t totalbytes;

    for (bufp = buf, bytestoread = size, totalbytes = 0; bytestoread > 0;
         bufp += bytesread, bytestoread -= bytesread) {
        bytesread = read(fd, bufp, bytestoread);
        if ((bytesread == 0) && (totalbytes == 0))
            return 0;
        if (bytesread == 0) {
            errno = EINVAL;
            return -1;
        }
        if ((bytesread) == -1 && (errno != EINTR))
            return -1;
        if (bytesread == -1)
            bytesread = 0;
        totalbytes += bytesread;
    }
    return totalbytes;
}
```



# Example

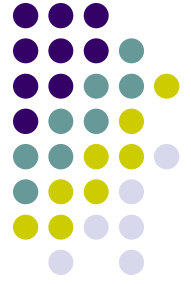
- A program to copy a file (Program 4.9)

```
#define READ_FLAGS O_RDONLY
#define WRITE_FLAGS (O_WRONLY | O_CREAT | O_EXCL)
#define WRITE_PERMS (S_IRUSR | S_IWUSR)

int main(int argc, char *argv[]) {
    if ((fromfd = open(argv[1], READ_FLAGS)) == -1) {
        perror("Failed to open input file");
        return 1;
    }

    if ((tofd = open(argv[2], WRITE_FLAGS, WRITE_PERMS)) == -1) {
        perror("Failed to create output file");
        return 1;
    }

    bytes = copyfile(fromfd, tofd);
    printf("%d bytes copied from %s to %s\n", bytes, argv[1], argv[2]);
    return 0;          /* the return closes the files */
}
```

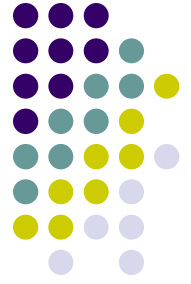


# Select

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

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

- Provides a method of monitoring file descriptors from a single process: for 3 conditions
  - `readfds` : whether read will not block
  - `writefds` : whether write will not block
  - `errorfds` : exceptions
- Returns the number of descriptors contained in the descriptor sets if successful. -1 with `errno` set if unsuccessful



# Select – 4 Macros

- Setting the corresponding bit

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

- Clearing the corresponding bit

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

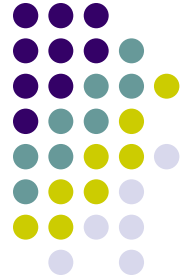
- Clearing all bits

```
void FD_ZERO(fd_set *fdset);
```

- Testing whether the corresponding bit is set

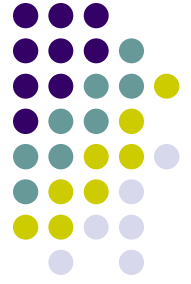
```
int FD_ISSET(int fd, fd_set *fdset);
```

# Example



- Monitoring File Descriptors (Program 4.14)

```
while (numnow > 0) {                                /* continue monitoring until all are done */
    FD_ZERO(&readset);                               /* set up the file descriptor mask */
    for (i = 0; i < numfds; i++)
        if (fd[i] >= 0) FD_SET(fd[i], &readset);
    numready = select(maxfd, &readset, NULL, NULL, NULL); /* which ready? */
    /* Error checking skipped */
    for (i = 0; (i < numfds) && (numready > 0); i++) { /* read and process */
        if (fd[i] == -1)                             /* this descriptor is done */
            continue;
        if (FD_ISSET(fd[i], &readset)) {             /* this descriptor is ready */
            bytesread = r_read(fd[i], buf, BUFSIZE);
            numready--;
            if (bytesread > 0)
                docommand(buf, bytesread);
            else { /* error occurred on this descriptor, close it */
                r_close(fd[i]);
                fd[i] = -1;
                numnow--;
            }
        }
    }
}
```

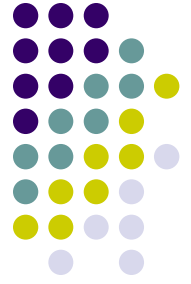


# Example

- Program 4.15

```
int waitfdtimed(int fd, struct timeval end) {
    fd_set readset;
    int retval;
    struct timeval timeout;

