Project 3: Unix Signals and File I/O

Due: Monday 29 April 2013, 23:59:00 (11:59 PM) Pacific USA time zone.

Points on this assignment: 150 points with 10 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. Submit your work for this assignment as a *single bzip file* through TEACH. The same single bzip file should also be submitted through BlackBoard.

Place all of the files you produce for this assignment in a single directory, maybe something called Homework3.

In this assignment, you will be working with Unix signals and file I/O system calls. This assignment is a significant step up in complexity from the previous 2 homework assignments. This assignment will also be only in C, with some Makefile fun mixed in. If you look through some of the C code on the previous 2 homeworks, 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.

1. **10 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.

2. **20 points**. Write a C program 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.

   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: `SIGUSR1`, `SIGUSR2`, and `SIGINT`. 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.

   When your application receives the `SIGUSR1` signal, it should print:

   ```
   Caught SIGUSR1
   ```

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

When your application receives the SIGUSR2 signal, it should print:

```
Caught SIGUSR2
```

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

When your application receives the SIGINT signal, it should print:

```
Caught SIGINT, exiting
```

When it receives the SIGINT signal, your application **should** exit (after printing the message of course).

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 this time.

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.

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

```
$ ./sig_demo
Caught SIGUSR1
Caught SIGUSR2
Caught SIGINT, exiting
$
```

3. **90 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 a `seek()` into a file.

   Write a C program on `os-class` or CS311 VM called `myar.c`. This program will illustrate the use of file I/O on Unix by maintaining a Unix archive library, in the standard archive format.

   Once compiled your program should run in a manner similar to the standard Unix command **`ar`**. You can look at the man page for **`ar`** for insight. However, for this assignment, the following is the syntax your program must support:

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

   where `archive-file` is the name of the archive file to be used, and `key` is one of the following options:

`-q`	Quickly append named files (members) to archive. 15 points (of the 90). Check the meaning of append in the notes below.
`-x`	Extract named members. 15 points (of the 90). 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).
`-t`	Print a concise table of contents of the archive. 5 points (of the 90). 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. 5 points (of the 90). 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. 20 points (of the 90). Make sure you read the note below about the `-d` option and creation of a new file.
`-A`	Quickly append all "regular" files in the current directory (Except the archive itself). 20 points (of the 90). There is not an option for the Unix `ar` command that does this.
`-w`	**Extra credit 5 points**: for a given timeout (in seconds), add all modified files to the archive (Except the archive itself). **See note h.** There is not an option for the Unix `ar` command that does this.

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

The options listed 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` is short for `-t -v` on the `ar` command. The `-A` and `-w` commands are new options not in the usual `ar` command.

Notes:

a) For the `-q` command `myar` should create an archive file if it doesn't exist, using permissions "666". For the other commands `myar` reports an error if the archive does not exist, or is in the wrong format.

b) You will have to use the system calls `stat` and `utime` to properly deal with extracting and restoring the proper timestamps. Since the archive format only allows one timestamp, store the `mtime` and use it to restore both the `atime` and `mtime`. Permissions should also be restored to the original value, subject to `umask` limitation.

c) The `-q` and `-A` commands do not check to see if a file by the chosen name already exists. It simply appends the files to the end of the archive.

d) The `-x` and `-d` commands operate on the first file matched in the archive, without checking for further matches. It is possible for a file name to exist more than once in an archive, use the first one that matches.

e) 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.

f) Since file I/O is expensive, do not make more than one pass through the archive file, an issue especially relevant to the multiple delete case.

g) You are required to handle multiple file names as members.

h) For the `-w` flag, 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.

i) Make sure you lookup what a "regular" file is in Unix.

j) You must have a `Makefile` for this portion of the homework. Your `Makefile` must contain the following 2 targets:

   1. `all` -- the `all` target should compile and link the final program. The `all` target should also be the default target (which only means that it is the first target in the Makefile).
   2. `clean` – the `clean` target should remove all binary applications (`myar`) and object files (.o's). In addition, you should remove any editor droppings from emacs, vi, or whatever. The sample Makefile from Homework1 includes a `clean` target. Make sure you use this before you bundle all your files together for submission.

