Producer-Consumer Documentation

Objective:

To implement a solution to the producer-consumer problem. This solution with consist of many producer and consumer threads running concurrently sharing a common buffer.

Global Variables:

- struct element - consists of value and the lock
- buffer - a buffer of elements
- max_buffer_size - size of the buffer
- count - number of elements in the buffer
- fd - file descriptor for the log file
- fileLock - mutex lock for file writing
- in - next slow for producer
- out - next slow for consumer
- proLock - lock
- val - value produced

Functions:

The program has the following functions:

- main()
- consumerFunction(void *args)
- producerFunction(void *args)

main():

The main method starts up setting up the problem. It checks for three arguments: the number of consumer threads, the number of producer threads, and the buffer size. If none are passed, it uses the minimums of 3, 4, and 20. The buffer is then created along with the log file.

Following, the threads are built. Using pthread_create, the threads are started and an error is thrown a message is logged.

The main method terminates by calling pthread_join() on each thread, and exits successfully.

producerFunction():

This function starts off by printing “Producer” with it’s ID by calling pthread_self. An infinite while loop then starts. The loop first checks by obtaining the lock and checking if the buffer is full. If the buffer is full, it logs it to the text file and sleeps for one second.

If it is not full, it then sets the value property in the buffer to val variable. It updates val, count and in. It logs the produced value to the text file.

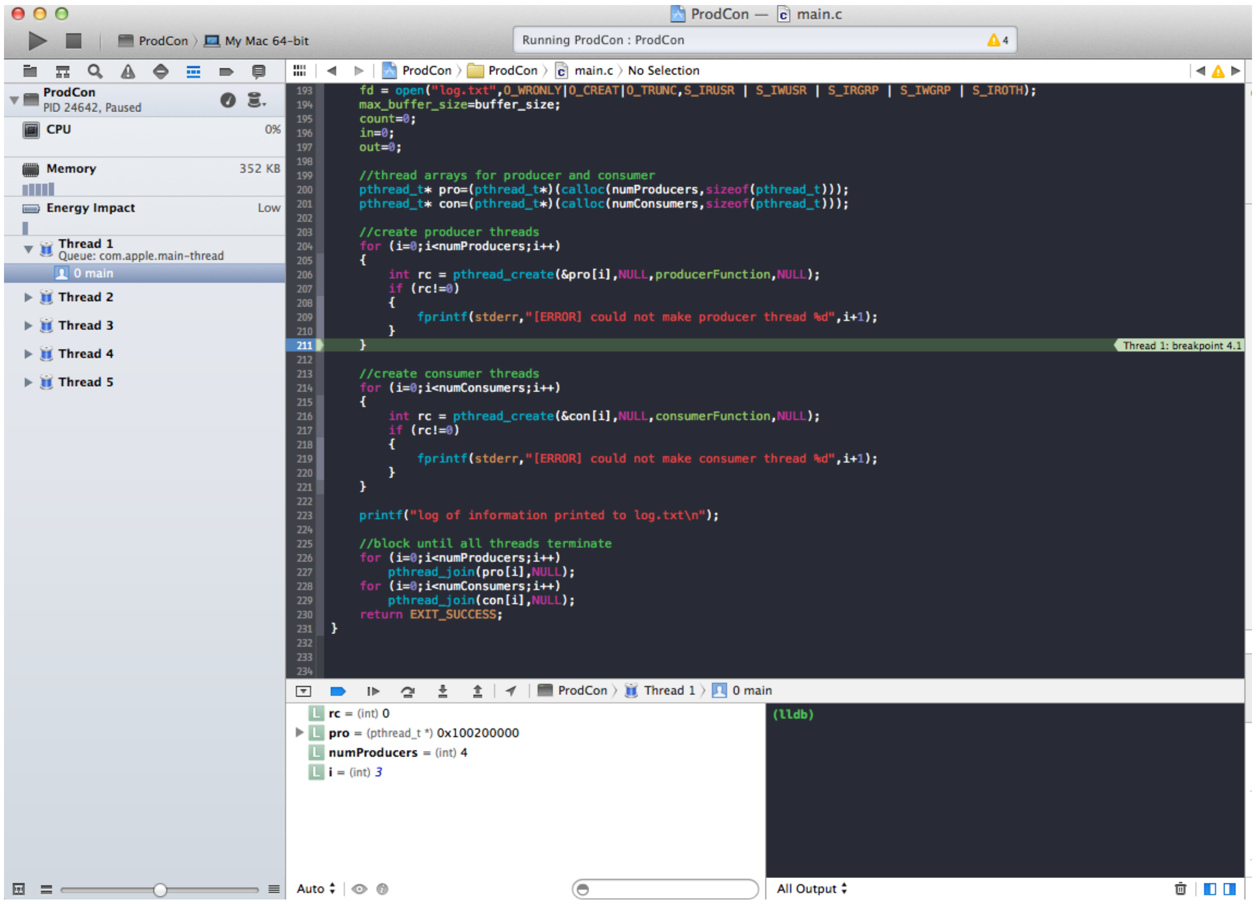
consumerFunction():

This function acts as the counter part to the producer function. It also prints it’s name to start, and enters a while loop. It checks to see if the buffer is full and logs if it is.

