Assignment 2: System Calls and Processes

CSE3201: Operating System

Computer Science and Engineering

University of Dhaka

Version 1.0

Contents

1 Due Date and Marks Distribution

- Due Date: before 23:59:59, 07 Nov, Monday 2018
- Marks: Worth 25 marks (of the 100 available for the lab class mark component of the course)

- The 10% bonus for one week early is available for the basic assignment. Deadline: 30 October, Monday

2 Introduction

In this assignment you will be implementing a set of file-and process-related system calls. Upon completion, your operating system will be able to run multiple copies of a single application at user-level and perform some basic file I/O.

A substantial part of this assignment is understanding how OS/161 works and determining what code is required to implement the required functionality. Expect to spend at least as long browsing and digesting OS/161 code as actually writing and debugging your own code.

Your current OS/161 system has minimal support for running executable, nothing that could be considered a true process. Assignment 2 starts the transformation of OS/161 into something closer to a true multi-tasking operating system. After this assignment, it will be capable of running multiple processes from actual compiled programs stored in your account. The program will be loaded into OS/161 and executed in user mode by System/161; this will occur under the control of your kernel. First, however, you must implement part of the interface between user-mode programs ("userland") and the kernel. As usual, we provide part of the code you will need. Your job is to design and build the missing pieces.

Our code can run one user-level C program at a time as long as it doesn't want to do anything but shut the system down. We have provided sample user programs that do this (reboot, halt, poweroff), as well as others that make use of features you might be adding in this and future assignments. So far, all the code you have written for OS/161 has only been run within, and only been used by, the operating system kernel. In a real operating system, the kernel's main function is to provide support for user-level programs. Most such support is accessed via "system calls." We give you one system call, reboot(), which is implemented in the function *sys_reboot*() in main.c. In GDB, if you put a breakpoint on *sys_reboot* and run the "reboot" program, you can use "backtrace" to see how it got there.

For those attempting the advanced version of the assignment, you will also be implementing the subsystem that keeps track of multiple tasks. You must decide what data structures you will need to hold the data pertinent to a "process" (hint: look at kernel include files of your favorite operating system for suggestions, specifically the proc structure.) The first step is to read and understand the parts of the system that we have written for you.

2.1 User-level programs

Our System/161 simulator can run normal C programs if they are compiled with a cross-compiler, cs161-gcc. This runs on a host (e.g., a Linux x86 machine) and produces MIPS executables; it is the same compiler used to compile the OS/161 kernel. To create new user programs, you will need to edit the Makefile in bin, sbin, or testbin (depending on where you put your programs) and then create a directory similar to those that already exist. Use an existing program and its Makefile as a template.

2.2 Design

In the beginning, you should tackle this assignment by producing a DESIGN DOCUMENT. The design document should clearly reflect the development of your solution. They should not merely explain what you programmed. If you try to code first and design later, or even if you design hastily and rush into coding, you will most certainly end up in a software "tar pit". Don't do it! Plan everything you will do. Don't even think about coding until you can precisely explain to your partner what problems you need to solve and how the pieces relate to each other. Note that it can often be hard to write (or talk) about new software design, you are facing problems that you have not seen before, and therefore even finding terminology to describe your ideas can be difficult. There is no magic solution to this problem, but it gets more comfortable with practice. The important thing is to go ahead and try. Always try to describe your ideas and designs to someone else. In order to reach an understanding, you may have to invent terminology and notation, this is fine. If you do this, by the time you have completed your design, you will find that you have the ability to discuss problems that you have never seen before efficiently. Why do you think that CSE is filled with so much jargon? To help you get started, we have provided the following questions as a guide for reading through the code. We recommend that you answer questions for the different modules and be prepared to discuss with your group member. Once you have prepared the answers, you should be ready to develop a strategy for designing your code for this assignment.

3 Code Walking

Discuss your answers to the code walk-through questions to your assignment partner.

3.1 Running user-level program

kern/{syscall,lib,vm}
These directories contain the files that are responsible for the loading and running of userlevel

programs. Currently, the files in these directories are loadelf.c, runprogram.c, and uio.c, and etc., although you may want to add more of your own during this assignment. Understanding these files is the key to getting started with the assignment, especially the advanced part, the implementation of multi-programming. Note that to answer some of the questions, you will have to look in files outside these directories.

syscall/loadelf.c

This file contains the functions responsible for loading an ELF executable from the filesystem and into virtual memory space. Of course, at this point this virtual memory space does not provide what is normally meant by virtual memory, although there is translation between the addresses that executables "believe" they are using and physical addresses, there is no mechanism for providing more memory than exists physically.

syscall/runprogram.c

This file contains only one function, *runprogram()* , which is the function that is responsible for running a program from the kernel menu. Once you have designed your file system calls, a program started by *runprogram()* should have the standard file descriptors (*stdout*, *stderr*) available while it's running.

