Advanced Embedded Software Development

Homework 3 (62 Pts) - Due Sunday February 17th (midnight)

Revision 2019.02.07

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, captions explaining images and screen captures, etc.**

Further, once you have GDB working in Problem 2, schedule time with the Student Assistants to demonstrate your ability to command-line debug (single step, restart, print variables) your application.

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.

- Mastering Embedded Linux Programming (Second Edition) – Simmonds
  - Chapter 9 – Interfacing with Device Drivers
  - Chapter 14 – Debugging with GDB
- Linux Kernel Development (Third Edition) - Love
  - Chapter 17 – Devices and Modules
  - Chapter 11 – Kernel Timers (only page 222-224)
- Chapter 1 & 2 - Linux Device Drivers – Corbet (skim/reference)
  https://lwn.net/Kernel/LDD3/

Resources:

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

- GDB Manual and information: https://www.gnu.org/software/gdb/documentation/
- Kernel Modules: https://wiki.archlinux.org/index.php/Kernel_module
- Kernel Timer Example: https://www.ibm.com/developerworks/linux/library/l-timers-list/?ca=drs-

3 Problem Set

[Problem 1 – 2 pts] Set up BBG ethernet networking

Modify your BBG configuration using Buildroot to enable ethernet networking with your host. Enable the packages for DHCP as well as SSH connectivity (e.g. Dropbear) to allow "scp" copying of files from Add an ethernet configuration file as a build image overlay that is properly placed in the BBG rootfs to enable DHCP IP addressing on the ethernet interface as a part of booting the BBG.

Screen capture and report the console boot sequence/dmesg log showing the BBG getting an ethernet address from the network.

Screen capture and report a (scp) copy of a file from your host to the BBG to the /usr/bin directory.

[Problem 2 - 20 Pts] Remote debugging your application with GBD

Write/port, compile and add your own application (e.g. from Assignment 2 Problem 2) out-of-tree from the Buildroot Linux directories, yet using the Buildroot target tool chain. Make sure the symbols haven't been stripped. Let's run it on the BBG using GDB and exercise debugging skills with it.

Next set up your BBG and host for remote debugging. We're going to use Buildroot (i.e. make menuconfig) to configure and rebuild the BBG target Linux image as well as create host executables so you can do rudimentary remote command-line debugging of code running on the BBG. This method will utilize the ethernet interface for debugging connectivity.

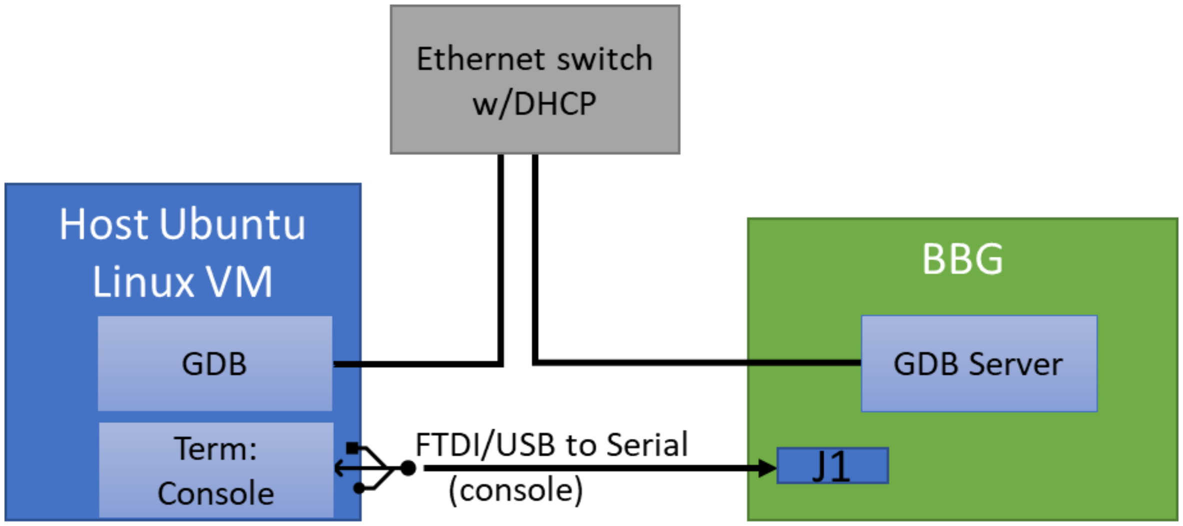

Figure 1 - Recommended GDB connectivity

The high-level items needed via Buildroot are:

- Enable running GDB on the host
- Enable inclusion of GDB Server on the target
- Enabling debugging with symbols

Besides our MELP textbook, utilize the GDB manual for assistance in executing remote debugging commands. Key points to consider are SYSROOT settings to allow the host access to code symbols and compiling the target code to include them.

On the BBG console via the serial cable (or ssh to the BBG in another terminal), start gdbserver without supplying an initial command to run or process ID to attach by using the '--multi' command line option.

```
root@bbg# gdbserver --multi comm &
```

Note: "comm" is the IP address:socket# and the "&" at the end of the command line to put the process in the background and allow you to continue to use the console command line.

