Programming Assignment 4 (PA4) - myls

Milestone Due: **Wednesday, May 25 @ 11:59pm**
Final Due: **Wednesday, June 1 @ 11:59 pm**

Assignment Overview

The purpose of this (last!) programming assignment is to further emphasize C and implement a real Unix command (along with using more Standard C Library routines). You will build a program called myls that will function very similarly to the real Unix command **ls**. This program will list information about the files (or directories) entered as a command line argument. If files/directories are not specified, ls prints information about the current directory by default.

Some of the system calls and Standard C Library routines you will use include `getopt_long()`, `lstat()`, `opendir()`, `readdir()`, `closedir()`, `malloc()`, `realloc()`, `free()`, `qsort()`, `getpwuid()`, `getgrgid()`, `strcmp()`, and possibly `ctime()`. We will try not to do too much handholding on this PA. You have some experience now and you need to become confident looking up information on your own. You will not be given nearly as much help/handholding in some of the upper division courses, so now is the time to start doing more on your own.

To specify which section of the man pages to search in: `man -s# <function_name>`
where # is replaced with the section number.

To determine which section of the man pages to search in: `man -l <function_name>`

Grading

- **README: 5 points** - See README File section.
- **Compiling: 5 points** - Using our Makefile; no warnings. If what you turn in does not compile with the given Makefile, you will receive 0 points for this assignment. **NO EXCEPTIONS!**
- **Style: 10 points** - See Style Requirements section.
- **Correctness: 80 points**
  - **Milestone (15 points)** - To be distributed across the Milestone functions (see below).
  - Make sure you have all files tracked in Git.
- **Extra Credit: 5 points** - View Extra Credit section for more information.
- **Wrong Language:** You will lose 10 points for each module in the wrong language, C vs. Assembly or vice versa.

NOTE: If what you turn in does not compile with given Makefile, you will receive 0 points for this assignment.

Getting Started

Follow these steps to acquire the starter files and prepare your Git repository.

Gathering Starter Files:
The first step is to gather all the appropriate files for this assignment.
Connect to ieng9 via ssh (replace cs30xzzz with YOUR cs30 account).

```
$ ssh cs30xzzz@ieng9.ucsd.edu
```

Create and enter the pa4 working directory.

```
$ mkdir ~/pa4
$ cd ~/pa4
```

Copy the starter files from the public directory.

```
$ cp -r ~/../public/pa4StarterFiles/* ~/pa4/
```

Starter files provided:

pa4.h	pa4Strings.h	pa4Globals.c
test.h	testnameCompare.c	Makefile

Preparing Git Repository:

Refer to previous writeups for preparing your Git repository. This will be required again for PA4.

Sample Output

A sample stripped executable provided for you to try and compare your output against is available in the public directory. Note that you cannot copy it to your own directory; you can only run it using the following command (where you will also pass in the command line arguments):

```
$ ~/../public/pa4ref
```

If there is a discrepancy between the sample output in this document and the `pa4ref` output, follow the `pa4ref` output.

Below are some brief example outputs of this program. Make sure you experiment with the public executable to further understand the program behavior. Bolded text is what you type in the terminal.

1. Command-line Parsing Errors

1.1. Invalid flag.

```
[cs30xzzz@ieng9]:pa4$ ./myls -g
./myls: illegal option -- g
Usage: ./myls [-ahlrtBF] [file ...]
```

2. Other Errors

2.1. File doesn't exist.

```
[cs30xzzz@ieng9]:pa4$ ./myls foobar
lstat --foobar: No such file or directory
```

2.2. Opening directory with no permissions.

```
[cs30xzzz@ieng9]:pa4$ ./myls noPermissionsDir
noPermissionsDir:
opendir --noPermissionsDir: Permission denied
```

2.3. Run out of memory when attempting to store a file in the table

```
[cs30xzzz@ieng9]:pa4$ ulimit -d 1
[cs30xzzz@ieng9]:pa4$ ./myls pa4.c
Attempted to allocate memory in buildFileInfoTable(): Not enough space
```

