THE UNIVERSITY OF SYDNEY

COMP2129 Assignment 5

Due: 11:59pm Sunday, 9th June 2013

Task description

In this task you are asked to implement a program which will simulate a simple device manager. A device manager is a program which listens for when devices are attached to the computer, and manages incoming data from each of them to feed to the rest of the system. It also monitors the status of the attached devices, including whether any errors have occurred along with their status being either connected or disconnected.

This assignment is split into three stages. Each of these stages will be marked independently of one another. The stages are designed so that each stage is an extension of the last: your program should keep doing everything defined in the previous stages, and you should add more functionality to it. It is highly recommended that you use version control to keep your track of any changes made to your assignment using a repository, allowing you to easily revert to previous working versions if required.

This assignment will be marked automatically – even so, you should keep your code neat, orderly and logical as this will help you in extending to the following stages of the assignments.

Your task is to write a program called **`devm`** implements those stages of operation outlined in the rest of this document. You are left to define C files and header files; you may find it helpful to be able to split your code up across multiple `.c` and `.h` files for this assignment. The Makefile must produce a **`devm`** binary after running the **`$ make`** command and also remove any binaries or object files your build process creates running the **`$ make clean`**. Upon submission using the **`$ make submission`** command, the **`Makefile`** along with all `.c` and `.h` files in your assignment folder will be submitted.

Hint: You are advised to follow Piazza to receive any updates concerning the assignment

FIFOs

FIFOs, or named pipes are features of UNIX systems which allow locations on the filesystem to be associated with pipes that would usually be shared between a parent and a child process. You can open and read FIFOs like normal files, with one important difference: any read operation on these will *block* until there is new data to be read, or another process has closed the FIFO. This means that you need to think about how and when to read from these named pipes in order to stop your program from becoming unresponsive. Any files which are FIFOs will be explicitly stated in this document.

Although FIFOs will be used during correctness testing, you are able to use regular files for your own testing. From the point of view of your program, a regular file will be like a FIFO that never blocks.

Stage one — Catching signals and identifying devices

For this stage, your program **devm** needs to catch signals issued by our testing program, which will indicate a new device connection. It then needs to read from a FIFO to identify the device type, and log all activity along the way.

A dictionary of devices which may be connected will be given, in a file called **known_devices**. Each device has a unique identifier (ID) associated with it, which you can look up in this dictionary to determine what type of device it is. For this assignment, only two types of devices will be connected: type **mouse** or **keyboard**. The following is an example of the **known_devices** file:

```
ID 0461:4d22 Primax Electronics, Ltd
ID 056a:00dd Wacom Co., Ltd Tablet
ID 045e:00dd Microsoft Corp. Comfort Curve Keyboard 2000 V1.0
ID 046d:c016 Logitech, Inc. Optical Wheel Mouse
ID 1d6b:0002 Linux Foundation 2.0 root hub
ID 413c:2003 Dell Computer Corp. Keyboard
ID 8087:0024 Intel Corp. Integrated Rate Matching Hub
ID 05fe:0011 Chic Technology Corp. Browser Mouse
```

The ID takes the form **0123:abcd** where any digit may be a hexadecimal number. Any hexadecimal digit will be a base-10 digit **[0-9]**, or a lowercase character **[a-f]**. You must search each line to find out what type of device the ID corresponds to; you should be looking for the string "mouse" or "keyboard", which will appear somewhere in the line, or not at all. Also note from the above example that you should be searching case-insensitively. If neither string "mouse" or "keyboard" appear in the line, you may discard that line (for the purposes of this assignment).

When your program **devm** starts, it should immediately start logging its activities to a file called **devm.log**. This file should be opened in append mode, such that previous data will not be erased and for the henceforth this document will be referred to as "the log file" or "the log".

Each line of the log file should look like this (using a UNIX timestamp[1]:

```
[<timestamp>] <message>
```

When **devm** first starts up, it should write the following message to the log file: **devm started**. In the log file itself, that line would look like:

```
[1368869693] devm started
```

Upon startup, your program should also write its PID (process ID) to a file called **devm.pid**. Upon termination of your program, you should remove this file. Your program should then read in the previously mentioned **known_devices** file, and then go into a waiting state. From this waiting state it should catch the signals SIGUSR1 and SIGTERM.

