Operating Systems, Security and Networks (207SE)
Lab 8: **Cache Library exercise**

This session we will explore an application that is a more elegant version of the last tutorial. It does this by using a structure that simulates a cache library.

Setting up the activities

Getting to required file

- Run MobaXterm and connect to the server
- Drag the zip file into MobaXterm or use filezilla to ftp the zip file to the server
- unzip cache-example.zip
- cd cache-example

The files you will download are:

cache_example.c - Main program, makes use of the library

cache_reader.c – The code for the library
cache_reader.h – The header file for the library

Makefile – this is what's used by the make system when you type make

text – a file with some text in it, which we are going to open and read.

Background Information

The cache_reader.* files make up a 'caching fixed length buffered file reader library'. The idea is that this library will provide some simple data buffering, using the same idea as a hard disk reading a track's worth of data (given a sector read request). Remember from the lecture - the hard disk reads more data than it has been asked to, putting the rest in the buffer, under the assumption that the next read is going to require the data which comes directly after the recently requested blocks/sectors on the media.

Basically, the cache_reader library is used to open a certain file (which you specify), and it will read a certain amount of the file's contents into a buffer referenced in the cr_file struct. You can request bytes from this buffer, to do whatever you wish with (e.g. print them out to the screen!), as shown in the cache_example.c file.

The API for cache_reader.c code is shown in cache_reader.h, but a brief explanation of each function is shown below:

cr_file* cr_open(char* filename, int buffersize);
This function's job is to open a file on the system, and initialise the buffer. It returns a pointer to a cr_file structure, which you can access using the -> notation. It accepts a file name as a string, and an integer representing the size of the buffer which should be used to buffer the file data.

void cr_close(cr_file *f);
This function closes the file which cr_file opened. It accepts a pointer to the cr_file structure which needs to be closed, but returns no value.

char cr_read_byte(cr_file* f);
This function requests a byte from the file represented by the cr_file pointer. It returns the next byte in that structure's buffer (Note that a char **is** one byte), and moves its current position in the buffer

along by one. When the buffer is empty, the refill function is called (this is not intended for use outside of the cache_reader.c file. Repeated calls to cr_read_byte should return successive bytes from the relevant cr_file's buffer.

int refill(cr_file* buff);
This function will refill a cr_file structure's buffer and change the position back to the start of the buffer. It accepts as an argument the pointer to a cr_file structure, and returns the amount of bytes read.

In the cache-example program, we simply use the caching library to open a file for us, and start reading data from the buffer provided by the caching library. When we receive a byte that represents the EOF (End Of File) constant (usually -1, but you can check with a simple c program) we know all the contents of the file have been read, and the program can exit.

Points(!) to remember

The buffer in the cr_file structure is represented by a pointer, not an array (as you might have expected if you've never seen this before). The memory address that this variable points to contains the value at the START of your buffer. In order to access elements after the memory start, you need to add an OFFSET to the value of the pointer (**hint:** the offset will be something to do with the usedBuffer variable).

If pointers are confusing at all to you, I would recommend you do some research as part of this minimum work (at the very least, make sure you've read the links we've provided for you), and experiment with things you're not familiar with.

Basic Portfolio Activity

At the moment the program will compile and run, but there will be no output. This is because the cr_read_byte function in cache_reader.c is incomplete. As per the description above, the cr_read_byte function needs to return the next byte in the buffer each time it is called (the current position in the buffer is given by the usedBuffer variable) as well as change the current position of usedBuffer.

cr_read_byte also needs to check whether usedBuffer is at the end of the buffer (by comparing it with the bufferlength specified in the cr_file structure. This is important, because in C you can walk off the edge of a data structure and start corrupting other memory (within reason – remember protection?). This will give you unpredictable results in your program.

It's up to you to read the code, understand what's going on, and implement cr_read_byte. Start with cache_example.c and understand what's happening here, then move on to the code behind the calls that cache_example is making in cache_reader.c. Hopefully then, the actual code required to complete the cr_read_byte function will be simple.

For your portfolio, you need to show the code you have written, and explain how it works in the context of the library (cache_reader.*). You need to demonstrate that you understand what is going on in the library's code and also that your code works correctly. If you have time, please try to attempt the further questions, as they are a relatively simple extension of your understanding of this minimum work.

The correct output from the program should be the contents of the 'text' file!

For writing the cr_read_byte function you will receive 3 out of 5.

Additional Portfolio Activity

1. How can you prove the file output is being buffered, given the output to the screen is essentially the contents of the file. Devise some way to show that each byte is being read, and when the buffer is being refilled. This may require (nominal) changes to the main program and the library code.

2. Provide some statistics in the cache_example.c code to show how many bytes were read in total, how many words were read in and how many times the buffer was refilled.

Complete these two activities in addition to the basic portfolio activity to get 5 out of 5.