Next, start GDB on your host, then using a series of GDB commands connect the host to the BBG target, configure the session, push the program you want to debug, and start debugging!

After manually confirming the commands necessary to set up and begin your debug session, demonstrate initializing the host GDB configuration to save some typing by capturing the commands into a gdbint file and adding a “-x gdbinit” as a parameter to your host GDB command.

Capture your host debugging session using the “manual” configuration command method

1) Start the host GDB session with the program name to debug
2) Show the host commands connecting to the target
   e.g. `(gdb) target extended-remote 192.168.0.8:5555`
3) Pushing your out-of-tree executable to the target
   e.g. `(gdb) remote put …`
4) Select the file to debug. This should be set to the filename on the target system.
   e.g. `(gdb) set remote exec-file filename`
5) Set breakpoints (e.g. main and others) and any other commands necessary, then run your program
6) Single stepping through your code and continuing execution
7) Manually showing (print) variable values at the command line
8) Capture the “console” printouts from the BBG output of your program.

Capture and repeat the above steps using using a host initialization file i.e. -x gdbinit , and repeat some debugging steps. Show the contents of your gdbinit file.

For the future, now that you have exercised the GDB command line basics you can download and set up DDD (a GUI) or perhaps Eclipse IDE to run the GBD remote sessions, as mentioned in MELP.

[Problem 3 - 20 Pts] Create a Kernel Module

Create your own “external” (not in the tree) kernel module that use a kernel timer to periodically wake up (fire) every 500msec by default. Each time the timer wakes up you should call a function that prints to the kernel log buffers

- Your name
- a count of how many times the timer has fired

Declare a statically allocated variable to track this count in your callback function.

Your kernel module also needs to take two input parameters, your name and the timer count time in seconds. The parameters should be settable at the installation of the module (E.g. sudo insmod myTimerModule name=”Nikhil” period=30) or via input parameter with module configuration files.

To create/initialize a timer, look for the kernel timer functions:

```
• void init_timer (struct timer_list * timer);
• void setup_timer (struct timer_list * timer,
       void (*function)(unsigned long),unsigned long data);
```

Be sure to add a callback, modify and/or delete your kernel timer appropriately:

- `void add_timer (struct timer_list * timer);`
- `void mod_timer (struct timer_list * timer, unsigned long expires);`
- `int del_timer (struct timer_list * timer);`

Confirm that your module has been loaded by running the command on your system:

```
$ lsmod | grep module-name
$ modinfo module-name
```

Submit your code to your Git repo and record it's the location (link) in your Canvas assignment submittal.
Install and run your kernel module 2 times, each time with a different set of command line parameters. For each run get screenshots of the following items to include in your report:

- Screenshots of the install and successful load of the module
- Output print buffer (dmesg log) of your count printing your name and the count and timestamps
- Screenshots of the module info showing you as the author
- Screenshots of the module remove

[Problem 4 - 20 Pts] Data Structures

In this problem pretend your new job is supporting an animal ecologist and process data she has provided you. Although this processing would be better implemented as a user-space application, we'll use it as a chance to explore Linux data structures and operations.

The objective of this exercise is to create animal-type "sorting" functions in a Linux kernel module. Based on command line (or .conf file) parameters for initialization at module insertion, it will construct and initialize 2 internal lists using the Linux kernel data structure definition, constructs and macros. (e.g. lists.h)

The module is defined with a seed static array of 50 animals types ("frog", "spider", "shark", "frog", "elephant", "fish", "toad", etc.) of your choice to be processed during initialization of the module. Note some types ***should*** be repeated multiple times in the seed. Process this seed array into two sets:

- An "ecosystem" set of all unique animal types, with duplicates removed. Each type should have associated count of occurrences that they appear in the seed array.
- A filtered "list" of animal "types of interest" based on the module input parameter(s). Filter options should be none, either or both:
  - Animal type explicit (e.g. animal_type="frog"
  - Count_greater_than (e.g. count_greater_than=2)

Your code should dynamically allocate memory used to construct "nodes" for each set and deallocate it upon removal of the module.

Report:

- The data structure type used for each set. (e.g. hash-map, linked list, rbtree, etc)
- The entries in the seed array.

- During module initialization, your module should report duplicate occurrences as they are processed.
- At the end of module initialization report the results for the 2 sets of processed data:

  Set 1:
  - the unique “ecosystem” list of animal types, reported alphabetically, along with the occurrences of each in the seed list.
  - The ecosystem size (number of nodes)
  - The total amount of memory dynamically allocated for nodes

  Set 2: Do three runs - none, with one filter, with two filters
  - The filter criteria
  - The filtered list content (animal types)
  - The filtered ecosystem size (number of nodes).
  - The total amount of memory dynamically allocated for nodes

- The amount of time to insert your module
- Upon removal of the module, show the amount of freed memory for each set.
- The amount of time to remove your module.