# Recitation 12: Tshlab + VM

Instructor: TAs

11 November 2019

## Outline

- Labs
- Signals
- IO
- Virtual Memory

## Tshlab and Proxylab

- **Tshlab due Thursday!**
  - 2 late days available

- **Proxylab is released after**
  - Checkpoint due November 26
  - Final due December 5

## Signals

- Parent process sends SIGINT to a child process.
  What is the behavior of the child?
- What is the default?
- What else could the child do?

## More Signals

- Parent process sends SIGKILL to a child process. What is the behavior of the child?

- What is the default?
- What else could the child do?

## Sending Signals

- Parent sends SIGKILL to a child process.

```
...
    pid_t pid = ...; // child pid
    kill(pid, SIGKILL);
    // At this point, what could have
    // happened to the child process?
```

## Blocking Signals

- The shell is currently running its handler for SIGCHLD.

- What signals can it receive?
- What signals can it not receive (i.e., blocked)?

## Errno

- **In shell lab, your signal handlers must preserve errno**
- **Only contains useful value if just returned error**

Consider successfully opening a file “temp.txt”. What is the value of errno?

# IO functions

## Needed for tshlab

- `int open(const char *pathname, int flags, mode_t mode);`
  - Can pass bitwise-or of flags:
    - File Creation: O_CREAT, O_TRUNC, etc.
    - File Status
    - Access Modes (must include at least one): O_RDONLY, O_WRONLY, O_RDWR
  - Mode: specifies what permission is associated with file when creating one

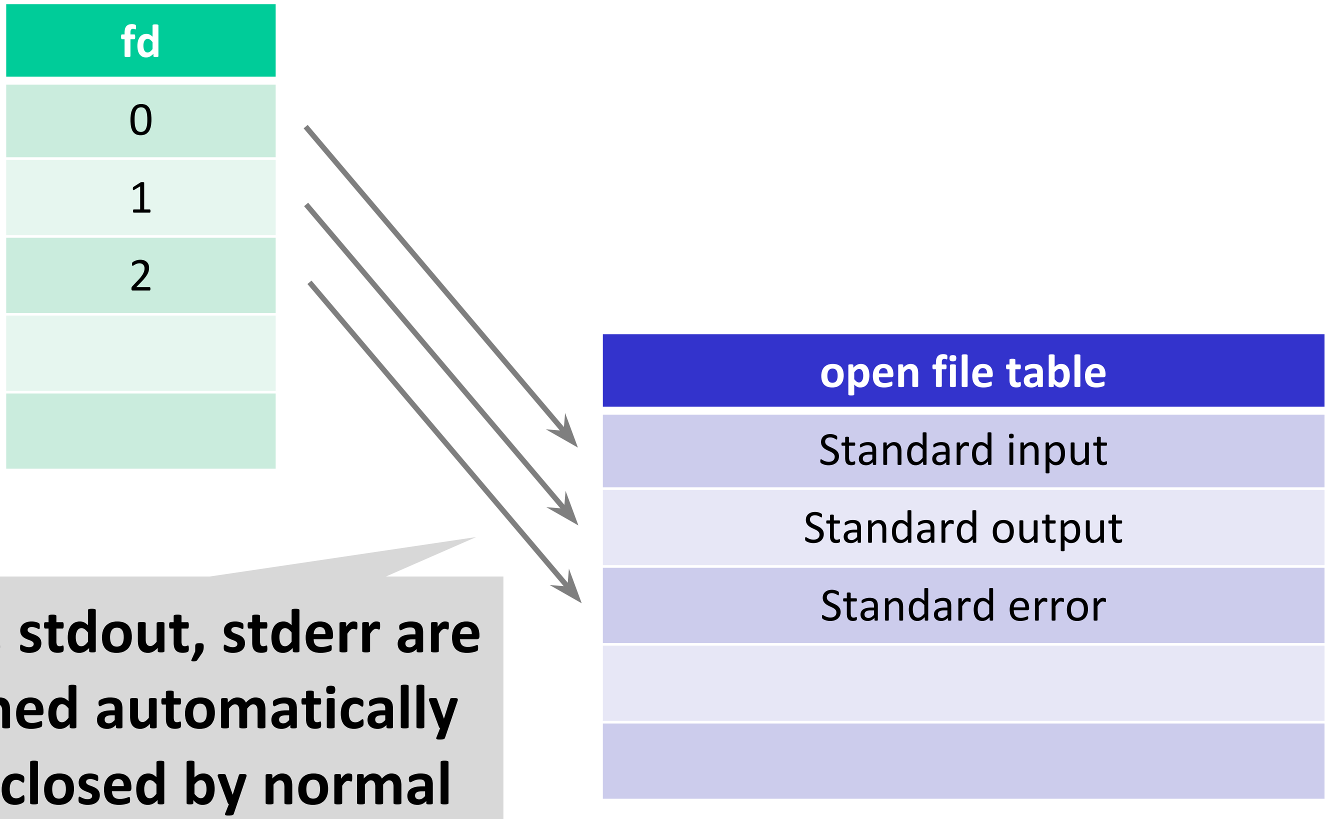
- `int close(int fd);`
- `int dup2(int oldfd, int newfd);`

