1 Installation of Raspbian and system calls

1.1 Resources for COSC440

Skeleton code for all exercises in COSC440 labs are available at:

http://www.cs.otago.ac.nz/cosc440/resources.php

Also the schedule, lectures notes and course reading materials are available at:

http://www.cs.otago.ac.nz/cosc440/schedule.php

Please ensure you read the materials before attending the lectures and labs.

1.2 Installation of Raspbian

Raspbian is a free operating system based on Debian optimized for the Raspberry Pi (RPI) hardware. You can find more information at http://www.raspbian.org/FrontPage.

NOOBS (New Out Of the Box Software) is an easy operating system install manager for the RPI. You can find out details of how to install Raspbian with NOOBS at http://lp.torchbrowser.com/?sysid=448&appid=260

Raspbian has been already installed on the SD card you have got. Just follow the instructions of the installation procedure.

Once you have installed Raspbian and reboot your pi, you can start a terminal window and run **raspi-config** to configure the options of your Raspbian.

Once your system boots, you should have logged in the user account **pi** but remember the password **raspberry** that will be needed when **sudo** is used.

You need to maintain your own Raspbian to hack in Linux kernel. This makes the hack real but affects no other users.

You should retrieve new lists of packages from the repository:

```
$ sudo apt-get update
$ sudo apt-get upgrade
```

You should install necessary packages:

```
$ sudo apt-get install build-essential
$ sudo apt-get install vim
$ sudo apt-get install libncurses5-dev
$ sudo apt-get install screen
```

1.3 Kernel configuration and compilation

You need kernel headers to compile your own kernel modules. To install the most recent Linux kernel headers on Raspberry Pi:

```
$ sudo apt-get install raspberrypi-kernel-headers
```

Alternatively, you can download Linux kernel source and compile your own Linux kernel. This exercise is good for you to know how to configure the kernel to suit your own purposes (e.g. for embedded systems or make the kernel leaner by removing unnecessary components/modules).

Note it will take a lot of time (several hours) to compile the kernel from the Linux source code. So continue the lab with the next section or read the reading material of the paper while the kernel is being compiled.

The following steps guide you to reconfigure, compile, and install the kernel image and its related modules.

Find the instructions to download, configure, compile, and install a RPI Linux kernel source at https://www.raspberrypi.org/documentation/linux/kernel/building.md.

Note: before copying the new kernel into /boot, make sure the old kernel **kernel7.img** is saved to a different name. In case the new kernel fails to work, you can still boot with the old kernel by changing the file **/boot/config.txt** with the following line:

```
kernel=old-kernel7.img
```

After the new kernel is installed, reboot the system:

```
$ sudo reboot
```

Once your RPI reboots, use:

```
$ uname -a
```

to check if your compiled kernel version is **5.10.xx**

In case your RPI fails to reboot, power off and then power on your RPI while holding down the **shift** key. You will be prompted with a recovery menu where you can change the **config.txt** file to boot the old kernel as suggested before.

1.4 System calls for file operations

While your kernel is being compiled, let's have a recap on system calls used in file operations in the user space. The rest of this lab can be completed using your host desktop OS X terminal.

Many devices are abstracted as a file in Linux/Unix. The standard operations are open(), close(), read(), write(), ioctl(), lseek(), and etc, which are called system calls and are the API for user applications to request for kernel services. They are different from the libc functions fopen(), fclose(), fread(), fwrite(), which are based on the system calls, but can prefetch and buffer data at the user space in order to avoid unnecessary system calls. System calls are more expensive than function calls because they involve context switch from user space to kernel space and then a switch back to user space. Below are the brief semantics of the file related system calls.

1.4.1 open()

```
int open(const char *pathname, int flags);

int open(const char *pathname, int flags, ... /* mode_t mode */);
```

When using the function, the following header files should be included in your program (we will put these files before each system call for the rest of the course material):

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
```

For a given file name pathname, open() returns a file descriptor, which is a small, non-negative integer for use in subsequent system calls such as read() and write(). A call to open() will cause the kernel to create a file structure in the kernel space for the calling process.

The second argument flags specifies the options of open(). A file can be open for reading only or for both reading and writing. For example, the following constants (defined in fcntl.h) are often used:

O_RDONLY	open for reading only
O_WRONLY	open for writing only
O_RDWR	open for reading and writing

The argument is formed by OR'ing together different options, e.g. O_RDWR| O_CREAT|O_NONBLOCK. To find out other options, use the Linux manual pages by the command

