COMP 3500 Introduction to Operating Systems

Project 2 – An Introduction to OS/161

Points Possible: 100

Submission via Canvas

Important! You can only use the PC assigned to your group in the OS laboratory (Shelby 2129). The minimum penalty for using other groups' PCs in the laboratory is 30 points.

There should be no collaboration among groups and/or students working individually. Students in one group should NOT share any project code with any other group. Collaborations among groups in any form will be treated as a serious violation of the University's academic integrity code.

Objectives:

- Use your installed CentOS to build OS/161 and run Sys/161
- Configure and build OS/161 kernels
- Discover important design aspects of OS/161 by examining its source code
- Manage OS/161 using a version control system called cvs; apply cvs to create a repository and tracking your source code changes
- Use GDB to debug OS/161

1. Introduction

The goal of assignment is to provide you with the opportunity to understand the structures of OS/161 and System/161. OS/161 is an operating system on which you will be working, whereas System/161 is a machine simulator on which the OS/161 runs.

This document is organized as follows. The Section 2 outlines CVS and GDB – the two programming tools. In Section 3, we will discuss the source code of the OS/161 and the aforementioned tools. Sections "Setting up your account" (see Section 4.1) and its subsequent sections provide instructions on what you have to do for project 2 (also referred to as assignment 0). This assignment is focused on giving you an opportunity to explore the OS/161, which is of crucial importance later on. Please carefully read and answer the questions. Furthermore, you must submit a small modification to OS/161 (see Section "Practice modifying your kernel" and add a few useful debugging hooks to the OS/161.

Although some of the questions allude to concepts we have not yet covered so far, the basic understanding of new concepts can easily be found by skimming the pages in the textbook. It is expected that you can consult the code base for answers.

It is more desirable to read or skim the entire assignment followed by set up your account (See Section "Setting up your account") and working through each section of project 2.

2. OS/161 and System/161

The code you will be working on is comprised of two main components:

- OS/161: a simplified operating system developed by the Systems Research at Harvard group, at Harvard University. You will augment the functionality of the OS/161 in subsequent programming assignments.

- System/161: the machine simulator used to emulate the hardware on which your OS/161 will be running. The focus of this course is to design and implement operating systems rather than developing or simulating hardware. You may not need to change the machine simulator, but are required to modify or augment any portion of the OS/161 code that runs on it.

The OS/161 distribution consists of a full operating system source tree, including some utility programs, libraries, and the like. After you build the OS you boot it on the simulator, analogous to booting a real operating system on real hardware. Using a simulated machine is useful in operating system development, because it is more difficult to develop and debug an operating system running on real and bare hardware. The System/161 machine simulator is an excellent platform for rapid development of operating system code, while retaining a high degree of realism. Apart from floating point support and certain issues relating to RAM cache management, the System/161 provides an accurate emulation of a MIPS R3000 processor.

There will be subsequent OS/161 programming assignments for each of the following topics:

- Synchronization
- Processes and Scheduling
- Memory Management
- File Systems

These projects are cumulative in the sense that you are expected make use of code developed in each assignment to complete subsequent ones. Note that you must reuse your own code to do all assignments, and you have to make sure your code is clean.

3. Tools for Project 2: CVS, GDB, and Script

We will learn the following two programming tools that will make your assignments a lot easier. You can find detailed information pertinent to these two tools from the two html files: (1) cvs.html and (2) gdb.html, which are available on Canvas.

3.1 CVS (Concurrent Versions System)

CVS is a source code version control system, which is intended to manage source files of a software package in a way that multiple programmers can develop software simultaneously. In CVS each programmer has a private copy of the source tree and is allowed to independently make modifications. CVS is capable of merging multiple programmers' modifications and detecting conflicts when merges fail.

Although CVS is a powerful system, in case of COMP3500 you are only required to understand a subset of its functionality. The cvs.html (Note: check Canvas for this file) provides the information you need to know for the OS projects. If you intend to learn more about CVS, please take a close look at a comprehensive documentation available at: https://www.cs.utah.edu/dept/old/texinfo/cvs/cvs_toc.html

3.2 GDB (Gnu Debugger)

GDB makes it possible to examine your programs while they are running. Specifically, GDB allows you to execute programs while viewing and setting important values of variables. In particular, GDB enable you to debug OS/161 you are building on the simulated System/161 machine.

