Homework 4: Unix Signals and (more) File I/O

Due: Monday April 28, 2014 11:55 PM Pacific USA time zone.

Points on this assignment: 470 points with 50 bonus points available.

Work submitted late will be penalized as described in the course syllabus. You must submit your work twice for this and all other homework assignments in this class. Ecampus wants to archive your work through Blackboard and EECS needs you to submit through TEACH to be graded. If you do not submit your assignment through TEACH, it cannot be graded (and you will be disappointed with your grade). Make sure you submit your work through TEACH and Blackboard. Submit your work for this assignment as a *single tar.bzip file* through TEACH and Blackboard. The same single tar.bzip file should also be submitted through Blackboard.

In **all** your source files (.c files, .h files, .txt files, and even the Makefile), you need to have 4 things at the top of every file as comments:

1. Your name
2. Your OSU email address (ONID or engineering)
3. The class name and section (this is CS311-400)
4. The assignment number (this is homework #4).

If you omit the 4 header lines, you can expect to lose 20% on your grade. It is easy to do, so just do it.

Place all of the files you produce for this assignment in a single directory, called `Homework4`. My dislike of spaces in directory names (file names in general) continues.

In this assignment, you will be working with UNIX signals and file I/O system calls. This assignment will be only in C, with a good bit of Makefile automation mixed in. If you look through some of the C code on the previous homework assignments, you can find some C code fragments that may be helpful in this assignment. You have 2 weeks to complete this assignment. However, I urge you to not delay beginning it. Starting this soon after or concurrently with Homework #3 should work very well. Homework #3 is intended to give you a jump start on this assignment.

This assignment is a significant step up in challenge and complexity from previous homework assignments. Expect to dedicate a significant amount of time to this assignment. Don't expect to start this on the Sunday before it is due and complete it on time. You'll be really grumpy.

Remember that the programming work in this class is intended to be individual work, not group work.

1. **5 points**. When you are ready to submit your files for this assignment, make sure you submit a *single bzip file*. Review homework #1, problem #1 if you need a refresher on how to do this. If your file is not a single bzip file, you cannot receive points on this assignment. By now, you are used to hearing this.

2. **40 points total**. Write a C program on `eos-class` to send and receive Unix signals. Don't go overboard on this portion of the assignment. You should be able to easily complete this portion with less than 50 lines of C code. It may take you less time to write the C code than it took me to write the problem description.

   2.1. You will write an application in C (`sig_demo.c`) that installs 3 different functions as signal handlers. Your C code will handle the following signals:

      2.1.1. `SIGUSR1`

      2.1.2. `SIGUSR2`

      2.1.3. `SIGINT`

   2.2. It may be tempting to use a single signal handler for all 3 signals, but don't. Create a separate single handler function for each signal.

   2.3. When your application receives the `SIGUSR1` signal, it should print (**10 points**):

```
SIGUSR1 has been caught
```

      2.3.1. Your application should not exit when the `SIGUSR1` signal is handled.

   2.4. When your application receives the `SIGUSR2` signal, it should print (**10 points**):

```
SIGUSR2 has been caught
```

      2.4.1. Your application should not exit when the `SIGUSR2` signal is handled.

   2.5. When your application receives the `SIGINT` signal, it should print (**20 points**):

```
SIGINT has been caught, terminating the program
```

      2.5.1. When your code receives the `SIGINT` signal, your application should exit (after printing the message of course).

   2.6. You may call `printf()` for these messages within your signal handler. I know that it is not strictly *safe* to call `printf()` within a signal handler, but we will, just this one time.

   2.7. Once your application has installed the 3 signal handlers, it should send the signals to itself, in this order: `SIGUSR1`, `SIGUSR2`, `SIGINT`. The `getpid()` system call can be your friend here. I want you to use `kill()` not `raise()` to send the signals.

   2.8. When I run your compiled program, I should see the following

```
$ ./sig_demo
SIGUSR1 has been caught
SIGUSR2 has been caught
SIGINT has been caught, terminating the program
$
```

2.9. If you find yourself struggling with this portion of this assignment, you need to review chapter 20 in TLPI and spend some time looking at sections 20.4 and 20.5 and listing 20-1. Again, think in terms of 50 lines of C code.