2.4. Run out of memory after storing some entries in the table

```
[cs30xzzz@ieng9]:pa4$ ulimit -d 8
[cs30xzzz@ieng9]:pa4$ ./myls
.:
Attempted to allocate memory in buildFileInfoTable(): Not enough space
Only partial info will be displayed.
Makefile
buildFileInfoTable.c
modTimeCompare.c
pa4.c
pa4.h
pa4Globals.c
pa4Strings.h
printGroupName.s
```

3. Valid Output

3.1. No command-line options.

```
[cs30xzzz@ieng9]:pa4$ ./myls
.:
Makefile
buildFileInfoTable.c
buildFileInfoTable.ln
buildFileInfoTable.o
modTimeCompare.c
modTimeCompare.ln
modTimeCompare.o
modTimeCompareRev.o
modTimeCompareRev.s
myls
nameCompare.o
nameCompare.s
nameCompareRev.c
nameCompareRev.ln
nameCompareRev.o
pa4.c
pa4.h
pa4.ln
pa4.o
pa4Globals.c
pa4Globals.ln
pa4Globals.o
pa4Strings.h
parseArgs.c
parseArgs.ln
parseArgs.o
printFileInfoTable.c
printFileInfoTable.ln
printFileInfoTable.o
printGroupName.o
printGroupName.s
```

```
printOwnerName.c
printOwnerName.ln
printOwnerName.o
printPermissions.c
printPermissions.ln
printPermissions.o
```

3.2. Mix of short and long command-line options (including the help flag).

```
[cs30xzzz@ieng9]:pa4$ ./myls -la --classify -B --help -r
Usage: ./myls [-ahlrtBF] [file ...]
List information about the files (the current directory by default).
Sort entries alphabetically if -t is not specified.

 -a, --all             do not ignore entries starting with .
 -l,                   use a long listing format
 -r, --reverse         reverse order while sorting
 -t,                   sort by modification time, newest first
 -B, --ignore-backups  do not list implied entries ending with ~
 -F, --classify        append indicator (one of */=>@|) to entries

 -h, --help            displays this help and exit
```

3.3. Mix of short and long command-line options (not including the help flag).

```
[cs30xzzz@ieng9]:pa4$ ./myls --all -r sampleDir
sampleDir:
test.link.txt
subDir
regFile.c
myls
..
.
```

3.4. Command-line options "`-F`" and "`-l`" using each kind of classifier in the long listing format.

```
[cs30xzzz@ieng9]:pa4$ ./myls -lF ~/../public/pa4SampleFiles
/home/solaris/ieng9/cs30x/cs30xzzz/../public/pa4SampleFiles:
drwxr-x---   2 cs30x10  cs30x       4096 May 18 20:44 directory/
-rw-r-----   1 cs30x10  cs30x          0 May 18 20:36 door>
---x--x--x   1 cs30x10  cs30x          0 May 18 20:48 executable*
----------   1 cs30x10  cs30x          0 May 18 20:49 noPermissions
-rw-r-----   1 cs30x10  cs30x          0 May 18 20:22 pipe|
-rw-r-----   1 cs30x10  cs30x          0 May 18 20:51 regularFile
-rwxr-x---   1 cs30x10  cs30x          0 May 18 20:21 socket=
lrwxrwxrwx   1 cs30x10  cs30x         11 May 18 20:06 symbolicLink@
```

Detailed Overview

The function prototypes for the various C and Assembly functions are as follows.

C routines:

```
int main( int argc, char * argv[] );
int parseArgs( int argc, char * argv[] );
int buildFileInfoTable( const char * filename, struct fileInfo ** tablePtr );
void printFileInfoTable( struct fileInfo * table, int entries, int printMode );
int modTimeCompare( const void * p1, const void * p2 );
int nameCompareRev( const void * p1, const void * p2 );
void printPermissions( const mode_t mode );
void printOwnerName( const uid_t uid );
```

Assembly routines:

```
int modTimeCompareRev( const void * p1, const void * p2 );
int nameCompare( const void * p1, const void * p2 );
void printGroupName( const gid_t gid );
```

For the Milestone, you will need to complete:

`parseArgs.c` `buildFileInfoTable.c` `modTimeCompareRev.s` `nameCompare.s`

C Functions to be Written

Listed below are the modules to be written in C.