## Permissions for open()

| | Read (R) | Write (W) | Executable (X) | All (RWX) |
|---|---|---|---|---|
| User (USR) | `S_IRUSR` | `S_IWUSR` | `S_IXUSR` | `S_IRWXU` |
| Group (GRP) | `S_IRGRP` | `S_IWGRP` | `S_IXGRP` | `S_IRWXG` |
| Other (OTH) | `S_IROTH` | `S_IWOTH` | `S_IXOTH` | `S_IRWXO` |

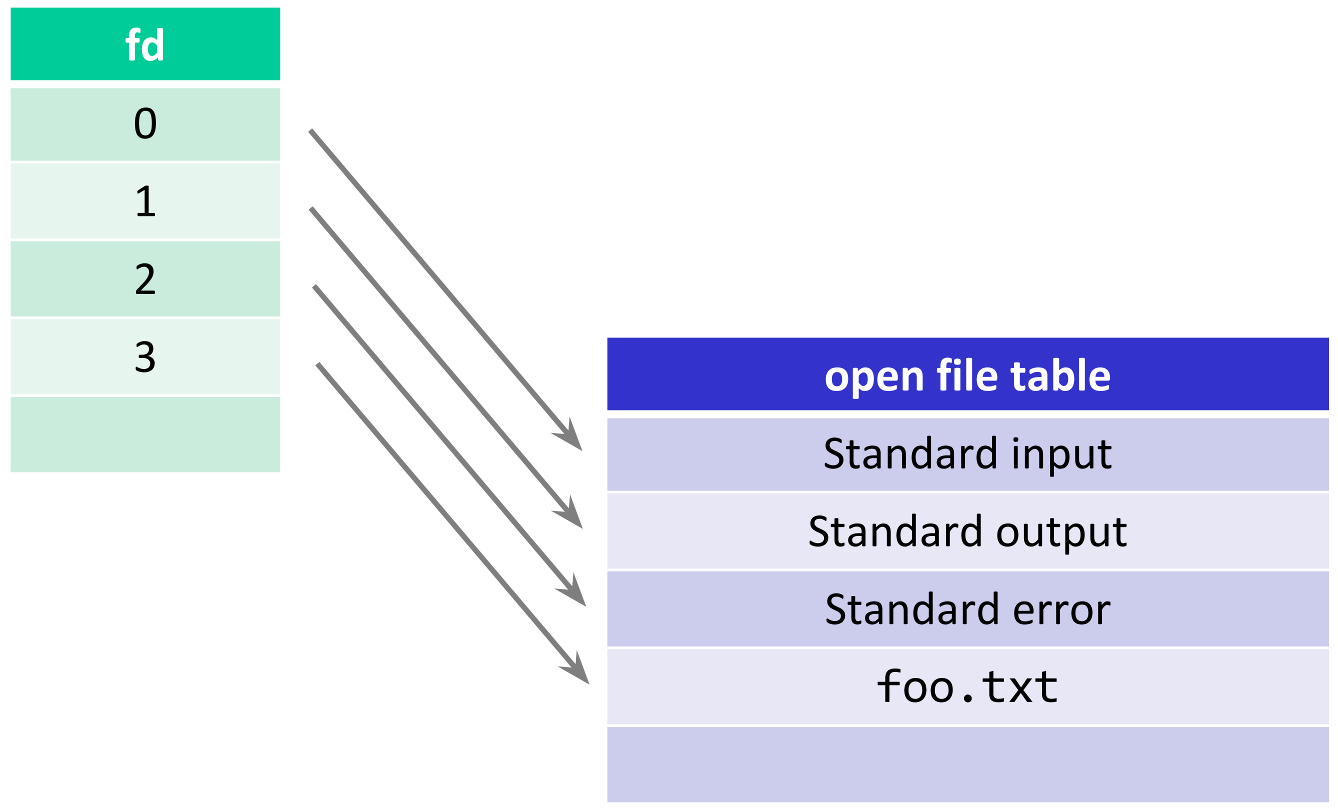
- **These constants can be bitwise-OR'd and passed to the third argument of open()**
- **What does `S_IRWXG | S_IXUSR | S_IXOTH` mean?**
- **How to create a file which everyone can read from but only the user can write to it or execute it?**

