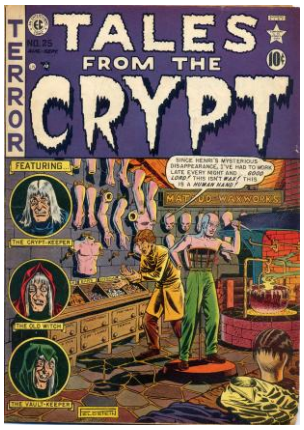
Lab 3:
THREADS FROM THE CRYPT

Please read this entire assignment. Take some notes. Think about it some. Take some more notes. Look for answers to your questions in this document. **Plan the work. Work the plan**.



Do not place ANY directories in your submitted tar file. I will not change into any sub-directories to hunt down your source files. When you create your `tar.gz` file to submit, do it within the directory where you created the source files, **NOT from a higher level directory. If I cannot find your source files in the same directory where I extract your submitted tar file, I will simply give you a zero on the assignment**. Submit a single `tar.gz` file to Canvas for this assignment.

In this assignment, you will be working with a multi-threaded application. Specifically, you'll be writing a program that uses PThreads and the `crypt()` function. **I urge you to not delay beginning it**.

1. **You must use `babbage` for this assignment, not `ada`.** `Babbage` is the general purpose server for this class, especially when we are doing multi-threaded or multi-process applications. If I find you running this application on `ada`, I will contact you. **If you continue to use (misuse) `ada`, I'll take stronger actions**.
2. You are going to be making a lot of calls to the `crypt()` function. **Let me just apologize right here and say the documentation is weak, bad, and in some places just wrong**. I'll give you several places to find information about the function. Taken together, they work pretty well. I just wish I could find one that worked well on its own.
   - **2.1.** When I make reference to the `crypt()` function, I mean the collection of functions listed in the `crypt()` man page. This includes: `crypt()`, `crypt_r()`, `crypt_rn()`, and `crypt_ra()`. **The `crypt()` function is not reentrant, hence not thread-safe. Do not use that one. I used the `crypt_rn()` function in my code.**
   - 2.2. See section 10 on page 6 for some references for the `crypt()` function.
3. **The foci for this assignment are**:
   - 3.1. **Write a multi-threaded application that demonstrates the performance improvement of using multiple threads** and
   - 3.2. **An introduction to the use of `crypt()` and salted passwords**.

4. Required command line options.

Option	Action
`-i filename`	Specify the name of the input file. This file contains the hashed password values that you will crack. **This is a required command line option.**
`-o filename`	Specify the name of the output file. If this option is not specified for input, `stdout` is written.
`-d filename`	Specify the name of the dictionary file that contains plain-text words that will be used to crack the passwords. **This is a required command line option.**
`-t #`	Specify the number of threads to use. The default is to use a single thread. You'll use up to 24 threads.
`-v`	Enable some verbose processing. • This is really to help you follow what your code is doing. • You need to accept this switch, have your code emit some diagnostics with this set. If your code simply prints a message that verbose is enabled, it will meet this objective. • **All the messages from the verbose output must be sent to `stderr`, NOT `stdout`. Do not comingle output and diagnostics. Comingle means Comangle.** • **Since we are using `stderr` to collect the accounting information, you need to be make sure you can disable all supplemental diagnostic information you send to `stderr` with verbose.**
`-h`	Output some helpful text about command line options. Make your output for this option look like mine. **This output must go to `stderr`.**
`-n`	Apply the `nice()` function to your running process. Pass the value 10 to `nice()` to lower the probability your process will be scheduled. We want to be good citizens.

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

Target	Action
`all`	Builds all dependent C code modules for your application in the directory. This should be the first target in your `Makefile`.
`thread_hash`	Builds all dependent C code modules for the `thread_hash` application in the directory. The dependency of `thread_hash` should be `thread_hash.o`.
`thread_hash.o`	Compiles the `thread_hash.c` module for the thread_hash application in the directory, based off of any

	changed dependent modules. The dependency for `thread_hash.o` should be `thread_hash.c`.
`clean`	Deletes all executable programs, object files (files ending in .o produced by `gcc`), and any editor chaff (`#` files from `vi` and `~` files from **`emacs`**). Make sure you use this before you bundle all your files together for submission.