In the advanced assignment, *runprogram()* is a good base for writing the *execv()* system call, but only a base. When writing your design doc, you should determine what more is required for *execv()* that *runprogram()* does not need to worry about. Once you have design your process framework, *runprogram()* should be altered to start processes properly within this framework.

lib/uio.c

This file contains functions for moving data between kernel and user space. Knowing when and how to cross this boundary is critical to properly implementing userlevel programs, so this is a good file to read very carefully. You should also examine the code in *vm/copyinout.c*.

Questions

3.1.1. What are the ELF magic numbers?

3.1.2. What is the difference between *UIO_USERISPACE* and *UIO_USERSPACE*? When should one use *UIO_SYSSPACE* instead?

3.1.3. Why can the *struct uio* that is used to read in a segment be allocated on the stack in *load_segment()* (i.e., where does the memory read actually go)?

3.1.4. In *runprogram()*, why is it important to call *vfs_close* before going to usermode?

3.1.5. What function forces the processor to switch into usermode? Is this function machine dependent?

3.1.6. In what file are *copyin* and *copyout* defined? *memmove*? Why can't *copyin* and *copyout* be implemented as simply as *memmove*?

3.1.7. What (briefly) is the purpose of *userptr_t*?

3.2 Traps and SystemCalls

kern/arch/mips

Exceptions are the key to operating systems; they are the mechanism that enables the OS to regain control of execution and therefore do its job. You can think of exceptions as the interface between the processor and the operating system. When the OS boots, it installs an **"exception handler"** (carefully crafted assembly code) at a specific address in memory. When the processor raises an exception, it invokes this, which sets up a **"trap frame"** and calls into the operating system. Since **"exception"** is such an overloaded term in computer science, operating system lingo for an exception is a "trap", when the OS traps execution. Interrupts are exceptions, and more significantly for this assignment, so are system calls. Specifically, *syscall.c* handles traps that happen to be *syscalls*. Understanding at least the C code in this directory is key to being a real operating systems junkie, so we highly recommend reading through it carefully.

locore/trap.c

mips_trap() is the key function for returning control to the operating system. This is the C function that gets called by the assembly exception handler. *enter_new_process()* is the key function for returning control to user programs. *kill_curthread()* is the function for handling broken user programs; when the processor is in **usermode** and hits something it can't handle (say, a bad instruction), it raises an exception. There's no way to recover from this, so the OS needs to kill off the process. The advance part of this assignment will include writing a useful version of this function.

syscall/syscall.c

syscall() is the function that delegates the actual work of a system call off to the kernel function that implements it. Notice that reboot is the only case currently handled. You will also find a function, *enter_forked_process()*, which is a stub where you will place your code to implement the fork system call.

Questions

3.2.1. What is the numerical value of the exception code for a MIPS system call?

3.2.2. Why does *mips_trap()* set *curspl* to *IPL_HIGH* "manually", instead of using *splhigh()*?

3.2.3. How many bytes is an instruction in MIPS? (Answer this by reading *syscall()* carefully, not by looking somewhere else.)

3.2.4. What is the contents of the *struct mips_syscall* stored?

3.2.5. What would be required to implement a system call that took more than 4 arguments?

3.2.6. What is the purpose of *userptr_t*?

3.3 userland/lib/crt0/mips/crt0.S : the C startup code)

There's only one file in here, *crt0.S*, which contains the MIPS assembly code that receives control first when a user-level program is started. It calls *main()* . This is the code that your *execv* implementation will be interfacing to, so be sure to check what values it expects to appear in what registers and so forth.

userland/lib/libc/unix/errno.c

This is where the global variable errno is defined.

userland/lib/libc/arch/mips/syscalls-mips.S

This file contains the machine-dependent code necessary for implementing the userlevel side of MIPS system calls.

syscalls.S

This file is created from *syscalls-mips.S* at compile time and is the actual file assembled to put into the C library. The actual names of the system calls are placed in this file using a script called *gensyscalls.sh* (in *libc/syscalls*) that reads them from the kernel's header files. This avoids having to make a second list of the system calls. In a real system, typically each system call stub is placed in its own source file, to allow selectively linking them in. OS/161 puts them all together to simplify the makefiles.