## File descriptors

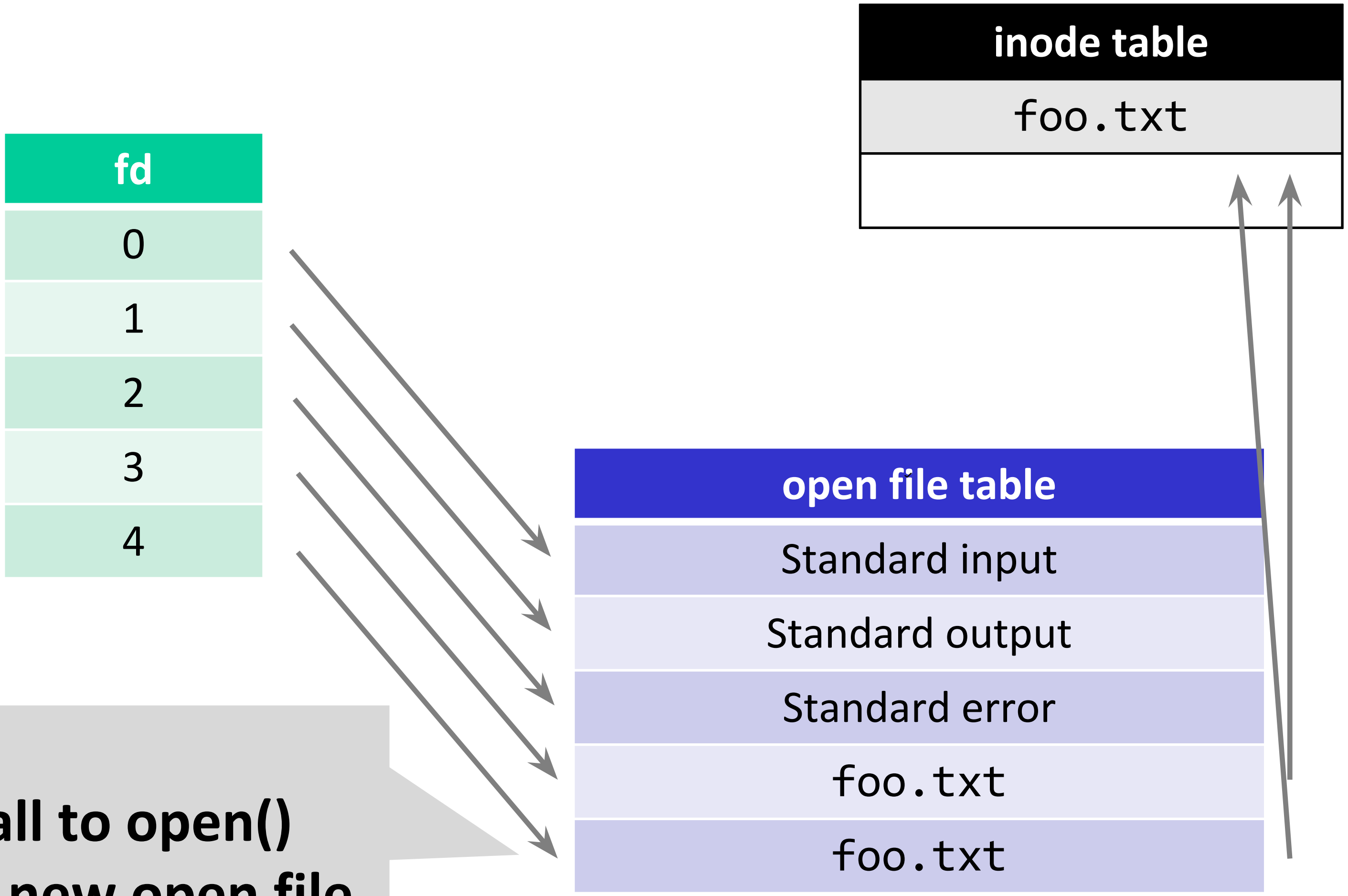


stdin, stdout, stderr are opened automatically and closed by normal termination or exit()