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

```
make clean all
```

to have it completely clean the directory and build `thread_hash`.

5.1.2. I also strongly recommend that you **use some form of revision control on your source files**. Not only does this reduce the possibility of catastrophic file loss, but it is a LOT better than making `.BAK1`, `.BAK1a`, `.BAK17-2`, `.BAK4c` copies of your code.

5.1.2.1. Putting this into your `Makefile`, as described in the notes about `make` would make your life better.

5.1.3. You must compile your program using the following flags for `gcc`

5.1.3.1. Putting these flags into the `Makefile` with the `CFLAGS` variable will make your life better.

5.1.3.2. When compiled with the flags, the `gcc` compiler should emit no errors or warnings.

5.1.3.3. Any warning from the compiler is an automatic 20% deduction.

```
-Wall -Wextra -Wshadow -Wunreachable-code
-Wredundant-decls -Wmissing-declarations
-Wold-style-definition -Wmissing-prototypes
-Wdeclaration-after-statement -Wno-return-local-addr
-Wunsafe-loop-optimizations -Wuninitialized -Werror
-Wno-unused-parameter
```

5.1.3.4. I actually found a few extra `gcc` options to put in my Makefile. **These are optional**, but did help me create cleaner code. I know, you thought there COULD NOT BE MORE `gcc` options.

```
-Wno-string-compare -Wno-stringop-overflow
-Wno-stringop-overread -Wno-stringop-truncation
```

6. Output

6.1. Each thread should output a line when a password is cracked. **Not all passwords can be cracked.** The output format must match the following. The order of the output lines does NOT need to match and you should expect it to vary each time you run your code. For a cracked password, the first word in the output is just the word “cracked.” That is followed by the plaintext password that cracked the hashed password. The last item in the output is the entire hash string. For a password that failed to be cracked, print 3 asterisks, “failed to crack”, and the hashed password.

```
cracked  dansant  $3$$33af7f5af46f48a1d3f1127764f75225
```

```
cracked  Nadean  $2b$08$ViWZtBSlLU8GWbTh0JBcm..JF0snkJK.jF4O1ONqUipt3XXSDq..m
cracked  Plumiera  $3$$34c3e869ca9981c167f427708d18ecd9
*** failed to crack  $3$$92b4342d5733bd0a6d9a3b189103e2d9
cracked  hysterogen  $1$w5i5r1ng$uuJZ88.zSFOmluyzsNpx..
```

6.2. Each thread must output the total time it was running, the count of how many of each hash algorithm it processed, and the total number of passwords processed and the total number of passwords that failed to be cracked. **This must go to `stderr`**. Look at the output of mine to see how this should look. It is very wiiiiiiide; it just does not fit into this document.

6.3. When determining which algorithm was used to hash a plaintext password, you can use the following:

6.3.1. If the **initial** letter in the hash **is not a '$'**, then it is the descrypt (DES) algorithm.

YZ9yYUyKJaSt6

6.3.2. All other hash algorithms begin the hash string with a '`$`'.

6.3.3. If the **second** letter is a **3**, it is the NT algorithm.

$**3**$$67816a38595fc99f6964276d48161a67

6.3.4. If the **second** letter is a **1**, it is md5crypt (MD5).

$**1**$.ygLbqdg$clOuJNT/KnD8mTr/2Sm9T0

6.3.5. If the **second** letter is **5**, it is SHA-256.

$**5**$rounds=1395$YOwlQfwfp4B9v6R.$D43YTPLc629/U5KY9fGJsgwVaF.J5JjrOHK5EfvBVX5

6.3.6. If the **second** letter is **6**, it is SHA-512.

$**6**$rounds=1111$8iC3OVHXja6yqB0X$eMABiFcNWMN2iIqKh5qMRaHeouxNtvY7WgIwHIlR/DIMsrneu1b/qms4xAAj/4t4sxp2jZyd0ZPKNPv9w5J9M0

6.3.7. If the **second** letter is **y**, it is yescrypt.

$**y**$j7T$2D/kcBdk1f4chh7zeJlGxgSffzkc6yFygJt.KxMyaAoP25GhBZWs8s5JO97eUpNK$SitklBwL35tgFSSb750gMj75CzkJmvoy4p6kRyAsmp6

