I/O System Calls

Chapter 3

modified from slides by Dr. B. Boufama and Dr. Quazi Rahman

Unix I/O System Calls

- most Unix I/O can be performed using system calls
  - open, close
  - read, write
  - lseek
- in contrast to standard I/O functions, they are unbuffered

Comparison

I/O system calls	standard I/O library
unbuffered I/O	buffering
not part of ISO C	specified in C standard

File Descriptors

- who use it
  - kernel refers to any open file by a *file descriptor*
- what is it
  - non-negative integer 0..OPEN_MAX
- how do you get it
  - typically returned by system call open() to open/create these
    - file
    - pipe
    - network communication channel

File Descriptors

<unistd.h>

	file descriptors	value
standard input	STDIN_FILENO	0
standard output	STDOUT_FILENO	1
standard error	STDERR_FILENO	2

System Call: open()

```
#include <fcntl.h>
int open(const char *fName, int oflag, .../* mode_t mode */ );
```

returns: file descriptor if ok, -1 on error

- *fName*
  - name of file to open
- *oflag*
  - list of values separated by bitwise 'OR' (|)
  - determines how the file is to be opened
  - its value defined in <fcntl.h> (file control)

System Call: open()

- value for *oflag*
  - must have one of these three

oflag value	meaning
O_RDONLY	read only
O_WRONLY	write only
O_RDWR	read and write

System Call: open()

- value for *oflag*
  - these are optional

oflag	meaning...
O_APPEND	open for writing at end of file
O_CREAT	• if file does not exist, create it • if this option is used, also need third parameter (*mode_t mode*) for access permission bits
O_EXCL	• generate error if O_CREAT is specified and file already exists • ensure that caller must create the file
O_TRUNC	if file exists and successfully opened for either write-only or read-only, truncate its length to zero

System Call: open()

- value for *mode* defined in <fcntl.h>

	mode mask	bit mask
access permission for owner: R, W, X	S_IRUSR	0400
	S_IWUSR	0200
	S_IXUSR	0100
access permission for group: R, W, X	S_IRGRP	0040
	S_IWGRP	0020
	S_IXGRP	0010
access permission for others: R, W, X	S_IROTH	0004
	S_IWOTH	0002
	S_IXOTH	0001

System Call: creat()

```
#include <fcntl.h>
int creat(const char *fName, mode_t mode);
```

returns: file descriptor if ok, -1 o.w.

equivalent to

open(fName, O_WRONLY | O_CREAT | O_TRUNC, mode)

System Call: close()

```
#include <unistd.h>
int close(int fd);
```

returns: 0 if ok, -1 on error

- what happens with this operator
  - file descriptor is set free
- when to return -1 ?
  - e.g. *fd* was already closed
- what happens when process terminates
  - all open files closed by kernel

System Call: lseek()

```
#include <unistd.h>
off_t lseek(int fd, off_t offset, int whence);
```

returns: new file offset if ok, -1 on error

- what does it do
  - set *file pointer* associated with file *fd*

System Call: lseek()

offset

- a long integer
- can be a negative number
- meaning: number of bytes
- initialized to 0 when file opened, unless O_APPEND is specified

System Call: lseek()

whence

whence is SEEK_SET	set file offset to *offset* bytes from beginning
whence is SEEK_CUR	set file offset to current value plus *offset*
whence is SEEK_END	set file offset to size of the file plus *offset*

System Call: lseek()

how to use lseek:

- use *lseek* to determine the current offset

```
off_t     currpos
currpos = lseek(fd, 0, SEEK_CUR)
```


System Call: lseek()

how to use lseek:

- use *lseek* to determine if a file is capable of seeking
  - *lseek* returns -1 if *fd* refers to pipe, socket, etc. (cannot seek)

System Call: lseek()

```
#include <unistd.h>

int main(void) {
        if (lseek(STDIN_FILENO, 0, SEEK_CUR)
== -1)
                printf("cannot seek\n");
        else
                printf("seek ok\n");
        exit(0);
}
```

pipe or socket

```
> a.out < testfile
seek ok
> cat testfile | a.out
cannot seek
```


