Project 3: Candy Kids

1.   Overview

In the notes we saw a solution to the producer-consumer problem with bounded buffers. In this assignment you will use that solution to manage access to a bounded buffer storing candy. One group of threads will model candy factories which generate candy one at a time and insert the candy into the bounded buffer. Another group of threads will model kids which eat candy one a time from the bounded buffer. Your program, called `candykids`, will accept three arguments:

        ./candykids <#factories> <#kids> <#seconds>

Example: `./candykids 3 1 10`

> **# Factories:**   Number of candy-factory threads to spawn.
>
> **# Kids:**   Number of kid threads to spawn.
>
> **# Seconds:**   Number of seconds to allow the factory threads to run for.

2.   Produce/Consumer Operation

Main

Your `main()` function will start and control the application. Its steps are as follows:

```
main() {
        // 1. Extract arguments
        // 2. Initialize modules
        // 3. Launch candy-factory threads
        // 4. Launch kid threads
        // 5. Wait for requested time
        // 6. Stop candy-factory threads
        // 7. Wait until no more candy
        // 8. Stop kid threads
        // 9. Print statistics
        // 10. Cleanup any allocated memory
}
```

1. **Extract Arguments**
    Process the arguments passed on the command line. All arguments must be greater than 0. If any argument is 0 or less, display an error and exit the program.

2. **Initialize Modules**
    Do any module initialization. You will have at least two modules: bounded buffer, and statistics. If no initialization is required by your implementation, you may skip this step.

3. **Launch factory threads**
    Spawn the requested number of candy-factory threads. To each thread, pass it its factory number: 0 to (number of factories - 1).
    - Hint: Store the thread IDs in an array because you'll need to join on them later.
    - Hint: Don't pass each thread a reference to the same variable because as you change the variable's value for the next thread, there's no guaranty the previous thread will have read the previous value yet. You can use an array to have a different variable for each thread.

4. **Launch kid threads**
    Spawn the requested number of kid threads.

5. **Wait for requested time**
   In a loop, call `sleep(1)`. Loop as many times as the "# Seconds" command line argument. Print the number of seconds running each time, such as "Time 3s" after the 3rd sleep. This shows time ticking away as your program executes.

6. **Stop factory threads**
   Indicate to the factory threads that they are to finish, and then call join for each factory thread. See section on candy-factory threads (below) for more.

7. **Wait until no more candy**
   While there is still candy in the bounded buffer (check by calling a method in your bounded buffer module), print "`Waiting for all candy to be consumed`" and sleep for 1 second.

8. **Stop kid threads**
   For each kid thread, cancel the thread and then join the thread. For example, if a thread ID is stored in `daThreadID`, you would run:
   > **pthread_cancel**(daThreadId);
   > **pthread_join**(daThreadId, NULL);

9. **Print statistics**
   Call the statistics module to display the statistics. See statistics section below.

10. **Cleanup any allocated memory**
    Free any dynamically allocated memory. You may need to call cleanup functions in your statistics and bounded buffer modules if they need to free any memory.

File Structure

You must split your code up into modules by using multiple .h and .c files.

Suggestion is to have the following files:

- `candykids.c`: Main application holding factory thread, kid thread, and `main()` function. Plus some other helper functions, and some `#defined` constants.

- `bbuff.h/.c`: Bounded buffer module (see below).

- `stats.h/stats.c`: Statistics module (see later section).

- `Makefile`: Must compile all the .c files and link together the .o files.

Coding Suggestions

- The factory creates candy and the kids consume it. The candy will be stored in a bounded buffer. To do this, you need a data type to represent the candy. The following `struct` is convenient:

  ```
  typedef struct {
          int factory_number;
          double time_stamp_in_ms;
  } candy_t;
  ```

- `factory_number` tracks which factory thread produced the candy item.

- `time_stamp_in_ms` tracks when the item was created. You can get the current number of milliseconds using the following function. **This code must be linked with the -lrt flag. Add it to CFLAGS in your Makefile[1].**

---

[1]If `-lrt` gives you build problems, try placing the option after the $(OBJS) (object files).

```
double current_time_in_ms(void)
{
        struct timespec now;
        clock_gettime(CLOCK_REALTIME, &now);
        return now.tv_sec * 1000.0 + now.tv_nsec/1000000.0;
}
```

Bounded Buffer

Create a bounded buffer module which encapsulates access to the bounded buffer. Your bounded buffer must be implemented using the technique shown on slide 28 of the 05-Synchronization notes.

Suggested public interface (the complete `bbuff.h` file) is shown below. Note that it operates on `void*` pointers instead of directly with `candy_t` structures. This is done so that the buffer need not know anything about the type of information it is storing. In this case, make the buffer array (declared in the .c file) of type `void*` such as: `void* da_data[DA_SIZE];`

```
#ifndef BBUFF_H
#define BBUFF_H

#define BUFFER_SIZE 10

void bbuff_init(void);
void bbuff_blocking_insert(void* item);
void* bbuff_blocking_extract(void);
_Bool bbuff_is_empty(void);

#endif
```

For example, an item can be inserted into the buffer with the following code which will dynamically allocate one candy element (pointer stored in `candy`), set the fields of the candy, and then call the bounded buffer function to insert it into the bounded buffer.

```
void foo() {
        candy_t *candy = malloc(...);
        candy->factory_number = ...;
        candy->time_stamp_in_ms = ...;
        bbuff_blocking_insert(candy);
}
```

The `bbuff_init()` function is used to initialize the bounded buffer module if anything needs to be initialized. Think of it *like* the constructor for your module: if this were object-oriented C++ then the constructor would do any needed initialization. But, in C there are no constructors, so `init()` functions are often used. If you do not need to initialize anything in your module, you can omit the `bbuff_init()` function entirely.

