Lecture Notes for Lecture 10 of CS 5600 (Computer Systems) for the Fall 2019 session at the Northeastern University Silicon Valley Campus.

Input and Output Operations

Philip Gust,
Clinical Instructor
Department of Computer Science

Part two of a two-part lecture on files, directories, and input/output.
These slide highlight content from suggested readings.

Lecture 9 Review

- In this lecture, we learned about disks, which are used to store information in a computer system. We saw that a disk is divided into platters, and platters into track and sectors.
- We saw how a block of information is stored in a sector of a track on a platter of a disk. Each block has a logical address that is numbered sequentially on a disk.
- Next, we learned about the files system, which organizes files into inodes that contain information about the file, and a list of blocks where the file information is stored.
- We also learned about a special kind of file called a directory, which names files and organizes them into a hierarchically named directory system.
- Finally, we saw how to mange files and directories in C through calls to the operating system

Input and Output Operations

Outline

- In this lecture, we will discuss different ways to access the information from input and output devices using C I/O functions.
- The C language provides a low-level *unbuffered I/O* functions that use *file ids* to write and read information directly to and from I/O devices. Each function is a call to the operating system.
- A higher-level *buffered I/O* functions use *streams* to buffer information within a program before writing or reading it to or from I/O devices to improve efficiency of I/O operations.
- Character-oriented stream I/O functions support reading and writing sequences of characters. Block-oriented stream I/O functions support transferring blocks of data to and from I/O devices.
- Finally, we will see how to access information in specific parts of a file using random-access I/O functions.

