Lecture 2. File I/O

Most File I/O under UNIX can be performed using just five functions: `open()`, `read()`, `write()`, `lseek()`, and `close()`. These functions are referred to as **unbuffered I/O**, meaning that no buffer and associated size are specified for the reading/writing of data. Rather, data is read/written *character by character*, as opposed to *in block* (with the block size specified), such as with buffered I/O. The preference for unbuffered I/O comes when output is desirable as soon as available (such as with `stderr`), as opposed to having to wait until the whole block has become available. I/O kernel system calls are unbuffered.

File Descriptors

Files are referred to by their file descriptors, small numbers returned by the kernel whenever a file is created or written to. By convention, file descriptor 0 is associated to the **standard input** of a process (input from the keyboard), 1 to the **standard output** (output to the terminal), and 2 to the **standard error** (error output to the terminal). To improve program readability, these numbers are usually constants defined as `STDIN_FILENO`, `STDOUT_FILENO`, and `STDERR_FILENO`. Their definitions can be found in the `<unistd.h>` header file.

File descriptors range from `0` to `OPEN_MAX-1`, with early UNIX implementations having a maximum of 19 (allowing maximum 20 open files per process). Subsequent system have increased this to 63.

`open()` and `openat()` Functions

A file is opened or created using one of the `open()` or `openat()` functions:

```
#include <fcntl.h>

int open(const char *path, int oflag, …, /* mode_t mode */);
int openat(int fd, const char *path, int oflag, …, /* mode_t
                mode */);
```

Returns file descriptor if OK, -1 otherwise

The "…" are a place holder for a varying type and number of remaining arguments.

The arguments are:

- *path* — the filename of the file to create or open;
- *oflag* — one or more of the following options, OR-ed together:

O_RDONLY	Open for reading only.
O_WRONLY	Open for writing only.
O_RDWR	Open for reading and writing.
O_EXEC	Open for execute only.
O_SEARCH	Open for search only (for directories, to evaluate search permissions at the time the directory is opened, and never again later).
O_APPEND	Append to the end of file on write.
O_CLOEXEC	Set the FD_CLOEXEC file descriptor flag.
O_CREAT	Create the file if it doesn't exist. Requires a `mode_t` type argument to `open()` and `openat()`, specifying the access permission bits of the new file. There are 9 permission bits for each file, divided into three categories (user, group, other), which you may know from using the `chmod` shell command.
O_DIRECTORY	Generate an error if path doesn't exist.
O_EXCL	Generate an error if O_CREAT is specified and the file already exists.
O_NOCTTY	If path refers to a terminal, do not allocate the device as the controlling terminal for this process (you may want to not bind your process to the terminal specified for I/O).
O_NOFOLLOW	Generate an error if path is a symbolic link.
O_NONBLOCK	If path is a FIFO, a block special file, or a character special file, sets the nonblocking mode for opening the file and all later I/O to that file.

The return value is guaranteed to be the lowest file descriptor available.

The *fd* parameter distinguishes `openat()` from `open()`. There are three possibilities:

1. *path* is an absolute pathname. Then the *fd* parameter is ignored and *openat()* functions exactly as *open()*;
2. *path* is a relative pathname. The *fd* parameter locates the file relative to the start directory. Is is obtained from the start directory of the relative pathname;
3. *path* is a relative pathname and `fd=AT_FDCWD`. Then *fd* locates the file in the current directory.

The `openat()` function is missing on some platforms: glibc 2.3.6, **Mac OS X 10.5**, FreeBSD 6.0, NetBSD 5.0, OpenBSD 3.8, Minix 3.1.8, AIX 5.1, HP-UX 11, IRIX 6.5, OSF/1 5.1, Cygwin 1.5.x, mingw, MSVC 9, Interix 3.5, BeOS. But the replacement function is not safe to be used in libraries and is not multithread-safe.