Debugging an operating system is similar to debugging ordinary programs. However, a machine has to be reboot when the operating system running on the machine crashes. Fortunately, a machine simulator makes this debugging task easier, because rebooting OS/161 on System/161 takes only a fraction of a second, which is not the case for real operating systems executing on real hardware. More importantly, the System/161 simulator provides information describing what the OS kernel did to cause the crash. Such information may not be easily available when running real OS on real hardware.

To efficiently debug the OS/161, you need to use our version of GDB (referred to as `cs161-gdb`), which is configured for MIPS and has been patched to be able to communicate with your OS kernel through System/161.

A difference between debugging a regular program and debugging an OS/161 kernel is that you need to ensure that you are debugging the operating system rather than the machine simulator. If you were to simply type the obvious thing:

```
% cs161-gdb sys161
```

you would be attempting to debug the simulator. For instructions regarding how to debug your OS kernel and for an introduction to GDB, please refer to the gdb.html file (Note: check Canvas for this file).

3.3 The script Command

The script command line tool allows you to save a session of your terminal. In addition to saving each command per line in a text file, the script command makes a typescript of everything that happens on your CentOS terminal. Screencasting tools to a desktop session(GUI) is what script is to a terminal. Let us demonstrate the usage of script through the following example:

```
$ script
$Script started, file is typescript
$ cd
$ ls
file1 file2 file3
$ exit
exit
Script done, file is typescript
```

Then, you may use the `mv` command to change the file name from `typescript` to any name you like. Alternatively, you may specify the name of your log file upfront as below:

```
$ script sample.script
$Script started, file is typescript
$ cd
$ ls
file1 file2 file3
$ exit
exit
Script done, file is sample.script
```

4. Getting Started

4.1. Setting up Your Account

Important! Please add to your PATH variable the directory containing all of the programs related to OS/161. If you are using bash as your shell you should add the following line near the end of the `~/.bashrc` file.

```
export PATH=~/cs161/bin:$PATH
```

If you use tcsh as your shell, you should add the following line near the end of the `~/.cshrc` file:

```
setenv PATH ~/cs161/bin:$PATH
```

The man page for `tcsh` includes a list of various things you can put in your prompt.

4.2. Getting the Distribution (submit)

You have to download, build and install the distributions of the OS/161, System/161 MIPS emulator, and the OS161 toolchain. You will then need to modify some of these instructions to fit the install paths on your machine. The following five compressed files can be found on Canvas.

1) OS/161: `os161-1.10.tar.gz`
2) tool chain: `cs161-binutils-1.4.tar`
3) cross compiler: `cs161-gcc-1.5.tar`
4) special gdb: `cs161-gdb-1.5.tar`
5) sys161 MIPS emulator: `sys161-1.14.tar`

Important! Please note that the above sequential order is also the required order for building. Detailed information on building these components is included in the `tgz` files. You may choose to quickly go through a file called "The MIPS toolchain for os161.txt" available on Canvas to learn how to extract and build source code files for the tool chain, cross compiler, and the special gdb. Please read a file named "How to build and run sys161.html" available on Canvas to learn how to compile sys161.

You may follow the instructions below to build the tool chain, cross compiler, special gdb, and sys161. Before starting the following steps, you need to save the above five tarred and compressed file in `~/cs161`

4.2.1 Build the tool chain

```
%cd ~/cs161
%tar vfxz cs161-binutils-1.4.tar
%cd cs161-binutils-1.4
%./toolbuild.sh
```

4.2.2 Build the cross compiler

```
%cd ~/cs161
%tar vfxz cs161-gcc-1.5.tar
%cd cs161-gcc-1.5
%./toolbuild.sh
```

4.2.3 Build the special gdb

```
%cd ~/cs161
%tar vfxz cs161-gdb-1.5.tar
%cd cs161-gdb-1.5
%./toolbuild.sh
```

4.2.4 Build the sys161 emulator

```
%cd ~/cs161
%tar vfxz sys161-1.12.tar.gz
%cd sys161-1.12
%./configure mipseb
%make
%make install
```

Important! If you do not complete the above steps (see Sections 4.2.1-4.2.4), you will not be able to compile OS161.

4.3. Scripting Your Session (submit)

4.3.1 Script the following session using the `script` command. This may seem a nitpicky item to require as a handin; however, slight deviation from these instructions can produce baffling, time-wasting results - the script can help you catch subtle mistakes.