Your program **devm** should catch the UNIX signal SIGUSR1, which will communicate to the program that a new device has been connected. Upon receiving this signal, **devm** should read a single line from the FIFO **new_devices**, which will contain exactly one ID corresponding to a device, as above. An example line from **new_devices** is:

```
045e:00dd
```

[1] Number of seconds since 1 January 1970. See http://en.wikipedia.org/wiki/Unix_time

This signal indicates that the device with the ID just read has been connected. You should log this activity in a message **`Device connected: <device type> <device id> at <device path>`**, where **`device type`** will be either **`keyboard`** or **`mouse`** corresponding to the ID read from **`known_devices`**, **`device id`** will be the ID read in from **`new_devices`**, and **`device path`** will be **`dev/USBx`**, where **`x`** corresponds to the connection order of the devices. The first device will be connected with pathname **`dev/USB0`**, the next with **`dev/USB1`**, and so forth. An example line from the log upon the connection of device ID **`045e:00dd`** would be:

```
[1368869693] Device connected: keyboard 045e:00dd at dev/USB0
```

Your program should also catch SIGTERM, which will indicate that **`devm`** should terminate. Upon catching this signal, you should write the message **`SIGTERM caught, devm closing`** to the logfile, and then your program should terminate. An example log line for this is:

```
[1368869693] SIGTERM caught, devm closing
```

For the first stage of the assignment, your program will be marked on the following:

- Writing the correct lines to the log file for open and close of **`devm`**.
- Catching SIGUSR1 and logging the correct device connection to the log file.
- Terminating on SIGTERM.
- Having a correct timestamp for each line of the log (with a tolerance of ± 5 seconds).
- Writing the correct PID, and removing the pidfile on exit.

The following is a state diagram of **`devm`** for this stage of the assignment.

It is intended to be a guide only — please refer to the descriptions above for exact behaviour.

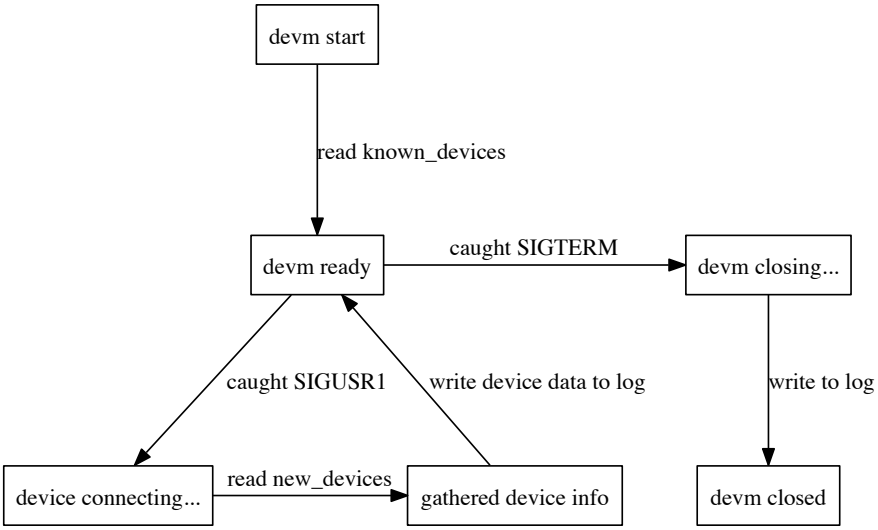


Figure 1: State diagram for Stage 1

Stage two — Multiplexing input

For this stage, upon a device connecting, your program needs to create a new process to act as an independent manager for that device. The manager should read data packets from the device in question, and indicate that the device has been disconnected when there are no more packets to read.

Upon a new device connecting, your program should create a new process (the "device process") for the device to run independently on (a monitoring process for the device). Each device connected is identified with a FIFO **`dev/USB0`**, **`dev/USB1`**, and so on in the order of connection. This FIFO will be henceforth called "the device file".

The device process should open the device file upon being started, and begin to read data from the device file. The data in the **`dev/USBx`** files correspond to binary packets from the device in question, and as such cannot be read as text. How you should read and interpret the data depends on what sort of device is being connected.

