# csci 3411: Operating Systems

## Final Project

## Requirements

If any of these requirements create a problem for you or are unclear, come talk to me.

- You should do this assignment in a small group of 3. You can get permission from me to have a group of 2 or 1, but please plan on 3. You should not discuss the code with other students who are not in your group. You may not use code downloaded from the Internet without specific permission from me.
- If you discover any ambiguities in this assignment, send email to me and I will clarify.
- Always watch your email for updates on the assignments.

## Goals

The goal of this assignment is to gain a better understanding of Linux and kernel programming, and the process of making a substantial and nontrivial change to a large code-base.

## Deadlines

1. Group Selection, Wednesday, Oct 26th, 11:59pm.
2. Project Selection, Thursday, Oct 27th, 11:59pm.
3. Initial Investigation and Design, Thursday, Nov 3rd, 11:59pm.
4. Progress Demo, Thursday, Nov 10th, 6:00pm.
5. Progress Report, Wednesday, Nov 23rd, 11:59pm.
6. Final Report, Friday, Dec 9th, 11:59pm.

## Grading

You will receive *individual* grades, even though you will work in groups. Your grade will be a composite of 1) the group grade on your project, and 2) an individual grade based on your and your team members group feedback. If your group has an amazing product, but your team members consistent say that you didn't contribute, your grade will suffer. Your group grade is broken down as follows: 13% each for Initial Investigation and Design, Progress Demo, and Progress Report. 61% for the Final Project.

## Assignment

Your group can choose one of the following project ideas, or create your own. If you choose to create your own, please contact the professor to decide if it is appropriate.

All projects should be viewed as multi-step projects, and you should program them that way. So at each stage you should have working copies of the first few steps or simplified

solutions, and can demo them and submit them – even while you may be working on more advanced features. You will not complete the project if you don't use this approach. Each project suggests several 'feature' steps, although you can make your own decisions about exactly where you want to break it up. You want to have multiple, incremental, solutions. You will get significant credit for partially completing the projects, but you must be able to demonstrate functional code for the subparts you did complete. Code that doesn't compile, and hasn't been tested for those parts will not result in credit.

I again recommend buying the Robert Love book, "Linux kernel development" if you don't feel comfortable with the Linux code base. Additionally, the documentation at the http://lwn.net/Kernel/ and http://lwn.net/Kernel/Index/ website should be very useful. Another good site is the Kernelnewbies site at http://kernelnewbies.org/ and the always useful LXR source index at http://lxr.linux.no/source/ .

## Project 1: Structured File Filesystem

File systems do not have to be generic storage systems that store and view data as binary blobs. A file system can be specialized to know about the structure and types of data that is stored in it. This can allow the file system to present a useful view of structured data and let users run standard tools like grep, ls, echo, and editors to view and modify files.

In this project you will select a file format that has a well defined structure. For example:

- scheme programs, as they are syntactically structured as a tree
- unix password or shadow files (or other software configuration files such as for webservers, email servers, etc)
- xml (maybe a specific form of xml that is restricted in some ways).
- latex (restricted somehow)
- …

You will need to create a new filesystem, for which the backing store (where the data is stored) is not a raw disk partition or block device, but rather a file of the selected format.

Your file system should:

1. Be able to be mounted and unmounted, where the file that should be the backing store is specified at mount time, or is somehow written to the kernel (i.e. via a character device with a "write" function defined).

2. Create a directory hierarchy based on the file format structure of directories representing the different elements of the file. For example, in the password case, you might make a directory for each user (i.e. each line of the password file) and then subdirectories for each field in the user record. In the latex case, you might make a directory for each top-level section command, and subdirectories for paragraphs.

3. Each piece of data in the file should be visible either as contents of a file in the file-system, or as metadata attached to a file or directory. For example if an XML

entity has an attribute of creation time that could be shown as the creation time on the file storing that entity. The names of the files in the directory tree should be appropriate to the specific format of the file.

