Datorteknik
Lab 4: Input-Output Operations and Buffers

Todor Stoyanov

October 11, 2018

1 Objectives and Lab Materials

In this lab you will learn how buffering affects I/O operations. This lab is directly taken from lab 10 in the course textbook.

The matrerials for this lab are:

- A Windows PC with qemu installed.
- A raspbian kernel for qemu.
- A disk image with raspbian.

Complete the following tasks. Provide all source code, including appropriate inline comments. In the report explain how the code works and describe what test you performed the validity of your code.

1.1 Preliminaries

Before starting with this lab please verify you have performed these preliminary steps (Note: same procedure as in Lab 2):

- Obtain the folder with qemu virtual machine settings from `N:\AASS\qemu_vms/` or ask the lab assistant if you have trouble finding it.
- Edit the file `start_qemu.bat` so that the kernel file and image file are taken from your local directory. Store the file in a local folder.
- Double-click on the `start_qemu.bat` file and verify that Raspbian boots up.
- In case you need them for later, the username for raspbian is pi and the password is DT509G_pi

- The rest of the lab assumes you are working within the raspbian emulator.
- The command-line text editors `nano` and `vi` are pre-installed on the raspbian image.

2 Task 1: Buffered input (7 points)

- Please note: you souhld only use standard C and your code should compile with the -ansi compile flag.
- Write a C program which uses the standard C API to open a file descriptor for reading in binary mode.
- Create a function `buf_in` which uses a character array of size $b = 16$ bytes to implement a read buffer.
- If the buffer is empty, `buf_in` should read in b bytes from the file and place them in the buffer.
- On subsequent calls, `buf_in` should return the next byte (charcter).
- Test and verify that `buf_in` works correctly on files of at least 32 bytes.

3 Task 2: Buffered output (8 points)

- Please note: you souhld only use standard C and your code should compile with the -ansi compile flag.
- Write a C program which uses the standard C API to open a file descriptor for writing in binary mode.
- Create a function `buf_out` which uses a character array of size $b = 16$ bytes to implement a write buffer.
- If the buffer is full, `buf_out` should write the contents to the file.
- Implement an additional function `buf_flush` which forces the contents of the write buffer to be written into the file.
- Test and verify that `buf_out` and `buf_flush` both work correctly for writing buffers of size less than 16 bytes, as well as buffers of sizes more than 16 bytes.

4 Task 3: Performance evaluation (10 points)

- Please note: you souhld only use standard C and your code should compile with the -ansi compile flag.
- Write a C main program which uses your buffered input/output functions for copying a file. The file names (source and destination) should be provided as command line arguments.
- Use the shell command `diff` to verify that the copied file is identical to the original.
- Use the C timing API to compute the time for reading and the time for writing a single byte.
- Calculate the average times and compare them to reading/writing a single byte (you will need to write a non-buffered version of your program to do this).
- Compare the performance over different file sizes, and using buffer sizes of 16, 32 and 64 bytes.
- Remove all debug output and timing information from your copying program and use the shell command `time` to measure how long it takes to copy a large file (¿100Kb).
- Compare the time it takes to the time when using the shell command `cp`.