Advanced Embedded Software Development

Homework 2 (100 Pts) - Due Sunday February 10th (before midnight)

Revision 2019.01.25

1 Guidelines

For this homework, both Canvas and Github online materials need to be submitted. **Turn in a *.pdf for your report to this assignment to Canvas. Please neatly format submissions with your name, date, homework # at the top, and enumerator your answers using headings indicating each problem number.**

And please format your answers - provide headings to the problems, context with captured data and descriptions for images and screenshots.

2 Reading & Resources

These are reading assignments that are good to complete the homework for the week as well as review materials to prepare for the upcoming content in the class.

- (MELP) Mastering Embedded Linux Programming (Second Edition) – Simmonds
  - Chapter 6 – Focus is on Buildroot and Adding your Own Code (Overlay)
- Linux Kernel Development – Love
  - Chapter 5 - System Calls (all)
  - Chapter 17 – Devices and Modules
- Linux Device Drivers – Corbet (skim/reference) https://lwn.net/Kernel/LDD3/
  - Chapters 1 & 2

Resources:

These are useful documents that will assist your work with the Buildroot toolchain and is an accompaniment to the MELP book, above.

- Buildroot
  - Buildroot Practical Labs - https://bootlin.com/doc/training/buildroot/buildroot-labs.pdf
  - Buildroot Manual - https://buildroot.org/downloads/manual/manual.pdf

These sections are not required but may help with doing the homework assignments.

- MELP - Chapter 3 – All about Bootloaders
- Ubuntu Linux Boot Procedure: https://wiki.ubuntu.com/Booting

3 Problem Set

[Problem 1] Record your Repository

Provide us a Github link to your repositories. Include a link to the repository on the top of your assignment to turn in.

[Problem 2 - 20 Pts] Track system calls and library calls with File IO

Let' refresh your file operations skills, as this will be important as we move towards implementing a device driver and writing programs that require user interaction. Additionally, you'll familiarize yourself with and use tools to support your program's development. Specifically, you'll use perf, ltrace and strace to collect information on your program that reads and writes to a file on your development host machine. Your goal is to write a program that can:

- Print to standard out an interesting string using printf
- Create a file
- Modify the permissions of the file to be read/write
- Open the file (for writing)
- Write a character to the file
- Close the file
- Open the file (in append mode)
- Dynamically allocate an array of memory
- Read an input string from the command line and write to the string to the allocated array
- Write the string to the file
- Flush file output
- Close the file
- Open the file (for reading)
- Read a single character (getc)
- Read a string of characters (gets)
- Close the file
- Free the memory

After writing this, run the ltrace and strace command line applications and collect the output of the system calls and library calls that were used to interact with your file. Additionally, use the "perf stat <your program>" command to collect some performance statistics of your program. **Show all 3 outputs (ltrace, strace, perf) in your report.**

[Problem 3 - 30 Pts] Setup Buildroot, then build and boot a Beaglebone Green (BBG) Linux image

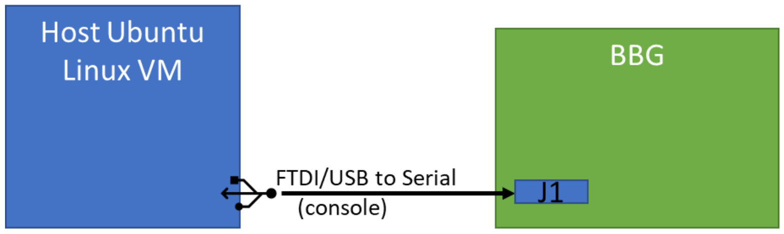
In this exercise you're building an micro SD memory card with everything to boot and run embedded Linux on a BBG. The images (uBoot, MLO, kernel, rootfs) for the BBG will be built using the Buildroot toolchain with MELP Chapter 6 as your guide.

(***Note***: We're moving on from Crosstools-NG to use the Buildroot Linux toolchain/development environment. As you begin, you'll want to make sure your Ubuntu (host) Linux VM has plenty of disk space (>100 GB). Secondly, over time, as you use install and setup the Buildroot tools, you'll want to adjust your dev environment (.bashrc, $PATH, $HOME/bin, etc) to access and use Buildroot's bin, etc.)

The process will involve:

- Setting up Buildroot along with the corresponding Linux source files (these are different files from your host Linux sources)
- configuring and building Buildroot toolchain
- making/configuring a target BBG Linux configuration (e.g. beaglebone_defconfig)
- building the Linux image

- burn the image files to your microSD memory card as a bootable file system
- set up your BBG HW
  - connect your dev host to BBG using a serial console cable for console interaction - With FTDI/USB serial port cable connected to your host and the other end to the BBG J1 connector (Black wire on cable is pin 1)



  - Download 'minicom' or 'screen' or 'gtkterm" or your favorite term emulator to your host.
    - `$ sudo screen /dev/ttyUSB0 115200`

    Or

    - `$ sudo apt-get install minicom`
    - `$ sudo minicom /dev/ttyUSB0 115200`

    Or

      - `$ sudo apt-get install gtkterm`
      - `$ sudo gtkterm -s 115200 -p /dev/ttyUSB0`
  - (Note: If you are using a guest Linux Virtual Machine (VM) for development on your host, there may be some configuration (mapping) needed of the FTDI/USB serial port from your host computer port through to your guest VM environment.)