4. The contents of any files can be read through standard system calls such as read, open, readdir. The first version of your file system should be 'read-only' meaning that the backing store file cannot be modified by writing or editing the files in your file system. You can return an ENOTSUPP error for any 'write' calls or operations that are requested by a userspace program. It is as if you mounted your filesystem readonly.

However, once that is working you should then make a 'version 2' of your file system that adds support for writes. This can be divided into two separate feature sets:

1. Only files support write operations. So the content of a record that is viewable as a file can be edited, but the directory structure and metadata can not be edited. So I can change the value of a particular users' password or change which shell they selected, but I can not create new users (as that would require making a new directory).

2. Directory and metadata operations are write-enabled. Now the directories can be deleted, added, renamed and metadata can be changed.

You can use the libfs file system library or the FUSE "user-space" filesystem library to help implement your file system.

### Evaluation

To test your filesystem you should create several different data files that are valid examples of your structured file format. You can use existing files but make sure the data in them is public since others will see it.

You should then write several shell scripts that test your file system by using standard unix commandline programs such as 'ls', grep', and cat for the read-only portion, and 'echo' and some editor to test the modifiecation of your file. You can also run some more complex tests by hand using standard editors such as emacs or vi, for these results capture a screenshot or a text log of the shell to include in your report to show the tests you did. Your tests should verify that all of the required features of the file system work and that in common error cases, reasonable error return values are generated.

## Project 2: Filesystem Attributes

One of the newer features of some file systems is file attributes. Read the explanation of how Mac OS X is using them at http://arstechnica.com/reviews/os/macosx-10.4.ars/7.

Think about how you would support file attributes in each of the three software levels:

1. Application: So a specific application stores attributes (like Word).

2. Desktop: A desktop environment like Gnome/KDE/Win32 stores attributes that are visible only through the desktop (not commandline).

3. Library: In a common library (like the C library) that many applications and other libraries use.

4. Kernel: Inside the OS and file system.

## User Defined Attributes

You will create a mechanism to allow users to attach specific attributes to files and directories. This will consist of several new system calls, modification to some existing file system code, and the creation of new user-space programs to display the attributes.

An attribute is attached to a specific file or directory and consists of a AttrName and a AttrValue.

All attributes will be stored in a special “attribute” directory that is created for each file or directory that has attributes attached. This attribute directory will have a name created by pre-pending the file name with a dot “.” and appending the file name with the string “_attr”. So a file testfile will have an attributed directory called .testfile_attr. Creating attributes to files that already start with a “.” is not allowed. All other files or directories can have attributes attached to them.

An attribute will be stored in the attribute directory as a file whose name is the AttrName and whose contents is the AttrValue. So an attributed named “Creator” whose value was “lolcat” attached to file “testfile” would be stored in .testfile_attr/Creator with the Creator file contents being “lolcat”.

The file permissions for the attribute directory and the attribute files should match the permissions of the file they are attached to.

## Specification

This specifies exactly what features you need to add and how they should work.

## System Calls

For all system calls, appropriate error values should be returned for situations like ‘removing an attribute that doesn’t exist’, ‘adding an attribute with an invalid name’, lack of resources, etc. What follows is a suggested (though not mandatory) system call API.

```
long cs3411_set_attribute(char *filename, char *attrname,
                          char *attrvalue, int size)
```