---

pa4.c

```
int main( int argc, char * argv[] );
```

This function is the main driver for the program and will be delegating a majority of the functionality to the other functions we are writing.

1. Parse the command line arguments via `parseArgs()`. If an error is indicated, print the short usage statement and return indicating failure.
2. If the help flag was set, print the long usage statement to `stdout` and return, indicating success.
3. If no files or directories were specified through the command line arguments, we can assume the files in "`.`" (the current working directory) will be displayed. If this is the case, proceed with step 5.
4. Otherwise, cycle through the command line arguments which specify the files and/or directories to display the information of.
5. For each file and/or directory to display, we need to build the `fileInfo` table (array of `fileInfo` structs), print the `fileInfo` table, and then free the memory that was allocated for the table.

Reasons for error:

- Errors returned from `parseArgs()`

Return Value: `EXIT_SUCCESS` on success, `EXIT_FAILURE` on failure.

parseArgs.c

```
int parseArgs( int argc, char * argv[] );
```

Parse the command-line options using `getopt_long()`, setting the user-specified flags in an int variable. One of the best resources for this function is the man page [`man -s3c getopt_long`]. It must support the short and long options listed below. Note that for each flag, we just want to set the corresponding bit in the int variable to be returned that indicates all the flags set--we are not actually implementing any of the flags' behaviors in this function.

When creating the array of `struct option` for `getopt_long()`, set the third member of the struct (`flag`) to `NULL` for each option since we will be using bit masks instead of int flags to keep track of the user-specified options.

Flags:

Short:	`-a`	`-l`	`-r`	`-t`	`-B`	`-F`	`-h`
Long:	`--all`		`--reverse`		`--ignore-backups`	`--classify`	`--help`

Reasons for error:

- `getopt_long()` detects an invalid flag and automatically prints an error message → set the `ERR_FLAG` bit

Return Value: An int variable with the specific bits set corresponding to the flags entered by the user.

---

buildFileInfoTable.c

```
int buildFileInfoTable( const char * filename, struct fileInfo ** tablePtr );
```

This function takes in a file/directory name and an empty table of `fileInfo` structs (which should have been passed in as the address of a pointer to a `struct fileInfo`). The purpose of this function is to populate the table with information about the argument file/directory. When this function returns, `tablePtr` should be pointing to the address of the first `struct fileInfo` in the array.

> Before diving into this file, it is highly advised to thoroughly read the following man page: `man -s2 stat`.

First, we want to get information on the passed in file using `lstat()`.

If `filename` is not a directory, then it must be a regular file. In this case, we will populate the table with just a single `struct fileInfo` containing the information for this file. We need to allocate space for the `struct fileInfo` and save within it: the `name` of the file and its `stat` info (ensuring the `name` is null-terminated). Finally, add the `fileInfo` struct to the table and return.

If `filename` is a directory, we want to first print the name of the directory to `stdout` using `STR_DIR_LABEL` defined in `pa4Strings.h`. Then we want to loop through each file in the directory, adding the information about each file to our table.

1. In order to access the entries in the directory, we need to first open the directory using `opendir()`, which returns a `DIR *` (this is similar to how `fopen()` returns a `FILE *`). We can then use `readdir()` to read one entry of the directory at a time, returned as a `struct dirent`. This returned struct contains the name of the file read from the opened directory [`man dirent`].

2. Once we have the filename of the current entry, we need to construct the pathname to pass to `lstat()` to get the file's information. Knowing the name of the directory and the name of the file, we can construct the pathname using `STR_BUILD_PATHNAME` defined in `pa4Strings.h`. Once we pass this pathname to `lstat()`, we will have the information needed for the next entry in our table.

3. We now need to make space for our new table entry by dynamically expanding the table (array) of `fileInfo` structs. Once we have reallocated the array to contain space for an additional `struct fileInfo` in the last slot of the table, we can store within it: the `name` of the file (not the full pathname) and its `stat` info (ensuring the `name` is null-terminated).

4. We can continue processing each file in the directory, making sure to keep track of the number of entries stored in the `fileInfo` table.