2.10. If you make use of some resources (such as web sites), make sure you reference them in your code. Using the references and not citing them is a violation of the code of student conduct.

2.11. Put a target in your `Makefile`. Lots of details about the single `Makefile` for assignment this are shown below, in section 3.9.

3. **315 points total**. This portion of the homework is about reading and writing files. You will need to `stat()` or `fstat()` files, check file permissions, check file time stamps, and perform seek through a file. You will need more than 50 lines of C code to complete this portion of the assignment, quite a bit more.

3.1. Write a C program on `eos-class` called `myar`. This program will illustrate the use of file I/O on UNIX by managing a UNIX archive library, in the standard archive format.

3.2. Once compiled your program should run in a manner similar to the standard UNIX command `ar`. You will need to spend some time looking at the man page for `ar`.

3.3. For this assignment, the following is the generic command line syntax **your** program must support (this does differ from the standard `ar` command line syntax):

```
myar key archive-file [member [...]]
```

3.3.1. The *`archive-file`* is the name of the archive file to be used, and *`key`* is one of the following options. Notice that only a single key can be on the command line.

Switch	Action
`-q`	"Quickly" append named files (members) to archive. **40 points**. Check the meaning of append in the notes below (section 3.6.3). If the key –q is used and no file members are on the command line, an archive file with no members will be created, just like `ar` does..
`-x`	Extract named members. **40 points**. Just as the regular `ar` command, if no member is named on the command line when extracting files, all files are extracted from the archive. The

	permissions on the extracted files should match permissions on the files before archiving (as described in the notes below). To "extract" a file is not to remove the file from the archive, it is to make a copy of the file outside of the archive file.
`-t`	Print a concise table of contents of the archive. **15 points**. The concise table of contents for your application (myar) should match exactly the output from the "`ar t`" command on the same archive file.
`-v`	Print a verbose table of contents of the archive. **30 points**. The verbose table of contents for your application (`myar`) should match exactly the output from the "`ar tv`" command on the same archive file. See the man page on `ar`.
`-d`	Delete named files from archive. **70 points**. Make sure you read the note below about the `-d` option and creation of a new file. To delete a file from the archive does remove it from the archive. If the `-d` key is used on the command line without any members, the archive file is unaltered.
`-A`	Quickly append all "regular" files in the current directory (Except the archive itself). **50 points**. There is not an option for the Unix `ar` command that does this. The `-A` key is used without any members listed on the command line. If a member is on the command line with the `-A` key, issue a warning that it is ignored.
`-w`	**Extra credit 25 points**: for a given timeout (in seconds), add all modified files to the archive (Except the archive itself). **See note 3.6.8.** There is not an option for the Unix `ar` command that does this. If you do this extra credit work, make sure you indicate so clearly in your assignment. You can only receive extra credit on the project of all the other command line options work correctly.

3.4. The archive file maintained must use ***exactly*** the standard format defined in `/usr/include/ar.h`, and in fact may be tested with archives created with the `ar` command.

3.4.1. The archive files created the regular `ar` command and those created with your `myar` must be interoperable.

3.4.2. **Do not copy or in any way modify the `ar.h` include file**.

3.4.3. The sizeof() operator is your friend.

3.5. The options listed in the table above are compatible with the options having the same name in the `ar` command, except for the following exceptions: the `-v` and `-t` command take no further argument, and list all files in the archive. `-v` option is short for `-t -v` (or `tv`) on the regular `ar` command. The `-A` and `-w` commands are new options not in the usual `ar` command.

3.6. Notes (lots of them and they are **all** important):

3.6.1. For the `-q` and `-A` command `myar` should create an archive file if it doesn't exist, using permissions "666".

3.6.1.1. For the other commands `myar` reports an error if the archive specified on the command line does not exist, or is in the wrong format.

3.6.1.2. A bad file format causes and error statement and your program to exit. Don't try and do anything with a file that has a bad format, just issue an error statement and exit.

3.6.2. You will have to use the system calls `stat()` and `utime()` to properly deal with extracting and restoring the proper timestamps.

3.6.2.1. Since the archive format only allows a single timestamp, store the `mtime` and use it to restore both the `atime` and `mtime`.