This call sets the attribute named “attrname” to the value in “attrvalue” which has length “size” to the file named “filename”. The filename must be an absolute path to the file. The strings should be null terminated, and reasonable limits should be enforced on all strings. The return value should be 0 for success, or a negative error value.

```
long cs3411_get_attribute(char *filename, char *attrname,
                          char *buf, int bufsize)
```
This call gets the value of the attribute named “attrname” attached to the file named “filename”. The value is stored in the buf pointer which must have at least “bufsize” bytes of storage. The filename must be an absolute path to the file. The strings should be null terminated, and reasonable limits should be enforced on all strings. The return value should be the number of bytes returned for success, or a negative error value.

```
long cs3411_get_attribute_names(char *filename, char *buf,
                                int bufsize)
```
This call gets all of the names of attributes that are set for the file “filename”. The list of attributes is returned as a “:” (colon) separated list in the “buf” string. The buf pointer must have at least “bufsize” bytes of storage. The filename must be an absolute path to the file. The strings should be null terminated, and reasonable limits should be enforced on all strings. The return value should be the number of bytes returned for success, or a negative error value. This system call can be used with the cs3411_get_attribute() call to list ‘all’ of the attributes attached to a specific file.

```
long cs3411_remove_attribute(char *filename, char *attrname)
```
This call removes the specified attribute from the file. It should remove the file that was created in the attribute directory. If this was the last user attribute for this file, the attribute directory should also be removed. Note, this will require some locks as some other process may be adding a new attribute at the same time, and should not fail because the directory was removed from under it. The return value should be 0 for success, or a negative error value. This system call can be used with the cs3411_get_attribute_names() call to remove ‘all’ of the attributes attached to a specific file.

## User space test programs

You should create two user-space programs that cat set attribute values and retrieve them. These programs must make use of the system calls specified above and can not directly access the attribute directory.

**setattr** The setattr program should take as input a single name=value pair and a list of files to apply the attributes to. For example (using shell wildcards for *, your program does not need to parse for wildcards)

```
> setattr "Professor=Dr. Jekyll" *.c myfile.txt
> setattr Type=worddoc *.doc
```

In the first case all .c files and the myfile.txt file should have the attribute of Professor added with value “Dr. Jekyll”. In the second case the Type attribute should be set to “worddoc” for all files in the directory with file extension .doc.

**listattr** This program should take as input an attribute name and a list of files, and should output the values and names of the attributes requested for those files. For example:

```
> listattr Owner *.c yourfile.asd
file.c Owner=lolcat
second.c Owner=Fred Flintstone
```

```
yourfile.asd Owner=Jones
> listattr Type *
file.c Type=Cfile
second.c Type=Cfile
myfile.txt Type=Text
test.doc Type=worddoc
```

The first should list the Owner name/value pair for all c files and the file “yourfile.asd”. The second should list the “Type” attribute for all files in the local directory. If a file does not have the attribute requested it should not be listed.

A special attribute name is “LISTALL” which will cause the listaddr program to list all of the attributes attached to the specified files.

### Kernel Support for Attributes

These attributes you have now created should be better integrated with the regular kernel file functions. For example if you remove a file, all of the attributes should be removed with it. If a file is renamed, the attributes should move with it.

Look at the kernel implementation of ‘unlink’ and ‘rename’ (see sys_unlink and sys_rename, respectively) and add support in them to use your system calls to correctly remove the attributes for unlink, or to move the attributes to the new file name for rename.

You will need to verify that your changes work on the ext3 file systems, but you do not have to have them work with any other specific file systems. However, many others will probably work if you implement your changes in the generic unlink and rename code. For example look at the system calls sys_rename and sys_unlink and see how they work. The lxr source browser will be very useful for this stage of the project to figure out how these work and where to place your changes. You cannot call system call implementations directly in the kernel as they expect user-level arguments. One way around this is to use get_fs and set_fs. This solution breaks important kernel functionality and should not be used.

## Project 3: Freezing the Filesystem

You will implement kernel changes to ensure that modifications to the filesystem are not persistent between reboots. Similar to a snapshot system, except that it ensures that the machine always boots into a known configuration, rather than allowing for the return to an arbitrary point in the state of the system.

At any time, the system can either be “thawed” or “frozen”. If a machine is thawed, then any changes made to the filesystem will remain. If the machine is put into a frozen state, then all changes to the filesystem will be reset when that part of your program is run.