5. After getting the name and information of all the files in the directory, close the directory using `closedir()` (just like we would close a file using `fclose()`), set the `tablePtr` to point to the first element in the table, and return.

Note: You do NOT have to recursively descend/read each directory.

Reasons for error:

- `lstat()` encountered an error → print `STR_LSTAT_ERR` and call `perror()`, passing in the name of the file.
    - If this happens with the initial `filename` passed in, return after printing errors.
    - If this happens while looping through a directory, skip to the next entry in the directory after printing errors.
- Memory allocation fails → print the error by calling `perror()`, passing in `STR_ALLOC_ERR`.
    - If any entries were successfully added to the table, also print `STR_PARTIAL_INFO`, and proceed with step 5 above.
- Directory couldn't be opened → print `STR_OPENDIR_ERR` and call `perror()`, passing in the name of the directory.

Return Value: The number of entries added to the table of `fileInfo` structs (if no entries were added to the table, return 0).

---

printFileInfoTable.c

```
void printFileInfoTable( struct fileInfo * table, int entries, int printMode );
```

This function handles printing all the output, displaying the information of each file in the file info table. To do so, we first need to sort the table using `qsort()` and the appropriate comparison function depending on the flags set by the user. After sorting, we can then print out the information of each file stored in the table. Make sure to adjust the output according to which flags are set.

Default behaviors:

- Sort the files alphabetically by name.
- Skip any hidden files starting with '`.`' (see `man -s3c basename` for how to access the filename and not the first character of the pathname--for example, `../pa4/Makefile` should not be considered a hidden file).
- Print the file name.
- Print a newline character after each entry.

Flags:

`-a`	`--all`	Do not ignore entries starting with '`.`'
`-l`		Use a long listing format
`-r`	`--reverse`	Reverse the order while sorting
`-t`		Sort by modification time, newest first
`-B`	`--ignore-backups`	Do not list implied entries ending with `~`
`-F`	`--classify`	Append indicator (one of `*/=>@|`) to entries
`-h`	`--help`	Displays the long usage and exits

Long listing format: (`-l`)
`man -s2 stat` will be very useful here. There is a space between every field in the long listing output.

1. Print the character corresponding to the file type (see the macros in `/usr/include/sys/stat.h` to determine the type of the file--e.g. `S_ISDIR` for is this file a directory, etc.).

`d`	Directory
`l`	Symbolic link (you do not need to display the link target)
`b`	Block device (print major and minor numbers instead of size)
`c`	Character device (print major and minor numbers instead of size)
`-`	Regular file

2. Print the file permissions.
3. Print the number of links (using `STR_NLINK` defined in `pa4Strings.h`).
4. Print the owner name.
5. Print the group name.
6. Print the file size in bytes (using `STR_SIZE` defined in `pa4Strings.h`). If the file is a block device or a character device, print the major and minor numbers instead of size (using `STR_MAJOR_MINOR` defined in `pa4Strings.h`). See `man -s3c makedev` for more information.
7. Print the modification time. If the modification time is greater than six months ago, format the time as `Month Day Year`. Otherwise, format the time as `Month Day Hour:Min`. See `man -s3c gettimeofday` for getting the current time. See `man -s3c ctime` for help formatting the time. Note that you will need to strip off the leading `Weekday` and trailing "`:Sec Year\n`".

8. Print the file name.
9. Print a newline character after each entry.

Append indicator to entries: (`-F`)
See the macros in `/usr/include/sys/stat.h` to determine the type of the file. Depending on the file type, append the corresponding character to the file name when the classify flag is set.

`/`	Directory
`@`	Symbolic link (you do not need to display the link target)
`*`	Executable (any of the execute permission bits are set)
`=`	Socket
`|`	Named pipe
`>`	Door

Return Value: None.

---

modTimeCompare.c

```
int modTimeCompare( const void * p1, const void * p2 );
```

This function will be used with `qsort()` to sort based on the modification times (newest first) of two `struct fileInfo` (which are passed in as void pointers). See `man -s2 stat` to figure out how to access the modification time of a `struct stat`.

Return Value: -1 if the first modification time is newer, 0 if the modification times are the same, or +1 if the first modification time is older.