```
$ man 2 open
```

The third argument is only used when a new file is being created, in which case a protection mode such as 0x700. However, to avoid errors, the following constants (defined in sys/stat.h) should be used:

S_IRWXU	00700 user (file owner) has read, write and execute permission
S_IRUSR	00400 user has read permission
S_IWUSR	00200 user has write permission
S_IXUSR	00100 user has execute permission
S_IRWXG	00070 group has read, write and execute permission
S_IRGRP	00040 group has read permission
S_IWGRP	00020 group has write permission
S_IXGRP	00010 group has execute permission
S_IRWXO	00007 others have read, write and execute permission
S_IROTH	00004 others have read permission
S_IWOTH	00002 others have write permission
S_IXOTH	00001 others have execute permission

mode must be specified when O_CREAT is in the flags, and is ignored otherwise.

If successful, open() returns a non-negative file descriptor; otherwise, it returns -1, in which case **errno** can be used to check the error number and message.

In order to check the value of **errno**, its header needs to be included:

```
#include <errno.h>
```

Then with the **errno**, it is useful to print a human-readable error message, and this can be done with

```
#include <errno.h>
#include <stdio.h>

void perror(const char *s);
```

which takes a string (usually the name of the system call) and prints out an error message.

Note that it is a standard for system programming to always check the return value of system calls and take corresponding actions when error occurs. The following common errors are worth noting:

- **EAGAIN/EWOULDBLOCK** Non-blocking I/O has been selected using O_NONBLOCK and there was no data immediately available for reading.
- **EFAULT** You provided an invalid buffer space that is outside your accessible address space.
- **EINTR** The call was interrupted by a signal before any data was read. You normally should repeat the system call on this error.

However, each system call has its own set of errors. For more detailed information about errors of each system call, use the Linux manual pages **man 2 <syscall-name>**.

1.4.2 creat()

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int creat(const char *pathname, mode_t mode);
```

It is equivalent to:

```
#include <unistd.h>

int open(pathname, O_WRONLY|O_CREAT|O_TRUNC, mode);
```

1.4.3 read()

```
#include <unistd.h>

ssize_t read(int fd, void *buf, size_t count);
```

It reads **count** bytes of data from an open file **fd** into the user buffer **buf**. If successful, the number of bytes read is returned; if the end of file is reached, **0** is returned; otherwise, **-1** is returned on error.

Note that the number of bytes returned can be less than the amount requested (i.e. count). For example, if there are 70 bytes left until the end of file and you try to read 100 bytes, read() returns only 70. Also if you read from a socket (a special kind of file descriptor), the amount of data requested has not yet arrived from the network and read() just returns whatever in the socket buffer. It is always a good practice to compare the return value with the requested amount (count) in order to react accordingly.

Here is a code snippet showing how read() is used to read from a file descriptor into a fixed-size buffer, then process the data in the buffer, and read again until the EOF is reached, or until the socket is empty:

```
#include <errno.h>
#include <unistd.h>

#define BUF_SIZE 100
char buf[BUF_SIZE];

size_t size_to_be_read = BUF_SIZE;

restart:

while ((size_read = read(fd, buf, size_to_be_read)) > 0) {
  /* process the data in the buffer..... */
}

if (size_read < 0) {
  if (EINTR == errno) {
    /* read() interrupted by a signal, so let's try again */
    goto restart;
  } else {
    /* other error, so print a message and quit */
    perror("read()");
    exit(EXIT_FAILURE);
  }
}

/* size_read == 0, so EOF reached, or socket disconnected
   task completed */
```

1.4.4 write()

```
#include <unistd.h>

ssize_t write(int fd, const void *buf, size_t count);
```

It requests the kernel to write **count** bytes from the buffer pointed by buf into the file fd. The return value usually equals to **count** if successful. However, the return value can be less than count when the physical medium of the file has no enough space. You should check the return value in order to make sure all data are successfully stored in the file. In case of socket, you should write the rest of the data again hoping the socket buffer or the network capacity is available soon.