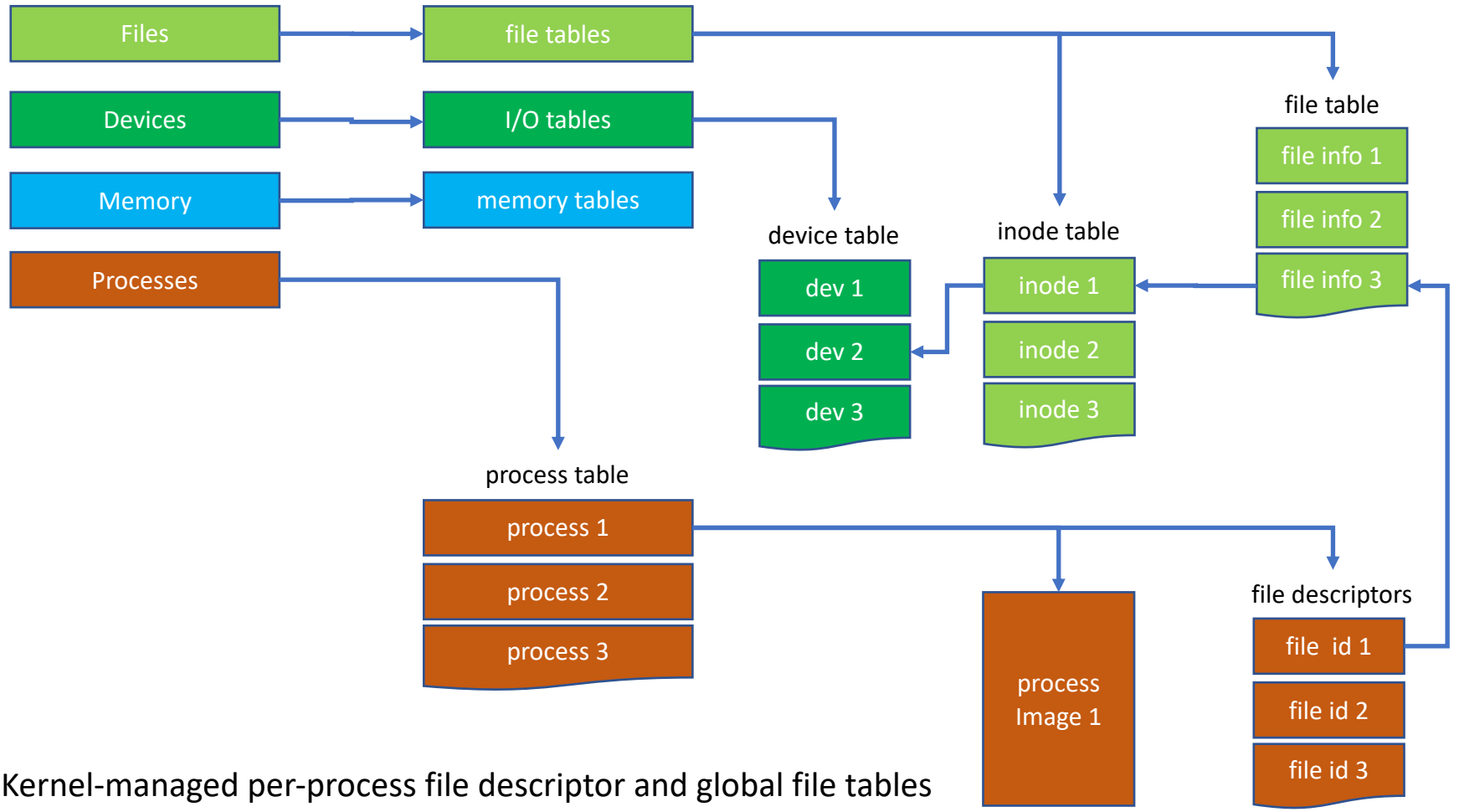
Input and Output Operations

Unbuffered Input and Output

- The operating system provides a per-process *file descriptor table* that uses an an integer file descriptor index into a system table of files opened by all processes.
- The *file table* records the mode that the file has been opened, including reading, writing, and appending. The file table in turn indexes into the file system inode table.
- Unbuffered I/O functions use file descriptors to read and write directly from the file system, block and character devices, and other objects such as networks that do not normally exist in the file system.

Input and Output Operations

Unbuffered Input and Output



Kernel-managed per-process file descriptor and global file tables

Input and Output Operations

Unbuffered Input and Output

- The file table entries maintain information about each open file in the file system. This information is used by the non-buffered file I/O operations.
- Here is a typical content of a file information record:

field	description
inode	the inode for the file
flags	open mode flags (read-only, write-only, read-write)
offset	current read/write offset
block id	current block id

Input and Output Operations

Unbuffered Input and Output

- C automatically creates three file descriptors associated with standard input (0), standard output (1), and standard error (2). The unbuffered I/O functions include:

function	description
open(path, flag, mode)	open or create a file specified by the path using flags and mode to determine the operation.
creat(path, mode)	uses open() with specific flags to create a file
read(fd, buf, nbytes)	read the next nbytes from fd into buffer and update file pointer
pread(fd, buf, nbytes, offset)	read the next nbytes from fd into buffer starting at offset without updating file pointer
write(fd, buf, nbytes)	write nbytes from buffer to fd and update file pointer
pwrite(fd, buf, nbytes, offset)	write the next nbytes from buffer to fd,starting at offset without updating file pointer
stat(path, statbuf) fstat(fd, statbuf)	returns file info in struct stat buffer
close(fd)	closes the open file

Input and Output Operations

Unbuffered Input and Output

- Here is an outline of the read(fd, buf, nbytes) function:

```
int read(fd, buf, nbytes) {
    get entry for fd;
    if (no entry or entry not open for reading) {
        set errno to EBADF and return -1;
    }
    set bytes read to 0
    adjust nbytes to bytes available in entry inode
    set block offset from current entry offset
```

Input and Output Operations

Unbuffered Input and Output

- Here is an outline of the read(fd, buf, nbytes) function:

```
while (bytes read < nbytes) {
    if entry offset not within entry current block {
        get block no. for offset from entry inode
        update entry current block no.
        set block offset to start of current block
    }
    read entry current block
    if (error reading block) {
        set errno to EIO and return -1;
    }
    transfer available bytes from block offset to buffer
    update bytes read
    update entry offset
}
return bytes read
}
```