System Call: lseek()

offset

- what happens if offset is greater than file's current size ?
  - file gets extended
  - any bytes not written are read back as 0

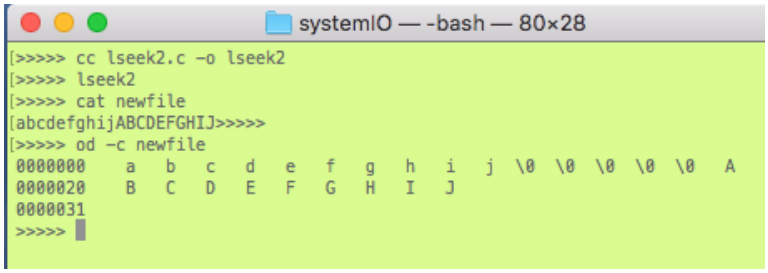
Example...
pay attention to file size and file content

```c
#include <unistd.h>
#include <fcntl.h>
#include <stdlib.h>

int main() {

        char buf1[] = "abcdefghij";
        char buf2[] = "ABCDEFGHIJ";
        int fd;

        if (( fd = creat("newfile", 0533 )) < 0) {
                printf("creat error\n");
                exit(1);
        }
        if (( write(fd, buf1, 10) ) != 10) {
                printf("write error\n");
                exit(1);
        }
        if (( lseek(fd, 15, SEEK_SET) ) == -1) {
                printf("write error\n");
                exit(1);
        }
        if (( write(fd, buf2, 10) ) != 10) {
                printf("write error\n");
                exit(1);
        }
        close(fd);
}
```

System Call: read()

```
#include <unistd.h>
ssize_t read(int fd, void *buf, size_t nbytes);
```

returns: number of bytes read,
0 if end of file
-1 on error

- read up to *nbytes*
- *ssize_t* usually defined as *signed size_t*
- in what situations: *return value = 0* ?
  (EOF has already been reached)

System Call: read()

```
#include <unistd.h>
ssize_t read(int fd, void *buf, size_t nbytes);
```

returns: number of bytes read,
0 if end of file
-1 on error

- in what situations: return value < *nbytes* ?
  - number of bytes read before EOF is less than *nbytes*
  - reading from keyboard – up to a line
  - reading from a network

System Call: write()

#include <unistd.h>
*ssize_t write(int fd, const void *buf, size_t nbytes);*

returns: nbytes if ok, -1 o.w.

Example: reversing a file

- what does it do
  - last byte becomes the first byte
  - second last byte becomes the second
  - …
- command
  > reverse fileIn fileOut
- two solutions
  - use lseek on fileIn (read bytes from end)
  - use lseek on fileOut (write bytes from end)

solution 1: use lseek() on fileIn

```
#include <unistd.h>
#include <fcntl.h>
int main(int argc, char *argv[]) {

    int fd1, fd2;
    char buffer;
    long int i = 0, fileSize = 0;

    fd1 = open(argv[1], O_RDONLY);
    fd2 = open(argv[2], O_CREAT|O_WRONLY|O_TRUNC, 0755);

    while (read(fd1, &buffer, 1) > 0)
        fileSize++;

    while (++i <= fileSize) {
        lseek(fd1, -i, SEEK_END);
        read(fd1, &buffer, 1);
        write(fd2, &buffer, 1);
    }
    close(fd1);
    close(fd2);
}
```

```
//solution 2: start writing at the end

int main(int argc, char *argv[]) {

    int fd1, fd2;
    char buffer;
    long int i = 0, fileSize = 0;

    fd1 = open(argv[1], O_RDONLY);
    fd2 = open(argv[2], O_CREAT|O_WRONLY|O_TRUNC, 0755);

    while(read(fd1, &buffer, 1)>0)
        fileSize++;

    lseek(fd2, fileSize-1, SEEK_SET);          // extend
    lseek(fd1, 0, SEEK_SET);

    while(++i <= fileSize) {
        read(fd1, &buffer, 1);
        lseek(fd2, -i, SEEK_END);
        write(fd2, &buffer, 1);
    }
    close(fd1);
    close(fd2);
}
```