---

nameCompareRev.c

```
int nameCompareRev( const void * p1, const void * p2 );
```

This function will be used with `qsort()` to sort based on the names of two `struct fileInfo`, in reverse alphabetical order.

Return Value: -1 if the first name is larger, 0 if the names are the same, or +1 if the first name is smaller.

---

printPermissions.c

```
void printPermissions( const mode_t mode );
```

This routine prints out the various rwx permissions for the owner, group, and others when the `-l` flag is set. Do not worry about SUID, SGID, or sticky bit settings. A - (dash) indicates that permission bit is not set.

See `man -s2 stat` for more information on `mode_t`.

Return Value: None.

printOwnerName.c

```
void printOwnerName( const uid_t uid );
```

This function will print the file owner name, given the `uid`. A call to `getpwuid()` will return a pointer to a `struct passwd`, from which we can determine the user's login name (which we want to print). If the `passwd` entry cannot be found, simply print the numeric value of the `uid`. Use the strings specified in `pa4Strings.h` (`STR_PWNAME` or `STR_UID`) to print out the owner name. These strings will ensure that either the owner's name or the `uid` will be printed with a field width of 8, left justified.

See `man -s3c getpwuid` for further information.

Return Value: None.

Assembly Functions to be Written

Listed below are the modules to be written in Assembly.

modTimeCompareRev.s

```
int modTimeCompareRev( const void * p1, const void * p2 );
```

This function will be used with `qsort()` to sort based on the modification times (oldest first) of two `struct fileInfo` (which are passed in as void pointers). See `man -s2 stat` to figure out how to access the modification time of a `struct stat`. Make sure to use the offset defined in `pa4Globals.c`.

This function must be a leaf subroutine.

Return Value: -1 if the first modification time is older, 0 if the modification times are the same, or +1 if the first modification time is newer.

nameCompare.s

```
int nameCompare( const void * p1, const void * p2 );
```

This function will be used with `qsort()` to sort based on the names of two `struct fileInfo`, in alphabetical order.

Return Value: -1 if the first name is smaller, 0 if the names are the same, or +1 if the first name is larger.

printGroupName.s

```
void printGroupName( const gid_t gid );
```

This function will print the group name, given the `gid`. A call to `getgrgid()` will return a pointer to a `struct group`, from which we can determine the name of the group. If the `group` entry cannot be found, simply print the numeric value of the `gid`. Use the strings specified in `pa4Globals.c` (`strGname` and `strGid`) to print out the group name. These strings will ensure that either the group's name or the `gid` will be printed with a field width of 8, left justified.

See `man -s3c getgrgid` for further information.

Return Value: None.

Unit Testing

You are provided with a basic unit test file for the Milestone functions of this assignment. This test file has minimal test cases and is only meant to give you an idea of how to write your own tests. **You must write unit test files for each of the Milestone functions, as well as add several of your own thorough test cases to all 4 unit test files. You will lose points if you don't do this!** You are responsible for making sure you thoroughly test your functions. Make sure you think about boundary cases, special cases, general cases, extreme limits, error cases, etc. as appropriate for each function. The Makefile includes the rules for compiling and running these tests. Keep in mind that your unit tests will not build until all required files for the unit tests have been written. These test files are not being collected for the Milestone and will only be collected for the final turnin (however, they should already be written by the time you turn in the Milestone because you should be using them to test your Milestone functions).

Unit tests you need to complete:

```
testparseArgs.c
testnameCompare.c
testmodTimeCompareRev.c
testbuildFileInfoTable.c
```

To compile:

```
$ make testnameCompare
```

To run:

```
$ ./testnameCompare
```

(Replace "`testnameCompare`" with the appropriate file names to compile and run the other unit tests)

README File

Your README file for this and all assignments should contain:

- High level description of what your program does.
- How to compile it (be more specific than: just typing "make"--i.e., what directory should you be in?, where should the source files be?, etc.).
- How to run it (give an example).
- An example of normal output and where that normal output goes (stdout or a file or ???).
- An example of abnormal/error output and where that error output goes (stderr usually).
- How you tested your program (what test values you used to test normal and error states) showing your tests covered all parts of your code (test coverage). (Be more specific than diff'ing your output with the solution output--i.e., what are some specific test cases you tried?, what different types of cases did you test?, etc.)
- Anything else that you would want/need to communicate with someone who has not read the assignment write-up but may want to compile and run your program.
- Answers to questions (if there are any).

