LAB 3: FILE HANDLING

Each running program, called a process, has several file descriptors associated with it.

When a program starts, it usually has three of these
descriptors already opened. These are:

0: Standard input
1: Standard output
2: Standard error

SYSTEM CALLS:

Write System Call:

The write system call arranges for the first nbytes bytes from buf to be written to the file associated with the file descriptor fildes. It returns the number of bytes written. This may be less than nbytes if there has been an error in the file descriptor or if the underlying device driver is sensitive to block size. If the function returns 0, it means no data was written; if it returns –1, there has been an error in the write call, and the error will be specified in the errno global variable.

Here’s the syntax:

```
#include <unistd.h>
size_t write(int fildes, const void *buf, size_t nbytes);
#include <unistd.h>
#include <stdlib.h>

int main()
{
        if ((write(1, “Here is some data\n”, 18)) != 18)
            write(2, “A write error has occurred on file descriptor 1\n”,46);
        exit(0);
}
```

This program simply prints a message to the standard output. When a program exits, all open file descriptors are automatically closed, so you don’t need to close them explicitly. This won’t be the case, however, when you’re dealing with buffered output.

```
$ ./simple_write
Here is some data
$
```

A point worth noting again is that write might report that it wrote fewer bytes than you asked it to. This is not necessarily an error. In your programs, you will need to check errno to detect errors and call write to write any remaining data.

Read System Call:

The read system call reads up to nbytes bytes of data from the file associated with the file descriptor fildes and places them in the data area buf. It returns the number of data bytes read, which may be less than the number requested. If a read call returns 0, it had nothing to read; it reached the end of the file. Again, an error on the call will cause it to return –1.

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

This program, simple_read.c, copies the first 128 bytes of the standard input to the standard output. It copies all of the input if there are fewer than 128 bytes.

```
#include <unistd.h>
#include <stdlib.h>
int main()
{
	char buffer[128];
	int nread;
	nread = read(0, buffer, 128);
		if (nread == -1)
			write(2, “A read error has occurred\n”, 26);
			if ((write(1,buffer,nread)) != nread)
				write(2, “A write error has occurred\n”,27); exit(0);
}
```

If you run the program, you should see the following:

```
$ echo hello there | ./simple_read
hello there
$ ./simple_read < draft1.txt
```

In the first execution, you create some input for the program using echo, which is piped to your program. In the second execution, you redirect input from a file. In this case, you see the first part of the file draft1.txt appearing on the standard output.

Open System Call:

To create a new file descriptor, you need to use the open system call.

```
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
int open(const char *path, int oflags);
int open(const char *path, int oflags, mode_t mode);
```

In simple terms, open establishes an access path to a file or device. If successful, it returns a file descriptor that can be used in read, write, and other system calls. The file descriptor is unique and isn’t shared by any other processes that may be running. If two programs have a file open at the same time, they maintain distinct file descriptors. If they both write to the file, they will continue to write where they left off. Their data isn’t interleaved, but one will overwrite the other. Each keeps its idea of how far into the file (the offset) it has read or written. You can prevent unwanted clashes of this sort by using file locking. The name of the file or device to be opened is passed as a parameter, path; the oflags parameter is used to specify actions to be taken on opening the file. The oflags are specified as a combination of a mandatory file access mode and other optional modes. The open call must specify one of the file access modes shown in the following table:

Mode	Description
O_RDONLY	Open for read only
O_WRONLY	Open for write only
O_RDWR	Open for reading and writing

The call may also include a combination (using a bitwise OR) of the following optional modes in the oflags parameter:

O_APPEND: Place written data at the end of the file.
O_TRUNC: Set the length of the file to zero, discarding existing contents.
O_CREAT: Creates the file, if necessary, with permissions given in mode.
O_EXCL: Used with O_CREAT, ensures that the caller creates the file. The open is atomic; that is, it's performed with just one function call. This protects against two programs creating the file at the same time. If the file already exists, open will fail.