Here is a code snippet showing the usage of write():

```
size_remaining = size_read;
size_written = 0;
```

```
while ((size_written_this_time = write(fd, &buf[size_written], size_remaining))
         < size_remaining) {
  if (size_written_this_time < 0) {
    /* something wrong */
    if (EINTR == errno) {
      /* interrupted, start again */
      continue;
    } else {
      perror("write()");
      exit(EXIT_FAILURE);
    }
  } else {
    /* write() wrote less than required, so we need to
       update size_written and size_remaining
       and call write() again to write out the remaining data */
    size_written += size_written_this_time;
    size_remaining -= size_written_this_time;
  }
}
```

1.4.5 close()

```
#include <unistd.h>

int close(int fd);
```

It closes a file referred to by the file descriptor **fd**, so that your program no longer accesses the file. It returns **0** on success but **-1** on error.

1.4.6 ioctl()

```
#include <sys/ioctl.h>

int ioctl(int fd, int request, ... /* char *argp */);
```

ioctl() is mainly used by device drivers which is an essential part of the paper. It manipulates the underlying device parameters through different requests. fd is normally a file descriptor referring to an opened device file such as /dev/tty. The request is device dependent. The third argument argp points to a buffer that contains data to be passed to the device driver. You will learn more about ioctl() when you work on the implementation of device drivers.

It returns **0** on success but **-1** on error.

1.4.7 lseek()

```
#include <sys/types.h>
#include <unistd.h>

off_t lseek(int fd, off_t offset, int whence);
```

Every open file has an associated "current file offset", which points to the current position where read() and write() apply. lseek() can change this position and repositions the offset of the open file fd to the argument offset according to the directive whence in the following ways:

```
SEEK_SET: The offset is set to offset bytes.

SEEK_CUR: The offset is set to its current location plus offset bytes.
```

```
SEEK_END: The offset is set to the size of the file plus offset bytes.
```

This means lseek() allows the current position to be set beyond the end of the file, which does not change the size of the file until data is written at the position. The subsequent reads of the data in the gap will return null (0) until data is actually written into the gap.

It returns the resulting offset position (in terms of the beginning of the file) on success.

There are many other system calls related to a file. However, the above system calls are the most important to device drivers and will be implemented when you go through the implementation of device drivers.

1.5 Assessment

In this lab, you should implement your cat program, which can read from a file and write the content of the file to the standard output referred by the file descriptor STDOUT_FILENO. Standard input, output and error files are automatically open when a program is loaded for execution and can be referred to by the file descriptors STDIN_FILENO, STDOUT_FILENO and STDERR_FILENO (defined in stdio.h). For example, the following command

```
$  my_cat test.txt
```

should display the content of "test.txt".

In the program my_cat, you should read from a file (the file name is given through command line argument argv and argc), check any errors such as EINTR (in which case you should repeat the same read()), write to standard output. You should also check the number of bytes read and if the end of file is reached. Also you should check the errors from open() in case of non-existing files or denied access and print out error messages accordingly.

The pseudocode of the cat program (exclude error-checking) is:

```
fd = open(filename, readonly)

while (there is data read from fd into buf) {
  write the data to the standout output
  check the size written,
  if smaller than the data size, update indices
  and called write() again to write out the remaining data
}

close(fd)
```

1.6 Setting up your Git repository

For any projects, such as source code, assignments and theses, it is important to:

1. keep track of everything you added/edited, so that you can revert to a given state should something goes wrong
2. have a distributed backup system so that if your working copy is lost, you can still retrieve your work.

Therefore a distributed versioning system such as git:

http://git-scm.com

should be used. You are required to make use of git to version all assignments in COSC440, and encouraged to do the same to other papers, and your COSC480 dissertation.

To begin with, first, create an account on the cloud-based git server bitbucket

http://bitbucket.org

Then after your account is set up, you can create a git repository for each project. For example cosc440-asgn1. You can also create a repository for the COSC440 labs, and this is a good way to share code between your work machine, and your home linux machine should you wish to work at home in the weekend.

Now, it is a good idea to create a git repository in bitbucket for your first programming assignment "cosc440-asgn1".

Important
Please make sure that the box "private" is clicked.

Then you can check out a working copy on your working machine by following the instructions on github.

Before you can do that on your machine, you need to install git first:

```
$ sudo apt-get install git-core
```

For general usage of git, please have a look at:

http://jonas.iki.fi/git_guides/HTML/git_guide/

http://gitref.org/basic/

1.7 Reference

1. Advanced programming in the UNIX environment, by W.R. Stevens, Addison Wesley
2. Linux manual pages, **man 2 <syscall-name>**