Below is a very simple example of `open()`:

```
#include <fcntl.h> /* for open() */
#include <stdio.h> /* for printf() */

int main(int argc, char *argv[])
{
     char *filename="/Users/awise/Stevens/Lecture2/foo";
     int fd=open(filename, O_RDWR);
     printf("File descriptor fd=%d\n", fd);
}
```

If the file doesn't exist (and the O_CREAT option wasn't used), this code prints out the value –1:

```
$ ./myopen
File descriptor absolute path fd=-1
```

If the file does exist, the code prints out the value of the file descriptor associated with newly opened file foo:

```
$./myopen
File descriptor absolute path fd=3
```

If we wanted to use the `open()` system call to also create the file if it didn't exist, we could have used the option O_CREAT bitwise OR with O_RDWR:

```
#include <fcntl.h> /* for open() */
#include <stdio.h> /* for printf() */

int main(int argc, char *argv[])
{
     char *filename="Users/awise/Stevens/Lecture2/foo";
     int fd=open(filename, O_RDWR|O_CREAT);
     printf("File descriptor fd=%d\n", fd);
}
```

Filename and Pathname Truncation

In order to illustrate the maximum number of characters allowed to construct a filename or a path on your system, you need to look at the synopsis of another system call, `pathconf()`. For a more detailed description, of course, always use the man pages:

```
$ man 2 pathconf
```

But, in the meantime:

```
#include <unistd.h>

long pathconf(const char *pathname, int name);
long fpathconf(int fd, int name);
                    Returns number of bytes in pathname, filename if OK, -1 otherwise
```

The arguments are:

- *`pathname`* — the string representing the absolute path for a file;
- *`fd`* — alternatively, the number representing the file descriptor;
- *`name`* — one of many options for `pathconf()`, listed as follows:

_PC_FILESIZEBITS	minimum # of bits needed to represent the maximum size of a regular file in the specified directory. This is a signed integer value;
_PC_LINK_MAX	maximum # of links a file may have;
_PC_MAX_CANON	maximum # of bytes on a terminal's input queue;
_PC_MAX_INPUT	# of bytes for which there is space on a terminal's input queue;
_PC_NAME_MAX	max # of bytes in a filename (w/o terminating null);
_PC_PATH_MAX	max # of bytes in a relative pathname (with terminating null);
PIPE_BUF	max # of bytes that can be written to a pipe;
_PC_TIMESTAMP_RESOLUTION	resolution in ns for file timestamps;
_PC_SYMLINK_MAX	# of bytes in a symbolic link

In order to find out the maximum number of characters a file's name may have, we could call `pathconf()` as in the following program:

```
#include <unistd.h> /* for pathconf() */
#include <stdio.h> /* for printf() */

int main(int argc, char *argv[])
{
     char *filename="/Users/awise/Stevens/Lecture2/foo";

     long max_name_bytes=pathconf(filename, _PC_NAME_MAX);
     printf("Name of file %s may have at most %ld bytes.\n",
filename, max_name_bytes);

     long max_path_bytes=pathconf(filename, _PC_PATH_MAX);
     printf("Path of file %s may have at most %ld bytes.\n",
filename, max_path_bytes);
return 0;
}
```

The output of this program is (I include the run command):

```
$ ./mypathconf
Name of file /Users/awise/Stevens/Lecture2/foo may have at most
255 bytes
Path of file /Users/awise/Stevens/Lecture2/foo may have at most
1024 bytes
```

The same value for the filename maximum size will be returned when using the constant FILENAME_MAX, defined in the `<stdio.h>` header:

```
#include <stdio.h>

void main(void)
{
     printf("Maximum allowable pathname size is %d bytes",
FILENAME_MAX);
}
```

This produces:

```
$ ./mypathconf2
Maximum allowable pathname size is 1024 bytes
```

There are two very useful websites to check also, when trying to find out a good reference on the POSIX UNIX standards (system calls and constants):

pubs.opengroup.org (reference for The Open Group Base Specifications, of IEEE)
www.freebsd.ord (reference for the FreeBSD man pages)

