Syscalls: Files 3: Basic I/O Syscalls

1. Library vs. Syscall I/O
2. Basic I/O Syscalls (part a)
3. **Basic I/O Syscalls (part b)**
   - **read**
   - **write**
   - **blocking vs nonblocking mode**
4. File Offset
5. Filesystem Syscalls

read() System Call

`read()` is the basic *input* syscall:

```
ssize_t read(int fd, void *buf, size_t count)
```

- `fd` is the file descriptor handle
- `buf` is a pointer to the start of the memory block/array where the bytes read in will be stored
- `count` is the number of *bytes* to be read
- returns the number of bytes actually read ($\leq$ `count`), else -1 on error
- a return of 0 (*zero*) indicates an **end-of-file** condition

read() System Call (contd.)

The amount of data to be read is specified in *bytes*.

`read()` does not understand C types; the `sizeof` operator is commonly used to determine the number of bytes to read.

No checking is performed to ensure that the `buf` memory block is large enough to hold `count` bytes—this is left to the programmer!

A `read()` call changes only the first n bytes of `buf` where n is the actual number of bytes that were read (it does not "zero out" the rest of `buf`).

Because the number of bytes read may be less than the number requested, programs will generally have to *capture the return* from `read()` to know how much of `buf` is valid.

read() System Call (contd.)

There are effectively three classes of returns from `read()`:

- > 0: data was read
- 0: an *end-of-file* condition was encountered
- -1: an error occurred

The number of bytes actually read may be *less* than `count` for several reasons:

- less than `count` bytes remain in a regular file
- less than `count` bytes are currently in the kernel buffer for a pipe or socket
- the default behavior for terminals is to return the next *line*
- a `read()` call may be interrupted by a **signal**
- some devices return only the next "record" (e.g., tape drives)

read() System Call (contd.)

Example of reading (up to) 100 characters of ASCII text and storing it as a legal C string:

```
int infd, nread;
char str[101];
...
infd = open(...);
...
if ((nread = read(infd,str,100)) == -1) {
  perror("Error reading from file");
  exit(EXIT_FAILURE); }
//Ensure str is a legal C string:
str[nread] = '\0';
...
```

read() System Call (contd.)

Example of reading a *line* of ASCII text and storing it as a legal C string:

```
int infd, pos, nread;
char next, line[101];
...
infd = open(...);
...
pos = 0;
while ((nread = read(infd,&next,1)) > 0 && next != '\n')
  line[pos++] = next;
line[pos] = '\0';
if (nread == -1)...
...
```

read() System Call (contd.)

Example of reading an `int` (in binary):

```
int infd, nread, fileint = 0;
...
infd = open(...);
...
if (read(infd,&fileint,sizeof(int)) <= 0) {
  fprintf(stderr,"Problem reading int from file");
  exit(EXIT_FAILURE); }
...
```

write() System Call

`write()` is the basic *output* syscall:

```
ssize_t write(int fd, const void *buf, size_t count)
```

- `fd` is the file descriptor handle
- `buf` is a pointer to the start of the memory that contains the bytes to be written
- `count` is the number of *bytes* to be written
- returns the number of bytes actually written ($\leq$ `count`), else -1 on error

write() System Call (contd.)

Example of writing a C string to a file:

```
char *str;
int outfd;
...
outfd = open(...);
...
if (write(outfd,str,strlen(str)) == -1) {
  perror("Error writing str to file");
  exit(EXIT_FAILURE); }
...
```

Note: a C string's *null sentinel char* is generally *not* written out to text files (files containing null's are considered *binary files*).

write() System Call (contd.)

Example of writing an array of `int`'s (in binary) to a file:

```
int outfd, vector[10];
...
outfd = open(...);
...
for (int i=0; i<10; i++) {
  if (write(outfd,vector+i,sizeof(int)) == -1) {
    perror("Error writing int to file");
    exit(EXIT_FAILURE); }
  }
...
```

Blocking Mode

By default, I/O occurs in **blocking mode**.

In blocking mode, a `read()` or `write()` call can **block**—cause the process/thread to be *suspended*—until data is able to be read or written.

Blocking mode is important to understand, because it can be possible processes/threads to block *indefinitely!*

This is the case for I/O with pipes/FIFOs, sockets, terminals, and non-block devices.

E.g., if you try to `read()` from an empty pipe (that still has its write end open), the process/thread will block until some data becomes available in the pipe.

Blocking Mode (contd.)

Note, however, that I/O involving **regular files** or **block devices** cannot cause indefinite blocking.

This is because data being transferred from/to regular files and block devices gets *automatically* **cached** (**buffered**) by the kernel, to speed up reading/writing of files to secondary storage.

The effect is that `read()`/`write()` cannot block indefinitely:

- `read()` may have to wait on a disk read if the requested data is not in the **page cache**, but it will "block" only briefly
- `write()` usually returns immediately because the data simply gets written to the page cache (written to storage later), or may block briefly if **synchronized I/O** has been set

Nonblocking Mode

The possibility of `read()` or `write()` blocking indefinitely can be a problem for some applications.

Then, a file descriptor can be put into **nonblocking mode**.

When in nonblocking mode, a `read()` or `write()` call to an FD *always returns immediately*:

- if it can "immediately" read/write data, then it does (just as in blocking mode)
- if it *cannot* immediately read/write data (so call would *block*), -1 is returned (error), and `ERRNO` is set to `EAGAIN` or `EWOULDBLOCK`

Nonblocking mode can be appropriate to use when I/O involves pipes/FIFOs, sockets, terminals, and some devices.

Nonblocking Mode (contd.)

A file descriptor can be set to nonblocking mode in various ways:

- `open()` and `O_NONBLOCK` flag
- `fcntl()` and `F_SETFD` command, to set `O_NONBLOCK` flag
- `pipe2()` and `O_NONBLOCK` flag (Linux specific)
- `accept4()` and `SOCK_NONBLOCK` flag (Linux specific)