ENCE360 Lab 5: Files and Performance Optimisation

File copying, mmap and resizable buffers

Objectives

This lab explores the performance of using buffered IO, mmap and how to use realloc to read an unknown amount of data into memory.

Source code

The files on Learn should contain the following files:

- `fread_fwrite`
- `read_write`
- `realloc`
- `mmap.c`
- `perf.sh`
- `Makefile`

You can build all the examples using the *Makefile* provided, simply change to the directory containing the code and type '**make**'

There is in addition a test script called *perf.sh* which can be run simply by *./sh perf.sh* which will run some performance tests and also check correctness of each of the programs.

Part 1 – using mmap to copy files

With memory-mapping a file, parts or all of a file are brought into main memory and are made available for reading, writing or executing. With this mechanism, an application no longer reads or writes data from/to the file, but reads/writes data directly from main memory, which is a much quicker operation. The Linux operating system works behind the scenes to bring the relevant parts of a file into memory and make them available for reading when they are needed. Similarly, after writing into memory the operating system transparently takes care of writing back any modified parts of the file to hard disk. The central system calls for dealing with memory-mapped files are *mmap*() and *munmap*():

```
void *mmap(void *addr, size_t length, int prot, int flags,  int fd, off_t offset);

int munmap(void *addr, size_t length);
```

Read the man pages thoroughly for mmap and implmenent copying.

Note – the permissions for *mmap* need to be the same as those used to open the file.

We wish to use *MAP_PRIVATE* for reading a file, because we don't care about synchronising with writes, and *MAP_SHARED* because we want our writes to actually be written back to disk.

What are the uses of mmap? Why would we use mmap instead of fopen/fread? (Why would we not?)

Part 2 – using realloc to read an unknown size of file

In part one above, we checked the size of the file using *fstat* and resized our file accordingly.

But what if we needed to read in a file of unknown size piece by piece, or receiving data over a socket?

The answer is we can use realloc. Your task here is to use realloc to implement a dynamically growing buffer. This kind of data structure is extremely common in practice and follows an algorithm like so:

```
Reserve some initial space
When some new data is inserted
     While(not enough space to fit)
          Double the reserved space
          Reallocate the data (use realloc)
     Append data to the end of the buffer
```

There are just two functions to implement, and the rest of the program tests the functions by using it to copy files (as with all the programs this week).

```
Buffer *new_buffer(size_t reserved);
void append_buffer(Buffer *buffer, char *data, size_t length);
```

Part 3 – performance analysis of buffered IO

Firstly, implement the file copying in `read_write.c` and `fread_fwrite.c` (this should be very familiar).

Then examine the output of *`./sh perf.sh`* to verify the file copy is working.

All the tests, if they're correctly implemented, should have the result:

files test.dat and output.dat are identical

Finally, graph the numbers `fread_fwrite` and `read_write`.

- Which one is faster/slower?
- What are the factors which make one faster than the other.?
- How do they compare to the mmap?