Questions

3.3.1. What is the purpose of the SYSCALL macro?

3.3.2. What is the MIPS instruction that actually triggers a system call? (Answer this by reading the source in this directory, not looking somewhere else.)

3.4 kern/include

The files *vfs.h* and *vnode.h* in this directory contain function declarations and comments that are directly relevant to this assignment.

Questions

3.4.1. How are *vfs_open*, *vfs_close* used? What other *vfs_()* calls are relevant?

3.4.2. What are *VOP_READ*, *VOP_WRITE*? How are they used?

3.4.3. What does *VOP_TRYSEEK* do?

3.4.4. Where is the *struct thread* defined? What does this structure contain?

3.5 fork()

fork()
Answer these questions by reading the fork() man page and the sections dealing with *fork()* in the textbook.

Questions

3.5.1. What is the purpose of the fork() system call?

3.5.2. What process state is shared between the parent and child?

3.5.3. What process state is copied between the parent and child?

4 Getting Started

4.1 Setting up your account

You're working in pairs for this assignment. Some of the instructions below need to be done for each group member, some need to be performed only once for the overall group. We will indicate this when appropriate, so **pay attention**.

Both partners will need to set up various environment variables for you to access the tools needed for the course. Do the following,

- You will need to add $HOME/os161/bin , and $HOME/os161/bin-${ARCH}, to your PATH.
- You will need to add $HOME/os161/man to your MANPATH .

Note: You must not have "." (or equivalent) prior to "/bin" in your PATH. The build will fail later if you do. Note: doing this is generally a bad idea for security reasons anyway.

Each partner will need to modify their umask to allow their partner to share the assignment files (if your interested, see man umask for details). Do this by modifying your .profile in your home directory. Change the umask command to be the following.

Use your two-digit roll number to create your group number for this session. One of you must create directory for the group as $HOME/osprjXXXX, where XXXX corresponds to your group number (two digits from each roll no, e.g. 0805). For the remainder of the document we will assume you replace references to osprjXXXX with your own group number (e.g., osprj0805).

You will be using cvs to manage the assignments for your group and cvs needs to be told where you will be storing its repository. The easiest way to do this is to set an environment variable in your shell's start up files. Edit your .profile or .bash_profile and add the following lines. Remember to adjust the group number.

```
CVSROOT="$HOME/osprjXXXX/cvsroot"
export CVSROOT
```

Now whenever you log in, you umask and environment variable CVSROOT will be set appropriately. Just on this occasion, run source .bash_profile (or .profile) to avoid having to log out and log back in again.

4.2 Obtaining and setting up the distribution in CVS

In this section, you will be setting up the cvs repository that will contain your code. Only one of you needs to do the following. We suggest your partner sit in on this part of the assignment.

- Change to your group's project (assignment) directory, and create a directory for your cvs repository.

  ```
  % cd $HOME/osprjXXXX
  % mkdir cvsroot
  %
  ```

- Check the permissions match as below.

  ```
  % ls -l
  total 4
  drwxrws---  4 <user_name> <group_name>   4096 Mar 10 15:22 cvsroot
  %
  ```

If not, you have done something wrong with your umask setup and need to fix it.

As a special case, if your permissions look like this:

```
drwxrwx---   4 <user_name> <group_name>  4096 Mar 10 15:22 cvsroot
```

Your directory does not have the setgid bit set. Run this command:

```
chmod g+s cvsroot
```

- Check your CVSROOT environment variable as below.

```
% echo $CVSROOT
$HOME/osprjXXXX/cvsroot
%
```

If your CVSROOT is not an appropriately modified version of the above, then you will need to fix it.

- Initialize the cvs repository. This should be the only time you need to do it for the rest of your assignments.

```
% cvs init
%
```

5 Basic Assignment

5.1 Setting Up Assignment 2

Remember to continue using your modified PATH for this assignment, as outlined in ASST0.

5.1.1 Obtaining and setting up ASST2 in CVS

In this section, you will be setting up the cvs repository that will contain your code. Only one of you needs to do the following. We suggest your partner sit in on this part of the assignment.

- Check your umask is still set appropriately.

```
% umask
007
%
```

- If not, you have done something wrong with your umask setup and need to fix it. Check your CVSROOT environment variable as below.

```
% echo \$CVSROOT
$HOME/osprjXXXX/cvsroot
%
```

XXXX indicate your group number such as osprj0805. If your CVSROOT is not an appropriately modified version of the above, then you will need to fix it.

