CS415—Systems Programming

Spring 2014

Programming Assignment 3

System Structure

You are required to create a system of four communicating concurrent processes which does some relatively lightweight manipulation of text files. The `driver` is the process responsible for creating and interconnecting the other three processes—it's the ultimate ancestor in this system. We define its role after specifying the function of the other three processes.

The Scanner

The `scanner` process accepts ASCII data on its standard input and detects *words*. A word is defined to be any consecutive sequence of non-whitespace characters, where a whitespace character is a SPACE, a TAB, or a NEWLINE. The scanner then transforms words in accordance with two rules:

- all punctuation characters (as defined by the `ispunct()` library function) are stripped out of the word, and
- all characters in the word are mapped to lower case by the `tolower()` function.

To give you a specific benchmark against which you can test your conversion. The binary executable file

`~kearns/public/415/p3/words`

takes a single command-line argument, the name of a file. It outputs the words found in the named file, one per line. It also shows the words after the two conversions defined above.

When the scanner detects a word which contains an even number of characters, it sends that word down a pipe to a process called `even`. It then terminates that word by sending a space down the pipe to `even`. It performs analogously when a word containing an odd number of characters is detected—the only difference is that a process called `odd` gets the space-terminated word down its pipe.

On detecting EOF on the standard input, `scanner` will close its output pipes and enter an infinite loop in which it sleeps for 1 second and then prints a single asterisk on its standard output. When it receives a `SIGTERM` signal from `even`, it enters a phase in which it expects to receive the output of `even` and `odd` (see below) and to display that information to the user by writing onto its standard output. The final output might appear as:

```
**********************************
Words with even letters:
   farp     2
   twit     6
Words with odd letters:
   fubar    1
   zip      8
```

Even and Odd

The processes even and odd expect to have standard input which appears as

word word ... word word

Each process also expects a single command line argument which should be the pid of the `scanner` process. If this command line argument is a 0, then `even` and `odd` are assumed to be running in standalone mode in which there is no `scanner` process.

Each constructs an internal table of word and count pairs, updating on incoming words. When `even` detects EOF on its standard input, it will wait for 10 seconds, signal the `scanner` as specified above, and will then write (to its standard output) a sequence of word and count pairs. The list should be in lexicographic word order (alphabetical order in the sense of the `strcmp()` library function). Note that if the pid of the `scanner` is given as 0, then no signal should be sent.

The process called `odd` will do exactly the same as `even`, *but it will never issue a signal.*

In the output generated by `even` and `odd`, the words will be space-terminated character strings, and the counts will be space-terminated strings which represent the associated integer count. The `scanner` receives and outputs those pairs in reasonably formatted human-readable form (see the example above). It is imperative that the standard output of `even` and `odd` be hooked up to the appropriate pipes which feed the `scanner` during its output phase.

If the environment variable `EODELAY` is set to a positive integer value, `even` and `odd` should delay that number of seconds after receiving a word. For example, if your system is invoked as

```
% ( export EODELAY=3; driver bigwordfile )
```

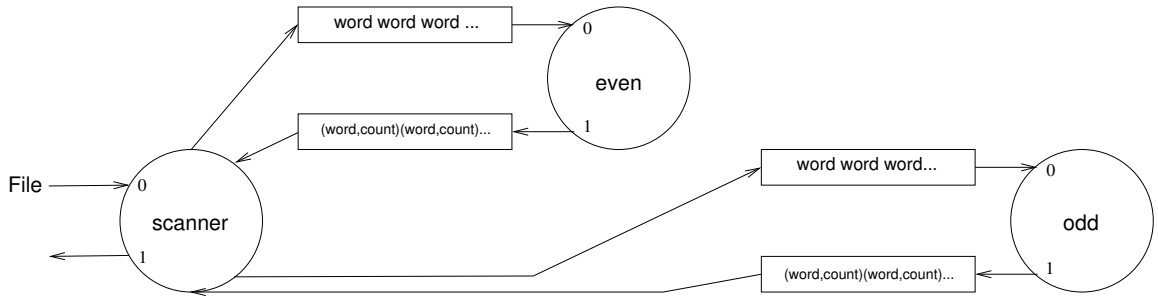
then `even` and `odd` will delay 3 seconds after receiving a word. If `EODELAY` is not set, or if its value is not a positive integer, there should be no delay.

You may not make assumptions about the size of the input file. This means that you should dynamically "grow" your data structures as input is processed.

The Driver

The `driver` is responsible for taking a single command line argument which specifies the file to be processed by this system. If that file can be opened, it must then create the three sub-processes: `scanner`, `even`, and `odd`. It must arrange for the standard input of `scanner` to come from the designated file. The standard output of `scanner` should be inherited from `driver`. The `driver`

must also arrange for the pipes which interconnect the processes in keeping with the figure below. The `driver` is not shown in the figure because it has no real role in processing the data; it must, however set up all components of the system.



Notes

- Be sure to use low level input/output operations when having the processes interact via the pipe(s). Specifically, you should be invoking the kernel calls `read()` and `write()` instead of things like `putc()` and `fscanf()` in order to do pipe i/o. You may communicate with the user (i.e, stdin and stdout of `scanner`) using whatever i/o constructs you choose (but you would be wise to use the standard i/o library). The `scanf()` function does a particularly nice job of picking out the words in a text file.
- EOF is never detected on a pipe unless the writer closes its write descriptor for that pipe. Until the input side is closed, a read will hang indefinitely.
- Be sure to terminate this multi-task system gracefully. You should call `exit()` explicitly in the three children of `driver`, and `driver` should explicitly `wait()` for their termination.
- You must submit three source files. The source for driver should be called `driver.c`; the scanner, `scanner.c`. One source program, `evenodd.c` should be written to be the even and odd children. Have the process use the value of `argv[0]` to adapt it's personality to be `even` or `odd`: the only real difference in behavior is that `even` will signal `scanner` while `odd` will not. In the directory where the binaries of the system will reside, there should be binaries for `driver`, `scanner`, and `evenodd`. There should be a link from `evenodd` to `even`, and another link from `evenodd` to `odd` in order to provide the illusion of separate binaries.
- We will test even/odd in standalone mode by invoking them as

  ```
  ./even 0 < sometestfile
  ```

  where `sometestfile` contains a list of words separated by spaces. In this case, `even` should output the word and count pairs on its standard output.

Due

See the project web page for the submission deadline. Use the following steps to prepare your submission file:

1. In the directory where your source resides, do

   ```
   tar zcvf system.tgz driver.c scanner.c evenodd.c
   ```

   This produces a gzipped tarball (`system.tgz`) consisting of your three source files.

2. In that same directory, execute

   ```
   gpg -r CSCI315 --sign -e system.tgz
   ```

   This signs and encrypts your gzipped tar archive so the TA can retrieve the submission.

3. Finally,

   ```
   mv system.tgz.gpg ~; chmod 644 ~/system.tgz.gpg
   ```

   This moves your submission to your home directory and protects it so that it can be copied by the TA.