Questions to Answer in the README

1. How can you encourage your peers to act with integrity, especially when they ask for your help on an assignment?

Extra Credit

There are 5 points total for extra credit on this assignment.

- Early turnin: **[2 Points]** 48 hours before regular due date and time
  [1 Point] 24 hours before regular due date and time
  (it's one or the other, not both)
- **[1 Point]** Display the targets of symbolic links with the long listing (see `--EC-show-link` below).
- **[2 Points]** Recursively list subdirectories encountered (see `--EC-recursive` below).

Extra Credit Flags

There are two flags that you will implement support for, `--EC-show-link`, and `--EC-recursive`. These correspond to the `-l` and `-R` flags, respectively, in the real `ls` utility (the one belonging to GNU Coreutils). We will test these flags individually (i.e., not mixed with other flags) when testing your extra credit implementation.

NOTE: You will not be creating any extra files for the extra credit. Simply add the support for the additional long flags to your existing implementation. The regular required functionality detailed above in the writeup should not be affected by your extra credit implementation.

A sample stripped executable provided for you to try and compare your output against is available in the public directory. Note that you cannot copy it to your own directory; you can only run it using the following command (where you will also pass in the command line arguments):

```
$ ~/../public/pa4refEC
```

`--EC-show-link`

When this flag is set, your program will have the same behavior as the `-l` flag you implemented above, however when symbolic links are encountered, it will also print out an arrow followed by the target of the symbolic link. When parsing the arguments, the long flag string `"EC-show-link"` should correspond to the short flag character `0x1`. You will need to add these values to `pa4Strings.h`.

For example, in the following directory, `regularFile` is the target of the symbolic link `symbolicLink`:

```
[cs30xzzz@ieng9]:pa4$ ./myls --EC-show-link ~/../public/pa4SampleFiles
/home/solaris/ieng9/cs30x/cs30xzzz/../public/pa4SampleFiles:
drwxr-x---   2 cs30x10  cs30x       4096 May 18 20:44 directory
-rw-r-----   1 cs30x10  cs30x          0 May 18 20:36 door
---x--x--x   1 cs30x10  cs30x          0 May 18 20:48 executable
----------   1 cs30x10  cs30x          0 May 18 20:49 noPermissions
-rw-r-----   1 cs30x10  cs30x          0 May 18 20:22 pipe
-rw-r-----   1 cs30x10  cs30x          0 May 18 20:51 regularFile
-rwxr-x---   1 cs30x10  cs30x          0 May 18 20:21 socket
lrwxrwxrwx   1 cs30x10  cs30x         11 May 18 20:06 symbolicLink -> regularFile
```

`--EC-recursive`

When this flag is set, your program will recursively print the contents of any subdirectories you encounter. At each step, you should print out all the files of the directory you're on (starting with the directory on which you call `myls`), and then, in alphabetical order recurse on any files that are directories, starting this process anew. When parsing the arguments, the long flag string `"EC-recursive"` should correspond to the short flag character `0x2`. You will need to add these values to `pa4Strings.h`.

For example, suppose we have a directory `dir`, whose directory structure looks like this:

```
dir
├── file1.h
├── file2.c
├── subdir1
│   ├── filea.pdf
│   └── subdir1a
│       └── foo.txt
└── subdir2
    └── fileb.py
```

Running your program on this directory should generate the following output:

```
[cs30xzzz@ieng9]:pa4$ ./myls --EC-recursive dir
dir:
file1.h
file2.c
subdir1
subdir2

dir/subdir1:
filea.pdf
subdir1a

dir/subdir1/subdir1a:
foo.txt

dir/subdir2:
fileb.py
```

Note how before we recursively print the contents of any subdirectory, we first print the directory we're currently on in its entirety (including the names of any folders). Then we can start listing the directories we wish to recurse into. For each of these directories, you need a way to print them and their contents.