- Install the micro SD memory card, and then apply power to the BBG . You may need to interrupt/change to normal BBG manufactured boot sequence to boot your image through the use of BBG buttons or console interaction …

- Change the BBG login greeting to include your name using make menuconfig and rebuild your image. Burn your new image to the microSD memory card. Congratulations, you've just build your own Linux embedded system!

Keep in mind the version of tools, kernel, etc of software mentioned in the book are newer now, so you'll have to be attentive to versions, paths, etc.

We're not going for any fancy functionality, but are just getting the tools set up, oriented to the location of files, and learning how to burn a micro SD memory card and get it to boot on the BBG. Please to not overwrite the eMMc memory that ships on the BBG – **you should only boot from your micro SD memory card**. (Hint: Maintaining the original manufacturers code on the BBG allows you to boot from the manufacturer's original image, by removing the micro SD memory card with your image, and see that the BBG is still functional. Good to know when you're wondering if it's a problem with your code or the HW!)

Boot your BBG, login to root, and try some commands on the console.

Report a screen capture of the last 20 lines of console boot sequence, as well as you logging in and executing your favorite commands at the console (e.g. ps -aux, lsmod, ls /).

[Problem 4 - 10 Pts] Port Your File IO program to BBG

Now that you have a cross development environment setup, let's port your program from Problem 1 to the BBG. The simplest way to do this is to cross-compile your program "out-of-tree" in some project directory outside of the Buildroot directories. (e.g. ~/projects/myProblem4).

Then create an overlay directory inside the Buildroot directory –

(e.g. ~/buildroot/board/beaglebone_rick/root-overlay/usr/bin) and copy your "arm compiled" executable there. Next configure Buildroot (make menuconfig) to add an overlay directory that will deliver your file (add to) the rootfs in next complete image you build and burn to the micro SD memory. This will put your executable in a good location is in the (target) BBG filesystem's /usr/bin directory.

Additionally, configure Buildroot to build/add to your BBG image the strace, ltrace, and perf executables using "make menuconfig" to include the correct packages (e.g. for ltrace search for "ltrace" then set symbol: BR2_PACKAGE_LTRACE [=y]).

Remake, burn a new complete micro SD memory card image, install and boot/run your new Linux image.

From a BBG console command line, run your program and collect the BBG version of ltrace, strace and perf stat of the interesting stats that you collected from Problem 1.

[Problem 5 - 30 Pts] Implement Your Own System Call on BBG

You are to create your own system call that can sort an array of numbers in kernel mode. This is more for practice of implementing a call than for making something for its utility. This system call needs to support the following features:

- A set of input parameters from user space including
    - Pointer to a buffer (input)
    - Size of that buffer (entries or bytes)
    - Pointer to a sorted buffer
- Validation of all input parameters
- Print information to the kernel buffer (log)
    - Log the input (user space) buffer contents when your syscall enters, exits, the size of the buffer,
    - Log the output buffer at start/completion of the sort (details provided below)
- Your system call needs to allocate dynamic memory to copy data in from user space
    - The user space array needs to be copied into kernel space presort
    - The array needs to be copied back to user space post sort.

Your syscall should be defined appropriately using the SYSCALL_DEFINE macro given your argument list size. The input buffer should be at least 256 int32_t data items. The buffer needs to be copied into kernel space. Your syscall will need to sort the data in an order from largest to smallest. Once sorted, it needs to pass the sorted data back to the calling user application in a sorted buffer array.

Review the System Call lecture for additional guidance on where to add/how to add your code. In other words, add your syscall code to the kernel image built by Buildroot's make. Keep in mind, we're building a Beaglebone (arm-based) image, so some source modifications will be in the Buildroot kernel source directory (e.g. ~/buildroot/output/build/linux-<something>/kernel) while other source code modifications will be in architecture specific directories (e.g. ~/buildroot/output/build/linux-<something>/arch/arm…).

Show that your kernel module works by writing a user space application that calls your system call numerous times. You can randomly generate this buffer of data elements using random() and time(). This application can be built "out-of-tree" from Buildroot but you should use (include) Buildroot Linux directories of source files (i.e. ~/buildroot/output/build/linux-headers-<xxx> and ~/buildroot/output/build/linux-<xxx>" used to compile the BBG kernel.

You should show that:

- System call works correctly (all input parameters valid and correct)
    - Print information showing that the sort worked correctly

- System call fails (input parameters are not valid and/or correct)
    - BONUS: All errors should return appropriate error values (defined in errno.h and errno-base.h)
- Bonus: use a sort algorithm with O(nLogn) performance

Your report should include screenshots of example output of your multiple program runs, including both successful and failure cases (annotated with the "use-case" comments), appropriate BBG kernel logs with printk comments noting the use-cases. Report the time it takes to perform your syscall using timestamps from the log files.

[Problem 6 - 10 Pts] Create a CRON/Systemd task on BBG

Write a C-program that uses your system call from problem 5 along with a few other system calls listed below. This program should run every 10 minutes and it should run to completion. Your program should collect the following information using system call APIs and print its output to a file (either write to the file or just redirect the output):

- Current Process ID
- Current User ID
- Current date and time
- Output of your system call

Report the collected information and the outputs from your system call for a period greater than 30 minutes.