4.3.2 Make a directory in which you will do all your programming assignments. Throughout this assignment, it is assumed that the directory is called cs161. Moreover, please create a directory called assignment1 in which you are going to place the files that you will submit.

```
% mkdir ~/cs161
% mkdir ~/cs161/asst0
% cd cs161
```

Note that ~ is your home directory in CentOS.

4.3.3 Unpack the OS/161 distribution by typing:

```
% tar xvfz os161-1.10.tar.gz
```

4.3.4 This will create a directory named `os161-1.10`

Rename your OS/161 source tree to just `os161`.

```
% mv os161-1.10 os161
```

4.3.5 End your script session by typing exit or by pressing Ctrl-D. Rename your typescript file to be `setup.script`.

```
% mv typescript ~/cs161/asst0/setup.script
```

4.4 Setting up your CVS repository (submit)

Script the following session using the `script` command.

4.4.1 Create your CVS repository directory. Throughout the rest of this handout, we will assume that you created `~/cs161/cvsroot`.

4.4.2 **Important!** Set your CVSROOT environment variable. This will keep you from having to specify the -d argument every time you use CVS.

```
% export CVSROOT=~/cs161/cvsroot
```

4.4.3 Add this `export` line to your `.bashrc` file as well (but do not include this editing session in your script output).

4.4.4 Echo `$CVSROOT` and make sure that it is what you expect (`~/csc325/cvsroot`).

4.4.5 Initialize your repository by typing:

```
% cvs init
```

This tells CVS to create all the files it uses to track stuff in your repository. (If it complains about "not known `CVSROOT`", you probably didn't complete Item 3 above).

4.4.6 Change directories into the OS/161 distribution that you unpacked in the previous section and import your source tree.

```
% cd ~/cs161/os161
% cvs import -m "Import of os161" src os161 os161-1_10
```

You can alter the arguments as you like; here's a quick explanation.

`-m "Initial import of os161"` is the log message that CVS records. (If you don't specify it on the command line, it will start up a text editor). `src` is where CVS will put the files within your repository. It will also be the name that you specify when you check out your system. `Os161` is the "branch tag." You don't need to worry about the full implications of this; think of it as giving CVS a name to help you remember from where you got this code. `Os161-1_10` is the name of the version of the code that you are importing. (Always use the actual version, whatever it is. Replace dots with underscores.) You can use this name later with `cvs diff` and other CVS commands.

4.4.7 Now, you can remove the source tree that you just imported.

```
% cd ..
% rm -rf os161
```

Don't worry - now that you have imported the tree in your repository, there is a copy saved away. In the next step, you'll get a copy of the source tree that is yours to work on. You can safely remove the original tree. You **cannot ever** remove your CVS repository located in `$CVSROOT`.

4.4.8 Now, checkout a source tree in which you will work.

```
% cd ~/cs161
% cvs checkout src
```

4.4.9 End your script session. Rename your script output to `cvsinit.script`.

```
% mv typescript ~/cs161/asst0/cvsinit.script
```

5. Code Reading (submit)

It is challenging that you will be working with a large body of code written by someone else. It is of importance that you grasp the overall organization of the entire code base, understand where different pieces of functionality are implemented, and learn how to augment it in a natural and correct fashion. As you and your partner develop code, although you needn't understand every detail of your partner's implementation, you still need to understand its overall structure, how it fits into the greater whole, and how it works.

To become familiar with a code base, you are required to sit down and read the code. You can use the code reading questions included below to help guide you through reviewing the existing code. Although there is no need to review every line of code in the system in order to answer all the questions, you are advised to review every file in the system.

The goal of this exercise is to understand our base system. You should aim at understanding how it all fits together so that you can make intelligent design decisions when you approach future assignments. This task is seemingly tedious, but if you understand the structure of the system, you will have much less difficulty completing future assignments. Although it may not be apparent yet, you have much more time to study the system's overall structure now than you will at any other point in the term.

The file system, I/O, and network sections may seem confusing since we have not discussed how these components work. However, it is still useful to review the code now and get a high-level idea of what is happening in each subsystem. It will be fine if you do not understand the low-level details at this point.

The questions below (which appear in red text) are not meant to be tricky--most of the answers can be found in comments in the OS/161 source, though you may have to look elsewhere (such as Silberschatz et al.) for some background information. Place your the answers to the following questions in a file called `~/cs161/asst0/code-reading.txt`.