Input and Output Operations

Unbuffered Input and Output

- Here is an outline of the close(fd, buf, nbytes) function:

```
int read(fd) {
    get entry for fd;
    set bytes read to 0
    adjust nbytes to bytes available in entry inode
    while (bytes read < nbytes) {
        if entry offset not within entry current block {
            get block no. for offset from entry inode
            update entry current block no.
        }
        read entry current block
        transfer available bytes to buffer
        update bytes read
        update entry offset
    }
    return bytes read
}
```

Input and Output Operations

Unbuffered Input and Output

- Here is a function that uses the unbuffered I/O functions to copy an input file to an output file.

```
#include <sys/stat.h>
#include <unistd.h>
#include <stdio.h>

long copyUnbuffered(const char* infile, const char* outfile) {
   // open infile read-only
   int infd =  open(infile, O_RDONLY);
   if (infd < 0) {
      perror(infile);
      return -1;
   }
```

Input and Output Operations

Unbuffered Input and Output

- Here is a function that uses the unbuffered I/O functions to copy an input file to an output file.

```
// create file write-only with rw- r-- r-- permission
mode_t mode = S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH;
int outfd = open(outfile, O_WRONLY | O_CREAT |  O_TRUNC, mode);
if (outfd < 0) {   // create file if does not exist
   perror(outfile);
   close(infd);
   return -1;
}
```

Input and Output Operations

Unbuffered Input and Output

- Here is a function that uses the unbuffered I/O functions to copy an input file to an output file.

```
 // copy bytes from infile to outfile
unsigned char bytes[128];
long count = 0;
size_t nread;
size_t nwrite = 0;
while ((nwrite >= 0) && (nread = read(infd, bytes, sizeof(bytes))) > 0) {
   unsigned char *bp = bytes;
   while ((nread > 0) && (nwrite = write(outfd, bp, nread)) >= 0) {
      nread -= nwrite;
      bp += nwrite;
      count += nwrite;
   }
}
```

Input and Output Operations

Unbuffered Input and Output

- Here is a function that uses the unbuffered I/O functions to copy an input file to an output file.

```
    // report errors
    if (nread < 0) {
        perror(infile);
    }
    if (nwrite < 0) {
        perror(outfile);
    }

    close(infd);
    close(outfd);
    return count;
}
```

Input and Output Operations

Unbuffered Input and Output

- Flags for open()
  - Must specify flag for how file will be opened:

```
#define O_RDONLY    0x0000    /* open for reading only */
#define O_WRONLY    0x0001    /* open for writing only */
#define O_RDWR      0x0002    /* open for read and write */
```

Input and Output Operations

Unbuffered Input and Output

- Flags for open()
  - If file does not yet exist, must OR in flag to create the file

    ```
    #define O_CREAT         0x0200          /* create if nonexistant */
    ```

  - If file already exists, can OR in flag to cause error, append contents, or truncate to 0 length
    - #define O_APPEND 0x0008 /* set append mode */
    - #define O_TRUNC 0x0400 /* truncate to zero length */
    - #define O_EXCL 0x0800 /* error if already exists */

Input and Output Operations

Unbuffered Input and Output

- Flags for open()
  - Examples:

```
// open existing file for read-only, error if file does not exist
int oflags = O_RDONLY;

// open for write-only; if file does not exist, create it,
// else truncate file to 0-length
int oflags = O_WRONLY | O_CREAT |  O_TRUNC;

// open for write-only; if file does not exist, create it,
// else append to existing file */
int oflags = O_WRONLY | O_CREAT |  O_APPEND;

// open for write-only; if file does not exist, create it,
// else flag error if file already exists */
int oflags = O_WRONLY | O_CREAT |  O_EXCL;
```

Input and Output Operations

Unbuffered Input and Output

- Permission modes for open()
  - When creating file, must specify permissions by OR-ing together combination of bit masks:

```
/* Read, write, execute/search by user */
#define S_IRWXU   0000700        /* [XSI] RWX mask for owner */
#define S_IRUSR   0000400        /* [XSI] R for owner */
#define S_IWUSR   0000200        /* [XSI] W for owner */
#define S_IXUSR   0000100        /* [XSI] X for owner */
```

