**Heterogeneous and Cloud Computing**

**Homework #8 (Part B)**

Max Points: 15

**Due: Mon, Nov 24 2014 By 11:59 PM (EST)**

**Email-based help cut-off: 24 hours prior to due date**

|  |
| --- |
| **Objective**: The objective of this exercise is to explore the use of:   1. Create a custom OS and run in in a virtual machine such as those on a Cloud 2. Emulate simple Software as a Service (SaaS) on a VM.   **Submission**: Save this MS-Word document to your computer prior to proceeding with this exercise. Upload the following at the end of the lab exercise:   1. This document saved as a PDF with the convention *MUid*\_Homework8\_PartB.pdf. 2. Your modified version of the startup script for your custom OS distribution in etc/init.d/rcS. 3. The C++ program that can find all possible anagrams from a given set of characters named with the convention MUid\_AnagramFinder.cpp (.h)   Where *MUid* is your Miami University unique ID. You may discuss any questions you may have with your instructor. |

# Part #1: Creating an Anagram Finder

## Background

Software as a Service (SaaS) is a standard cloud delivery mechanism in which a virtual machine is created and dedicated to specifically running a custom piece of software. In this homework, a C++ program will serve as the sample application to run as a SaaS example on a custom OS. The OS we will be running will be absolutely barebones with just the following major components:

* A standard Linux kernel will be used to boot up a VM
* The kernel will have a simple startup system that lives in RAM and runs the C++ program (you will not even have glibc installed! I did say barebones)

## Anagram Finder

The SaaS service functionality is to generate all valid anagrams from a given set of letters. You are supplied with a dictionary containing valid English words in a file called english.txt for identifying valid words. The C++ program (do not use OpenCL or Map-Reduce logic; instead develop a regular C++ program) performs the following operations:

* Obtain set of letters for generating anagrams as a single string using one of the following two approaches:
  + If the environment variable QUERY\_STRING is set (i.e., getenv(“QUERY\_STRING”) returns non-NULL value) then prefer to use it. If QUERY\_STRING is specified then print the HTTP header “Content-type: text/plain\n\n” before generating any output. The idea is that we will use exactly the same executable to run under bare bones apache as a web service via a RESTful API for testing elastic load balancing on EC2 later on).
  + If the environment variable is not set, then repeatedly (in a loop) prompt the user for a word (see sample outputs) and print its anagrams until the user presses Control+D (signifying logical end of file).
* Print all possible anagrams (with minimum length of 3 and maximum length of 7) that can be created from the given string in alphabetical order. The std::next\_permutation method will come in handy here.

## Sample Outputs

The sample outputs from independent runs of the program are shown below:

|  |  |  |
| --- | --- | --- |
| **$ ./AnagramFinder**  Enter word (Ctrl+D to quit): easy  aye  ayes  easy  say  sea  yea  yeas  yes  Enter word (Ctrl+D to quit): |  | **$ export QUERY\_STRING=sweets**  **$ ./AnagramFinder**  Content-type: text/plain  ewe  ewes  see  sees  set  sets  sew  sews  stew  stews  sweet  sweets  tee  tees  twee  wee  wees  weest  west  wet  wets |
|  |  |
| **$ ./AnagramFinder**  Enter word (Ctrl+D to quit): anagram  agar  anagram  arm  gram  man  mar  nag  rag  raga  ram  ran  rang  Enter word (Ctrl+D to quit): |  |

# Part #2: Creating a custom OS for SaaS

**Background**: Many modern operating systems are often created by combining the core set of GNU (acronym for GNU is not Unix) tools (namely: gcc, binutils, emacs, to name a few) along with the Linux kernel. Such operating systems include: Fedora, Ubuntu, Mint, and Android. These distributions are created using the following two major components:

1. The core GNU tools (<http://www.gnu.org/software/software.html>)
2. The Linux kernel (obtained from <https://www.kernel.org/>) compiled using GNU tools

