CS 618 Lab 3 Makefile generation and debugging using gdb
Deadline 21st August 11.59PM, Individual assignment

Task 1: Build using make and run a program

For this task you will download, compile, and run a piece of open source software from the web using make. The program is called jp2a (JPEG to ASCII converter). Here are the steps to follow:

1. Download the jp2a tar.gz file from the following link: https://sourceforge.net/projects/jp2a/files/jp2a/1.0.6/
2. Find the appropriate unix command to uncompress the tar file on a terminal.
3. After decompressing the archive, change into the directory just created and **run the configure program** to set up the build process for your particular computer type/architecture. The configure program analyzes your system and produces a Makefile that will work for your computer to build and install the program
4. Normally configure sets up the Makefile to install the app to the directory /usr/local/bin, but since you may not have root access on the system you're using, you may not be able to install it there. We suggest you tell the configure tool to install the app to your current directory, the directory where you decompressed the jp2a source files. To do so, run the command as follows:

   ./configure --prefix=$(pwd)

If it worked properly, you'll see several lines of output such as: `checking for a BSD-compatible install... /usr/bin/install -c config.status: executing depfiles commands`

5. Assuming that the configure program completes successfully, you are ready to compile and install the program using make. Run: make, and once it completes, run **make install**
6. If make install worked properly, you should now have a bin/ subdirectory within your jp2a source folder. In this folder should be a newly built executable called jp2a. You can run the program while in that directory by typing: ./jp2a [options] filename.jpg
7. In order to make the jp2a command available to you regardless of your current directory you will need to add the directory that contains the jp2a executable to your PATH. This can be done by

adding a line to your bash_profile that modifies your PATH environment variable. Figure out how to add the PATH environment variable.

8. If you have done this correctly, the command which jp2a should print the location of the jp2a executable on your machine. You can also now run the jp2a program from any location on your machine using the following: (note the ./ prefix is no longer required) **jp2a [options] filename.jpg**
9. To complete this task you should download a .jpg file of your choice from the web and convert it to ASCII using jp2a. You can use Google Image Search to find an image
10. The jp2a program should output an ASCII version of the image. Redirect the jp2a ASCII output to capture the output in a file named task1_lab3_jp2a.txt
11. In the report provide details about environmental variables that you have added and other details that you feel necessary to be included.

Task 2: Write a Makefile

For this task you will write a Makefile for a small set of C program files provided to you in `lab3_task2.tar.gz`. These files represent a linked list library stored in `linkedlist.c` and `linkedlist.h` along with some client program C files that use this library to perform simple tasks.
Your Makefile should have the following **six properties**:

• A target that builds an object file named linkedlist.o from the source code found in linkedlist.c. If linkedlist.c or linkedlist.h is modified, the linkedlist.o file should be rebuilt. In other words, it depends on both of those files.
• A target that builds an executable file named **list_check** from the source file use_ll_2.c and the compiled object file **linkedlist.o**. If **linkedlist.o** or any of its dependencies are modified, **list_check** should be rebuilt.
• A target that builds an executable file named **list_run** from the source file use_linkedlist.c and the compiled object file **linkedlist.o**. If **linkedlist.o** or any of its dependencies are

modified, **list_ run** should be rebuilt. (You can test the list_check and list_run programs by running them once they have been compiled.)
• A target named **clean** that removes the list_check and list_ run executables along with any .o files from the directory.
• The Makefile's default **target** (the one that runs if make is not given any parameters) should build both the list_check and list_run executables.
• Use at least one of make's advanced features. For example, declare at least one variable and use it in your rules, and/or try to use some of the special variables such as $^ or $< .

Task 3:Debugging *(for the debugging exercises, you will submit the corrected programs and the corresponding analysis details in the written report)*

The goal of the following debugging exercises is not simply to find the errors; it is to learn about and use gdb. For each of the first two exercises, you should run the programs under the control of the gdb debugger, and use it to find the bugs. **You should write about What debugger commands did you use? Why? What did they reveal? What were the bugs?**

a) Debug the program selection.c. This program tries to use the selection sort algorithm to sort an array. However, it crashes with a dreaded Segmentation Fault. Usually, to debug a segfault, you want to run the program under the debugger and let it crash. You can then use debugger commands to get information about the state of the program when it crashed. The *backtrace* command can be particularly useful.

After you have fixed the segmentation fault, you might find that the program still doesn't sort the array correctly. You should find and fix that bug too.
After selection.c correctly sorts the array, the program will go into an infinite loop when it tries to use the *binary_search* function. Use the debugger to find and fix the problem. Since you don't usually know where a program is going into an infinite loop, note that you can use Control-C inside *gdb* to pause the program. You can then inspect the state of the program, install breakpoints, and so on.

b) For this exercise, you should explain the error in insertion.c. This program tries to use the insertion sort algorithm to sort an array. However, the result is not correct. Your job is to explain, clearly and completely, how and why the program produces the output that it does.

Task 4

1. Download the source C file t4_msg.c
2. Create a Makefile and add the following targets:
   - `release`: Compiles the program normally, e.g. `gcc -Wall -o program program.c`.
   - `debug`: Compiles the program with `-g` option (which includes debug symbols in the binary).
3. Build and run the program with: make debug followed by ./program
4. You will notice that the output is not correct. Use gdb to locate the bug in the code. Specifically, we are asking you to not simply perform printf debugging where you insert code and recompile.
5. You need to upload the corrected source code and Makefile for this task.
6. In your report, provide the details of 1)how you changed the code, 2)Makefile, 3)the message that should have been displayed and 4)how you found the bug.
7. How does GDB know where functions and data are located in the executable when you are debugging?
8. Run objdump on the executable you compiled. Run objdump -help to see what options it offers. Experiment with the options to see what information you can get it to display.

Submission instructions:

1. Make a new branch called lab3 in your parent CS618 github repository
2. For each task make a separate directory in lab3 named as taskx (x is the number of the task eg. task1 task2)
3. Inside the directory corresponding to each task, populate the required files as mentioned in the handout

4. Make a report and provide the required details corresponding to each task and upload it in the lab 3 folder. The report should be named as lab3_rollno (rollno is of the format cs24mtxx)