- Download into you home directory "asst2.zip", unzip, and import the OS/161 sources into your repository as follows

```
% cd asst2/src
% cvs import -m ‘‘Initial import of asst2 OS/161 sources" asst2-src os161 asst2-bas
```

You have now completed setting up a shared repository for both partners. **The following instructions are now for both partners** (you can create a directory for you partner).

- Create "cse3211" in your home directory and in partner's directory. Change to your directory (or parner's directory).

```
% cd ~/cse3211 or cd ~/partner_directory/cse3211
```

- Now checkout a copy of the os161 sources to work on from your shared repository.

```
% cvs checkout asst2-src
```

- You and your partner should now have a asst2src directory to work on. Partner should take a copy of his/her code from "cse3211" of partner's directory and put it to the "$HOME-/cse3211" of his/her system. Do not delete your partner's directory from your system. You will need it to update the assignment code.

6 Building and testing your code

6.1 Configure OS/161 for Assignment 2

Before proceeding further, configure your new sources.

```
% cd ~/cse3211/asst2-src
% ./configure
```

Unlike the previous assignment, you will need to build and install the user-level programs that will be run by your kernel in this assignment.

```
% cd ~/cse3211/asst2-src
\% bmake
```

Note: "bmake" in this directory does both "bmake" and "bmake install".

For your kernel development, again we have provided you with a framework for you to run your solutions for ASST2.

You have to reconfigure your kernel before you can use this framework. The procedure for configuring kernel is the same as in ASST0 and ASST1, except you will use the ASST2 configuration file:

```
% cd ~/cse3211/asst2-src/kern/conf
% ./config ASST2
```

You should now see an ASST2 directory in the compile directory.

6.2 Building for ASST2

When you built OS/161 for ASST1, you ran make from compile/ASST1. In ASST2, you run make from (you guessed it) compile/ASST2.

```
% cd ../compile/ASST2
% bmake depend
% bmake
% bmake install
```

If you are told that the compile/ASST2 directory does not exist, make sure you ran config for ASST2.

6.3 Command Line Arguments to OS/161

Your solutions to ASST2 will be tested by running OS/161 with command line arguments that correspond to the menu options in the OS/161 boot menu.

IMPORTANT: Please DO NOT change these menu option strings!

6.4 Running "asst2"

For this assignment, we have supplied a user-level OS/161 program that you can use for testing. It is called asst2, and its sources live in "src/userland/testbin/asst2".

You can test your assignment by typing "p /testbin/asst2" at the OS/161 menu prompt. As a short cut, you can also specify menu arguments on the command line, example: sys161 kernel "p /testbin/asst2".

Running the program produces output similar to the following prior to starting the assignment.

```
Unknown syscall 55
Unknown syscall 55
Unknown syscall 55
Unknown syscall 55
Unknown syscall 55
Unknown syscall 55
Unknown syscall 55
Unknown syscall 55
Unknown syscall 55
Unknown syscall 55
Unknown syscall 55
Unknown syscall 55
Unknown syscall 55
Unknown syscall 55
Unknown syscall 55
Unknown syscall 55
Unknown syscall 55
Unknown syscall 55
Unknown syscall 55
Unknown syscall 55
Unknown syscall 55
Unknown syscall 55
Unknown syscall 55
Unknown syscall 55
Unknown syscall 55
```

```
Unknown syscall 55
Unknown syscall 55
Unknown syscall 55
Unknown syscall 55
Unknown syscall 55
Unknown syscall 55
Unknown syscall 55
Unknown syscall 55
Unknown syscall 55
Unknown syscall 55
Unknown syscall 55
Unknown syscall 55
Unknown syscall 55
Unknown syscall 55
Unknown syscall 55
Unknown syscall 55
Unknown syscall 55
Unknown syscall 55
Unknown syscall 3
.................
.................
```

asst2 produces the following output on a (maybe partially) working assignment.

```
OS/161 kernel [? for menu]: p /testbin/asst2
Operation took 0.000212160 seconds
OS/161 kernel [? for menu]:
**********
* File Tester
**********
* write() works for stdout
**********
* write() works for stderr
**********
* opening new file "test.file"
* open() got fd 3
* writing test string
* wrote 45 bytes
* writing test string again
* wrote 45 bytes
* closing file
**********
```

```
* opening old file "test.file"
* open() got fd 3
* reading entire file into buffer
* attemping read of 500 bytes
* read 90 bytes
* attemping read of 410 bytes
* read 0 bytes
* reading complete
* file content okay
**********
* testing lseek
* reading 10 bytes of file into buffer
* attemping read of 10 bytes
* read 10 bytes
* reading complete
* file lseek okay
* closing file
**********
* testing fork
* Forked, in parent
* Forked, in child
Unknown syscall 0
```