## open("foo.txt")

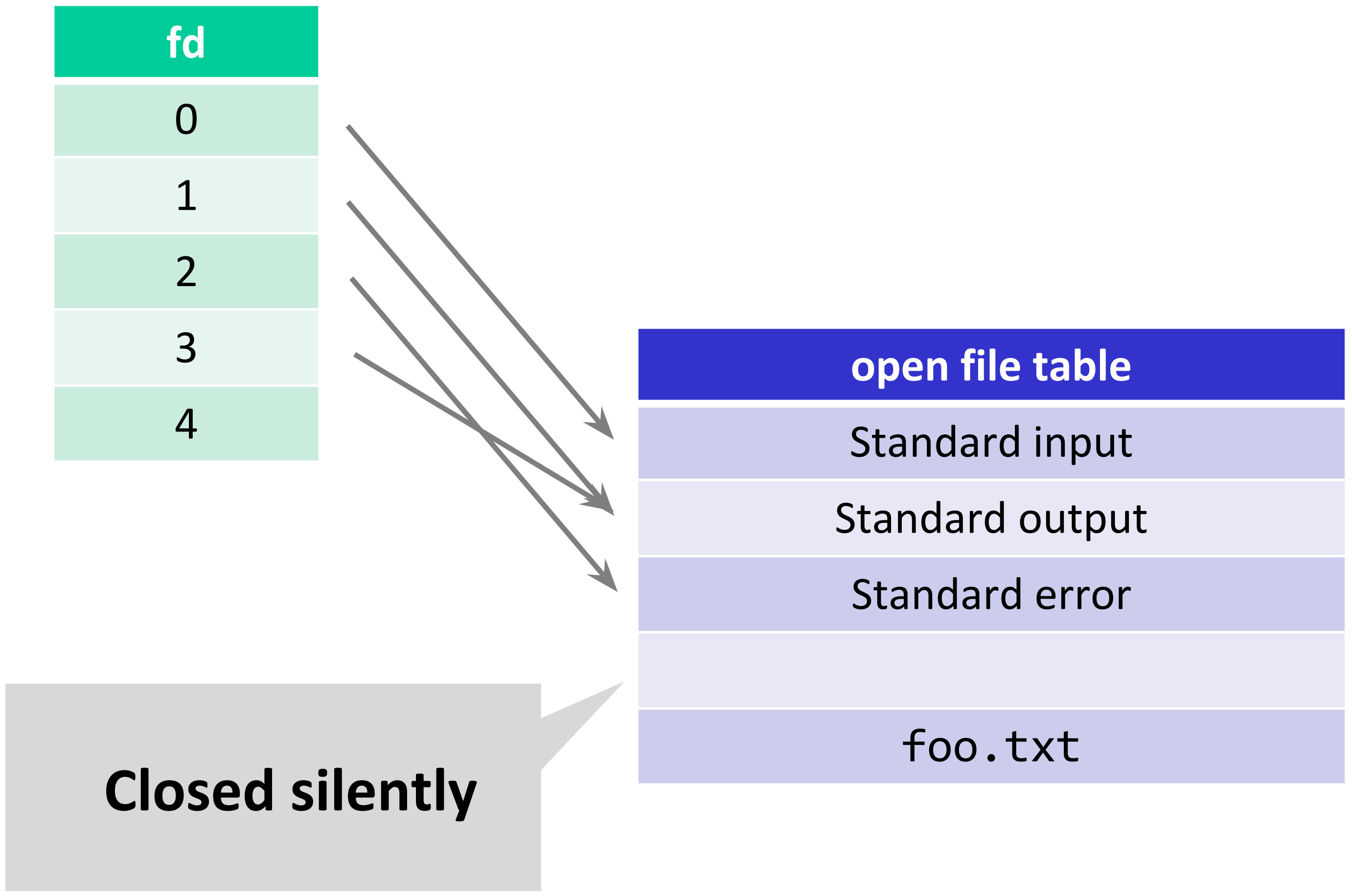

fd
0
1
2
3
open file table
Standard input
Standard output
Standard error
foo.txt

# open("foo.txt")



Each call to open() creates a new open file descriptor

## dup2(STDOUT_FILENO, 3)



## IO and Fork()

- File descriptor management can be tricky.
- How many file descriptors are open in the parent process at the indicated point?
- How many does each child have open at the call to execve?

```
int main(int argc, char** argv)
{
    int i;
    for (i = 0; i < 4; i++)
    {
        int fd = open("foo", O_RDONLY);
        pid_t pid = fork();
        if (pid == 0)
        {
            int ofd = open("bar", O_RDONLY);
            execve(...);
        }
    }
    // How many file descriptors are open in the parent?
```

## Redirecting IO

- File descriptors can be directed to identify different open files.

```
int main(int argc, char** argv) {
    int i;
    for (i = 0; i < 4; i++)
    {
        int fd = open("foo", O_RDONLY);
        pid_t pid = fork();
        if (pid == 0)
        {
            int ofd = open("bar", O_WRONLY);
            dup2(fd, STDIN_FILENO);
            dup2(ofd, STDOUT_FILENO);
            execve(...);
        }
    }
    // How many file descriptors are open in the parent?
}
```

## Redirecting IO

- At the two points (A and B) in main, how many file descriptors are open?

```
int main(int argc, char** argv)
{
    int i, fd;
    fd = open("foo", O_WRONLY);
    dup2(fd, STDOUT_FILENO);
    // Point A
    close(fd);
    // Point B
    ...
```

## Sample Exam Question

What is the possible output given contents of foo.txt are “ABCDEFG”?

```
int main(int argc, char *argv[]) {
        int fd1 = open("foo.txt", O_RDONLY);
        int fd2 = open("foo.txt", O_RDONLY);
        read_and_print_one(fd1);
        read_and_print_one(fd2);
        if(!fork()) {
                read_and_print_one(fd2);
                read_and_print_one(fd2);