In this project, you most likely will not have time to implement a full freezing of the filesystem. Instead, you will need to identify a set of changes that you can log and roll-back.

## Kernel Support

You will need to hook in to system calls that modify the filesystem in order to log changes that are made. The strace command will be very helpful in identifying the appropriate system calls. Syscalls receive file pointers that need to be translated to paths via path_lookup and dpath to determine precisely which file is being processed.

The log entries will need to be pushed to a userspace application that can write the log to persistent storage. Additional system calls may be helpful to tell the kernel how to find the userspace application.

## Userspace Helper

You will need to maintain a buffer between a userspace application and the kernel modifications. Your userspace application will commit log entries to a file. When activated, your userspace application (same or different) will parse the log file and roll back any modifications to the filesystem.

Proceed through this project in phases:

1. Logging from Kernel space to user space
   First establish hooks in system calls that access/modify the filesystem, and set up a logging infrastructure that you can test. Figure out which system calls you will monitor, ultimately freezing (by logging) all changes would be ideal, but you will need to limit the scope of your project.

2. Tracking changes
   Add the capability of your logging to detect changes in the filesystem. You may find it is easiest to identify a file as “dirty” if a write is performed. Tracking a minimal set of changes may prove difficult. Also, be careful that actions performed by your userspace application are not themselves logged, as this could cause an infinite loop in the kernel.

3. Backing up files and directories
   Using the tracking capability, you will store clean copies of files and directory structures (with the userspace app) so that you can undo the changes.

4. Automatic restore on command
   Extend your userspace app to roll back changes that were logged.

## Evaluation

For determining correctness, generate some simple test cases and observe that your logs reflect your test cases. Your test cases will probably consist of scripts that make deterministic accesses/modifications to the filesystem.

# Project 4: Earliest Deadline First Scheduler

This project focuses on the design, implementation, and evaluation of an Earliest Deadline First (EDF) scheduler in Linux. Linux has three main scheduling classes: SCHED_FIFO, SCHED_RR, and SCHED_OTHER. These can be used as arguments to `sched_setscheduler`. These schedulers implement first in, first out, fixed priority round robin, and a generic timesharing algorithm, respectively. This project involves adding a new scheduler class, SCHED_EDF.

## Adding Support for Deadlines

Not all threads in the system will use EDF. You will need to provide support to set specific threads to execute under the EDF scheduling class. This capability can be provided by either adding your own system calls, or by modifying the Linux system calls for modifying a thread's scheduler, `sched_setscheduler` and `sched_getscheduler.`

Additionally, Linux has no abstraction for deadlines. This will need to be added via system calls for setting the deadline (T) for a process. This deadline is assumed to be periodic (i.e. every T time units, a new deadline is created T time units into the future). Setting this deadline for a thread can be accomplished by adding your own new system call, or by modifying the `sched_setparam/sched_getparam` system calls. You can assume that deadlines are specified in "jiffies", or multiples of the timer tick. You will also need to add support for specifying the amount of processing time that task is allowed to carry out within each period, using either a new, or existing scheduler system call.

## Adding a New Scheduler

The Linux O(1) scheduler uses an array of 140 different "priority levels" to track the queues of threads at each of the priorities. You can use the highest priority entry in the array for EDF. You must add support so that threads in this runqueue will be scheduled according to EDF: the thread with the earliest deadline has the highest priority. Additionally, you will need to add support to 1) track task deadlines and reset them to the next period when one occurs, 2) put threads to sleep when they have expended their given processing time (and, comparably wake them up when their deadline finishes).

## Evaluation

You will need to evaluate your new scheduling policy with a set of threads to ensure that your policy is working correctly. When a set of threads are configured with different deadlines and processing times, their share of the processor can be observed with the "top" command to ensure they are given proper CPU allocations.