3.6.2.2. Permissions should also be restored to the original value, subject to `umask` limitations.

3.6.3. The `-q` and `-A` commands do not check to see if a file by the chosen name already exists within the archive file. They simply append the file(s) to the end of the archive.

3.6.4. The `-x` and `-d` commands operate on the **first file name** matched in the archive, without checking for further matches.

3.6.4.1. It is possible for a file name to exist more than once in an archive; use the first one that matches.

3.6.5. In the case of the `-d` option, you will have to build a new archive file to recover the space. Do this by unlinking the original file after it is opened (or after you've completed reading it), and creating a new archive with the original name.

3.6.6. Since file I/O is slow, do not make more than one pass through the archive file; an issue especially relevant to the multiple member delete case.

3.6.7. You are **required** to handle multiple file names as members on the command line.

3.6.7.1. Expect a 90% deduction if you cannot handle multiple file names on the command line as members.

3.6.8. For the `-w` flag (optional extra credit), the command will take as long as specified by the timeout argument. You should print out a status message upon adding a new file. This may result in many different copies of the same file in the archive.

3.6.9. Make sure you lookup what a "regular" file is in UNIX. Comment on this meaning in your code. Cite the references you use to develop your understanding of "regular" files.

3.6.10. It is not a requirement that you use `getopt()` to process `argv` from the command line. If you have a simpler method to use, that is fine. You'll probably get more practice using `getopt()` later in the class.

3.6.11. I have created some sample files that you can use for testing your application.

3.6.11.1. They are all plain text, so you can actually just `cat` the archive file after you've created it to see how it looks and compare it to one created with `ar`. **Try it, it can make the mystery of this assignment completely go away**.

3.6.11.2. The sample files are: `1-s.txt`, `2-s.txt`, `3-s.txt`, `4-s.txt`, and `5-s.txt`.

3.6.11.3. One of the things you'll note about the test files that some of them have an **even number of bytes** and others have an **odd number of bytes**. Think about why I might have pointed that out to you.

3.6.11.4. The sample files can be found on `eos-class` in:

`/usr/local/classes/eecs/spring2013/cs311/src/Homework4`

3.6.12. Because I know this is tricky and can cause undue grief, ***you need to carefully read the following 2 web pages and search for the word "even":*** http://en.wikipedia.org/wiki/Ar_(Unix) and http://www.unix.com/man-page/opensolaris/3head/ar.h/

3.6.12.1. One of the other things you learn in this assignment is how to carefully read (and re-read, and re-re-read) specifications and man pages.

3.6.12.2. Just take my word for this and carefully read the 2 web pages.

3.6.13. The length of file names that I use to test your code will not exceed 15 characters for this assignment.

3.6.14. Make sure your code compiles before you submit it. If your code does not compile, you are not going to be happy with your grade. See the syllabus section 12.9.1.

3.6.15. Because you are *potentially* working with binary files, you **must** use the `read()` and `write()` system calls, not `fscanf()` and `fprintf()`, for the file I/O. You can use `printf()` for terminal output.

3.6.16. Do yourself a big favor and do not try adding binary (non-text) files into your archive file for testing against the regular `ar` command.

3.6.16.1. Don't try putting your .o and other binary files into your archive using `myar`.

3.6.16.2. You'll get some odd messages about a table of contents that won't be very helpful.

3.6.16.3. Test things against `ar` using plain text files.

3.6.16.4. I'll only be testing your code using plain text files.

3.6.17. Printing in octal (such as for permissions) is probably not something you've done for quite a while. C can make it pretty easy for you. This web page gets low marks for beauty, but high marks for summarizing all those options for `printf()`: http://wpollock.com/CPlus/PrintfRef.htm. This is a nice page from "The C Book", listed as a recommended reference for C programming: http://publications.gbdirect.co.uk/c_book/chapter9/formatted_io.html.

3.6.18. For **25 points extra credit**, any time a file is added that already exists, remove the old copy from the archive, but only if it is not the same. If identical, do not add the new file. Make sure you clearly comment your meaning of "identical."

