ENCE360 Lab 4: Files and Performance Optimisation

Objectives

This lab explores the performance of using `mmap()` and how to use `realloc()` to read an unknown amount of data into memory.

The aims of this lab are for you to understand:

1. How to use `mmap()` to have the OS dynamically load files into memory.
2. The performance differences between managed and `mmap()`d copying.

Preparation

Download and extract your Lab 4 files from `lab4MemMap.zip` on Learn. This contains the files `realloc.c`, `mmap.c`, `perf.sh`, and `Makefile`.

Simply executing `make` in the lab directory should build all the examples. In addition, running `./perf.sh` will measure the performance and assess the correctness of each program.

Once you've completed the tasks below, make sure you also complete the quiz on the quiz server.

Program `mmap.c`

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 page for `mmap()` and implement file copying using the system call.

Note that 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.

When/why would we use `mmap()` instead of `fopen()` / `fread()`? When/why would we not?

Program `realloc.c`

In `mmap.c`, 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 continuously receive data over a connection?

The answer is we can use `realloc()`, similar to what we did in ENCE260. 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 received:
    while (/* Not enough space to fit */) {
        // Double the reserved space.
        // Reallocate the data via realloc().
    }
    // Append data to the end of the buffer.
```

There are two functions to implement:

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

The rest of the program tests the functions by using it to copy files (as with all programs in this lab).

Then, examine the output `./perf.sh` to verify your implementation is working. All the tests, if they're correctly implemented, should have the result:

```
files test.dat and output.dat are identical
```

Compare the real, user, and system times taken by each program. What patterns are there, and can you explain them?