The core set of components remains the same in the various distributions. However, the differences arise in the way other software (such as: open office, Java, Eclipse, chrome, etc.) are packaged, distributed, installed, and managed by the various distributions. One of the most conspicuous differences is usually in the graphical Desktop and window management systems supported by the various distributions. Of course Wikipedia has a nice page about Linux distributions at <http://en.wikipedia.org/wiki/Linux_distribution> from where the following image was obtained:

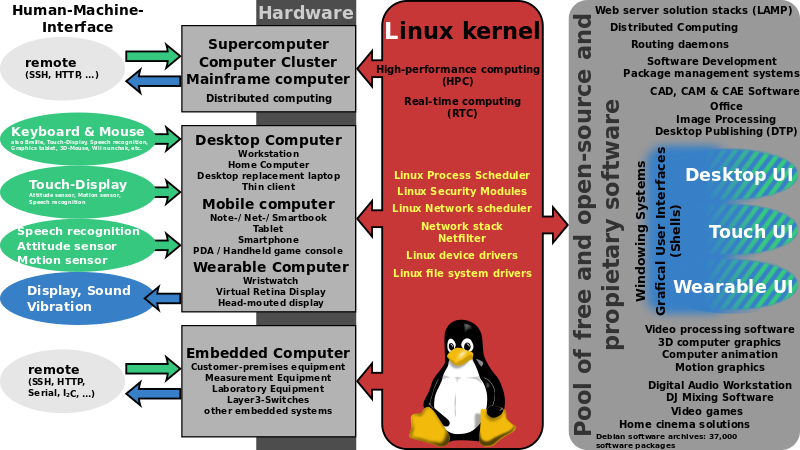


Figure 1: Obtained from Wikipedia page http://en.wikipedia.org/wiki/Linux\_distribution

## Running the bare kernel

In order to get started with running a bare Linux kernel in a virtual machine you will need the following software using the following:

1. QEMU (<http://www.qemu.org/>): QEMU is a generic and open source machine emulator and virtualizer. When used as a machine emulator, QEMU can run an OS and underlying programs made for one machine (e.g. an ARM board) on a different machine (e.g. your own PC). By using dynamic translation, it achieves very good performance.When used as a virtualizer, QEMU achieves near native performances by executing the guest code directly on the host CPU.
2. Prebuilt Linux kernel: You may download a custom kernel and compile it. However, kernel compilation is a time consuming process. Therefore, to save time, a prebuilt Linux kernel will be used for this lab exercise. Specifically, the following kernel that was actually run on the server will be used: /boot/vmlinuz-3.5.0-23-generic.

Run the bare bones Linux kernel (it is not an operating system but just the core kernel that can handle critical hardware devices and do context switching) using the following command:

|  |
| --- |
| **$** qemu-system-x86\_64 –k en-us -no-reboot -no-kvm -show-cursor -kernel vmlinuz-3.13.0-24-generic |

The above call to qemu will create a virtual machine that will be displayed in a separate graphical window and you should observe the following:

1. The qemu virtual machine will run a default BIOS.
2. Next, the qemu boot loader (not a standard/conventions boot loader) will load the specified kernel.
3. The kernel will boot up.
4. At the end of the booting process the kernel will panic and crash complaining that it did not find a root file system (as we have not specified one) to mount and work with.
5. Kill the qemu process by simply closing the graphical QEMU window.