Needless to say, recursion (as in, recursive function calls) lends itself very well to this task—just make sure you have a base case. Also, see to it that you don't recurse on the `.` and `..` directories, or you'll recurse indefinitely.

Milestone Turn-in Instructions

Milestone Turn-in - due Wednesday night, May 25 @ 11:59 pm [15 points of Correctness Section]

Before final and complete turnin of your assignment, you are required to turnin several modules for the Milestone check.

Files required for the Milestone:			
`parseArgs.c`	`buildFileInfoTable.c`	`modTimeCompareRev.s`	`nameCompare.s`

Each module must pass all of our unit tests in order to receive full credit.

A working Makefile with all the appropriate targets and any required header files must be turned in as well. All Makefile test cases for the milestone functions must compile successfully via the commands `make test***`.

In order for your files to be graded for the Milestone Check, you must use the milestone specific turnin script.

```
$ cse30_pa4milestone_turnin
```

To verify your turn-in:

```
$ cse30verify pa4milestone
```

Final Turn-in Instructions

Final Turn-in - due Wednesday night, June 1 @ 11:59 pm

Once you have checked your output, compiled, executed your code, and finished your README file (see above), you are ready to turn it in. Before you turn in your assignment, you should do `make clean` in order to remove all the object files, lint files, core dumps, and executables.

Files required for the Final Turn-in:

buildFileInfoTable.c	pa4.h	printGroupName.s
modTimeCompare.c	pa4Globals.c	printOwnerName.c
modTimeCompareRev.s	pa4Strings.h	printPermissions.c
nameCompare.s	parseArgs.c	Makefile
nameCompareRev.c	printFileInfoTable.c	README
pa4.c		
testparseArgs.c	testnameCompare.c	testmodTimeCompareRev.c
testbuildFileInfoTable.c		

Use the above names *exactly* otherwise our Makefiles will not find your files.

How to Turn in an Assignment

Use the following turnin script to submit your full assignment before the due date as follows:

```
$ cse30turnin pa4
```

To verify your turn-in:

```
$ cse30verify pa4
```

Up until the due date, you can re-submit your assignment via the scripts above. Note, if you turned in the assignment early for extra credit and then turned it in again later (after the extra credit cutoff), you will no longer receive early turn-in credit.

Failure to follow the procedures outlined here will result in your assignment not being collected properly and will result in a loss of points. Late assignments WILL NOT be accepted.

If there is anything in these procedures which needs clarifying, please feel free to ask any tutor, the instructor, or post on the Piazza Discussion Board.

Style Requirements

You will be graded on style for all the programming assignments. The requirements are listed below. Read carefully, and if any of them need clarification do not hesitate to ask.

- Use reasonable comments to make your code clear and readable.
- Use file headers and function header blocks to describe the purpose of your programs and functions. Sample file/function headers are provided with PA0.
- Explicitly comment all the various registers that you use in your assembly code.
- In the assembly routines, you will have to give high level comments for the synthetic instructions, specifying what the instruction does.
- You should test your program to take care of invalid inputs like non-integers, strings, no inputs, etc. This is very important. Points will be taken off if your code doesn't handle exceptional cases of inputs.
- Use reasonable variable names.
- Error output goes to stderr. Normal output goes to stdout.
- Use #defines and assembly constants to make your code as general as possible.
- Use a local header file to hold common #defines, function prototypes, type definitions, etc., but not variable definitions.
- Judicious use of blank spaces around logical chunks of code makes your code easier to read and debug.
- Keep all lines less than 80 characters, split long lines if necessary.
- Use 2-4 spaces for each level of indenting in your C source code (do not use tab). Be consistent. Make sure all levels of indenting line up with the other lines at that level of indenting.
- Do use tabs in your Assembly source code.
- Always recompile and execute your program right before turning it in just in case you commented out some code by mistake.
- Do #include only the header files that you need and nothing more.
- Always macro guard your header files (#ifndef ... #endif).
- Never have hard-coded magic numbers (any number other than -1, 0, or 1 is a magic number). This means we shouldn't see magic constants sitting in your code. Use a #define if you must instead.