You can check the behavior of your system when you exceed the maximum allowable file size by running the following program:

```
#include <fcntl.h> /* for open() */
#include <unistd.h> /* for pathconf() */
#include <string.h> /* for strcat() */
#include <stdlib.h> /* for malloc() */
#include <errno.h> /* for errno */

int main(int argc, char *argv[])
{
     char *pathname;
     pathname="/Users/awise/Stevens/Lecture2/";
     char filename[1000];
     for (int i=0; i<999; i++)
          (void) strcat(filename, “a");
     char *abs_path;
     if    (    (abs_path=malloc(strlen(pathname)+strlen(filename)
+1))!=NULL )
     {
          abs_path[0]='\0';
          strcat(abs_path, pathname);
          strcat(abs_path, filename);
     }
     else
          printf("malloc() failed\n”);

     printf("Long path: %s\n", abs_path);
     int fd=open(abs_path, O_RDWR|O_CREAT|O_TRUNC);
     long e=pathconf(abs_path, _PC_NO_TRUNC);
     printf("%s\n", strerror(errno));
return 0;
}
```

This program produces the following output:

```
Long path: /Users/awise/Stevens/Lecture2/aaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
```

```
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaa
File name too long
```

The `creat()` Function

```
#include <fcntl.h>

int creat(const char *path, mode_t mode);
```
Returns file descriptor for write-only if OK, -1 if error

This is equivalent to:

```
open(path, O_WRONLY|O_CREAT|O_TRUNC, mode);
```

The arguments are:

- *path* — the absolute file path;
- *mode* — the file access mode, set in the file permissions.

With `creat()`, a file is open only for writing. This is why `open()` is used instead, with the following options:

```
open(path, O_RDWR|O_CREAT|O_TRUNC, mode);
```

The `close()` Function

```
#include <unistd.h>

int close(int fd);
```
Returns 0 if OK, -1 if error

The kernel closes automatically any open files when a process terminates. The explicit closing of a file may be necessary for the safe access of the file by other processes (the release of the lock on the file by the current process may be necessary).

The `lseek()` Function

The offset of an open file can be returned by calling `lseek()`:

```
#include <unistd.h>

off_t lseek(int fd, off_t offset, int whence);
                                        Returns new file offset if OK, -1 if error
```

The arguments are:

- *`fd`* — file descriptor;
- *`offset`* — position of the "cursor" in the file;
- *`whence`* — an integer constant, defined as one of the three options:

SEEK_SET	the offset is set to *`offset`* bytes from the beginning of the file;
SEEK_CUR	the offset is set to the current value plus *`offset`*;
SEEK_END	the offset is set to the size of the file plus the *`offset`* (which can be negative).

For example, to determine the current offset, you can call `lseek()` with a 0 argument for *`offset`*, and a SEEK_SET argument for *`whence`*:

```
off_t current_position;
current_position=lseek(fd, 0, SEEK_CUR);
```

The `read()` Function

```
#include <unistd.h>

ssize_t read(int fd, void *buf, size_t nbytes);
                                        Returns number of bytes read if OK, -1 if error
```

The arguments:

- *fd* — file descriptor;
- *buf* — the string being read;
- *nbytes* — the number of bytes to be read.

The historical definition of the function used to be:

```
int read(int fd, char *buf, unsigned nbytes);
```

The number of bytes returned by the `read()` function may be less than the specified number *nbytes* in one of the following circumstances:

- EOF
- reading from a terminal (one line at a time)
- reading from a network
- reading from a pipe (FIFO)
- reading from tape
- when interrupted by a signal

The `write()` Function

```
#include <unistd.h>

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

Returns number of bytes written if OK, -1 if error

Arguments:

- *fd* — file descriptor;
- *buf* — the string being written;
- `nbytes` — the number of bytes to write.

For a regular file, the write starts at the current offset. If O_APPEND was on when the file was opened, the write starts at the end of the file.

To illustrate the usefulness of the `lseek()` system call, I used the idea of the library database project from a previous semester (I'm sure you all had it). The program should open and write into a text file, `books.txt`, in which each row is a simplified book record, consisting only of two fields, the author name, and the title. The fields are space-separated, and the end-of-lines have the `'\n'` EOL character.

The program inserts an 'X' at the beginning byte of a specified row. You will adapt this program for your homework.