5.1. Top Level Directory

The top level directory of many software packages is called `src` or `source`. In UNIX, if the operating system source is installed, it is typically found in `/usr/src`. The top of the OS/161 source tree is also called `src`. In this directory, you will find the following files:

`Makefile`: top-level makefile; builds the OS/161 distribution, including all the provided utilities, but does not build the operating system kernel.

`configure`: this is an autoconf-like script. It sets up things like `How to run the compiler.' You needn't understand this file, although we'll ask you to specify certain pathnames and options when you build your own tree.

`defs.mk`: this file is generated when you run `./configure`. You needn't do anything to this file.

`defs.mk.sample`: this is a sample defs.mk file. Ideally, you won't be needing it either, but if configure fails, use the comments in this file to fix `defs.mk`.

and the following directories:

`bin`: this is where the source code lives for all the utilities that are typically found in /bin, e.g., `cat`, `cp`, `ls`, `etc`. The things in bin are considered "fundamental" utilities that the system needs to run.

`include`: these are the include files that you would typically find in `/usr/include` (in our case, a subset of them). These are user level include files; not kernel include files.

`kern`: here is where the kernel source code lives.

`lib`: library code lives here. We have only two libraries: libc, the C standard library, and hostcompat, which is for recompiling OS/161 programs for the host UNIX system. There is also a crt0 directory, which contains the startup code for user programs.

`man`: the OS/161 manual ("man pages") appear here. The man pages document (or specify) every program, every function in the C library, and every system call. The man pages are HTML and can be read with any browser.

`mk`: this directory contains pieces of makefile that are used for building the system. You don't need to worry about these, although in the long run we do recommend that anyone working on large software systems learn to use make effectively.

`sbin`: this is the source code for the utilities typically found in `/sbin` on a typical UNIX installation. In our case, there are some utilities that let you halt the machine, power it off and reboot it, among other things.

`testbin`: these are pieces of test code.

It is unnecessary to understand every line in every excutable in bin and sbin, but it is worth the time to peruse a couple to see how they work. Eventually, you will want to modify these and/or write your own utilities and these are good models. Similarly, you don't need to read and understand everything in lib and include, but you should know enough about what's there to be able to get around the source tree easily. The rest of this code walk-through is going to concern itself with the `kern` subtree.

5.2 The Kern Subdirectory

Once again, there is a Makefile. This Makefile installs header files but does not build anything. In addition, we have more subdirectories for each component of the kernel as well as some utility directories.

`kern/arch`

This is where architecture-specific code goes. By architecture-specific, we mean the code that differs depending on the hardware platform on which you're running. For our purposes, you need only concern yourself with the `mips` subdirectory.

`kern/arch/mips/conf`

`conf.arch`: This tells the kernel config script where to find the machine-specific, low-level functions it needs (see `kern/arch/mips/mips`).

`Makefile.mips`: Kernel Makefile; this is copied when you "`config a kernel`".

`kern/arch/mips/include`

These files are include files for the machine-specific constants and functions.

1. Which register number is used for the stack pointer (sp) in OS/161?

2. What bus/busses does OS/161 support?

3. What is the difference between `splhigh` and `spl0`?

4. Why do we use typedefs like u_int32_t instead of simply saying "int"?

`kern/arch/mips/mips`

These are the source files containing the machine-dependent code that the kernel needs to run. Most of this code is quite low-level.

5. What does splx return?

6. What is the highest interrupt level?

`kern/asst1`

This is the directory that contains the framework code that you will need to complete assignment 1. You can ignore it for now.

`kern/compile`

This is where you build kernels. In the compile directory, you will find one subdirectory for each kernel you want to build. In a real installation, these will often correspond to things like a debug build, a profiling build, etc. In our world, each build directory will correspond to a programming assignment, e.g., ASST1, ASST2, etc. These directories are created when you configure a kernel (described in the next section). This directory and build organization is typical of UNIX installations but is not universal across all operating systems.

`kern/conf`

config is the script that takes a config file, like ASST0, and creates the corresponding build directory. So, in order to build a kernel, you should:

```
% cd kern/conf
% ./config ASST0
% cd ../compile/ASST0
% make depend
% make
```

This will create the ASST0 build directory and then actually build a kernel in it. Note that you should specify the complete pathname ./config when you configure OS/161. If you omit the "./", you may end up running the configuration command for the system on which you are building OS/161, and that is almost guaranteed to produce rather strange results!