    FD_ZERO(&readset);
    FD_SET(fd, &readset);
    if (gettimeofday(&timeout, NULL) == -1) return -1;
    while (((retval = select(fd + 1, &readset, NULL, NULL, &timeout)) == -1)
        && (errno == EINTR)) {
        if (gettimeofday(&timeout, NULL) == -1) return -1;
        FD_ZERO(&readset);
        FD_SET(fd, &readset);
    }
    if (retval == 0) {
        errno = ETIME;
        return -1;
    }
    if (retval == -1) return -1;
    return 0;
}
```



# Poll

```
#include <poll.h>
```

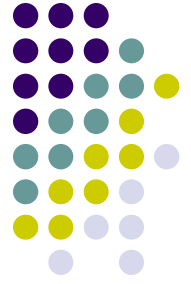
```
int poll(struct pollfd fds[], nfd_t nfd, int timeout);
```

- Provides a method of monitoring file descriptors
- Organizes the information by file descriptor: `struct pollfd`
  - Select: by the type of event
- Returns the number of descriptors that have events if successful. 0 if timeout, or -1 with `errno` set if unsuccessful



# Poll

- `struct pollfd`
  - File descriptor: `int fd;`
  - Requested Events: `short events;`
  - Returned Events: `short revents;`
- Event Flags: Table 4.2

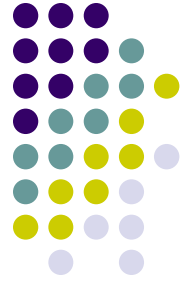


# Example

- Monitoring an array of file descriptors (Program 4.17)

```
if ((pollfd = (void *)calloc(numfds, sizeof(struct pollfd))) == NULL)
    return;
for (i = 0; i < numfds; i++) {
    (pollfd + i)->fd = *(fd + i);
    (pollfd + i)->events = POLLRDNORM;
}

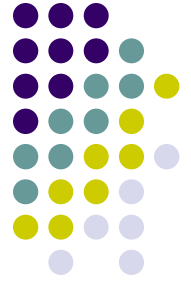
/* Continue monitoring until descriptors done */
while (numnow > 0) {
    numready = poll(pollfd, numfds, -1);
    if ((numready == -1) && (errno == EINTR))
        continue; /* poll interrupted by a signal, try again */
    else if (numready == -1)/* real poll error, can't continue */
        break;
```



# Example (continued)

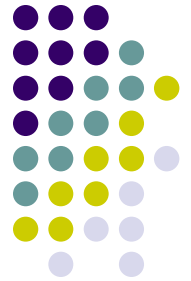
- Monitoring an array of file descriptors (Program 4.17)

```
for (i = 0; i < numfds && numready > 0; i++) {
    if ((pollfd + i)->revents) {
        if ((pollfd + i)->revents & (POLLRDNORM | POLLIN) ) {
            bytesread = r_read(fd[i], buf, BUFSIZE);
            numready--;
            if (bytesread > 0)
                docommand(buf, bytesread);    // some command to call on the data
            else
                bytesread = -1;                /* end of file */
        } else if ((pollfd + i)->revents & (POLLERR | POLLHUP))
            bytesread = -1;
        else
            /* descriptor not involved in this round */
            bytesread = 0;
        if (bytesread == -1) {                /* error occurred, remove descriptor */
            r_close(fd[i]);
            (pollfd + i)->fd = -1;
            numnow--;
        }
    }
}
}
```



# File Representation

- File Descriptor
  - Represents a file or device that is open
  - An `int`-type index into the file descriptor table of each process
  - Can refer to files, directories, blocks, sockets, pipes
- File Pointer
  - points to a data structure called a FILE structure in the user area of the process
  - Buffering is used

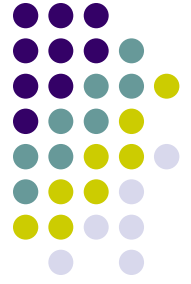


# Buffering

- How does the output appear when the following program executes? (Exercise 4.26)

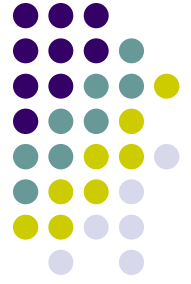
```
#include <stdio.h>

