CSc 139 Spring 2015
Department of Computer Science, CSUS
4/1/2015

Programming Assignment 4

- **The due day: April 16.**
- **No credit for a program that does not compile or does not run.**

This assignment requires you to write a multi-threaded C program for the same problem as in the Assignment 3: the *bounded-buffer producer/consumer* approach for file copying, but using **mutex** locks and **condition** variables (in POSIX thread library) instead of semaphores. Make sure that your submitted program compiles and runs in Athena.

1. Your program should be run with two file names provided at the command line: **infile** and **outfile**. Otherwise, your program prompts the user with the message: "Correct Usage: pc infile outfile" and then terminates.
2. In your program, the main thread checks for correct number of input, verifies and opens the infile, and creates the (empty) outfile. The main thread then spawns a producer thread and a consumer thread, and waits for both the producer and the consumer to finish before it terminates. The producer and the consumer share **a buffer of 9 slots with each slot having a size of 18 bytes**. The producer reads a string of the buffer slot size from the infile and places it into the next available buffer slot. The consumer takes the next available string from a buffer slot and writes it into the outfile. **The outfile is again a verbatim copy of the infile**.
3. The buffer can only be accessed in a mutually exclusive fashion, which is enforced through the use of a *mutex* lock **`buf_lock`** and **pthread_mutex_lock** and **pthread_mutex_unlock** operations.
4. When the buffer is full, the producer must wait until a buffer slot becomes available before it can place a string into it. When the buffer is empty, the consumer must wait for an item (a string) to be available. These synchronization conditions between the producer and the consumer are facilitated through two *condition* variables **`empty_slot`** and **`avail_item`,** and the **pthread_cond_wait** and **pthread_cond_signal** operations.
5. A *condition* variable is always used in conjunction with *a mutex* lock and *a condition*, and should be declared global. A thread uses a condition variable to (a) either notify other cooperating threads (with access to the same condition variable) that a condition has been met (**pthread_cond_signal**); (b) or block and wait for some condition to be met (**pthread_cond_wait**). When a thread blocks on a condition variable, *it automatically releases the associated mutex* lock, allowing other threads to gain the *mutex* lock. In the context of the producer/consumer approach, the producer can be synchronized with the consumer in the following manner when the buffer is full:

Producer
```
pthread_mutex_lock(&buf_lock);
while (buffer is full) {
  pthread_cond_wait(&empty_slot, &buf_lock);
}
   {fill a buffer slot, update variables, ……}
pthread_cond_signal(&avail_item);
pthread_mutex_unlock(&buf_lock);
```

Consumer
```
pthread_mutex_lock(&buf_lock);
while (buffer is empty) {
 pthread_cond_wait(&avail_item, &buf_lock);
}
   {empty a buffer slot, update variables, ……}
 pthread_cond_signal(&empty_slot);
 pthread_mutex_unlock(&buf_lock);
```

When the producer has the lock to the buffer first, and then realizes that the buffer is full, it uses a wait operation to block itself on the condition variable **`empty_slot`** and releases **`buf_lock`**. Later when the consumer gains **`buf_lock`** and empties a slot, it uses a signal operation on the condition variable **`empty_slot`** to notify the producer that there is an empty slot so that the producer can continue. After being awakened, the producer reacquires **`buf_lock`** and continues from the line immediately after where it was blocked.

Alternatively, the consumer can be synchronized with the producer in a similar manner when the buffer is empty (it should wait on the condition variable **`avail_item`** instead). *A wait operation will unconditionally block the calling thread, but a signal operation on a condition variable on which there is not any waiting thread is not remembered.*

6. A condition variable can be declared and statically initialized as follows:

```
pthread_cond_t        empty_slot = PTHREAD_COND_INITIALIZER;
```

Condition variable can be destroyed using **pthread_cond_destroy**(&**empty_slot**). The wait operation takes both a condition variable and a *mutex* lock, while the signal operation just takes a condition variable as its argument.

```
pthread_cond_wait(&empty_slot, &buf_lock);
pthread_cond_signal(&empty_slot);
```

7. A mutex lock can be declared and initialized as follows:

```
pthread_mutex_t       buf_lock = PTHREAD_MUTEX_INITIALIZER
```

The following calls can be used with a *mutex* lock:

```
pthread_mutex_lock(&buf_lock);
pthread_mutex_unlock(&buf_lock);
```

8. The data type for the buffer are given as follows:

```
#define   SLOTSIZE   18
#define   SLOTCNT    9
char      buffer[SLOTCNT][SLOTSIZE];
```

Like in Assignment 3, you need to have some additional variables to handle things like (a) the next available slot for the producer; (b) the next available item for the consumer; (c) number of items available in the buffer; (d) number of bytes in a slot; and (e) a flag to indicate when producing/consuming process ends.

9. Test your program with your own data. Then make sure that your program works for the following as the content of the infile:

```
Instead of using semaphores to synchronize concurrent activities,
This assignment offers an opportunity for you to use the mutex locks
and condition variables to coordinate the producer and the consumer
activities in the context of the file copying application.
In contrast to the semaphore-based solution where the order
of two semaphore wait operations is of pivotal importance to avoid
potential deadlocks, the mutex-lock-condition-variable approach allows
a thread to suspend on a particular condition and release the mutex
lock at the same time via the pthread_cond_wait operation, therefore
avoiding potential deadlocks. This exercise helps reinforce the point
that there are different synchronization approaches to a given problem.
```

10. Submission Requirements. Your program must include adequate commenting (**points deduction for programs with inadequate comments**). Compile your program with the following:

```
gcc   prog4.c   -o   prog4   -lpthread
```

Submit your **prog4.c** file to “Program4 Submission” in SacCT.