Input and Output Operations

Unbuffered Input and Output

- Permission modes for open()
  - When creating file, must specify permissions by OR-ing together combination of bit masks:

```
/* Read, write, execute/search by group */
#define S_IRWXG    0000070        /* [XSI] RWX mask for group */
#define S_IRGRP    0000040        /* [XSI] R for group */
#define S_IWGRP    0000020        /* [XSI] W for group */
#define S_IXGRP    0000010        /* [XSI] X for group */
```

Input and Output Operations

Unbuffered Input and Output

- Permission modes for open()
  - When creating file, must specify permissions by OR-ing together combination of bit masks:

```
/* Read, write, execute/search by others */
#define S_IRWXO    0000007        /* [XSI] RWX mask for other */
#define S_IROTH    0000004        /* [XSI] R for other */
#define S_IWOTH    0000002        /* [XSI] W for other */
#define S_IXOTH    0000001        /* [XSI] X for other */
```

Input and Output Operations

Unbuffered Input and Output

- Permission modes for open()
  - Examples:

```
// 0755 or “rwx  r-x  r-x”
mode_t mode = S_IRWXU | S_IRGRP | S_IX_GRP | S_IROTH | S_IXOTH;

// 0644 or “rw-  r--  r--”
mode_t mode = S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH;
```

Input and Output Operations

Buffered Input and Output

- *Streams* are an abstraction to interact with I/O devices in a uniform way. All streams have similar properties independent of the individual characteristics of the physical media they are associated with.
- Streams are implemented by the C I/O libraries and are not directly supported by the operating system. Instead, streams use file descriptors and call the unbuffered C I/O functions.
- Streams provide buffering of I/O operations to improve efficiency and enable operations that require multiple bytes of input to be available.

Input and Output Operations

Buffered Input and Output

- A stream is identified by a struct FILE that encapsulates the underlying file id, and any state information related to buffering of input or output. It is an opaque type containing stream information:
  - platform-specific identifier of associated I/O device, such as a file descriptor
  - the buffer
  - stream orientation indicator (unset, narrow, or wide)
  - stream buffering state indicator (unbuffered, line buffered, fully buffered)
  - I/O mode indicator (input stream, output stream, or update stream)
  - binary/text mode indicator
  - end-of-file indicator
  - error indicator
  - the current stream position and multibyte conversion state
  - reentrant lock (required as of C11)

Input and Output Operations

Buffered Input and Output

- C automatically creates three FILE streams associated with standard input (*stdin*), standard output (*stdout*), and standard error (*stderr*). The C buffered I/O file access functions include:

function	description
fopen(path, mode)	open or create a stream specified by the path using mode to determine the operation.
fdopen(fd, mode)	associates a stream with the existing file descriptor
fflush(stream)	synchronizes output stream with actual file
fclose(stream)	closes the open stream
feof(stream)	returns non-zero if stream at end of file, 0 otherwise
fileno(stream)	returns file descriptor associated with stream
setvbuf(stream, buf, type, size)	sets buffering mode for stream
tmpfile(void)	returns stream to a temporary file in TMPDIR, which is unlinked before this function returns

Input and Output Operations

Buffered Input and Output

- The C buffered unformatted I/O functions include:

function	description
fgetc(stream) getc(stream)	reads a byte from the input stream
getchar()	reads a byte from stdin
fgets(str, size, stream)	reads at most one less than size characters from input stream to string
fputc(c, stream) putc(c, stream)	writes a byte to the output stream
putchar(c)	writes a byte to stdout
fputs(str, stream)	writes a string to the output stream
putw(w, stream)	write an int word to the output stream
ungetc()	puts a byte back into an input stream