For a mouse device, we will use a common (real world) data format for a three button mouse (scroll wheel inclusive). A data packet from a mouse has a size of exactly three bytes, so you should read from a mouse input file in three byte chunks. Once a packet has been read, you should write the message **`Device packet: <packet> from <device type> <id> at <device path>`**, where **`packet`** is a hexadecimal representation of the bytes sent. For example, upon receiving the bytes **`0xaa`**, **`0xbb`** and **`0xcc`** from a mouse, you might output:

```
[1368869695] Device packet: 0xaabbcc from mouse 1234:4321 at dev/USB1
```

A real world keyboard has a complicated data format, so throughout this assignment we will use a simplified version. A data packet from a keyboard has a size of exactly one byte, and as such you should read from a keyboard input file in as a simple character. Once a packet has been read, you should write a similar line to when you log a mouse packet, here the number of the number of bytes read in will only be 1. A sample line might look like:

```
[1368869694] Device packet: 0xaa from keyboard 5678:8765 at dev/USB0
```

Once there is no more data to read from the device file, you should close it, and log a disconnected event for the device in the following format (following the above example)

```
[1368869694] Device disconnected: keyboard 5678:8765 at dev/USB0
```

There is a chance that when reading a multiple byte packet from a mouse, you will receive an EOF (end of file) partway through the packet, after reading only one or two bytes. In this case, you should log an error and disconnect the device. The relevant log lines for this particular case would look like:

```
[1368869694] Error while receiving packet: keyboard 5678:8765 at dev/USB0
[1368869694] Device disconnected: keyboard 5678:8765 at dev/USB0
```

For the second stage of the assignment, you will be marked on the following

- Creating a new process for each new device connected.
- Correctly reading packets, and logging them in the correct order.
- Correctly identifying when an error has occurred.
- Logging a disconnect and terminating the process when the end of file is reached for the device file, or an error has occurred.

Note also that you must continue to fulfill the requirements of stage one — this stage is an extension, not a replacement. It is also important that upon receiving SIGTERM to your main process that your entire process terminates properly: you must log a disconnect for all devices and make sure you kill all of the device processes, and close all open files.

Hint: Ensure that you do not leave zombie processes hanging around for too long!

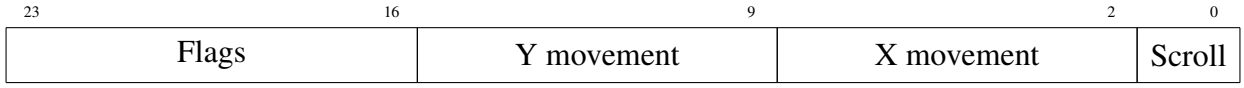
Stage three — Interpreting device packets

For each data packet received from each device, your program should now interpret it and write it in a sensible format to a file called **`output/USBx`**, where **`x`** corresponds to the connected order

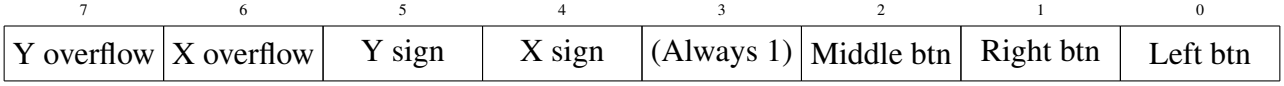
Hint: Ensure you create an **`output/`** directory if it does not already exist

Mouse packets

Mouse packets were previously stated to be three bytes long. These bytes are a compacted message that requires decoding. The packet is an encoded four field wide message of characters:



Flags The flags byte is expanded as follows:



where each box in this diagram indicates one bit.

The overflow bits indicate whether the X or Y movement bytes have overflowed (the values have become too big for a byte to hold). For this assignment, we do not care about this, simply ignore the values in these bits. If the X or Y sign bits are set, this means that the value in the X or Y movement byte should be negated. The "always 1" bit should always be 1, but simply ignore it (it's just there for padding really). Finally, if any of the button bits are set, that corresponds to a button on the mouse being held down.

