

Record I/O

CS3411 Fall 2017

Program Three

Due: November 7, 2017, AoE

Unix abstraction of files as byte streams is powerful and versatile. However, there are times having a file implementation which is aware of *records* would simplify developing certain types of programs. A record based file i/o operates on units of records which can be fixed or varying length. Although the Unix abstraction of byte streams work well with fixed length records, dealing with variable length records is notably more difficult. A typical example would be the need to read and write a given file line at a time where the length of each line varies.

In this assignment, our goal is to create a set of abstractions imitating regular open/read/write/lseek/close kernel calls but implement record based I/O semantics. The abstractions and their parameters are listed later in this description.

Representing Record Files

The easiest way to represent record files would be to encode the length of each record in the file. However, this approach would make the file contents inaccessible to other Unix utilities. Instead, we will create an *index* file which accompanies the data file and store the record length and position information in the index file. Each record in the data file will then be represented using a *record descriptor* with the following layout:

```
struct
{
int position;
int length;
} record_descriptor;
```

The index file is a **binary file** that is read and written by performing a kernel i/o for the size of the structure:

```
record_descriptor r;

r.position = record_position;
r.length = strlen(buf);

n = write(inx, &r, sizeof(record_descriptor));
```

The index file for record i/o is named by following the convention `.rinx.<data file name>` (i.e., if the data file is named `myfile.txt`, its index will be `.rinx.myfile.txt`). This convention conveniently hides the index files while permitting easy access to them based on the data file name. As an example, assume the following program fragment is executed using the abstractions you have created. Here, `all_hands_abandon_ship` is an error routine that prints an error message to the user and terminates the program. Note, how the return values from the new abstractions match with what would have been returned from the corresponding kernel call.

```
char * record;

if (( d = rio_open("myfile.txt", O_RDWR | O_CREAT, 0600)) < 0) all_hands_abandon_ship(RIO_OPEN);
if (( n = rio_write(d,"Systems\n",8)) != 8) all_hands_abandon_ship(RIO_ERROR_WRITE);
if (( n = rio_write(d,"programming is cool\n",20)) != 20) all_hands_abandon_ship(RIO_ERROR_WRITE);
if (rio_close(d) < 0) all_hands_abandon_ship(RIO_CLOSE);

if (( d = rio_open("myfile.txt", O_RDONLY, 0600)) < 0) all_hands_abandon_ship(RIO_OPEN);
if (( n = rio_lseek(d,1, SEEK_SET)) != 1) all_hands_abandon_ship(RIO_ERROR_LSEEK);

record = rio_read(d, &n);
if (n == 0) { fprintf(stderr,"Unexpected EOF on record i/o\n"); exit(-1);}
if (n < 0) all_hands_abandon_ship(RIO_ERROR_READ);
write(1,record, n);
free(record);
```

If there are no errors, this program should print *programming is cool* as `rio_lseek` seeks the file to the beginning of the second record in the file (`rio_lseek(d,1, SEEK_SET)` is zero relative). The contents of the resulting file is shown below. Note the index file contents are shown in decimal and comma separated for readability:

Data File	Index file
-----	-----
Systems	0, 8
programming is cool.	8, 20

Simplifications and Notes

In order to complete the project at a reasonable time, you can make the simplification that if the file is not at the end of file and a write is performed, the write length should be less than or equal to the length of the existing record in the file. In this case, you should write the data to the file and update the corresponding index record with the new length. Your developed abstractions should permit multiple record files to be opened and manipulated.

Semantics of Record i/o Routines

int rio_open(const char *pathname, int flags, mode_t mode)

If create is requested, open both the data file and the index file with O_CREAT. If O_RDONLY or O_RDWR is requested, make sure that the index file is present and return an error otherwise.

void * rio_read(int fd, int * return_value)

Allocate a buffer large enough to hold the requested buffer, read the next record and return the pointer to the allocated area. The I/O result should be returned through the return_value argument. It is the responsibility of the caller to free the allocated buffer.

int write(int fd, const void *buf, int count)

If appending to the file, create a record_descriptor and fill-in the values. Write the descriptor to the index file and the supplied data to the data file for the requested length. If updating a record, read the record descriptor, check to see if the new record fits in the allocated area and rewrite. Return an error otherwise.

int rio_lseek(int fd, int offset, int whence)

Seek both files to the beginning of the requested record so that the next I/O is performed at the requested position. *whence* assumes the same values as lseek whence argument.

int rio_close(int fd)

Close both files. Even though a single descriptor is passed along, your abstraction must close the other file as well. It is suggested that you return the descriptor obtained by opening the data file to the user and keep the index file descriptor number in the abstraction and associate them.

Submission

Your submission must be written in C.

1. Develop the abstraction into a single C file which is titled recordio.c and a prototype file recordio.h. A user who wishes to use your abstraction should be able to do so by including recordio.h in her file and linking with recordio.o.
2. Develop a single standalone program called *indexer* which creates an index file for a text file whose title is supplied as an argument to the program. This program should read the text file from beginning till end, find the beginning and ending of each line (ending with the newline character), create a descriptor and write the descriptor to the created index file. The program should not modify the text file which is supplied as an argument.
3. Develop a test program called testio.c which includes recordio.h and is linked against recordio.o. This program should expect a single argument which supplies a file name. The program should rio_open the supplied argument and report any errors to the user. Upon a successful open it should execute a series of rio_read statements, read the file record at a time and write each record to the standard output as shown above.
4. Test your testio.c and indexer. Use indexer to index itself (i.e., indexer.c) and use testio to display the file one line at a time.

Use Canvas to submit a tar file named prog3.tgz that contains:

1. A copy of the source files recordio.c recordio.h indexer.c testio.c **with comments**.
2. A makefile with *all*, *indexer*, and *testio*.
3. A README file which contains a description of what works and what not.

When I execute the commands: `gtar xzf prog3.tgz; make` against your submission, two executables named `indexer` and `testio` should be created in the current directory. No hardcopy submission is required.