Input and Output Operations

Buffered Input and Output

- The C buffered formatted I/O functions include:

function	description
scanf(fmt, …) fscanf(stream, fmt, …) sscanf(str, stream, fmt, …)	reads formatted chars from stdin, stream, or string
vscanf(fmt, va_list) vfscanf(stream, fmt, va_list) vsscanf(stream, str, fmt, va_list)	Reads formatted input chars from stdin, stream , or string using variable argument list
printf(fmt, …) fprintf(stream, fmt, …) sprintf(str, stream, fmt, …)	prints formatted chars to stdout, stream, or string buffer
vprintf(fmt, va_list) vfprintf(stream, fmt, va_list) vsprintf(str, stream, fmt, va_list)	prints formatted chars to stdout, stream, or string buffer using variable argument list
perror(str)	writes description of current error to stderr

Input and Output Operations

Buffered Input and Output

- Here is a function that uses buffered I/O to copy an input file to an output file

```
#include <stdio.h>

long copyBuffered(const char* infile, const char* outfile) {
   FILE *fp1, *fp2;
   if ((fp1 = fopen(infile,"rb")) == NULL) {  // read-only, must exist
      printf("\nInput file cannot be opened");
      return -1;
   }
   if ((fp2 = fopen(outfile, "wb")) == NULL) {  // write only, truncate
      printf("\nOutput file cannot be opened");
      fclose(fp1);
      return -1;
   }
```

Input and Output Operations

Buffered Input and Output

- Here is a function that uses buffered I/O to copy an input file to an output file

```
// copy bytes from infile to outfile
int ch;
long count = 0;
while ((ch = fgetc(fp1)) != EOF) {
   if (fputc(ch, fp2) == EOF) {
      break;
   }
   count++;
}
fclose(fp1);
fclose(fp2);
return count;
}
```

Input and Output Operations

Buffered Input and Output

- Buffered I/O fopen() mode flags
  - "r" read: Open file for input operations. The file must exist.
  - "w" write: Create an empty file for output operations. If a file with the same name already exists, its contents are discarded and the file is treated as a new empty file.
  - "a" append: Open file for output at the end of a file. Output operations always write data at the end of the file, expanding it.

    Repositioning operations (fseek, fsetpos, rewind) are ignored. The file is created if it does not exist.

Input and Output Operations

Buffered Input and Output

- Buffered I/O fopen() mode flags
  - "r+" read/update: Open a file for update (both for input and output). The file must exist.
  - "w+" write/update: Create an empty file and open it for update (both for input and output). If a file with the same name already exists its contents are discarded and the file is treated as a new empty file.
  - "a+" append/update: Open a file for update (both for input and output) with all output operations writing data at the end of the file.

    Repositioning operations (fseek, fsetpos, rewind) affects the next input operations, but output operations move the position back to the end of file. The file is created if it does not exist.

Input and Output Operations

Buffered Input and Output

- Buffered I/O fopen() mode flags
  - "b" binary mode: With the mode specifiers above the file is open as a text file. In order to open a file as a binary file, a "b" character has to be included in the mode string.

    This additional "b" character can either be appended at the end of the string (thus making the following compound modes: "rb", "wb", "ab", "r+b", "w+b", "a+b") or be inserted between the letter and the "+" sign for the mixed modes ("rb+", "wb+", "ab+").

Input and Output Operations

Buffered Input and Output

- Text vs. binary mode I/O
  - Text mode I/O attempts to handle differences in newline and end-of-file representation between operating systems (e.g. MS Windows)
  - Open file in text mode
    - fopen(infile,"r") // open existing read-only text file
    - fopen(outfile, “w”) // create, or open and truncate write-only text file
    - Optional “t” supported to explicitly specify text mode (e.g. “rt”, “wt”)

Input and Output Operations

Buffered Input and Output

- Text vs. binary mode I/O
  - Binary mode I/O reads bytes without additional processing
  - Open file in binary mode
    - fopen(infile,"rb") // open existing read-only binary file
    - fopen(outfile, “wb”) // create, or open & truncate write-only binary file
  - Note: On many systems (e.g. Unix/Linux/MacOS) there is no special processing for text, so “r”, “rt”, and “rb” do the same thing, but it is best to specify the mode to ensure portability across operating systems.