                close(fd2);
                fd2 = dup(fd1);
                read_and_print_one(fd2);
        } else {
                wait(NULL);
                read_and_print_one(fd1);
                read_and_print_one(fd2);
                printf("\n");
        }
        close(fd1);
        close(fd2);
        return 0;
}
```

```
void read_and_print_one(int fd) {
        char c;
        read(fd, &c, 1);
        printf("%c", c);
        fflush(stdout);
}
```

## Memory Access

- The processor tries to write to a memory address.
- List different steps that are required to complete this operation.

## Memory Access

- The processor tries to write to a memory address.
- List different steps that are required to complete this operation. (non exhaustive list)

- Virtual to physical address conversion (TLB lookup)
- TLB miss
- Page fault, page loaded from disk
- TLB updated, check permissions
- L1 Cache miss (and L2 … and)
- Request sent to memory
- Memory sends data to processor
- Cache updated

## Address Translation with TLB

- Translate 0x15213, given the contents of the TLB and the first 32 entries of the page table below.
- 1MB Virtual Memory
  256KB Physical Memory
  4KB page size

| Index | Tag | PPN | Valid |
|---|---|---|---|
| 0 | 05 | 13 | 1 |
| | 3F | 15 | 1 |
| 1 | 10 | 0F | 1 |
| | 0F | 1E | 0 |
| 2 | 1F | 01 | 1 |
| | 11 | 1F | 0 |
| 3 | 03 | 2B | 1 |
| | 1D | 23 | 0 |

| VPN | PPN | Valid | VPN | PPN | Valid |
|---|---|---|---|---|---|
| 00 | 17 | 1 | 10 | 26 | 0 |
| 01 | 28 | 1 | 11 | 17 | 0 |
| 02 | 14 | 1 | 12 | 0E | 1 |
| 03 | 0B | 0 | 13 | 10 | 1 |
| 04 | 26 | 0 | 14 | 13 | 1 |
| 05 | 13 | 0 | 15 | 18 | 1 |
| 06 | 0F | 1 | 16 | 31 | 1 |
| 07 | 10 | 1 | 17 | 12 | 0 |
| 08 | 1C | 0 | 18 | 23 | 1 |
| 09 | 25 | 1 | 19 | 04 | 0 |
| 0A | 31 | 0 | 1A | 0C | 1 |
| 0B | 16 | 1 | 1B | 2B | 0 |
| 0C | 01 | 0 | 1C | 1E | 0 |
| 0D | 15 | 0 | 1D | 3E | 1 |
| 0E | 0C | 0 | 1E | 27 | 1 |
| 0F | 2B | 1 | 1F | 15 | 1 |

