Advanced Practical Embedded Software

Homework 2 (120 Pts) - Due Sunday 2/4 Midnight

Revision 2018.1.26

Guidelines

Turn in a *.pdf with your answers to this assignment to D2L. **Please format your assignment nicely.** Don't make these assignments irritating to grade.

Reading & Resources

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

- Operating Systems Concepts - Silberschatz http://iips.icci.edu.iq/images/exam/Abraham-Silberschatz-Operating-System-Concepts---9th2012.12.pdf
    - Chapter 2 & 3
- Linux Device Drivers - Corbet - https://lwn.net/Kernel/LDD3/
    - Chapter 2 (all) & Chapter 7 (skim/reference)
- Linux Kernel Development https://github.com/yuanhui-yang/Linux-Kernel-Development/blob/master/Linux%20Kernel%20Development%20-%203rd%20Edition.pdf
    - Chapter 5 - System Calls (All)
    - Chapter 11 - Kernel Timers (Only pgs. 222 - 224)
- Test Driven Development for Embedded C - Chapter 2 Unit Testing
    - https://doc.lagout.org/programmation/C/Test-Driven%20Development%20for%20Embedded%20C%20%5BGrenning%202011-05-05%5D.pdf

Resources:

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

- Ubuntu Linux Boot Procedure: https://wiki.ubuntu.com/Booting
- Kernel Modules: https://wiki.archlinux.org/index.php/kernel_modules#Blacklisting
- Kernel Timer Example: https://www.ibm.com/developerworks/linux/library/l-timers-list/?ca=drs-

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.
https://goo.gl/forms/LFe5zX8B5EvkN8N82

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

For for some refreshing on your file operations as this will be important as we move towards the driver implementation and interaction portion of the course. In this problem you are to use perf, ltrace and strace to collect information on a program that reads and writes to a file. 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 and input string from the command line and write to the string
- 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** command to and monitor some performance statistics of your program. Show this output in your report.

[Problem 3 - 30 Pts] Implement Your Own System Call

You are to create your own system call that can sort a string in kernel mode. This is more for practice of implementing a call then for its utility. The point of this call is to sort a data array. This system call needs to support the following features:

- Print information to the kernel buffer
    - Log when syscall enters, exits, the size of the buffer, and the start/completion of the sort (details provided below)
- Your system call needs to take a set of input parameters from user space including a
    - Pointer to a buffer
    - Size of that buffer
    - Pointer to a sorted buffer
- You need to validate all input parameters
- 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 system call should be defined appropriately using the SYSCALL_DEFINE macro given your argument list size.

- Create a new directory for your syscall and add your directory to the sources list in the main kernel makefile.

The buffer will need to be copied into kernel space. Your syscall will need to sort your buffer in an order of largest to smallest. Once sorted, it needs to pass the buffer back to the sorted buffer array. You should create a buffer of at least 256 int32_t data items. You can randomly generate this buffer of data elements using rand and time.

Show that your kernel module works by writing a piece of software that calls your system call numerous times. 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)
    - All errors should return appropriate Error values (defined in **errno.h** and **errno-base.h**)

You should include screenshots of example output of your program in addition to the source code in your assignment. Report the time it takes to perform your syscall.

[Problem 4 - 10 Pts] Create a CRON/Systemd task

Write a C-program that uses your system call from problem 2 along with a few other system calls listed below. This program should run **every 10 minutes** and it should run to completion. This program should print its output to a file (either write to the file or just redirect the output). Your program should collect the following statistics using system call APIs:

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

[Problem 5 - 20 Pts] Create a Kernel Module

Create your own kernel module that use a kernel timer to wake up every 500msec (by default). Each time the timer wakes up you should call a function that prints to kernel log buffers a count of how many times the timer has fired. Declare a statically allocated variable to track this count in your callback. Your kernel module also needs to take two input parameters, your name and the timer count time. Your name should be printed along with the count. The parameter should be setable with the install of the module or via input parameter along with module configuration files.

