CS 511: Assignment #2
Using Semaphores for Buffer Synchronization

Due before class on Wednesday, October 14

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 one or more semaphore(s).

In this assignment one thread will read lines of text from an input 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' access to the buffer.

1.1 Part 0

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 an 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 1

Write a program named "transfer1" that creates two additional threads that run simultaneously. One additional thread (the "fill thread") reads lines of text from the input file and writes them into a buffer. The second additional thread (the

"drain thread") copies the lines from the buffer then writes them to the output file. You must synchronize the two threads' access to the buffer using a single semaphore acting as a lock.

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

The buffer should be big enough to store the longest line in the input file, but not big enough to store ALL the input lines at once; in other words, the buffer might fill up if the fill thread runs more often than the drain thread.

The fill thread should print a message whenever it successfully writes a line into the buffer. Also, the fill thread should print a message whenever it detects that it cannot write into the buffer because there is not sufficient empty space in the buffer. In cases where there is not enough space to write the next line, the fill thread has no alternative but to loop, continually checking the buffer's available space, until sufficient space becomes available.

The drain thread should print a message whenever it successfully reads a line from the buffer. Also, the drain thread should print a message whenever it detects that it cannot read from the buffer because the buffer is empty. In cases where the the buffer is empty, the drain thread has no alternative but to loop, continually checking the buffer's occupancy, until a line to read becomes available.

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 each thread executes. So that we can explore different thread execution rates, each thread should call `usleep` at the beginning of each loop, outside its critical section. This call should be the only sleep call in your program.[1]

transfer1 should take four command line arguments: the name of the input file; the name of the output file; the sleep time for the fill thread; and the sleep time for the drain thread.

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

```
line 1
slightly longer
```

[1]Note that `usleep` is NOT being used to ensure correctness, but to experiment with different timings. To be correct, a concurrent program must never depend on timing "tricks" like sleeping.

```
line 3
the longest line
line 5
```

then here is the output of an execution of transfer1 with both sleep times set to zero:

```
$ ./transfer1 input.txt output.txt 0 0
buffer size = 32
drain thread: no new string in buffer             <-- several such lines
fill thread: wrote [line 1
] into buffer (nwritten=8)
fill thread: wrote [slightly longer
] into buffer (nwritten=17)
fill thread: wrote [line 3
] into buffer (nwritten=8)
fill thread: could not write [the longest line
] -- not enough space (7)                         <-- hundreds such lines
drain thread: read [line 1
] from buffer (nread=8)
drain thread: read [slightly longer
] from buffer (nread=17)
drain thread: read [line 3
] from buffer (nread=8)
drain thread: no new string in buffer             <-- hundreds such lines
drain thread: read [the longest line
] from buffer (nread=18)
drain thread: no new string in buffer             <-- many such lines
fill thread: wrote [the longest line
] into buffer (nwritten=18)
fill thread: wrote [line 5
] into buffer (nwritten=8)
drain thread: no new string in buffer
drain thread: read [line 5
] from buffer (nread=8)
drain thread: read [QUIT] from buffer (nread=5)
fill thread: wrote [QUIT] into buffer (nwritten=5)
```

Clearly the "spin wait" approach is very inefficient. Setting sleep times to non-zero values (say, 10 and 0 or 0 and 10) produce similarly inefficient executions,

with the needless thread schedulings biased one way or the other based on the sleep time values.

1.3 Part 2

Write a program named "transfer2" that is the same as transfer1 except that it uses two semaphores for buffer synchronization and a third semaphore for mutex as suggested in lecture. Perform a "P(spaces)" operation each time a line is available to be written into the buffer; perform a "V(spaces)" operation each time a line is removed from the buffer. Operate similarly on the "items" semaphore.

The coordination provided by the two "signaling" semaphores eliminates many of the needless thread schedulings. You will probably see NO needless schedulings when the fill thread sleeps longer than the drain thread. When the drain thread sleeps longer or when neither thread sleeps you will probably see some needless schedulings, always of the fill thread. Why do these needless schedulings happen? Can you think of a way to eliminate the loops in both threads and eliminate all needless thread schedulings?

1.4 Implementation Notes

1. Linux has two semaphore interfaces, the old ("System V" or "XSI") interface and the newer POSIX interface. Use the POSIX interface: `sem_overview(7), sem_wait(3), sem_post(3), sem_init(3), sem_destroy(3).`

2. The main program should open both files before it creates either thread, to eliminate the possibility of file open affecting thread timing.

3. Take care to make critical sections no longer than they need be.

4. Minimize the use of global variables.

5. Use "-lpthread" to link with the pthreads library.

6. Code for a circular buffer (`cbuf.h, cbuf.c`) has been posted along with the assignment. You are welcome to use all, some, or none of this code.

2 Submission Instructions

You must work alone on this assignment. Submit `rw.c`, `transfer1.c`, and `transfer2.c` via Canvas by the indicated date and time. Late work is not accepted and Canvas will shut you out at the deadline. Since your clock and Canvas's clock may differ by a few minutes, be sure to submit at least several minutes before the deadline.

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

```
gcc -Wall -pedantic-errors primes.c -o primes
```

If you choose to use a different development environment, note that it might take non-trivial time to port your code from your environment to `linux-lab`, even if your code is fully working in your environment; be sure to budget for that time.