EEL-4736/EEL-5737 Principles of Computer System Design

Homework #1

Assigned: 8/21/2019; Due: 9/4/2019 - To be done individually

This assignment consists of three parts and will prepare you with an environment that will be used for development of design assignments (and a final project) throughout the semester. The assignments/project will be based on a file system design of increasing complexity, using Linux and the Python programming language. **Part A** will guide you through the setup and configuration of a virtual machine environment with Linux for development and testing of your assignments/project; it will also expose you to the Python language through a brief tutorial and test. The goal is to get you prepared with the environment that will be used throughout the semester, and to ensure you are able to go through the basic steps in programming and use of this environment. **Part B** will guide you through examples to learn about the Linux system call interface. **Part C** will guide you through the lowermost, "block layer" of the file system that will be a basis for future assignments.

What to submit:

Note: please keep in mind that the guidelines below are meant to help us efficiently grade all assignments, and that **points will be deducted if you don't follow naming directions.** Turn in through CANVAS **a single zip file** packing the following files with the provided names:

1. **Python code** solutions for Step 4 in Part A (**named list1.py, string1.py, wordcount.py**)
2. **A PDF document** with answers for questions in parts B (**named HW1_partB.pdf**)
3. **Python code** with your solution for part C (**named HW1.py**)

Part A

The setup consists of a virtual machine running the Linux operating system and Python. It is assumed that you have an x86-based Windows, Linux or Mac laptop, 4GB+ of RAM, and around 16GB of free disk space to set up this environment. If you are unsure if your computer meets these requirements, please contact the TA or instructor. If you have any problems during these steps, it is suggested that you use the class discussion list to ask and answer questions. That way you may get a response from the TA, instructor or fellow students, and others who might have the same question will benefit from it too.

1) Install a virtual machine monitor in your computer. There are two choices of free virtual machine technologies suggested – VirtualBox (www.virtualbox.org), or you can also use VMware Player (www.vmware.com). You only need one of these installed and either will work for the class, but VirtualBox is recommended if you don't have a virtual machine in your computer yet.

2) Download a Ubuntu Linux 16.04 64-bit desktop edition ISO from Ubuntu website (this distribution has Python version 2.7, which is recommended for the tutorials below). Creating a new virtual machine and installing the operating system is straightforward. Just use the default options in VirtualBox/VMware and follow default the installation steps. It is recommended that you have these at least 1 GB of RAM and 8 GB of hard disk for the VM.

3) Boot up your virtual machine and familiarize yourself with the Ubuntu environment. If you are unfamiliar with UNIX or Linux, it is highly recommended that you go through a tutorial. There are various tutorials online on sites such as http://www.ee.surrey.ac.uk/Teaching/Unix/, http://www.tldp.org/LDP/intro-linux/html/, and YouTube.

4) Go through the following Google Education Python tutorial and complete the basic exercises by following these URLs:

https://developers.google.com/edu/python

https://developers.google.com/edu/python/exercises/basic

Part B

In this section, you will use the setup you created in Part A to get acquainted with the basics of the Linux system call interface for file system operations, and the code that will be used as a basis for future assignments, as described below.

First, you will use the "strace" Linux command, which captures and displays all system calls issued by an application, showing the system call type, arguments, and return values. Type "man strace" on your Linux machine and/or search the Internet for "strace" to learn more about this command

Run the following commands under "strace" so you can capture a trace of system calls (the following command will trace only file descriptor-related system calls and show all arguments and return values). Run one command at a time, take the time to inspect the output logs of strace at each step to answer the questions below.

```
strace -v -e trace=desc ls
strace -v -e trace=desc touch myfile.txt
strace -v -e trace=desc echo "hello" > hello.txt
strace -v -e trace=desc cat < hello.txt > hello2.txt
strace -v -e trace=desc cp hello.txt hello3.txt
strace -v -e trace=desc rm hello3.txt
```

Part B Questions:

B.1 Describe in one sentence the purpose of each of the following Unix commands: ls, touch, echo, cat, cp, rm

B.2 For each of the commands above, select one representative system call shown in the strace output that is important for the implementation of each command (ls, touch, echo, cat, cp, and rm). Describe, in your own words, what the system call does, and why it is important for the implementation of the command. For each system call you select, provide its name, and describe its argument(s), and return value.