## Creating a root file system aka initrd

As observed in the previous part, the Linux kernel requires a valid root file system to operate with. In this part of the exercise a minimal initial file system called initrd or initramfs will be created. In order to aid creating the basic set of directories (they were manually created using the mkdir command shown below) that the Linux kernel requires in initramfs, a zip file containing the directory structure is supplied for this exercise. Use the supplied zip file using the following commands:

1. Download the supplied initial initramfs.zip file that has the initial directory structure setup for you for convenience. The zip file also includes very minimal configuration and a single executable that serves for running multiple commands.
2. Unzip the supplied zip file to create the basic directory structure to be included in the initial ram disk (initramfs):

|  |
| --- |
| $ unzip initramfs.zip  $ cd initramfs |

1. View the directory structure in initramfs and observe how the file named busybox has been symbolically linked to various commands including the init program. Virtual box is a versatile static binary that provides simple implementations for many standard commands such as: ls, ps, cat, less, grep, etc. The supplied initramfs uses busybox as the init executable required by Linux kernel using a symbolic link (busybox uses the file name to detect how it should behave). Recollect that init is the first process that is ever run by the Linux kernel.
2. Now, create your own initial ram file system (initramfs) using the following command where MUid is your unique ID (ensure you are in the initramfs directory):

|  |
| --- |
| $ find . -print0 | cpio --null -ov --format=newc > ../MUid-initramfs.cpio |

1. Finally run the kernel with the newly created initramfs using the command below from the directory where the *MUid*-initramfs.cpio file is present. The kernel should start up and drop you to the root-shell prompt #.

|  |
| --- |
| **$** qemu-system-x86\_64 –k en-us -no-reboot -no-kvm -show-cursor -kernel /boot/vmlinuz-3.5.0-44-generic –initrd MUid-initramfs.cpio |

1. Note that you are root (or super user) on the virtual machine. You can try a few of the standard shell commands at the # prompt. In order to quit your virtual machine type reboot at the # prompt.

## Customizing your SaaS initrd

The initramfs supplied to you is a barebones distribution that has very few commands and libraries. However, that is sufficient to run a complied C++ binary. No other baggage in the form libraries and VMs are needed.

1. Compile your anagram finder to run as a self-contained (static) executable using the the following command line:

|  |
| --- |
| $ g++ **-static** -O3 -Wall -std=c++11 AnagramFinder.cpp -o AnagramFinder |

1. Ensure your Type-2 hypervisor (aka qemu process) is not running.
2. Copy the generated AnagramFinder executable to the appropriate bin directory in your initramfs directory.
3. Now, recreate your own initial ram file system (initramfs) using the following command where MUid is your unique ID (ensure you are in the initramfs directory):

|  |
| --- |
| $ find . -print0 | cpio --null -ov --format=newc > ../MUid-initramfs.cpio |

1. Finally run the kernel with the newly created initramfs using the command below. The kernel should start up and drop you to the root-shell prompt #. Run the AnagramFinder in the bin directory and ensure it is operating correctly.

## Further Customizing your OS Startup for SaaS

Now it is time to add a bit of flair to your OS distribution to print a startup banner for your distribution in the following manner:

1. Ensure your virtual machine (aka qemu process) is not running.
2. On the Linux server for this course, locate the etc/init.d/rcS file. This file is essentially a shell script that is automatically run by busybox’s init program present in the root directory of your custom file system.
3. Edit the etc/init.d/rcS using emacs and add code to print messages to the end of this file as shown below (watch out for messages for single quotes in them – they will cause problems if you don’t replace single quotes with some other character):

|  |
| --- |
| echo ’your message goes here’ |

1. Finally, add “bin/AnagramFinder” to your startup script to have the AnagramFinder automatically run as your Software as a Service (SaaS).
2. Using the commands shown earlier create an updated initrd image and run your custom OS distribution and verify that your banner is correctly generated and your SaaS works.
3. Next make a screenshot of your banner and paste it in the space below:

|  |
| --- |
| Your screenshot goes here instead (that means you have to delete the text) of this text. |

# Part #3: Submit files to Niihka

Save this MS-Word document to your computer prior to proceeding with this exercise. Upload the following at the end of the lab exercise:

1. This document saved as a PDF with the convention *MUid*\_Homework8\_PartB.pdf.
2. Your modified version of the startup script for your custom OS distribution in etc/init.d/rcS.
3. The C++ program that can find all possible anagrams from a given set of characters named with the convention MUid\_AnagramFinder.cpp (.h)