# Project 5: User-level Threading Library

This is the only project that will be completed entirely at user-level. However, this is still one of the more challenging projects. Please read through this write-up and analyze it before deciding on this project. For this project you will implement not only thread manipulation functions to create and join threads, but also the synchronization primitives that can be used for coordination between them. This task will be made easier by following the technique proposed in the paper "Portable Multithreading - The Signal Stack Trick For User-Space Thread Creation". This paper can be retrieved at http://www.gnu.org/software/pth/rse-pmt.ps or by searching for the paper's title.

## User-level Thread Library Interface

Specifically, you will implement the following functions:

1. `gwthd_t gwthd_create(gwthd_init_fn_t f, void *data)` – where `gwthd_init_fn_t` is simply a function that will be called when the thread is created of type `void *f(void *d)`. The new thread is created, and it begins executing in `f` with the argument `data`. This function returns the identifier (of type `gwthd_t`) of the newly created thread to the thread that created it.
2. `void gwthd_kill(void *data)` – This function will terminate the *current thread* with a return value (that another thread can retrieve via `gwthd_join`) of `data`.
3. `int gwthd_join(gwthd_t id, void **data)` – The current thread will wait for the thread identified by `id` to return from its `gwthd_init_fn_t`, or kill itself. Specifically, this thread will block until thread `id` terminates in one way or the other, then it will wake. The return value of that thread will get placed in the `data` argument. This function will return `0` if thread `id` exists, and the join is successful (thus `data` is set to the value returned from the other thread), and `-1` otherwise.
4. `void gwthd_yield(gwthd_t id)` – This call (and the mutexes below) enables cooperative multi-threading. If `id == GWTHD_NIL`, then the calling thread is placed at the end of the runqueue. If `id` is the id of an actual thread, then this is a "directed yield" in which the current thread switches to the thread `id` directly *if* thread `id` is in the runqueue. This enables a thread to *donate* its processing time to another specific thread.
5. `struct gwthd_mux *gwthd_mutex_crt(void)` – Create a mutex! Return `NULL` if there is an error (i.e. cannot malloc the mutex).
6. `int gwthd_mutex_del(struct gwthd_mux *m)` – Delete a mutex! Return `0` on success, and `-1` on failure. Do *not* delete a mutex with threads blocked on it (i.e. return `-1`).
7. `int gwthd_mutex_take(struct gwthd_mux *m)` – Take a mutex. The semantics of this call should match what is expected from a mutex. Return `0`

on success. Return `-1` if this mutex is already taken by this thread, and do *not* block this thread.

8. `int gwthd_mutex_release(struct gwthd_mux *m)` – Release a mutex. This will wake any threads blocked on the mutex. Return `0` on success, and `-1` if the current thread does *not* actually hold the mutex.

Behind this abstraction layer, you will have to implement the user-level thread abstraction complete with run-states (blocked on mutex, running, runnable, etc...), a scheduler (I suggest non-preemptive FIFO), and the required mutex machinery including queues of blocked processes, and the necessary atomic instruction usage.

### Evaluation

To evaluate this project, you will implement two simple applications.

1. The Fibonacci number function, where each recursive call is implemented with a different thread, and the return value from that "call" is retrieved via `gwthd_join`. This will test `gwthd_create` and `gwthd_join`. You should test a version that uses `gwthd_kill` instead of just returning from each thread's initial function.
2. The producer-consumer problem where multiple threads generate data into a buffer, and multiple threads consume data from the buffer. After consuming one data-item, the consumer will call `gwthd_yield`, and after producing one data-item, the producer will also call `gwthd_yield`. Every `N` times the mutex is taken, the current thread will also call yield. This will test your mutex implementation.

### Taking it further

Preemptive scheduling can be provided by using signals to deliver a "timer-tick" to your process. Talk to me about this extension if interested.

Additionally, "wrappers" around blocking system calls (such as read and write) can be made that will ensure that those calls are non-blocking (a normal UNIX functionality), and allowing you to switch to another user-level thread if the thread would otherwise block.