Candy-Factory Thread
Each candy-factory thread should:

1. Loop until `main()` signals to exit (see below)

    1. Pick a number of seconds which it will (later) wait. Number randomly selected between 0 and 3 inclusive.

    2. Print a message such as: "`\tFactory 0 ships candy & waits 2s`"

    3. Dynamically allocate a new candy item and populate its fields.

    4. Add the candy item to the bounded buffer.

    5. Sleep for number of seconds identified in #1.

2. When the thread finishes, print the message such as the following (for thread 0):
    "`Candy-factory 0 done`"

Thread Signaling
The thread will end when signaled to do so by `main()`. This is not using Linux signals but rather just a `_Bool` global variable which is set to true when it's time to end the thread. For example, name it `stop_thread` and have it be `false` to start. Then have `main()`, when it wants to end the thread, set this variable to `true`. Have your thread continually check this `_Bool` variable (often called a flag) to see if it should end. Here is some pseudo-code that may help:

```
_Bool stop_thread = false;
void* dathread_function(void* arg) {
        while (!stop_thread) {
                // Do the work of the thread
        }
        printf("Done!");
}

void main() {
        // Spawn thread
        pthread_id daThreadId;
        pthread_create(&daThreadId, ...)

        // Wait
        sleep(...)

        // Tell thread to stop itself, and then wait until it's done.
        stop_thread = true;
        pthread_join(daThreadID, NULL)
}
```

Kid Thread

Each kid thread should do the following:

1. Loop forever

    1. Extract a candy item from the bounded buffer.

        - This will block until there is a candy item to extract.

    2. Process the item. Initially you may just want to `printf()` it to the screen; in the next section you must add a statistics module that will track what candies have been eaten.

    3. Sleep for either 0 or 1 seconds (randomly selected).

The kid threads are canceled from `main()` using `pthread_cancel()`. When this occurs, it is likely that the kid thread will be waiting on the semaphore in the bounded buffer. This should not cause problems.

3. Statistics

Create a statistics module tracking:

1. Count the number of candies each factory creates. Called from the candy-factory thread.

2. Count the number of candies that were consumed from each factory.

3. For each factory, the min, max, and average delays for how long it took from the moment the candy was produced (dynamically allocated) until consumed (eaten by the kid). This will be done by the factory thread calling the stats code when a candy is created, and the kid thread calling the stats code when an item is consumed.

Suggested .h file (`stats.h`):

```
#ifndef STATS_H
#define STATS_H

void stats_init(int num_producers);
void stats_cleanup(void);
void stats_record_produced(int factory_number);
void stats_record_consumed(int factory_number, double delay_in_ms);
void stats_display(void);

#endif
```

Internally in `stats.c`, you will likely need to track a number of values for each candy-factory. It is suggested you create a `struct` with all required fields, and then build an array of such `structs` (one element for each candy-factory).

The `stats_init()` function can initialize your data storage and get it ready to process produced and consumed events (via the respective functions). The `stats_cleanup()` function is used to free any dynamically allocated memory. This function should be called just before `main()` terminates.

Displaying Stats Summary

When the program ends, you must display a table summarizing the statistics gathered by the program. For example, it should resemble quite closely:

Statistics:

Factory#	#Made	#Eaten	Min Delay[ms]	Avg Delay[ms]	Max Delay[ms]
0	5	5	0.60498	2602.81274	5004.28369
1	5	5	0.40454	2202.97290	5005.06494
2	7	7	0.60107	2287.86067	4004.16162
3	8	8	1001.12012	2377.36115	5004.13159
4	5	5	0.40186	2202.63008	5005.38330
5	4	4	1003.22095	2503.94049	4006.16309
6	5	5	1003.24487	2603.35894	4005.19873
7	6	6	3002.30640	3836.61743	4005.16089
8	4	4	3001.74048	3753.03259	5004.31177
9	4	4	3002.76660	4253.44440	5005.13550

- **Factory #:** Candy factory number. In this example, there were 10 factories.

- **# Made:** The number of candies that each factory reported making (as per the call from the candy-factory thread).

- **# Eaten:** The number of candies which kids consumed (as per the call from the kid threads).

- **Min Delay[ms]:** Minimum time between when a candy was created and consumed over all candies created by this factory. Measured in milliseconds.

- **Avg Delay[ms]:** Average delay between this factory's candy being created and consumed.

- **Max Delay[ms]:** Maximum delay between this factory's candy being created and consumed.

Requirements:

- The table must be nicely formatted (as above).
    - Hint: For the title row, use the following idea:
      ```
      printf("%8s%10s%10s\n", "First", "Second", "Third");
      ```
    - Hint: For the data rows:
      ```
      printf("%8d%10.5f%10.5f\n", 1, 2.123456789, 3.14157932523);
      ```

- If the #Made and #Eaten columns don't match, print an error: "ERROR: Mismatch between number made and eaten."

4.   Testing

`valgrind` will be used to check for memory leaks. Don't worry if `valgrind` reports "still accessible" memory which was allocated from any function called from `pthread_exit()`; you may get a few such warnings. **But all memory that *you* allocate must be freed and not be "*still accessible*".**

You can run `valgrind` with the following command:
```
valgrind --leak-check=full --show-leak-kinds=all --num-callers=20 ./candykids 8 1 1
```

See online for some sample outputs.

5.   Deliverables

Submit an archive (.zip) to CourSys of your code and `Makefile`. We will build your code using your `Makefile`, and the run it using the command: `./candykids 2 2 10`

Please remember that all submissions will automatically be compared for unexplainable similarities.