7 The Assignment: File System Calls

Implement the following file-based system calls. The full range of system calls that we think you might want over the course of the semester is listed in *kern/include/kern/syscall.h*. For this assignment you should implement: *open*, *read*, *write*, *lseek*, *close*, *dup2*. You should also implement the *fork* system call. Note: You are implementing the kernel code that implements the system call functionality within the kernel. The C stubs that user-level applications call to invoke the system calls are already automatically generated when you build OS/161.

Note: the file-system related system calls are worth 85% of this assignment. You should only implement fork when you are confident your file-system syscalls works. If you find that you are having trouble with this assignment, you can still get a good mark by concentrating on the file-system related system calls. Note, however, that even if you do not implement fork, your implementation should not assume a single process.

It's crucial that your syscalls handle all error conditions gracefully (i.e., without crashing OS/161.) You should consult the OS/161 man pages included in the distribution and understand fully the system calls that you must implement. You must return the error codes as described in the man pages.

Additionally, your syscalls must return the correct value (in case of success) or error code (in case of failure) as specified in the man pages. Some of the auto-marking scripts rely on the return of appropriate error codes; adherence to the guidelines is as important as the correctness of the implementation.

The file *include/unistd.h* contains the user-level interface definition of the system calls that you will be writing for OS/161. This interface is different from that of the kernel functions that you will define to implement these calls. You need to design this interface and put it in *kern/include/syscall.h*. As you discovered (ideally) in Assignment 0, the integer codes for the calls are defined in *kern/include/kern/syscall.h*. You need to think about a variety of issues associated with implementing system calls. Perhaps, the most obvious one is: can two different user-level processes (or user-level threads, if you choose to implement them) find themselves running a system call at the same time?

`open(), read(), write(), lseek(), close(), and dup2()`

For any given process, the first file descriptors (0, 1, and 2) are considered to be standard input (stdin), standard output (stdout), and standard error (stderr) respectively. For this basic assignment, the file descriptors 1 (stdout) and 2 (stderr) must start out attached to the console device ("con:"). You will probably modify runprogram() to achieve this. Your implementation must allow programs to use dup2() to change stdin, stdout, stderr to point elsewhere. Although these system calls may seem to be tied to the filesystem, in fact, these system calls are really about manipulation of file descriptors, or process-specific filesystem state. A large part of this assignment is designing and implementing a system to track this state. Some of this information (such as the cwd) is specific only to the process, but others (such as offset) is specific to the process and file descriptor. Don't rush this design. Think carefully about the state you need to maintain, how to organize it, and when and how it has to change.

While this assignment requires you to implement file-system-related system calls, you actually have to write virtually no low-level file system code in this assignment. You will use the existing VFS layer to do most of the work. Your job is to construct the subsystem that implements the interface expected by user-level programs by invoking the appropriate VFS and vnode operations.

While you are not restricted to only modifying these files, please place most of your implementation in the following files: function prototypes and data types for your file subsystem in *src/kern/include/file.h*, and the function implementations and variable instantiations in *src/kern/syscall/file.c*.

fork()

For the basic assignment, you will implement a simplified form of the *fork()* system call. Your implementation of fork should be the same as that described in the man page, except that it should return 1 to the parent (rather than the child's PID). The amount of code to implement fork is quite small; the main challenge is to understand what needs to be done. We strongly encourage you to implement the file-related system calls first, with fork in mind.

Some hints:

- Read the comments above mips_usermode() in *kern/arch/mips/locore/trap.c*
- Read the comments in kern/include/addrspace.h, particularly as_copy.
- You will need to copy the trapframe from the parent to the child. You should be careful how you do this, as there is a possible race condition (where?/why?).
- You may wish to base your implementation on the thread_fork() function in kern/thread-/thread.c .

A note on errors and error handling of system calls

The man pages in the OS/161 distribution contain a description of the error return values that you must return. If there are conditions that can happen that are not listed in the man page, return the most appropriate error code from kern/errno.h. If none seem particularly appropriate, consider adding a new one. If you're adding an error code for a condition for which Unix has a standard error code symbol, use the same symbol if possible. If not, feel free to make up your own, but note that error codes should always begin with E, should not be EOF, etc. Consult Unix man pages to learn about Unix error codes.