3.6.18.1. If you do this extra credit work, make sure you indicate so clearly in your assignment.

3.6.18.2. You can only receive this extra credit if all the (non-extra-credit) command line options work correctly.

3.7. If you find yourself puzzling/lamenting/anguishing over even and odd file sizes, make sure you **read the entire assignment description** (including all the many notes, like this one 3.6.12).

3.8. If you find that your C programming or Makefile skills are still a bit rusty (or nearly non-existant), you may want to peruse some of the tutorials shown at the bottom of the week 3 assignments web page on Blackboard. The book "The C Book" is pretty good and free: http://publications.gbdirect.co.uk/c_book/

3.9. You must have a single `Makefile` for this homework. Your `Makefile` must contain (at least) the following targets:

Target	Action
all	Builds all dependent C code modules for both applications in the directory. The dependency of the all target should be myar and sig_demo. (**10** points)
myar	Builds all dependent C code modules for the myar application in the directory. The dependency of myar should be myar.o. (**10 points**)
myar.o	Compiles the myar.c module for the myar application in the directory, based off of any changed dependent modules. The dependency for myar.o should be myar.c. (**10 points**)
sig_demo	Builds all dependent C code modules for the sig_demo application in the directory. The dependency of sig_demo should be sig_demo.o. (**10 points**)
sig_demo.o	Compiles the sig_demo.c module for the sig_demo application in the directory, based off of any changed dependent modules. The dependency for sig_demo.o should be sig_demo.c. (**10 points**)
clean	Deletes all executable programs (homework3), object files (files

	ending in .o produced by gcc), and any editor droppings (# files from vi and ~ files from emacs). Make sure you use this before you bundle all your files together for submission. (**20 points**)

3.9.1. When I build your assignment, I should be able to just type

```
make clean
make
```

to have it completely clean the directory and build both `sig_demo` and `myar`.

3.9.2. There are additional requirements for your `Makefile` below (assignment section 4).

3.10. Since there are quite a few parts to completing this portion of the homework, I suggest/encourage/admonish that you start small.

3.10.1. Make a plan for how you will break down the assignment into smaller pieces. Don't just start hacking away.

3.10.2. Plan the work. Work the plan.

3.10.3. My recommendation on how to proceed with this assignment is:

3.10.3.1. Create the Makefile and keep it current as the project is developed.

3.10.3.2. Create an archive file using the standard `ar` command.

3.10.3.2.1. Use all the small sample files provided, `1-s.txt` through `5-s.txt`.

3.10.3.2.2. Cat the archive file to the screen.

3.10.3.2.3. This will take a lot of the mystery out of this assignment.

3.10.3.2.4. Create a couple more archive files like this and cat them to improve your comfort with the file format.

3.10.3.2.5. It's not scary.

3.10.3.3. Back to your C code, process all command line options using `getopt()` (or however you decide to parse the command line). Just use empty stubs for command line options you've not yet completed. You'll fill in the stub code as your work progresses.

3.10.3.4. Create code that performs the `-t` option on an archive file you've created with the regular `ar`. Once you have stepped through the file for the –t option, you have a good idea how to read and manage the archive file.

3.10.3.4.1. Having completed homework #3, you already have a head start on completion of the `-t` and `-v` options of the program. Seek and you shall succeed.

3.10.3.5. Move to the `-v` option on an archive file. You'll need to pull out more data and deal with messy dates, permissions, and octal numbers. Not as bad as lions, tigers, and bears, but close.

3.10.3.6. After that, I'd go `-q`, `-x`, `-d`, and `-A`, but go in the order that makes a clear progression for you.

3.10.4. **Remember**, the archive files you create with your `myar` need to be interoperable with those created with the standard `ar`. So you can easily test one from the other.

4. **110 points total**. Put tests as targets in your `Makefile`. Since you need to have a number of tests for your code, put them into the `Makefile` so it is easy for you to run them easily and consistently (and frequently). This is a simple form of regression testing. Something in the back of my mind tells me that this could be useful for other assignments in this class. Hmmmm…

4.1. You are going to want to have local copies or (a better choice) symbolic links to the sample test files for the following steps. It just makes command line shorter.

4.2. Put a target called `testq12345` in your `Makefile`. The `testq12345` target will do the following:

4.2.1. Remove any file named `ar12345.ar`. If the file `ar12345.ar` does not exist, don't show an error. Read the `man` page for `rm`.