## If you get stuck on tshlab

- **Read the writeup!**
- **Do manual unit testing before `runtrace` and `sdriver`!**
- **Post private questions on piazza!**

- **Read the man pages on the syscalls.**
  - Especially the error conditions
  - What errors should terminate the shell?
  - What errors should be reported?

## man wait

Taken from **http://man7.org/linux/man-pages/man2/wait.2.html**

```
WAIT(2)                    Linux Programmer's Manual                  WAIT(2)

NAME

       wait, waitpid, waitid - wait for process to change state

SYNOPSIS

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

       pid_t wait(int *wstatus);

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

       int waitid(idtype_t idtype, id_t id, siginfo_t *infop, int options);
                       /* This is the glibc and POSIX interface; see
                          NOTES for information on the raw system call. */
```

## man pages (probably) cover all you need

- What arguments does the function take?
  - read SYNOPSIS
- What does the function do?
  - read DESCRIPTION
- What does the function return?
  - read RETURN VALUE
- What errors can the function fail with?
  - read ERRORS
- Is there anything I should watch out for?
  - read NOTES
- Different categories for man page entries with the same name
- Looking up man pages online is not an academic integrity violation

## Function arguments

- Should I do dup2(old, new) or dup2(new, old)?
- Read the man page:

```
$ man dup2

SYNOPSIS

       #include <unistd.h>

       int dup(int oldfd);
       int dup2(int oldfd, int newfd);
```

## Function behavior

- How should I write my format string when I need to print a long double in octals with precision 5 and zero-padded?
- Read the man page:

```
$ man printf
```

```
DESCRIPTION
Flag characters
       The character % is followed by zero or more of the following flags:

       #      The value should be converted...
       0      The value should be zero padded...
       -      The converted value is to be left adjusted...
       ' '    (a space) A blank should be left before...
       +      A sign (+ or -) should always ...
```

## Function return

- **What does waitpid() return with and without WNOHANG?**
- **Read the man page:**

```
$ man waitpid
```

```
RETURN VALUE

       waitpid(): on success, returns the process ID of the child whose
       state has changed; if WNOHANG was specified and one or more
       child(ren) specified by pid exist, but have not yet changed state,
       then 0 is returned.  On error, -1 is returned.

       Each of these calls sets errno to an appropriate value in the case of
       an error.
```

## Potential errors

- **How should I check waitpid for errors?**
- **Read the man page:**

```
$ man waitpid
```

```
ERRORS

       ECHILD (for waitpid() or waitid()) The process specified by pid
              (waitpid()) or idtype and id (waitid()) does not exist or is
              not a child of the calling process.  (This can happen for
              one's own child if the action for SIGCHLD is set to SIG_IGN.
              See also the Linux Notes section about threads.)

       EINTR  WNOHANG was not set and an unblocked signal or a SIGCHLD was
              caught; see signal(7).

       EINVAL The options argument was invalid.
```

## Get advice from the developers

- I sprintf from a string into itself, is this okay?
- Read the man page:

```
$ man sprintf
```

```
NOTES

       Some programs imprudently rely on code such as the following

           sprintf(buf, "%s some further text", buf);

       to append text to buf.  However, the standards explicitly note that
       the results are undefined if source and destination buffers overlap
       when calling sprintf(), snprintf(), vsprintf(), and vsnprintf().
       Depending on the version of gcc(1) used, and the compiler options
       employed, calls such as the above will not produce the expected
       results.

       The glibc implementation of the functions snprintf() and vsnprintf()
       conforms to the C99 standard, that is, behaves as described above,
       since glibc version 2.1.  Until glibc 2.0.6, they would return -1
       when the output was truncated.
```