Other possible values for oflags are documented in the open manual page, which you can find in section 2 of the manual pages (use man 2 open). Open returns the new file descriptor (always a nonnegative integer) if successful, or –1 if it fails, at which time open also sets the global variable errno to indicate the reason for the failure. We look at errno more closely in a later section. The new file descriptor is always the lowest-numbered unused descriptor, a feature that can be quite useful in some circumstances. For example, if a program closes its standard output and then calls open again, the file descriptor 1 will be reused and the standard output will have been effectively redirected to a different file or device.

Initial Permission

When you create a file using the O_CREAT flag with open, you must use the three parameter form. mode, the third parameter, is made from a bitwise OR of the flags defined in the header file sys/stat.h.
These are:
S_IRUSR: Read permission, owner
S_IWUSR: Write permission, owner
S_IXUSR: Execute permission, owner
S_IRGRP: Read permission, group
S_IWGRP: Write permission, group
S_IXGRP: Execute permission, group
S_IROTH: Read permission, others
S_IWOTH: Write permission, others
S_IXOTH: Execute permission, others

For example,
open (“myfile”, O_CREAT, S_IRUSR|S_IXOTH);

Unmask system call

The umask is a system variable that encodes a mask for file permissions to be used when a file is created. You can change the variable by executing the umask command to supply a new value. The value is a three-digit octal value. Each digit is the result of ORing values from 1, 2, or 4; the meanings are shown in the following table. The separate digits refer to “user,” “group,” and “other” permissions, respectively.

Digit	Value	Meaning
1	0	No user permissions are to be disallowed.
4		User read permission is disallowed.
2		User write permission is disallowed.
1		User execute permission is disallowed.
2	0	No group permissions are to be disallowed.
4		Group read permission is disallowed.
2		Group write permission is disallowed.
1		Group execute permission is disallowed.
3	0	No other permissions are to be disallowed.
4		Other read permission is disallowed.
2		Other write permission is disallowed.
1		Other execute permission is disallowed.

Close System Call

You use close to terminate the association between a file descriptor, fildes, and its file. The file descriptor becomes available for reuse. It returns 0 if successful and –1 on error.

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

Note that it can be important to check the return result from close. Some file systems, particularly networked ones, may not report an error writing to a file until the file is closed, because data may not have been confirmed as written when writes are performed.

A File Copy Program

```
#include <unistd.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdlib.h>
int main()
{
    char c;
    int in, out;
    in = open(“file.in”, O_RDONLY);
    out = open(“file.out”, O_WRONLY|O_CREAT, S_IRUSR|S_IWUSR);
    while(read(in,&c,1) == 1)
        write(out,&c,1);
    exit(0);
}
```

WITH fcntl.h how the system calls can manipulate the files.

I/O SYSTEM CALLS

I/O through system calls is simpler and operates at a lower level than making calls to the C file-I/O library.

There are seven fundamental file-I/O system calls:
creat() - Create a file for reading or writing.
open() - Open a file for reading or writing.
close() - Close a file after reading or writing.
unlink()- Delete a file.

write() - Write bytes to file.
read() - Read bytes from file.

The creat() System Call

The "creat()" system call creates a file.
It has the syntax: int fp; /* fp is the file descriptor variable */

```
fp = creat( <filename>, <protection bits> );
```

Ex: fp=creat(“students.dat”,RD_WR);

This system call returns an integer, called a "file descriptor", which is a number that identifies the file generated by "creat()". This number is used by other system calls in the program to access the file. Should the "creat()" call encounter an error, it will return a file descriptor value of -1.

The "filename" parameter gives the desired filename for the new file.

The "permission bits" give the "access rights" to the file. A file has three "permissions" associated with it:

Write permission - Allows data to be written to the file.
Read permission - Allows data to be read from the file.
Execute permission - Designates that the file is a program that can be run.