Movement There are three movement fields: X movement; Y movement; and Scroll movement. The encoding scheme assumes the maximum required resolution for mouse **`dx`** and **`dy`** calls for a seven bit integer. Further, the scroll wheel state is transmitted as either rotated up (2), down (1), or remaining stationary (0), requiring an extra two bits at the end of the word N.B. the state **`11`** is redundant. For instance, if the movement word **`0x0902`** was received it would translate to:

$$\texttt{0x0902} = \overbrace{\texttt{0000100}}^{\texttt{dy}}\ \overbrace{\texttt{1000000}}^{\texttt{dx}}\ \overbrace{\texttt{10}}^{\text{scroll}}$$

Upon receiving each three byte packet from a mouse device, you should unpack its data, and write a line to the corresponding **`dev/USBx`** file, which is of the following format:

```
dx: <dx>, dy: <dy>, sc: <scroll>, left: <on/off>, middle: <on/off>, right: <on/off>
```

where **`dx`** and **`dy`** correspond to the X or Y movement, including the plus or minus sign

For example, upon receiving the input:

```
0x090205
0x2a0902
```

your program should output:

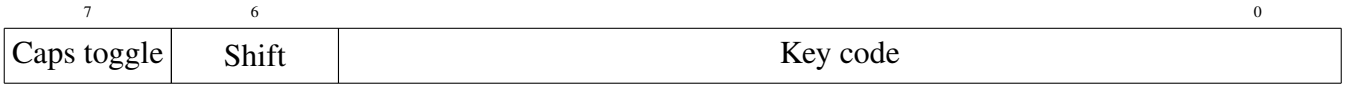
```
dx: +1, dy: +1, sc: -1, left: on, middle: off, right: off
dx: +64, dy: -4, sc: +1, left: off, middle: off, right: on
```

If any of **dx**, **dy**, or **sc** are 0, you should output **+0**

Keyboard packets

Real-world keyboard PCI involves communicating one or more 8 bit words representing the key codes of the keyboard. The key codes are transmitted as either MAKE or BREAK keycodes, to instruct the host whether the key is still being pressed or has been released, respectively. In addition to key codes, commands might be exchanged between the keyboard and host.

To minimize workload, we are transmitting a truncated, ad hoc scheme for key codes. The packet is a single byte consisting of 6 bits for the standard key code pressed, 1 bit for shift key, and 1 bit for *toggling* the caps lock.



The standard character set used in this assignment is a constant string literal called **CHAR_SET**. Note that in c string literals, the '\' is an escape character used to embed the double quotation mark ('\"'). To use the backslash, another backslash must be placed after it ('\\').

```
const char CHAR_SET[] =
    "0123456789" // {0-9}
    "-=[]\\;',./" // {10-19}
    "abcdefghijklmnopqrstuvwxyz" // {20-45}
    /* standard set end */
    ")!@#$%^&*("
    "_+{}|:\"<>?"
    "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
    /* shifted set end */
```

The shift and caps lock keys shall behave as expected of a typical keyboard. Whilst the shift key bit is set, all ascii *letters*, *numbers*, and *punctuation* characters from the lowercase set and replaced with the respective upper case character (i.e. **`shift'+`a' = `A'**, **`shift'+`0' = `!'**). If the caps lock is 'on' (i.e. it has been toggled on) the ascii *letters only* shall be replaced with the respective upper case character (i.e. **`shift'+`a' = `A'**, **`shift'+`0' = `0'**).

For example, for the following sequence of messages:

```
0x14 (`a')
0x54 (`shift'+`a')
0x15 (`b')
0x55 (`shift'+`b')
0x01 (`1')
0x41 (`shift'+`1')
0x80 (`toggle caps lock'+`0')
0x14 (`a')
0x15 (`b')
0x01 (`1')
0x91 (`toggle caps lock'+`,')
0x15 (`b')
0x01 (`1')
```

Your program should output

```
a
A
b
B
1
!
0
A
B
1
,
b
1
```

Marking

As mentioned previously in the assignment specification, marks are allocated base on the correctness of each stage. The automatic marker will test all stages regardless of whether the previous stages failed or not. Simply put, the marker is an optimist and looks favourably towards the future.

4 **marks** are assigned based on automatic tests for your correctness in stage one.

4 **marks** are assigned based on automatic tests for your correctness in stage two.

2 **marks** are assigned based on automatic tests for your correctness in stage three.

Warning: Any attempts to deceive the auto marker will result in an immediate 0 for the assignment.