CSCI 2800 — Computer Architecture and Operating Systems (CAOS)
Lab 5 (document version 1.1)

- This lab is due by the end of your lab session on Wednesday, October 1, 2025
- This lab is to be completed **individually**; do **not** share your code with anyone else
- You **must** show your code and your solutions to a TA, CA, or mentor to receive credit for each checkpoint
- Labs are available no later than the Sunday before your lab session; plan to start each lab early and ask questions during office hours, in the Discussion Forum on Submitty, and during your lab session
- Note that all of your C code **must** successfully compile via `gcc` with no warning messages when the `-Wall` (i.e., warn all) compiler option is used; also use `-Werror`, which treats all warnings as critical errors

1. **Checkpoint 1:** Download the `lab05-checkpoint1.c` code from Submitty. This code creates a pipe (un-named), then calls `fork()` to share the pipe descriptors with a new child process. Next, the parent process asks the user to input positive integers, sending each integer to the child process via the pipe.

   Compile and run this code. Make sure you understand what each line of code is doing. How does each loop in the parent and child processes end? A sample execution is shown below.

```
bash$ ./a.out
PARENT: Enter positive integer (0 to exit): 1
PARENT: Wrote 1 to pipe (4 bytes)
PARENT: Enter positive integer (0 to exit): CHILD: Read 1 from pipe (4 bytes)
1
CHILD: Read 1 from pipe (4 bytes)
PARENT: Wrote 1 to pipe (4 bytes)
PARENT: Enter positive integer (0 to exit): 2
PARENT: Wrote 2 to pipe (4 bytes)
PARENT: Enter positive integer (0 to exit): CHILD: Read 2 from pipe (4 bytes)
3
CHILD: Read 3 from pipe (4 bytes)
PARENT: Wrote 3 to pipe (4 bytes)
PARENT: Enter positive integer (0 to exit): 5
CHILD: Read 5 from pipe (4 bytes)
PARENT: Wrote 5 to pipe (4 bytes)
PARENT: Enter positive integer (0 to exit): 8
CHILD: Read 8 from pipe (4 bytes)
PARENT: Wrote 8 to pipe (4 bytes)
PARENT: Enter positive integer (0 to exit): 0
```

The output interleaves and makes this program rather hard to use.

To fix this, refactor this code into two separate programs called `lab05-checkpoint1-reader.c` and `lab05-checkpoint1-writer.c` that accomplish the same inter-process communication using a named pipe (`fifo`) instead. Specify the name of the `fifo` on the command line for each. Also, fix any bugs that were in the original code.

Run these two executables in two separate terminals. The example from the previous page becomes the following:

```
bash$ ./lab05-fifo-writer.out /tmp/fifo | bash$ ./lab05-fifo-reader.out /tmp/fifo
Enter positive integer (0 to exit): 1   | Read 1 from pipe (4 bytes)
Wrote 1 to pipe (4 bytes)               | Read 1 from pipe (4 bytes)
Enter positive integer (0 to exit): 1   | Read 2 from pipe (4 bytes)
Wrote 1 to pipe (4 bytes)               | Read 3 from pipe (4 bytes)
Enter positive integer (0 to exit): 2   | Read 5 from pipe (4 bytes)
Wrote 2 to pipe (4 bytes)               | Read 8 from pipe (4 bytes)
Enter positive integer (0 to exit): 3   | bash$
Wrote 3 to pipe (4 bytes)               |
Enter positive integer (0 to exit): 5   |
Wrote 5 to pipe (4 bytes)               |
Enter positive integer (0 to exit): 8   |
Wrote 8 to pipe (4 bytes)               |
Enter positive integer (0 to exit): 0   |
bash$                                   |
```

Do not square bracket notation; instead, use pointer arithmetic.

In your new programs, determine how many bytes are statically allocated. How many bytes are dynamically allocated?

Are you sure you have addressed all bugs in your code?

2. **Checkpoint 2:** For this checkpoint, we will extend Checkpoint 1 above, so be sure to complete Checkpoint 1 before starting this one.

   For this checkpoint, first notice how similar the use of a pipe or `fifo` is to the use of a file on disk. Once we have the necessary file descriptors, `read()` and `write()` calls are essentially the same in terms of syntax and the code we write. The underlying mechanism is certainly very different, though, since pipes are entirely in memory whereas files are on disk.

   Given the above, modify your "writer" code to write the values entered by the user to a file on disk if run with a `-f` (file) flag. Specifically, the usage is as follows:

   ```
   USAGE: ./lab05-fifo-writer.out {<fifo-name> | -f <filename>}
   ```

   In other words, you can run the writer as you did in Checkpoint 1 or you can use `-f` to specify an output file name.

   Below is an example program execution using this new feature.

   ```
   bash$ ./lab05-fifo-writer.out -f output.txt
   Enter positive integer (0 to exit): 1
   Wrote 1 to "output.txt" (4 bytes)
   Enter positive integer (0 to exit): 1
   Wrote 1 to "output.txt" (4 bytes)
   Enter positive integer (0 to exit): 2
   Wrote 2 to "output.txt" (4 bytes)
   Enter positive integer (0 to exit): 3
   Wrote 3 to "output.txt" (4 bytes)
   Enter positive integer (0 to exit): 5
   Wrote 5 to "output.txt" (4 bytes)
   Enter positive integer (0 to exit): 8
   Wrote 8 to "output.txt" (4 bytes)
   Enter positive integer (0 to exit): 0
   bash$ hexdump -C output.txt
   00000000  01 00 00 00 01 00 00 00  02 00 00 00 03 00 00 00  |................|
   00000010  05 00 00 00 08 00 00 00                           |........|
   00000018
   bash$
   ```

   Remember that your `hexdump` output might be different based on the endianness of the platform you are running on.

3. **Checkpoint 3: (v1.1)** In support of Homework 2, your task in this third checkpoint is to complete the code given in `lab05-checkpoint3.c`. Download and run this code; though it compiles with no warnings or errors and runs, it does not do anything useful yet.

   Specifically, implement the required functions described below. Be sure to use the given function prototypes exactly as shown. Comments in the given source file should clarify what each function should do.

   Also be sure that there are no memory leaks or buffer overflows or use of initialized memory.

   Implement the following functions, most likely in the order shown:

   - `void upcaseword( char * word, int wordlen );`
   - `int trieindex( char letter );`
   - `void addword( trie_t * trie, const char * word, int wordlen );`
   - `void freetrie( trie_t * trie );`
   - `int isvalidword( const trie_t * trie, const char * word, int wordlen );`

   [**OPTIONAL**] Also implement the following function:

   - `char * suggestword( const trie_t * trie, const char * word, int wordlen );`

   Running your code should produce the following interactive spell-checking program:

```
bash$ ./a.out
Trie contains valid words:
AA
AAH
AAHED
AAHING
AAHS
AAL
AALII
AALIIS
AALS
AARDVARK
CAOS
Enter word to look up (CTRL-D to exit): CAOS
==> CAOS is valid
Enter word to look up (CTRL-D to exit): CAORS
==> CAORS is not valid (CAOS)
Enter word to look up (CTRL-D to exit): canos
==> CANOS is not valid
Enter word to look up (CTRL-D to exit): aaLL
==> AALL is not valid (AALS)
Enter word to look up (CTRL-D to exit): AARDVARKY
==> AARDVARKY is not valid (AARDVARK)
Enter word to look up (CTRL-D to exit): AARDVA
==> AARDVA is not valid (AAH)
Enter word to look up (CTRL-D to exit): <CTRL-D>
bash$
```