These permissions can be set for three different levels:

User level: Permissions apply to individual user.
Group level: Permissions apply to members of user's defined "group".
System level: Permissions apply to everyone on the system

The open() System Call

The "open()" system call opens an existing file for reading or writing.
It has the syntax:

```
<file descriptor variable> = open( <filename>, <access mode> );
```

The "open()" call is similar to the "creat()" call in that it returns a file descriptor for the given file, and returns a file descriptor of -1 if it encounters an error. However, the second parameter is an "access mode", not a permission code.

There are three modes (defined in the "fcntl.h" header file):

O_RDONLY Open for reading only.
O_WRONLY Open for writing only.
O_RDWR Open for reading and writing

For example, to open "data" for writing, assuming that the file had been created by another program, the following statements would be used:

```
int fd;
fd = open( "students.dat", O_WRONLY );
```

A few additional comments before proceeding:
A "creat()" call implies an "open()". There is no need to "creat()" a file and then "open()" it.

The close() System Call

The "close()" system call is very simple. All it does is "close()" an open file when there is no further need to access it.

The "close()" system call has the syntax:

```
close( <file descriptor> );
```

The "close()" call returns a value of 0 if it succeeds, and returns -1 if it encounters an error.

The write() System Call

The "write()" system call writes data to an open file.
It has the syntax:

```
write( <file descriptor>, <buffer>, <buffer length> );
```

The file descriptor is returned by a "creat()" or "open()" system call. The "buffer" is a pointer to a variable or an array that contains the data; and the "buffer length" gives the number of bytes to be written into the file.

While different data types may have different byte lengths on different systems, the "sizeof()" statement can be used to provide the proper buffer length in bytes.

A "write()" call could be specified as follows:

```
float array[10];
write( fd, array, sizeof( array ) );
```

The "write()" function returns the number of bytes it writes. It will return -1 on an error.

The read() Sytem Call

The "read()" system call reads data from an open file.
Its syntax is the same as that of the "write()" call:

```
read( <file descriptor>, <buffer>, <buffer length> );
```

The "read()" function returns the number of bytes it returns. At the end of the file, it returns 0 or returns -1 on error.

lseek

The lseek system call sets the read/write pointer of a file descriptor, fildes; that is, we can use it to set wherein the file the next read or write will occur. We can set the pointer to an absolute location in the file or a position relative to the current position or the end of file.

```
#include <unistd.h>
#include <sys/types.h>
off_t lseek(int fildes, off_t offset, int whence);
```

The offset parameter is used to specify the position, and the whence parameter specifies how the offset is used. whence can be one of the following:

SEEK_SET: offset is an absolute position
SEEK_CUR: offset is relative to the current position
SEEK_END: offset is relative to the end of the file

lseek returns the offset measured in bytes from the beginning of the file that the file pointer is set to or -1 on failure. The type off_t, used for the offset in seek operations, is an implementation-dependent type defined in sys/types.h.

Errors:

EACCES Permission denied.
EMFILE Too many file descriptors in use by the process.
ENFILE Too many files are currently open in the system.
ENOENT Directory does not exist, or the name is an empty string.
ENOMEM Insufficient memory to complete the operation.

Lab Exercises:

1. Write a program to print the lines of a file that contain a word given as the program argument (a simple version of grep UNIX utility).
2. Write a program to list the files given as arguments, stopping every 20 lines until a key is hit. (a simple version of more UNIX utility)
3. Demonstrate the use of different conversion specifiers and resulting output to
allow the items to be printed.
4. Write a program to copy character-by character copy is accomplished using calls to the functions referenced in stdio.h.

Additional Exercises:

1. Write a program that shows the user all his/her C source programs and then prompts interactively as to whether others should be granted read permission; if affirmative such permission should be granted.
2. Use lseek() to copy different parts (initial, middle and last) of the file to others. (For lseek() refer to man pages)