int main(void) {
    int i;
    fprintf(stdout, "a");
    scanf("%d", &i);
    fprintf(stderr, "a has been written\n");
    fprintf(stdout, "b");
    fprintf(stderr, "b has been written\n");
    fprintf(stdout, "\n");
    return 0;
}
```



# Serialization

- Process of converting an in-memory object into a linear sequence of bytes
  - e.g. converting an object to a byte-string for network transmission
- Involved in saving an object onto a storage medium (file, memory buffer, etc.)
- Antonym: Unserialization
  - Restoring the object from the serialized byte sequence



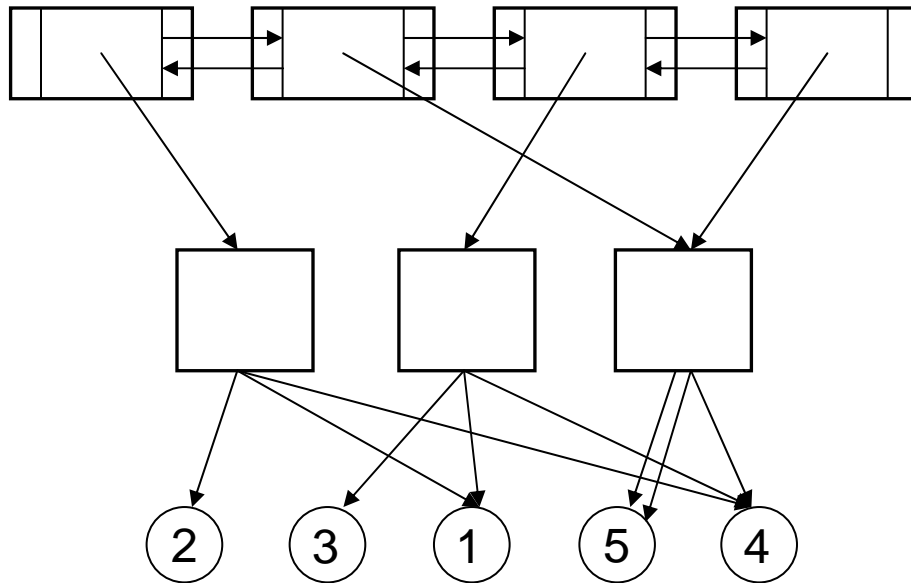
# Example

- Given two data structures:
  - Floats: A structure containing three float pointers
  - FloatList: A doubly linked list with the element at each node being a struct Floats pointer
- Tasks to do
  - Serialization: store the FloatList onto a char buffer
  - Unserialization: restore the FloatList from the char buffer