Input and Output Operations

Buffered Input and Output

- How content is buffered depends on the buffering mode, which can be set using the *setvbuf(stream, buf, type, size)* function.
- Three types of buffering are available: unbuffered, block buffered, and line buffered.
  - When a stream is *unbuffered*, bytes are intended to appear from the source or at the destination as soon as possible. Otherwise bytes may be accumulated and transmitted as a block.
  - When a stream is *fully buffered*, bytes are intended to be transmitted as a block when a buffer is filled.
  - When a stream is *line buffered*, bytes are intended to be transmitted as a block when a newline byte is encountered.

Input and Output Operations

Buffered Input and Output

- Normally, all files are block buffered. When the first I/O operation occurs on a file, malloc(3) is called and an optimally-sized buffer is obtained.
- If a stream refers to a terminal (as stdout normally does), it is line buffered. The standard error stream stderr is always unbuffered.
- The type argument can be set using one of the following macros:
  - _IONBF unbuffered
  - _IOLBF line buffered
  - _IOFBF fully buffered

Input and Output Operations

Buffered Input and Output

- Buffered Block I/O
  - Block/IO reads and writes blocks of a type.
  - Useful for transferring arrays of a type to and from disk.
  - Type can be a primitive type or a struct
  - Avoid using with pointer types or fields

function	description
fread(buf, size, nitems, stream)	reads nitems objects, each size bytes long, from the stream into the buffer
fwrite(buf, size, nitems, stream)	writes nitems objects, each size bytes long, from the buffer to the stream

Input and Output Operations

Buffered Input and Output

- Buffered Block I/O
  - Example:

```
#include <stdio.h>
FILE* fileptr = fopen(fname, “w+b”);
// write array to file
int a[5] = {1, 2, 3, 4, 5);
size_t nwritten = fwrite(a, sizeof(int), 5, fileptr);
```

Input and Output Operations

Buffered Input and Output

- Block I/O
  - Example:

```
// read file into second array
int b[5];
rewind(fileptr);  // move fileptr back to start of file
size_t nread = fread(b, sizeof(int), 5, fileptr);

// compare the two arrays
for (size_t i = 0; i < size; i++) {
    if (a[i] != b[i]) {
        printf("Written and read array differ at position %lu\n", i);
        return;
    }
}
printf("Output and input arrays are the same\n");
```

Input and Output Operations

Buffered Input and Output

- Random access I/O
  - Allows files to be accessed randomly rather than sequentially.
  - Move to any point in the file to begin reading or writing
  - Useful for implementing structured files and databases

function	description
fseek(stream, offset, whence)	seek to location in file. Offset can be negative. Values of whence are: SEEK_SET: from beginning of the file SEEK_CUR: from current point in the file SEEK_END: from end of the file
rewind(stream)	sets location to beginning of the file
ftell(stream)	returns current file position

Input and Output Operations

Buffered Input and Output

- Random access I/O
  - Example: Block I/O using random access I/O to modify file value:

```
#include <stdio.h>

FILE* fileptr = fopen(fname, “w+b);
// write array to file
int a[5] = {1, 2, 3, 4, 5);
size_t nwritten = fwrite(a, sizeof(int), 5, fileptr);
printf("fileptr at position: %lu after writing array\n", ftell(fileptr));

// update value at position 2 to “-3”
fseek(fileptr, 2*sizeof(int), SEEK_SET); // position before position 2
int newval = -3;
fwrite( &newval, sizeof(int), 1, fileptr);
```

Input and Output Operations

Buffered Input and Output

- Random access I/O
  - Example: Block I/O using random access I/O to modify file value

```
// read file into second array
int b[5];
rewind(fileptr);  // move fileptr back to start of file
size_t nread = fread(b, sizeof(int), 5, fileptr);
 // compare the two arrays
for (size_t i = 0; i < size; i++) {
    if (a[i] != b[i]) {
        printf("Written and read array differ at position %lu\n", i);
        return;
    }
}
printf("Output and input arrays are the same\n");
```