To create/initialize a timer, look for the 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 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
```

Get a screenshot of the following items to include in your report in addition to your kernel code:

- Screenshots of the install and successful load of the module
- Output print buffer of your count printing your name and the count
- Screenshots of the module info
- Screenshots of the module remove

[Problem 6 - 20 Pts] Doubly Linked List Data Structure

You are to implement a doubly linked data structure in its own library (*.c and*.h) that you could use in Linux and also on our TIVA board. You must write your own code (no copying from the internet). The data structure must be written independent of the elements you are trying to store. An example structure definition is below where the node is defined independent of the data you are trying to store. The desired info data structure that you are trying to create a linked list of, should include a linked list node element at the bottom of the structure definition. All code must adhere to our coding style guidelines. **All functions for these structures must take a pointer to the data structure as an input.**

```
struct node {
   struct node * prev;
   struct node * next;
};

struct info {
   uint32_t data;
   struct node;
};
```

The doubly Linked List library must have the following features

- A *.c and *.h with all public function documentation in the header file
- The linked list functions must take a head pointer
- If a link list has not been allocated, you should allocate one with any of the add functions.
- The ends pointers of the linked list should be terminated with a NULL pointer
- All operations must be done with pointers.
- If a linked list operation must fail, return a NULL pointer
- The linked list must use dynamic memory allocation

You need to implement a macro that can return the pointer of the outer containing data structure the linked list is contained in give a pointer to the node structure. This macro needs to be called **GET_LIST_CONTAINER**. This should be defined as:

GET_LIST_CONTAINER(addr, type, member)

addr = Pointer to your linked list
type = Containing structure type
Field = linked list member name from the structure (node)

You need to support the following functions:

- destroy
  - **Description:** Destroy all nodes in the linked list by freeing the memory
  - **Input Parameters:** a linked list data structure pointer
  - **Returns:** Pointer to the head of the linked list
- insert_at_beginning
  - **Description:** Add a node to the beginning of the linked list. Should add head node if it does not exist.
  - **Input Parameters:** the head node pointer and the data to add
  - **Returns:** Pointer to the head of the linked list
- insert_at_end
  - **Description:** Add a node to the end of the linked list. Should add head node if it does not exist.
  - **Input Parameters:** the head node pointer and the data to add
  - **Returns:** Pointer to the head of the linked list
- insert_at_position
  - **Description:** Add a node to a given position of the linked list.
  - **Input Parameters:** a base node pointer, data to add, and the index of where to add the data
  - **Returns:** Pointer to the head of the linked list
- delete_from_beginning
  - **Description:** Delete a node to the beginning of the linked list
  - **Input Parameters:** the head node pointer
  - **Returns:** Pointer to the head of the linked list
- delete_from_end
  - **Description:** Delete a node to the end of the linked list
  - **Input Parameters:** the head node pointer
  - **Returns:** Pointer to the head of the linked list
- delete_from_position
  - **Description:** Delete a node to a given position of the linked list
  - **Input Parameters:** a base node pointer and the index of where to add the data
  - **Returns:** Pointer to the head of the linked list
- Peek_value
  - **Description:** Return the data from the linked list item at a given position
  - **Input Parameters:** A base node pointer and the index to where to get data from
  - **Returns:** The data of the item pointer you have
- size
  - **Description:** Determine the number of links in your linked list
  - **Input Parameters:** a node pointer
  - **Returns:** size of linked list in nodes

[Problem 7 - 20 Pts] Unit Tests

You are to now implement unit tests for the data structures you coded in the previous problem. You are welcome to use either cmocka (https://cmocka.org/) or unity (http://www.throwtheswitch.org/unity/) Or other unit test frameworks of your choice.

Examples can be seen at:

- https://github.com/afosdick/ecen5013/tree/develop/tutorials/unit_tests
- https://github.com/ThrowTheSwitch/Unity/blob/master/examples/example_1/test/TestProductionCode.c

You need to have at least 1 test case for each function. These test should look at boundary conditions, undefined operation, expected operation, etc. For example, a double linked list should be able to insert at the head, in the middle and at the tail. The insert and delete should return expectedly if the list is empty. How you run your unit tests should be documented in a README for each of the two data structures. Add your unit tests code to your git repository in an appropriate directory. Include your unit test output in pdf that you turn in to **D2L**.