- Answer: Look at floatlist.{c,h} (given in MP3) and analyze them.



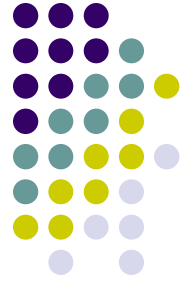
# Floatlist.c

- An illustration of given data structures



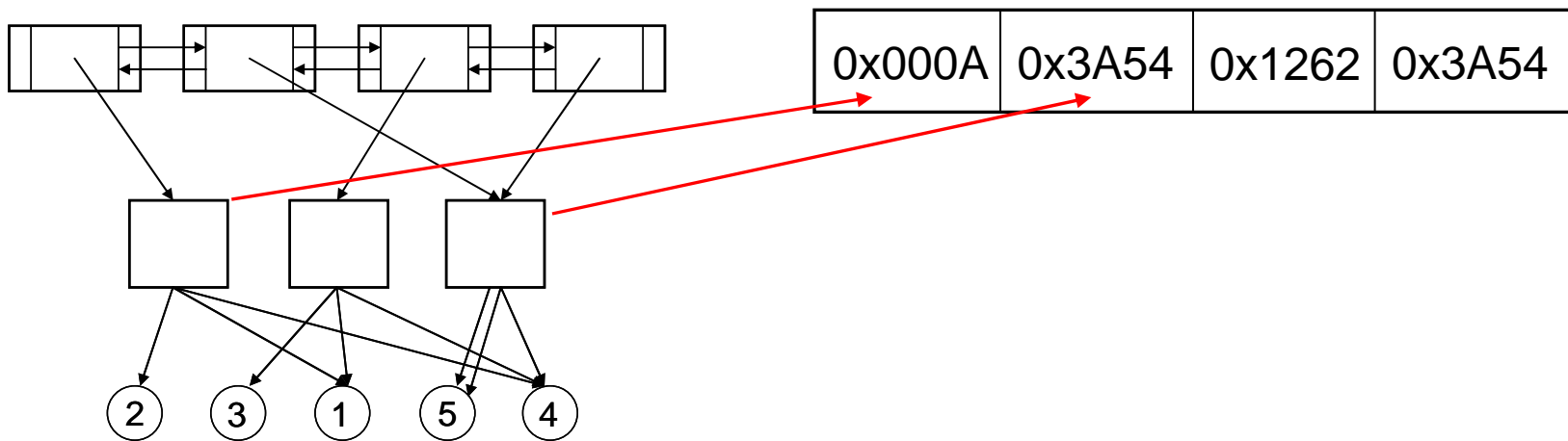
```
struct FloatList {  
    struct Floats *floats;  
    struct FloatList *next;  
    struct FloatList *last;  
};
```

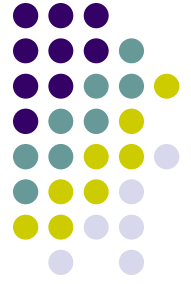
```
struct Floats {  
    float *a;  
    float *b;  
    float *c;  
};
```



# Floatlist.c

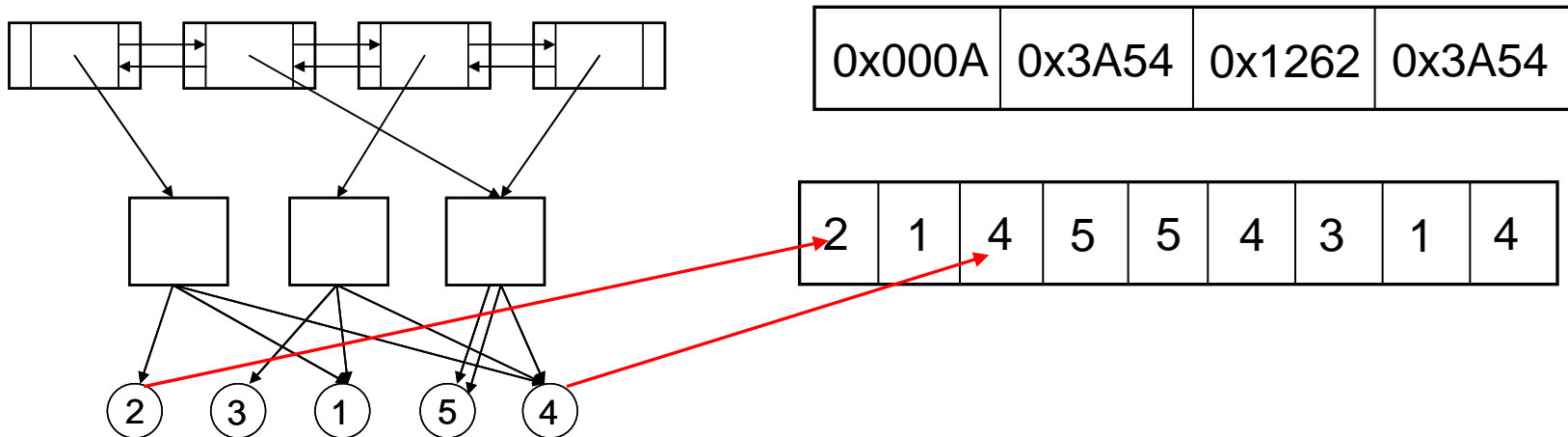
- Step 1: Traverse the list and figure out the number of unique struct Floats stored.





# Floatlist.c

- Step 2: Figure out the number of unique floats stored.

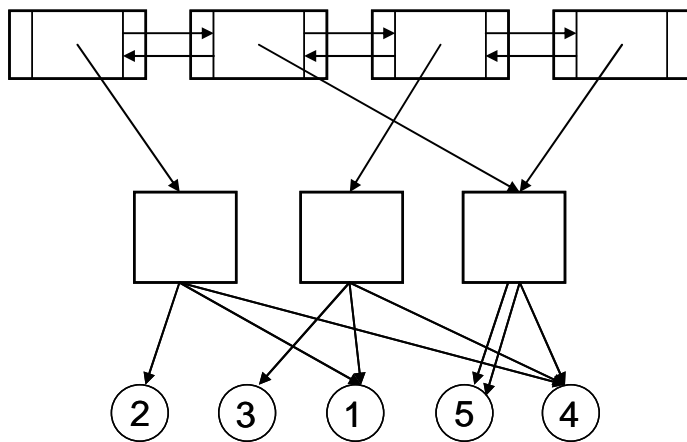






# Floatlist.c

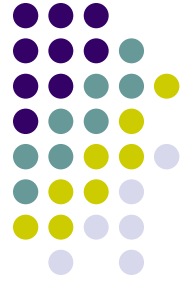
- Step 4: Store the indices of intermediate arrays



0x000A	0x3A54	0x1262	0x3A54
--------	--------	--------	--------