6.3.8. If the **next** letters are **gy**, it is gost-yescript.

$**gy**$j6T$o7/J1n.CI638iMtLEi/1mU.QIa8hnhH6iTR7vb50QaMJQvD/ABMEoOctNk7y/o5V$Po68cXkF.w41ebdewO.iGkaM0B0T/IOIGoSks7zr079

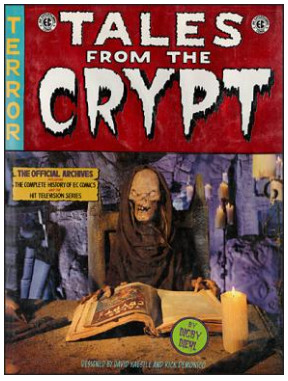
6.3.9. If the **next** letters are **2b**, it is bcrypt.

$**2b**$06$RHYay6xfytpkstHx8M5z6O6vfF/HYShYRd.WdlM2xgDJos7ydqVg6

6.4. When calculating the time for each thread and the total runtime for the process, use the same technique used in the matrix multiplication video assignment (`mm2.c`).

6.5. You will need a wiiiiiiiiide window to see the output without it wrapping.

7. It is important that you pair your password input file with the correct plaintext password file. The `passwords100.txt` must be paired with `hashes100.txt`. In a like fashion, the `passwords500.txt` must be paired with `hashes500.txt` file. You might be tempted to apply your code to a very large number of passwords. You will run out of patience before the file with 400k password is complete.
8. **Steps to success**. This code is pretty straight forward. My C code solution has about 580 lines (and a bunch of that is **`#ifdef`** experiments). This is only a suggested set of steps. It is not required and may not be complete.
   8.1. Your C code module must be named `thread_hash.c`. There is also a `thread_hash.h` file that has a couple helpful pieces of code. The way to define the enumerated type and names for the hash algorithms is terrific.
   8.2. Create a `Makfile`. Keep it current. Don't wait until the end to create the `Makfile` and find you have a bazillion warnings to track down in the last 3 minutes before the assignment is due.
      8.2.1. When you link your code (stage 4 of the 4 stages of compilation of a C program), you will need to link with `-lcrypt` (just as the `man` page for `crypt` says). Go ahead, put it in your `Makefile` on the link target.
      8.2.2. Put all the required targets into your `Makefile`.
      8.2.3. Check your `Makefile` into revision control.
   8.3. Parse the command line with `getopt()`. **Only after you have completely finished parsing the command line** can you begin to consider some default values for variables affecting the process. For instance, the name of the input file, the possible name of the output file, the name of the plaintext file.
      8.3.1. Check your code into revision control.
   8.4. You might start off not worrying about the multi-threaded portion of the assignment and just wrestle with the use of the `crypt_r()` function. Be sure to read section 2.1 about which variant of the `crypt()` functions to use. Remember, the output from your code is going to be in a different order than mine. That is okay and expected. Mine come out in a different order each time I run the program (with more than 1 thread).



      8.4.1. Check your code into revision control.
   8.5. When you've finally lassoed your `crypt()` function of choice, you'll need to start working toward multi-threading. You need to think carefully about what the shared resources in your code may be. If someone asks during class, I'll describe the approach I took on this. It is … … … interesting.
      8.5.1. Check your code into revision control.

8.6. Once you step into the multi-threaded portion, the output from your code will differ in the order of the passwords cracked, even with the same command line options. That is okay and even expected. Don't try and keep the output in the same order as the input file. Once you've cracked a hashed password, output it immediately.

8.7. You need to keep track of how many words were hashed by each thread. At the end of the program, output how many words were hashed by each thread. At example result will look as shown below. The exact number of words hashed by each thread will vary each time the program is executed; this is expected. **You must use dynamic load balancing with this assignment.** You should see each thread does a different number of hashes. **This output MUST got to `stderr`.**

8.7.1. Check your code into revision control.

8.8. Free any memory you allocated from the heap.

8.8.1. Check your code into revision control.

8.9. Clean up any compiler warnings.

8.9.1. Check your code into revision control.

8.10. Submit your code into Canvas.

8.11. Celebrate.

9. Some requirements

9.1. Your code must pass through `valgrind` without any memory leaks (lost memory) or any messages from `valgrind` about unsafe memory accesses. When running your multi-threaded configuration through `valgrind`, **use a small input file** (such as `passwords10.txt`). It will be very slow through `valgrind`.