Part C

Background and setup: In this part of the assignment, you are given an implementation of a raw block layer interface to a storage system. As a preview, this implementation realizes the lowermost layer of the file system case study we'll go through in this class (Textbook Section 2.5) – for the scope of this homework, you only need to be concerned that the block layer implements a simple interface to read/write data blocks:

BLOCK_NUMBER_TO_DATA_BLOCK(block_number): this is a read operation - it returns the contents of the block indexed by the argument block_number

update_data_block(block_number, block_data): this is a write operation - it writes the contents of block_data (an array of bytes) to the block indexed by block_number

As a starting point to the assignment, you are given the following Python files:

MemoryInterface.py – exposes BLOCK_NUMBER_TO_DATA_BLOCK(), update_data_block()

Memory.py – this actually implements the block storage in memory in your computer, and exposes interfaces to MemoryInterface.py

DiskLayout.py – this implements the mapping of a disk layout, based on Figure 2.20 of the textbook. The Disk_Layout,pdf file distributed with this assignment provides details about the naming and layout used. For this assignment, you don't need to worry about the superblock, bitmap and inode table areas of the disk layout – you only need to worry about the data blocks

config.py – this holds parameters that are used to specify the disk layout. For this assignment, you should not modify any of the parameters. The most important parameter of relevance to this homework is BLOCK_SIZE, which specifies how many bytes are stored in each data block

HW1.py – this is a "skeleton" for the code you need to implement in this assignment. The skeleton code provides the implementation of two template functions, **read()** and **write()**. The given **write()** function takes a string as input, "slices" it across multiple blocks of data, and writes to the storage system, starting from block number 0. The given **read()** function reads a string from the storage system, starting from block 0, and prints to the screen.

In your assignment, *you will implement the **write_at_offset()** function that is blank in the skeleton code*. The ***write_at_offset()*** function takes two arguments: a character string **newstring**, and an integer **offset**. The behavior of **write_at_offset()** function is as follows:

- If there is a valid block storing data at **offset,** this function will overwrite the storage system, starting at this offset, with the contents of **newstring**. The data in the storage system before the offset, or after the length of **newstring**, should remain unmodified. You may need to allocate additional blocks to accommodate **newstring**
- If the offset lands on a non-allocated block, you should return an error

Note that your **write_at_offset()** implementation should not use the read() and write() methods. These are just examples for your reference. Instead, you should build **write_at_offset()** upon the primitive **BLOCK_NUMBER_TO_DATA_BLOCK()** and **update_data_block()** methods. Your **write_at_office()** implementation should also make use of two functions that are used to find available blocks – **get_valid_data_block()**, which returns a block number that you can use to store a block, and **free_data_block(block_number)**, which frees up the block associated with block_number.

Programming assignment: you will write a Python program, called **HW1.py**, which takes three command-line arguments: **initialstring**, **newstring**, and **offset.** Your **HW1.py** program should extend the skeleton code provided to you to implement the **write_at_offset()** function (step 3 below):

1) Write **initialstring** across multiple data blocks. This is implemented by the write() function given to you.
2) Read **initiastring** back from the storage system, and print out to the screen. This is implemented by the read() function given to you. You can print the status of the block storage using the status() interface.
3) Write **newstring** starting from **offset** in the storage system – i.e., starting from the position given as an argument, again using multiple data blocks and supporting arbitrary sizes for string2
4) Read the string from the storage system, and print out to the screen, using read()

For instance, suppose the block size is 4 bytes, if HW1.py is executed with the following arguments:

HW1.py "Hello world" "! This is a test" 11

The arguments are strings – the quotes "" mark the beginning and end of each string argument.

In step 1) in this example, your program should store "Hell" in the first block, "o wo" in the second block, and "rld" in the third block. In step 2), you should read from these three blocks and print "Hello world". In step 3), the third block should end up with "rld!", the next block with " Thi", etc. In step 4), you should read the whole string, and print "Hello world! This is a test"

HW1.py "Today has been a bland day" "great" 17

This should print at the end of execution "Today has been a great day"

HW1.py "Hello world" "! This is a test" 12

This should return an error, as only three were allocated, and offset 12 lands in a fourth block