   The `Makefile` may (should) contain other targets the build any out of date modules. Again, the sample `Makefile` from Homework1 gives some guidance on this. Targets like `all` and `clean` are sometimes called phony targets because they don't actually produce a file.

   When I build your program, I should be able to just type `make` to have it completely build.

k) For **5 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."

l) It is not necessary that you use `getopt()` to process `argv` from the command line. If you have a simpler method to use, that is fine.

m) I have created some sample files that you can use for testing your application. They are all plain text, so you can actually just `cat` the archive file after you've create it to see how it looks and compare it to one created with `ar`. The sample files are: 1-s.txt, 2-s.txt, 3-s.txt, 4-s.txt, and 5-s.txt. 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. This will be important to you.

n) Because I know this is tricky, ***you need to carefully read the following 2 web pages and search for the word "even":*** http://en.wikipedia.org/wiki/Ar_%28Unix%29 and http://www.unix.com/man-page/opensolaris/3head/ar.h/ One of the other things you learn in this assignment is how to carefully read specifications and man pages. Just take my word for this and carefully read the 2 web pages.

o) Since there are quite a few parts to completing this problem in the homework, I suggest that you start small. Start with just creating code that performs the `-t` option on an archive file you've created with `ar`. From there, move to the `-v` option. From there, I'd go `-q`, `-x`, `-d`, `-A`, and `-w`, but go in the order that make a clear progression for you.

p) The length of file names used will not exceed 15 characters for this assignment.

q) Make sure your code compiles before you submit it. Simply being able to compile your code is worth 5 points (of the 90).

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

4. **30 points**. 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 then easily and consistently. This is a simple form of regression testing.
   a. Put a target called `testq12345` in your `Makefile`. The `testq12345` target will do the following:
      i. Remove any file named `ar12345.ar`. The file `ar12345.ar` does not exist, don't show an error.
      ii. Remove any file named `myar12345.ar`. The file `myar12345.ar` does not exist, don't show an error.
      iii. 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
```
      iv. Create a file named ar12345.ar using this call:
```
myar -q myar12345.ar 1-s.txt 2-s.txt 3-s.txt 4-s.txt 5-
s.txt
```
      v. Compare the files create by ar and myar.
```
    diff ar12345.ar myar12345.ar
```
      vi. The result of the diff command should show no differences.
   b. 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).
   c. Put a target in your `Makefile` called `testq` that will run the `testq12345`, `testq135`, and `testq24` targets. 5 Points (of the 30).
   d. Put a target called `testt12345` in your `Makefile`. The `testt12345` target will do the following:
      i. Remove any file named `ar12345.ar`. The file `ar12345.ar` does not exist, don't show an error.
      ii. 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
```
      iii. Run the following commands:
```
ar t ar12345.ar > ar-ctoc.txt
myar -t ar12345.ar > myar-ctoc.txt
```
      iv. Compare the table of contents files by using this command:

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

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

e. 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).

f. Put a target in your `Makefile` called `testt` that will run the `testt12345`, `testt135`, and `testt24` targets. 5 Points (of the 30).

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

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

   ii. 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
   ```

   iii. Run the following commands:

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

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

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

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

h. 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).

i. Put a target in your `Makefile` called `testv` that will run the `testv12345`, `testv135`, and `testv24` targets. 5 Points (of the 30).

j. Put a target in your `Makefile` called `tests` that will run the `testq`, `testt`, and `testv` targets. With this, you should only need to run "`make tests`" to run a fairly complete set of tests on your code. 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. Think about using macros in your `Makefile` for some of these. Test early and test often. 15 Points (of the 30).

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

Things to include with the assignment (in a single bzipped file):

1. C source code for the solutions to the posed problem (all files).
2. A Makefile to build your code.
3. A file describing how you ran your tests.

A file describing how you ran your tests, including where you found input files
Please combine all of the above files into a single bzip file prior to submission.