Advanced Programming in the UNIX Environment

Week 02, Segment 3:
read(2), write(2), lseek(3)

Department of Computer Science
Stevens Institute of Technology

Jan Schaumann
jschauma@stevens.edu
https://stevens.netmeister.org/631/

read(2)

```
#include <unistd.h>

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

Returns: number of bytes read; 0 on EOF, -1 on error

`read` begins reading at the current offset, and increments the offset by the number of bytes actually read.

There can be several cases where read returns fewer than the number of bytes requested. For example:

- EOF reached before requested number of bytes have been read
- reading from a network, buffering can cause delays in arrival of data
- record-oriented devices (magtape) may return data one record at a time
- interruption by a signal

write(2)

```
#include <unistd.h>

ssize_t write(int fd, void *buf, size_t num);
```

Returns: number of bytes written if OK; -1 on error

- `write` returns the number of bytes written
- For regular files, write begins writing at the current offset (unless `O_APPEND` has been specified, in which case the offset is first set to the end of the file).
- After the write, the offset is adjusted by the number of bytes actually written.

write(2)

Some manual pages note:

> *If the real user is not the super-user, then write() clears the set-user-id bit on a file. This prevents penetration of system security by a user who "captures" a writable set-user-id file owned by the super-user.*

Think of specific examples for this behavior.

Write a program that attempts to exploit a scenario where `write(2)` does not clear the setuid bit, then verify that your evil plan will be foiled.

Terminal — 80×24

```
Trying to create './newfile' with O_RDONLY | O_CREAT...
'./newfile' created. File descriptor is: 4
-rw-------  1 jschauma  users  0 Sep  2 01:27 newfile

Checking if './newfile' exists...
-rw-------  1 jschauma  users  0 Sep  2 01:27 ./newfile
Trying to create './newfile' with O_RDONLY | O_CREAT | O_EXCL...
Unable to create './newfile': File exists
Closing failed: Bad file descriptor

Trying to open './openex.c' with O_RDONLY...
'./openex.c' opened. File descriptor is: 5
'./openex.c' closed again

Trying to open (non-existant) './nosuchfile' with O_RDONLY...
Unable to open './nosuchfile': No such file or directory

Copied 'openex.c' to 'newfile'.
-rw-------  1 jschauma  users  3192 Sep  2 01:28 newfile
Trying to open './newfile' with O_RDONLY | O_TRUNC...
'./newfile' opened. File descriptor is: 5
'./newfile' truncated -- see 'ls -l newfile'
-rw-------  1 jschauma  users  0 Sep  2 01:28 newfile
apue$
```

lseek(2)

```
#include <sys/types.h>
#include <fcntl.h>

off_t lseek(int fd, off_t offset, int whence);
```

Returns: new offset if OK; -1 on error

lseek(2)

```
#include <sys/types.h>
#include <fcntl.h>

off_t lseek(int fd, off_t offset, int whence);
```

Returns: new offset if OK; -1 on error

The value of `whence` determines how `offset` is used:

- `SEEK_SET` bytes from the beginning of the file
- `SEEK_CUR` bytes from the current file position
- `SEEK_END` bytes from the end of the file

"Weird" things you can do using `lseek(2)`:

- seek to a negative offset
- seek 0 bytes from the current position
- seek past the end of the file

```
$ ssh apue
Last login: Thu Sep  3 22:53:40 2020 from 10.0.2.2
NetBSD 9.0 (GENERIC) #0: Fri Feb 14 00:06:28 UTC 2020

Welcome to NetBSD!

apue$ cd 02
apue$ vim lseek.c
apue$ cc lseek.c
apue$ ./a.out
seek OK
apue$ ./a.out <lseek.c
seek OK
apue$ cat lseek.c | ./a.out
cannot seek
apue$ mkfifo /tmp/fifo
apue$ ls -l /tmp/fifo
prw-r--r--  1 jschauma  wheel  0 Sep  3 22:55 /tmp/fifo
apue$ ./a.out </tmp/fifo
cannot seek
apue$ 
```

