CS 511: Assignment #2
Using Semaphores for Buffer Synchronization

Due Sunday, October 5 at 11pm.

1 Assignment

The purpose of this assignment is to give you some experience writing a simple multi-threaded program whose threads are synchronized by use of a semaphore.

In this assignment one thread will read lines of text from a file then write the lines into a buffer in memory. A second thread running simultaneously will copy the lines from the buffer then write them to an output file. When the program finishes, the output file should be equal to the input file. The concurrency problem is to synchronize the threads's access to the buffer.

1.1 Part 1

As a first step, write a single-threaded program named "rw" that repeatedly reads a line of text from an input file, writes the line into a buffer, then writes the line from the buffer to the output file. The program should take two command line arguments, the first being the name of the input file, the second being the name of the output file.

The purpose of this part of the assignment is to have you write and debug the file-handling code before tackling the concurrency aspects. Library calls that might be useful include `fopen(3)`, `getline(3)`, and `fwrite(3)`.

1.2 Part 2

Write a two-threaded program named "transfer1" in which one thread (the "fill thread") reads lines of text from the input file and writes them into a buffer. A second thread running simultaneously (the "drain thread") copies the lines from the buffer then writes them to an output file. You must synchronize the two threads's

access to the buffer using a single semaphore. The buffer should be initialized to all zeroes.

Once the fill thread has detected end of the input file it should write "QUIT" into the buffer. When the drain thread reads QUIT it should terminate. The fill thread should wait for the drain thread to terminate and, after the drain thread has terminated, the fill thread should deallocate the buffer and the semaphore then terminate the program.

The fill thread should print a message whenever it writes a line into the buffer. The drain thread should print a message whenever it reads a line from the buffer.

The fill thread should write each line into the buffer following the previous line. You need not worry about the buffer overflowing – allocate a buffer that is big enough to hold all the lines of the input file, including the terminating null character for each string. The drain thread must be able to examine the buffer and determine whether a new line has been written into it since the last time the drain thread executed. To make this determination, the drain thread should check whether the next byte it plans to read still has the initial zero value or something different; if the byte is no longer zero that means a string has been written into the buffer starting at that position.

The drain and fill threads will run at unpredictable rates, based on the operating system's scheduling decisions. Your program should operate properly regardless how often or when either thread executes. So that we can explore different thread execution rates, the fill thread should call `usleep` at the end of each loop, after it has left its critical section. This call should be the only sleep call in your program.[1]

transfer1 should take three command line arguments, the first being the name of the input file, the second being the name of the output file, the third being the sleep time.

Assuming this is the input file `input.txt`:

```
line 1
slightly longer
line 3
the longest line
line 5
```

[1]Note that `usleep` is NOT being used to ensure correctness, but to perform timing experiments. To be correct, a concurrent program must never depend on timing.

Here is the output of an execution of transfer1 with a sleep time of 0:

```
$ ./transfer1 input.txt output.txt 0
fill thread: wrote [line 1
] into buffer
fill thread: wrote [slightly longer
] into buffer
fill thread: wrote [line 3
] into buffer
fill thread: wrote [the longest line
] into buffer
fill thread: wrote [line 5
] into buffer
fill thread: wrote QUIT into buffer
drain thread: read [line 1
] from buffer
drain thread: read [slightly longer
] from buffer
drain thread: read [line 3
] from buffer
drain thread: read [the longest line
] from buffer
drain thread: read [line 5
] from buffer
drain thread: read [QUIT] from buffer
```

Notice that the fill thread ran first and ran long enough so that it had time to put all its lines into the buffer. Only then was the drain thread scheduled; it removed all lines. There was no concurrency in this execution. Each thread runs for only a short while and completes its task during that time.

Here is the output of an execution of transfer1 with a sleep time of 1000:

```
$ ./transfer1 input.txt output.txt 1000
fill thread: wrote [line 1
] into buffer
drain thread: read [line 1
] from buffer

(thousands of this line deleted:
```

```
drain thread: no new string in buffer)

fill thread: wrote [slightly longer
] into buffer
drain thread: read [slightly longer
] from buffer

(thousands of this line deleted:
drain thread: no new string in buffer)

fill thread: wrote [line 3
] into buffer
drain thread: read [line 3
] from buffer

(thousands of this line deleted:
drain thread: no new string in buffer)

fill thread: wrote [the longest line
] into buffer
drain thread: read [the longest line
] from buffer

(thousands of this line deleted:
drain thread: no new string in buffer)

fill thread: wrote [line 5
] into buffer
drain thread: read [line 5
] from buffer

(thousands of this line deleted:
drain thread: no new string in buffer)

fill thread: wrote QUIT into buffer
drain thread: read [QUIT] from buffer
```

The fill thread sleeping outside its critical section for a millisecond allowed the drain thread to be scheduled many times during that millisecond. However, each time the drain thread ran while the fill thread was sleeping there was no new line

in the buffer. This is extremely wasteful: only 6 lines will ever be written into the buffer; the drain thread ran needlessly thousands of times.

1.3 Part 3

Write a two-threaded program named "transfer2" that is the same as transfer1 except that it uses two semaphores for buffer synchronization as explained in lecture.

The coordination provided by the two semaphores eliminates all needless thread scheduling. Here is the output of an execution of transfer2 with a sleep time of 1000:

```
$ ./transfer2 input.txt output.txt 1000
fill thread: wrote [line 1
] into buffer
drain thread: read [line 1
] from buffer
fill thread: wrote [slightly longer
] into buffer
drain thread: read [slightly longer
] from buffer
fill thread: wrote [line 3
] into buffer
drain thread: read [line 3
] from buffer
fill thread: wrote [the longest line
] into buffer
drain thread: read [the longest line
] from buffer
fill thread: wrote [line 5
] into buffer
drain thread: read [line 5
] from buffer
fill thread: wrote QUIT into buffer
drain thread: read [QUIT] from buffer
```

There are no unnecessary thread schedulings. Each thread waits for the other thread's "post" operation on the semaphore.

2 Submission Instructions

You must work alone on this assignment. Submit `rw.c`, `transfer1.c`, and `transfer2.c` via Moodle by the indicated date and time. Moodle will shut you out at the deadline. Since your clock and Moodle's clock may differ by a few minutes, be sure to submit at least several minutes before the deadline.

Please note: *late work is not accepted; late submissions will not be graded and will receive a score of zero.* Also, points will be deducted if files have names different from `rw.c`, `transfer1.c`, and `transfer2.c`.

Your code should compile on `linux-lab.cs.stevens.edu` with no errors or warnings using a compilation command such as:

```
gcc -Wall -pedantic-errors transfer1.c -lpthread -o transfer1
```

Be sure to start early and contact the instructors if you have questions or encounter problems. Instructor office hours are posted in Moodle. Do not assume that instructors will be able to reply to emails over the final weekend of October 4-5, especially Sunday night. You should manage your time so that you finish by Friday, Oct 3. Do NOT write the entire program then start debugging. Instead, write a part, debug it, then write the next part. Remove extra print statements from a section of code only once you're confident that the section is working.