`kern/dev`

This is where all the low level device management code is stored. For the programming assignments, you can ignore most of this directory.

`kern/include`

These are the include files that the kernel needs. The kern subdirectory contains include files that are visible not only to the operating system itself, but also to user-level programs. (Think about why it's named "kern" and where the files end up when installed.)

7. How frequently are hardclock interrupts generated?

8. What functions comprise the standard interface to a VFS device?

9. How many characters are allowed in a volume name?

10. How many direct blocks does an SFS file have?

11. What is the standard interface to a file system (i.e., what functions must you implement to implement a new file system)?

12. What function puts a thread to sleep?

13. How large are OS/161 pids?

14. What operations can you do on a vnode?

15. What is the maximum path length in OS/161?

16. What is the system call number for a reboot?

17. Where is STDIN_FILENO defined?

`kern/lib`

These are library routines used throughout the kernel, e.g., managing sleep queues, run queues, kernel malloc, etc.

`kern/main`

This is where the kernel is initialized and where the kernel main function is implemented.

`kern/thread`

Threads are the fundamental abstraction on which the kernel is built.

18. Is it OK to initialize the thread system before the scheduler? Why or why not?

19. What is a zombie?

20. How large is the initial run queue?

`kern/userprog`

This is where code is added to create and manage user level processes. As it stands now, OS/161 runs only kernel threads; there is no support for user level code.

`kern/vm`

This directory is also fairly vacant. In Assignment 3, you'll implement virtual memory and most of your code will go in here.

`kern/fs`

The file system implementation has two subdirectories. We'll talk about each in turn.

`kern/fs/vfs`

This is the file-system independent layer (vfs stands for "Virtual File System"). It establishes a framework into which you can add new file systems easily. You will want to go look at vfs.h and vnode.h before looking at this directory.

21. What does a device name in OS/161 look like?

22. What does a raw device name in OS/161 look like?

23. What lock protects the vnode reference count?

24. What device types are currently supported?

`kern/fs/sfs`

This is the simple file system that OS/161 contains by default. You can ignore this directory for now.

6. Building a Kernel (submit)

Now for the fun part, it is time to build a kernel. As described in See `kern/conf`, you will need to configure a kernel and then build it.

Script the following steps using the `script` command.

6.1 Configure your tree for the machine on which you are working. We assume that you work in the directory `~/cs161`. Please note that if you intend to work in a directory that's not `~/cs161` (which you will be doing when you test your later submissions), you will have to use the `–ostree` option to specify a directory in which you are working. `./configure -help` explains the other options.

```
% cd ~/cs161/src
% ./configure
```

6.2 Configure a kernel named ASST0.

```
% cd ~/cs161/src/kern/conf
% ./config ASST0
```

6.3 Build the ASST0 kernel.

```
% cd ../compile/ASST0
% make depend
% make
```

6.4 Install the ASST0 kernel.

```
% make install
```

6.5 Now also build the user level utilties.

```
% cd ~/cs161/src
% make
```

6.7. End your script session. Rename your script output to `build.script`.

```
% mv typescript ~/cs161/asst0/build.script
```

7. Running your kernel (submit)

Download the file `sys161.conf` from Canvas and place it in your OS/161 root directory (`~/cs161/root`).
Script the following session.
7.1 Change into your root directory.

```
%cd ~/cs161/root
```

7.2 Run the machine simulator on your operating system.

```
%./sys161 kernel
```

7.3 At the prompt, type `p /sbin/poweroff <return>`. This tells the kernel to run the "`poweroff`" program that shuts the system down.

7.4 End your script session. Rename your script output to `run.script`.

```
%mv typescript ~/cs161/asst0/run.script
```

8. Practice modifying your kernel (submit)

8.1 Create a file called `~/cs161/src/kern/main/hello.c`.

8.2 Add the new file to your CVS repository:

```
%cvs add ~/csc325/src/kern/main/hello.c
```

8.3 In this file, write a function called hello() that uses kprintf() to print "Hello World\n".

8.4 Edit `kern/main/main.c` and add a call (in a suitable place) to hello().

8.5 Make your kernel build again. You will need to edit `kern/conf/conf.kern`, then reconfig, and then rebuild.

8.6 Make sure that your new kernel runs and displays the new message. Once your kernel builds, script a session demonstrating the config and build of your modified kernel (step 5 above). Call the output of this script session `newbuild.script`.

```
% mv typescript ~/cs161/asst0/newbuild.script
```

9. Using GDB (submit)

You will require two windows for the following portion.