```
apue$ echo $(( 10240020 / 512 ))
20000
apue$ df .
Filesystem   512-blocks       Used      Avail %Cap Mounted on
/dev/wd0a      30497436    7378524   21594044  25% /
apue$ echo $(( 7378524 - 7378428 ))
96
apue$ ls -ls file.hole
96 -rw-------  1 jschauma  users  10240020 Sep  4 03:10 file.hole
apue$ hexdump -c file.hole
0000000   a   b   c   d   e   f   g   h   i   j  \0  \0  \0  \0  \0  \0
0000010  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0
*
09c4000  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0   A   B   C   D   E   F
09c4010   G   H   I   J
09c4014
apue$ cp file.hole file.nohole
apue$ ls -l file*
-rw-------  1 jschauma  users  10240020 Sep  4 03:10 file.hole
-rw-------  1 jschauma  users  10240020 Sep  4 03:12 file.nohole
apue$ ls -ls file.*
   96 -rw-------  1 jschauma  users  10240020 Sep  4 03:10 file.hole
20064 -rw-------  1 jschauma  users  10240020 Sep  4 03:12 file.nohole
apue$ hexdump -c file.nohole
```

I/O Efficiency

In simple-cat.c from last week, we used:

```
#define BUFFSIZE 32768
[...]
    while ((n = read(STDIN_FILENO, buf, BUFFSIZE)) > 0) {
        if (write(STDOUT_FILENO, buf, n) != n) {
            fprintf(stderr, "Unable to write: %s\n", strerror(errno));
            exit(EXIT_FAILURE);
        }
    }
```

How/why did we pick 32768? What if we increased/decreased that number?

Let's create a benchmark test:

```
mkdir -p tmp
for n in $(jot 10); do
        echo "Creating file number $n...";
        dd if=/dev/urandom of=tmp/file$n bs=$(( 1024 * 1024)) count=100 2>/dev/null;
done

for n in 3145728 1048576 32768 16384 4096 1024 256 128 64 1; do
        cc -Wall -DBUFFSIZE=$n simple-cat.c;
        i=$(( $i + 1 ));
        for j in $$(jot 5); do
                /usr/bin/time -p ./a.out <tmp/file$i >tmp/out;
        done
done
```

Terminal — 80×24

```
BUFFSIZE = 16384
        0.47 real         0.01 user         0.34 sys
        0.32 real         0.00 user         0.30 sys
        0.31 real         0.00 user         0.26 sys
        0.29 real         0.02 user         0.25 sys
        0.30 real         0.00 user         0.27 sys


BUFFSIZE = 4096
        0.59 real         0.02 user         0.41 sys
        0.43 real         0.01 user         0.36 sys
        0.45 real         0.02 user         0.36 sys
        0.45 real         0.00 user         0.41 sys
        0.44 real         0.01 user         0.41 sys


BUFFSIZE = 1024
        0.87 real         0.02 user         0.72 sys
        0.69 real         0.04 user         0.62 sys
        0.68 real         0.00 user         0.62 sys
        0.68 real         0.01 user         0.66 sys
        0.70 real         0.05 user         0.63 sys
```

Conclusion

Compare the manual pages for `read(2)`, `write(2)`, and `lseek(2)` on different OS.

`lseek(STDIN_FILENO, 0, SEEK_CUR)` succeeds when connected to a terminal - what happens when you try to seek to the end or the beginning? Does that even make sense?

If you create a new file, write, say, 10 bytes of data, and then seek to the end of the file, where do you end up? Why?

Play around with the creation and handling of sparse files as well as repeat our benchmark test on different operating- and file systems.

In our next segment:

How can we visualize multiple processes accessing the same files simultaneously?

Are there additional considerations regarding atomicity when using `read(2)` and `write(2)`?

How does file descriptor redirection work in the shell?