```
#include <fcntl.h> /* for open() */
#include <sys/stat.h> /* for file access permission bits */
#include <stdio.h> /* for FILENAME_MAX */
#include <unistd.h> /* for pathconf() */
#include <string.h> /* for strcat() */
#include <stdlib.h> /* for malloc() */
#include <errno.h> /* for errno */

int getChar(int filedes);
int getBytePosition(int row, int filedes);
void putChar(int filedes, char c, off_t byte);

int main(int argc, char *argv[])
{
     off_t curr_byte;
     ssize_t num_bytes;
     char buf[200];
     int row=3;

     char *path_name="/Users/awise/Stevens/Lecture2/books.txt";
     int fd=open(path_name, O_CREAT|O_RDWR);

     /* 0 bytes from current offset to find current offset */
     curr_byte=lseek(fd, 0, SEEK_SET);
     printf("Cursor is at the %lldth byte.\n", curr_byte);

     /* 0 bytes from EOF to find the number of bytes in file */
     num_bytes=lseek(fd, 0, SEEK_END);
     printf("The file has %zd bytes.\n", num_bytes);

     /* 0 bytes from current offset to find current offset */
     curr_byte=lseek(fd, 0, SEEK_SET);
     printf("Cursor is at the %lldth byte.\n", curr_byte);

     /* Read file contents, knowing total bytes in the file */
     ssize_t read_bytes=read(fd, buf, num_bytes);
     printf("Reading %zd bytes.\n", read_bytes);
     printf("Reading --->%s<---.\n", buf);

     /* Output the contents of buffer onto terminal */
     ssize_t written_bytes=write(1, buf, read_bytes);
     printf("Writing %zd bytes.\n", written_bytes);
     printf("Writing --->%s<---.\n", buf);

     /* Get the byte # at beginning of specified row */
```

```
     off_t b=(off_t)getBytePosition(row, fd);
     printf("Byte number at row %d is %lld.\n", row, b);
     putChar(fd, '\n', b);
     putChar(fd, 'X', b+1);
return 0;
}

/* Returns the byte # at the beginning of specified row # */
int getBytePosition(int row, int filedes)
{
     printf("In getBytePosition()...\n");
     char buf[200];
     int i;
     /* Find the total number of bytes in file */
     ssize_t num_bytes=lseek(filedes, 0, SEEK_END);
     printf("num_bytes=%zd\n", num_bytes);
     /* IMPORTANT! Bring cursor back to BOF */
     lseek(filedes, 0, SEEK_SET);
     ssize_t read_bytes=read(filedes, buf, num_bytes);
     printf("read_bytes=%zd\n", read_bytes);
     int num_rows=1;
     /* Loop goes through the number of bytes in the file */
     for (i=0; i<num_bytes; i++)
     {
          printf("buf[%d]=%c\n", i, buf[i]);
          /* On EOL character, we increment the row # */
          if (buf[i]=='\n')
          {
               printf("***EOL\n");
               num_rows++;
               /* If row # equals row specified, ret. byte # */
               if (num_rows==row)
                    return i;
          }
     }
return -1;
}

/* Writes a character into a file at specified byte */
void putChar(int filedes, char c, off_t byte)
{
     lseek(filedes, byte, SEEK_SET);
     /* The write starts at the current offset */
     write(filedes, &c, 1);
}
```

Homework (due Tuesday, Feb-9-2016):

Write a C program that replaces or inserts a row in the `books.txt` file. You can specify your own row number as the row that needs to be replaced. You can supply your own record (for instance, your favorite book and its author). The `books.txt` file contains book records, each with two fields: a book title field (with no blank spaces in it) and an author field (again, with no blank spaces in it). You should be able to replace the record at the row number specified, with your own.