4.2.2. Remove any file named `myar12345.ar`. If the file `myar12345.ar` does not exist, don't show an error.

4.2.3. Create a file named `ar12345.ar` using this call:

```
ar q ar12345.ar 1-s.txt 2-s.txt 3-s.txt 4-s.txt 5-s.txt
```

4.2.4. Create a file named `myar12345.ar` using this call:

```
myar -q myar12345.ar 1-s.txt 2-s.txt 3-s.txt 4-s.txt 5-
s.txt
```

4.2.5. Compare the files created by `ar` and `myar`.

```
diff ar12345.ar myar12345.ar
```

4.2.6. The result of the diff command should show no differences.

4.3. Put targets in your `Makefile` called `testq135` and `testq24` which are like the target `testq12345`, but only use the noted subset of files (1, 3, 5, and 2, 4).

4.4. Put a target in your `Makefile` called `testq` that will run the `testq12345`, `testq135`, and `testq24` targets. **20 Points**.

4.5. Put a target called `testt12345` in your `Makefile`. The `testt12345` target will do the following:

4.5.1. Remove any file named `ar12345.ar`. If the file `ar12345.ar` does not exist, don't show an error.

4.5.2. Create a file named `ar12345.ar` using this call:

```
ar q ar12345.ar 1-s.txt 2-s.txt 3-s.txt 4-s.txt 5-s.txt
```

4.5.3. Run the following commands:

```
ar t ar12345.ar > ar-ctoc.txt
myar -t ar12345.ar > myar-ctoc.txt
```

4.5.4. Compare the table of contents files by using this command:

```
diff ar-ctoc.txt myar-ctoc.txt
```

4.5.5. The result of the diff command should show no differences.

4.6. Put targets in your `Makefile` called `testt135` and `testt24` which are like the target `testt12345`, but only use the noted subset of files (1, 3, 5, and 2, 4).

4.7. Put a target in your `Makefile` called `testt` that will run the `testt12345`, `testt135`, and `testt24` targets. **20 Points**.

4.8. Put a target called `testv12345` in your `Makefile`. The `testv12345` target will do the following:

4.8.1. Remove any file named `ar12345.ar`. The file `ar12345.ar` does not exist, don't show an error.

4.8.2. Create a file named `ar12345.ar` using this call:

```
ar q ar12345.ar 1-s.txt 2-s.txt 3-s.txt 4-s.txt 5-s.txt
```

4.8.3. Run the following commands:

```
ar tv ar12345.ar > ar-vtoc.txt
myar -v ar12345.ar > myar-vtoc.txt
```

4.8.4. Compare the table of contents files by using this command:

```
diff ar-vtoc.txt myar-vtoc.txt
```

4.8.5. The result of the diff command should show no differences.

4.9. Put targets in your `Makefile` called `testv135` and `testv24` which are like the target `testv12345`, but only use the noted subset of files (1, 3, 5, and 2, 4).

4.10. Put a target in your `Makefile` called `testv` that will run the `testv12345`, `testv135`, and `testv24` targets. **20 Points**.

4.11. Put a target in your `Makefile` called `tests` that will run the `testq`, `testt`, and `testv` targets.

4.11.1. With this, you should only need to run "`make tests`" to run a fairly complete set of tests on your code.

4.11.2. Being able to easily run a consistent set of tests on your code and help you quickly locate inadvertent changes in your program that might not have been caught until much later. **50 Points**.

4.12. Think about using macros in your `Makefile` for some of these.

4.13. Test early and test often. Make testing part of your normal development cycle.

4.14. I'm sure you will want to have additional tests; you can put them in your `Makefile` as well.

Things to include with the assignment (in a single tar.bzip file):

1. C source code for the solutions to the posed problems (all files).
2. A Makefile to build your code.
3. A plain simple text file describing what tests you ran.
4. Remember to put your name in every file you submit.
5. You do not need to (and should not) include a copy of the data files from eos-class in your assignment submission.

Please combine all of the above files into a single tar.bzip file prior to submission. Run the "`make clean`" before creating the tar.bzip file.