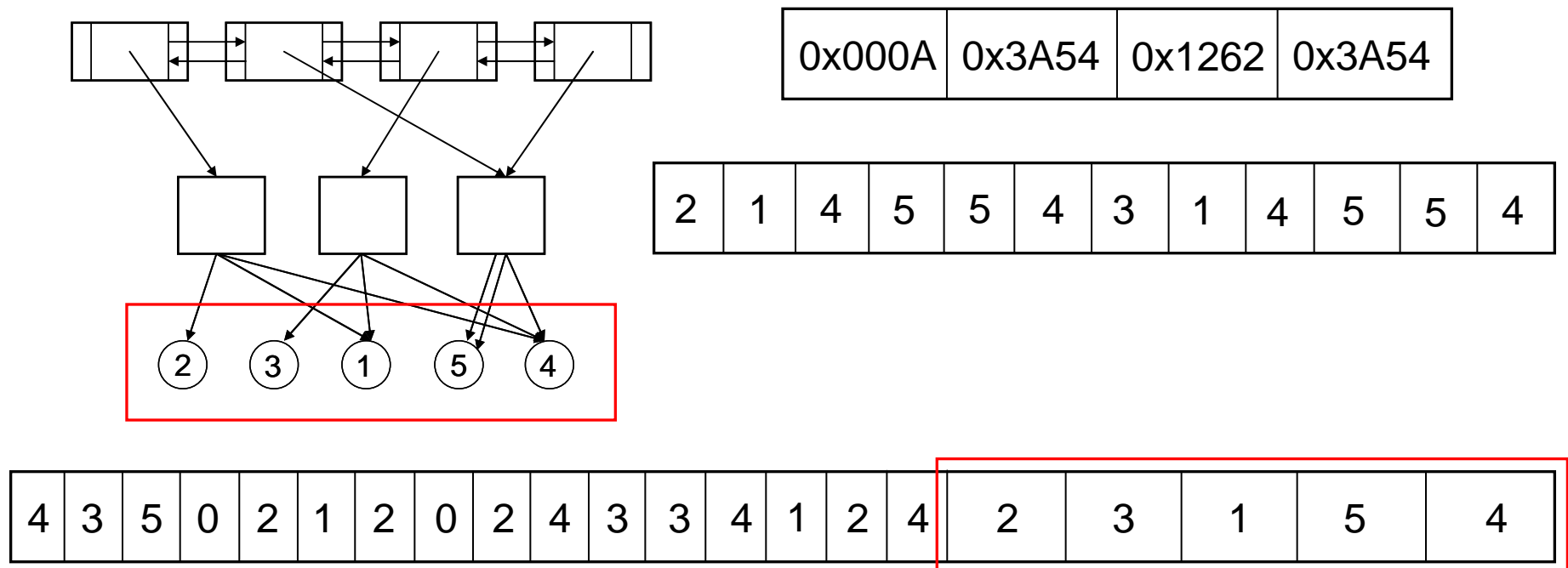
2	1	4	5	5	4	3	1	4
---	---	---	---	---	---	---	---	---

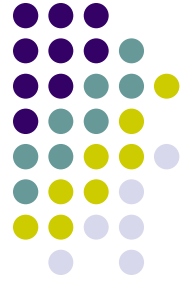
4	3	5	0	2	1	2	0	2	4	3	3	4	1	2	4
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---



# Floatlist.c

- Step 5: Copy over the floats themselves





# Floatlist.c

- Unserialization
  - Reverse process of what has been explained
- Make sure you understand the function

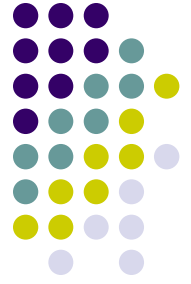


# memcpy

```
#include <string.h>
```

```
void *memcpy(void *s1, const void *s2, size_t n);
```

- Copies  $n$  bytes from memory area  $s2$  to  $s1$ 
  - Similar to `strcpy`, but not affected by NULL
  - Should not be used if the memory areas overlap
- Returns  $s1$



# Summary

- UNIX I/O
  - Basic Operations
    - `read, write, open, close`
  - `select, poll`
  - File Representation
    - Buffering
- Serialization
  - Example in MP3 given files