## Project 6: Full-Featured Web-Server

This project will entail significantly expanding your previous web-server projects by adding more functionality including 1) the ability to create, delete, update, and get data on the server using the HTTP commands POST, DELETE, and GET, 2) a thread-safe hierarchical data-store using the file-system as a backing store, 3) a thread-safe malloc and free implementation for memory allocation with specialized "bin allocation sizes" for common objects in your server. This project can safely be considered the one that will require the most code. Your server must use the "thread-pool" paradigm and a shared

data-structure (e.g. ring-buffer) for passing connections between the master and the worker threads.

## Hierarchical Data-Store

You will implement a hierarchical tree data-structure much like a ram-based file-system that will store data that is accessed hierarchically (i.e. using a path through a tree like /home/gparmer/csci3411/final_project_11.pdf). The data in the tree will actually be in your system's file system at a given root (i.e. /home/gwstudent/ws_root/*). When the data is accessed by clients, it is read from the file system and cached in your hierarchical data-structure. You will maintain up to a fixed amount of data in this data-structure (say 10 MB), and after you reach that, you will use an LRU policy to write data from your data-structure out to disk. You will use fine-grained locking (one lock per “file system object”) to protect your data-structure from concurrent access.

## Enhanced HTTP Support

Your server will enable clients access data in your hierarhical RAM file-system. It will allow them to create file-system objects (HTTP POST) at specific paths (which might create that entire path in your data-structure), delete file-system objects (and subtrees) using HTTP DELETE, and access data in your file-system using HTTP GET.

## Specialized Dynamic Memory Management

You will implement your own malloc and free functionality as functions wsmalloc and wsfree. These functions will maintain separate freelists for allocations of common sizes (e.g. sizeof(struct request)). Malloc requests for memory of these sizes will be allocated from separate freelists associated with the different sizes. Requests for any other size allocation will be passed onto the default malloc implementation. When memory is freed, you must determine if it is allocated by your allocator, in which case it should be placed on the appropriate freelist, or allocated by the default malloc, in which case it should be freed by the default free. The easiest way to do this is to maintain a “header” for each allocation that will store the allocation's size. The header can be found when free is called.

You must implement this in a thread-safe manner. You should use mutexes, one per allocation size your allocator is specialized for. Thus separate threads can concurrently make allocations to different size allocations, but the freelists are kept consistent for each individual size.

Your server will only call wsmalloc and wsfree.

## Evaluation

To evaluate this project, you will use httperf, wget, and any other client tools for generating traffic. You must generate a diverse set of GET, DELETE, and POST

behaviors. If you are unclear if you are extensively enough evaluating your web-server, then please contact me.

## Project 7: Additional Functionality in the Composite Component-based OS

We are developing a micro-kernel-like OS here at GW that is still in the early phases of development. The system is structured around the idea of a "component". A component is a piece of functionality that can be accessed through an interface; you can think of it much like an object. Example components define the scheduler, memory manager, and mutexes. This system is limited current in that it doesn't have a file system, nor a number of interesting schedulers (including EDF).

This project is more difficult than the others, mainly because you have already been introduced to Linux, and have some notion of how to work with it. You would have to learn quite a bit about this system before starting to work. Thus, if you are interested in this project, please contact me very soon to discuss it.

## Deliverables and Deadlines:

For all emails, make sure you follow the naming conventions for the email titles, and please always check if you are to submit an attachment, or put your detail in the body of the email. If you do not, you will not get credit. You should only send .txt and .pdf documents. Any .doc or .docx documents will result in no credit.

## Group Choice (Due Wednesday, Oct. 26th 11:59pm)

You must choose your group by this deadline. You must have 3 group members (or less with permission), all willing to work. If you cannot find a team, please email me as soon as possible, and I will assign one for you. Please send me and the TA an email (one per group) with the title "csci3411-11:GC", and with the body of the email letting us know the composition of your team. Do not send an attachment.