9.1.1. Having memory leaks or other warnings messages from `valgrind` will be an automatic 20% deduction.

10. References for the `crypt()` function and other useful resources. To be honest, I find it disappointing that there is so little useful information out there about these longstanding and important functions.

Reference
The `man` page on Babbage. I spent **LOTS** of time in here. This is **`man 3 crypt`**. It is very okay.
The **`man 5 crypt`** man page on Babbage. **I really wish I'd looked at this sooner. This may be the best single source for you.**

The `crypt.h` file in `/usr/include` on Babbage. Yes, it is a little scary, but be intrepid.
https://man7.org/linux/man-pages/man3/crypt.3.html
https://ftp.gnu.org/old-gnu/Manuals/glibc-2.2.3/html_node/libc_650.html
https://www.oreilly.com/library/view/practical-unix-and/0596003234/ch04s03.html
https://www.mankier.com/5/crypt
https://www.openwall.com/
bcrypt: https://en.wikipedia.org/wiki/Bcrypt
yescrypt/gost-yescrypt https://unix.stackexchange.com/questions/690679/what-does-j9t-mean-in-yescrypt-from-etc-shadow/724514#724514 **I find it unbelievable that this really is the best I could find on these algorithms!**
crypt: https://en.wikipedia.org/wiki/Crypt_(C)#Key_derivation_functions_supported_by_crypt
The code at `uncrypt_example.c` in the Lab directory.
Simple examples: https://www.dcode.fr/crypt-hashing-function

11. A table of the runtime numbers for your code should look similar this (they will vary).

For input of 100 hashes (`hashes-100.txt`)	
Threads	Time (seconds)
1	39.218
2	19.715
3	15.046
4	12.666
5	11.226

For input of 250 hashes (`hashes-250.txt`)	
Threads	Time (seconds)
2	120.39

4	65.69
8	36.37
16	21.67
20	19.56

For input of 500 hashes (`hashes-500.txt`)	
Threads	Time (seconds)
2	431.07
4	229.74
8	121.28
16	73.28
20	63.70

For input of 1000 hashes (`hashes-1000.txt`)	
Threads	Time (seconds)
2	1539.68
4	801.95
8	445.30
16	262.16
20	228.23

12. When running your code, you need to be sure to carefully match the name of the input hashed password file with the name of the plaintext words file. The hashes are different for each file.

Passwords file	**Hashes file**
`passwords-10.txt`	`hashes-10.txt`
`passwords-50.txt`	`hashes-50.txt`
`passwords-100.txt`	`hashes-100.txt`
`passwords-250.txt`	`hashes-250.txt`
`passwords-500.txt`	`hashes-500.txt`
`passwords-1000.txt`	`hashes-1000.txt`

`passwords-1500.txt`	`hashes-1500.txt`
`passwords-2000.txt`	`hashes-2000.txt`
`passwords-5000.txt`	`hashes-5000.txt`

I have password and plaintext files up to 20,000. You really don't want to try to use files that large. I actually have a file with over 400k hashes from which the other files were sampled.

13. When you want to compare the output from your code to mine, you need to be able to decode how I generated the output file names when I collected data. I used bash scripts to generate the data and then run the program to collect the output.

File name	Meaning
`cracked-p100-t4.out`	This is an output file collected either from the default `stdout` or from the `-o <filename>` command line option. • The `p100` indicates that the data input files were `password-100.txt` and `hashes-100.txt`. • The `t4` indicates that it was collected running the program with 10 threads. This file has 100 lines in it, one for each password hash that was attempted to be cracked.
`cracked-p100-t4.err`	This is the accounting information that was written to `stderr` while the process was running. • The `p100` indicates that the data input files were `password-100.txt` and `hashes-100.txt`. • The `t4` indicates that it was collected running the program with 4 threads. This file contains the accounting information for each thread and a line for the total.