Note that if you add an error code to src/kern/include/kern/errno.h, you need to add a corresponding error message to the file src/userland/lib/libc/string/strerror.c.

8 Design Questions

Here are some additional questions and issues to aid you in developing your design. They are by no means comprehensive, but they are a reasonable place to start developing your solution.

What primitive operations exist to support the transfer of data to and from kernel space? Do you want to implement more on top of these?

You will need to "bullet-proof" the OS/161 kernel from user program errors. There should be nothing a user program can do to crash the operating system when invoking the file system calls. It is okay in the basic assignment for the kernel to panic for an unimplemented system call (e.g. execv()), or a user-level program error.

Decide which functions you need to change and which structures you may need to create to implement the system calls.

How you will keep track of open files? For which system calls is this useful?

For additional background, consult one or more of the following texts for details how similar existing operating systems structure their file system management:

- Section 10.6.3 and "NFS implementation" in Section 10.6.4, Tannenbaum, Modern Operating Systems.
- Section 6.4 and Section 6.5, McKusick et al., The Design and Implementation of the 4.4 BSD Operating System.
- Chapter 8, Vahalia, Unix Internals: the new frontiers.
- The original VFS paper is available here.

9 Documenting your solution

This is a compulsory component of this assignment. You must submit a small design document identifying the basic issues in this assignment, and then describe your solution to the problems you have identified. The design document you developed in the planning phase (outlined above) would be an ideal start. The document must be plain ASCII text. We expect such a document to be roughly 500 - 1000 words, i.e. clear and to the point.

The document will be used to guide our markers in their evaluation of your solution to the assignment. In the case of a poor results in the functional testing combined with a poor design document, we will base our assessment on these components alone. If you can't describe your own solution clearly, you can't expect us to reverse engineer the code to a poor and complex solution to the assignment.

Create your design document to the top of the source tree to OS/161 (/asst2-src), and include it in cvs as follows.

```
% cd ~/cse3211/asst2-src
% cvs add design.txt
```

When you later commit your changes into your repository, your design doc will be included in the commit, and later in your submission.

Also, please word wrap you design doc if your have not already done so. You can use the unix *fmt* command to achieve this if your editor cannot.

10 Basic Assignment Submission

You will be submitting a diff of your changes to the original tree.

You should first commit your changes back to the repository using the following command. Note: You will have to supply a comment on your changes. You also need to coordinate with your partner that the changes you have (or potentially both have) made are committed consistently by you and your partner, such that the repository contains the work you want from both partners.

```
% cd ~/cse3211/asst2-src
% cvs commit
```

If the above fails, you may need to run cvs update to bring your source tree up to date with commits made by your partner. If you do this, you should double check and test your assignment prior to submission.

Beware! If you have created new files for this assignment, they will not be included in your submission unless you add them, using *cvs add* :

```
% cvs add filename.c
```

If you add files after running *cvs commit*, you will need to run *cvs commit* again.

Now tag the repository so that you can always retrieve your current version in the future.

```
% cd ~/cse207/asst2-src
% cvs tag asst2-finish
```

Now generate a file containing the diff.

```
% cvs -q rdiff -r asst2-base -r asst2-finish -u asst2-src > ~/asst2.diff
```

10.1 Testing Your Submission

?? We will post information on course Noticeboard for testing and submitting your assignment.

10.2 Submitting Your Assignment

?? Will be posted later on.

Note: If for some reason you need to **change and re-submit** your assignment after you have tagged it asst2-final, **you will need to either delete the asst2-final tag, commit the new**

changes, re-tag, and re-diff your assignment, or choose a different final tag name and commit the new changes, tag with the new tag, and re-diff with the new tag. To delete a cvs tag, use

```
% cvs rtag -d asst2-final asst2-src
```

Even though the generated diff output should represent all the changes you have made to the supplied code, occasionally students do something "ingenious" and generate non representative diff output.

We strongly suggest keeping a your cvs repository intact to allow for recovery of your work if need be.

11 Plagiarism

We take cheating seriously!!!. We will *moss* to detect code similarity.

Penalties include

- Copying of code: 0
- Help with coding: negative half the assignment's max marks
- Originator of a plagiarised solution: 0 for the particular assignment
- Team work outside group: 0 for the particular assignments