## Project Choice (Due Thursday Oct 27th, 11:59pm)

Your group must choose its project by this deadline. Please send an email, one per group to both the TA, and Professor with the title "csci3411-11: PC". The body of the email should include the project your group will undertake. Do not send an attachment. If your project requires embellishment (either because it is the Composite project, or you are defining your own), then we must talk before the deadline.

## Initial Investigation and Design (IID) (Due Thursday, Nov 3rd, 11:59pm)

Please send a document detailing 1) any initial investigations you've done into the details of the project, and implementation directions you've taken, 2) a break down of the project into separate pieces with each piece being a stepping-stone to the final implementation, with an associated timeline, and 3) the breakdown of the work between different team members. Please send this in an email (one per group to both the professor and the TA), with the title "csci3411-11:IID". The body of the email should be the investigations, break down, and timeline. Please include in this email the names of all group members. This email should not include an attachment.

For your timeline, and break-down of work, please include three deadlines with the work you will have done by that deadline:

1) Progress demo (see below),
2) Progress report (see below), and
3) Final report and demo (see below).

Additionally, each member of the group should independently email your professor and report what the break down of the work has been between members so far and report on any current imbalances or anticipated future imbalances. This email should have the title "csci3411-11:TF1" (team-feedback one). If you do not send this email, then it will be assumed that you have no grievances with your team, and that you don't anticipate they have any with you.

## Progress Demo (Due Thursday, Nov 10th, lab time)

You do not need to send an email this week. Instead, during Lab, you will give a short demo of the functionality you have so far, or of the investigations you've undertaken. The purpose of the demo is to convince us that you have been making progress on your project, that the breakdown of work between members is fair, and that you are on track to complete you project (given the timeline you set out with).

Specifically, come prepared to demonstrate the work you said you would do in the IID, or justify why it is not completed, and how you will amend the schedule to catch up. Please email your professor and TA a more detailed and updated IID (one per group with members listed at the top of the email). At this point, you should understand the problem better, thus should be able to include more detail in future deadlines.

As before, each member of the group should independently email your professor and report what the break down of the work has been between members so far and report on any current imbalances or anticipated future imbalances. This email should have the title "csci3411-11:TF2".

## Progress Report (Due Wed, Nov 23rd, midnight)

You should report your progress so far. Specifically, elaborate on the deliverables you promised in IID. Justify why any portion of these deliverables are not completed, and include a plan on how to catch up. Include a break-down of how the pending work will be divided between group members. Please send an email titled “csci3411:PR” to both the professor and the TA. Please send this in the body of the email, *not* as an attachment. Be sure to include in this document, the names of all group members. Also include an amended IID to include significantly more detail since you now know your project better.

As before, each member should independently email your professor and report what the breakdown of work has been between members and any effort/work imbalances. Please title this email “csci3411-11:TF3”.

## Final Report (Due Friday, Dec 10th, 12:59pm)

Your group's final document should present the implementation you have completed up to this point, including your design decisions. Report any difficulties and challenges your group hit in the implementation. If you did not reach your goals, explain where the road-blocks were and what you did to try and get around them. This document should be either a .txt or .pdf document. A .doc or .docx document will not be graded. Be sure to include in this document, the names of all group members. Please create a .tgz (called 3411-11_final_project.tgz) that includes 1) your source code, 2) any associated data or scripts you used to setup and/or test your implementation, and 3) the final document explaining your project. Additionally, include a README text file in the root directory of your .tgz including directions on how to build and use your code, how to use any scripts you used to test, and generally how to test your project. Include a MEMBERS text file including the names of each of your group members. You can create the .tgz using the following command:

```
tar -zcvf 3411_final_project.tgz 3411_final_project/
```

Upload this .tgz to blackboard.

You will give a demonstration of your code around the due date, which you will have to schedule with the professor.