9.1 Script the following gdb session (that is, you needn't script the session in the run window, only the session in the debug window). Be sure both your run window and your debug window are on the same machine.

9.2 Run the kernel in gdb by first running the kernel and then attaching to it from gdb.

```
#In the run window
  % cd ~/cs161/root
  % ./sys161 -w kernel

#In the debug window
  cd ~/cs161/root
  % cs161-gdb kernel
  (gdb) target remote unix:.sockets/gdb
  (gdb) break menu
  (gdb) c
  #gdb will stop at menu()

  (gdb) where

  #displays a nice back trace

  (gdb) detach
  (gdb) quit
```

9.3 End your script session. Rename your script output to gdb.script.

```
 % mv typescript ~/cs161/asst0/gdb.script
```

10. Practice with CVS (submit)

In order to build your kernel above, you already checked out a source tree. Now we'll demonstrate some of the most common features of CVS. Create a script of the following session (the script should contain everything except the editing sessions; perform those in a different window so they don't appear in the script). Call this file `cvs-use.script`.

10.1 Edit the file `kern/main/main.c`. Add a comment with your name in it.

10.2 Execute

```
% cvs diff -c kern/main/main.c
```

to display the differences in your version of this file.

10.3 Now commit your changes using `cvs commit`.

10.4 Remove the first 100 lines of `main.c`.

10.5 Try to build your kernel (this ought to fail).

10.6 Realize the error of your ways and get back a good copy of the file.

```
% rm main.c
% cvs update -d main.c
```

10.7 Try to build your tree again (see Section 6).

10.8 Now, examine the DEBUG macro in `src/kern/include/lib.h`. Based on your earlier reading of the operating system, add ten useful debugging messages to the source code of your operating system os161.

A sample source code is given below:

```
DEBUG(DB_VM, "VM free pages: %u\n", free_pages);
```

In the above code, `DB_VM` is a flag defined in `src/kern/include/lib.h`

If the debug DB_VM flag is set, the debug message will be printed on the console.

10.9 Now, display the locations where you inserted these DEBUG statements by doing a diff.

```
% cd ~/cs161/src
% cvs diff -c
```

10.10 **Important!** Finally, you should create a release.

```
% cd ~/cs161
% cvs commit
% cvs tag asst0-end src
% cd ~/cs161/asst0
% cvs export -rasst0-end src
% tar vfcz <group_ID>_asst0.tgz .
```

Note that <group_ID> is the ID number of your group. Do not forget to put a dot (i.e., .) at the end of the above command. If you ignore the dot, you will fail in creating a tarball.

11. Deliverables

Important! Your `asst0` directory, which you tarred and compressed above, should contain everything you need to submit, specifically:

```
1) setup.script
2) cvsinit.script
3) code-reading.txt
4) build.script
5) run.script
6) newbuild.script
7) gdb.script
8) cvs-use.script
```

Now, submit your tarred and compressed file named <group_ID>_asst0.tgz through Canvas. You must submit your single compressed file through Canvas (no e-mail submission is accepted. For example, suppose that I am a member of group6, my submitted file would read "group6_asst0.tgz".

12. Grading Criteria

1) Programing environment setup (i.e., setup.script): 10%
2) Setup CVS (i.e., cvsinit.script): 10%
3) Reading Code (i.e., code-reading.txt): 30%
4) Build os161 (i.e., build.script): 10%
5) Run os161 (i.e., run.script): 10%
6) Rebuild os161 (i.e., newbuild.script): 10%
7) Use GDB (i.e., gdb.script): 10%
8) Use CVS (i.e., cvs-use.script): 10%

13. Late Submission Penalty

- Ten percent (10%) penalty per day for late submission. For example, an assignment submitted after the deadline but up to 1 day (24 hours) late can achieve a maximum of 90% of points allocated for the assignment. An assignment submitted after the deadline but up to 2 days (48 hours) late can achieve a maximum of 80% of points allocated for the assignment.
- Assignment submitted more than 3 days (72 hours) after the deadline will not be graded.

14. Rebuttal period

- You will be given a period of one week to read and respond to the comments and grades of your homework or project assignment. The TA may use this opportunity to address any concern and question you have. The TA also may ask for additional information from you regarding your homework or project.