It then stores a local copy of the value element that is obtained from the buffer slot it is reading. It then updates the global variables, and logs the consumed message to the text file. After a successful log message is written, the unlock function is called.

Testing

Testing was done using Apple’s xCode on a Mac running Unix.

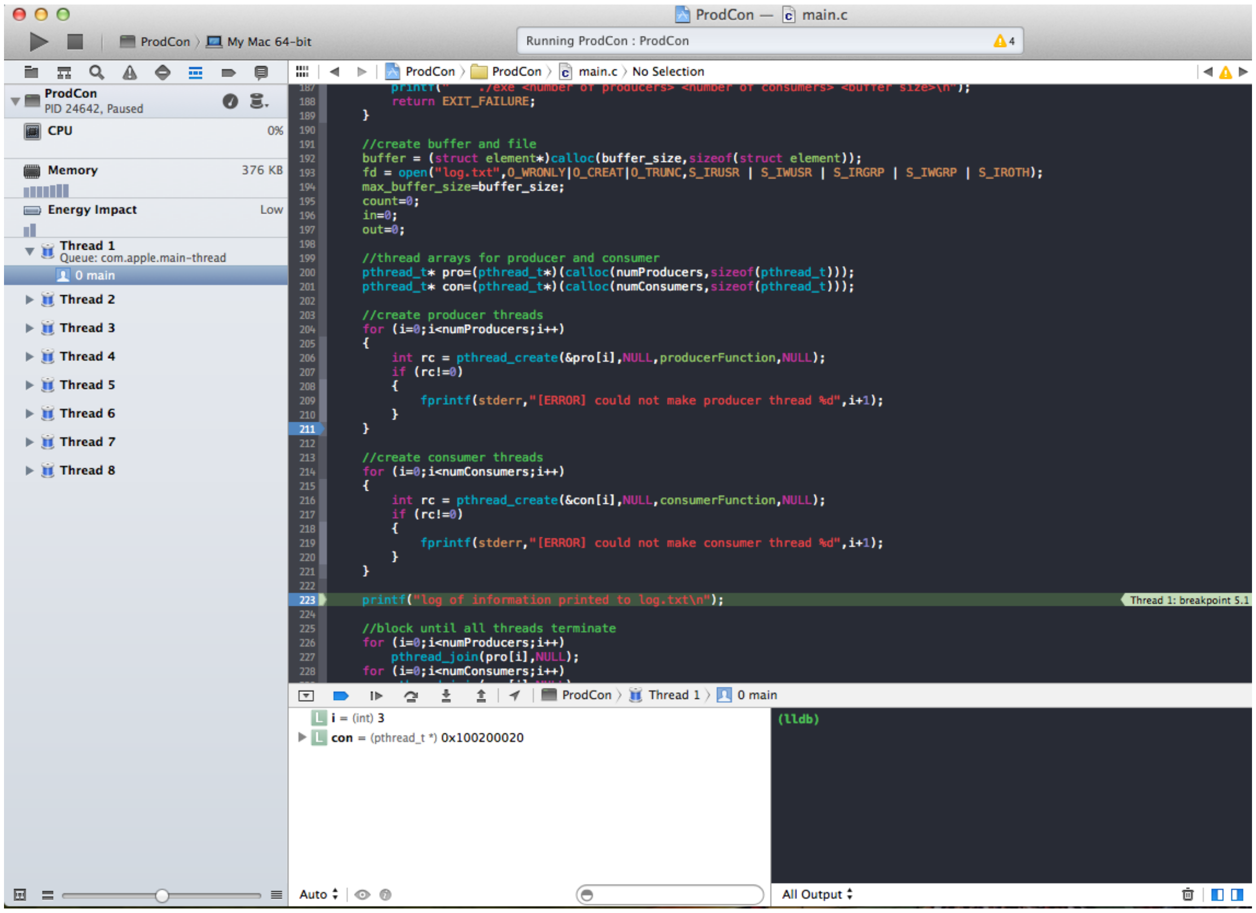


```c
    fd = open("log.txt",O_WRONLY|O_CREAT|O_TRUNC,S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP | S_IROTH);
    max_buffer_size=buffer_size;
    count=0;
    in=0;
    out=0;

    //thread arrays for producer and consumer
    pthread_t* pro=(pthread_t*)(calloc(numProducers,sizeof(pthread_t)));
    pthread_t* con=(pthread_t*)(calloc(numConsumers,sizeof(pthread_t)));

    //create producer threads
    for (i=0;i<numProducers;i++)
    {
        int rc = pthread_create(&pro[i],NULL,producerFunction,NULL);
        if (rc!=0)
        {
            fprintf(stderr,"[ERROR] could not make producer thread %d",i+1);
        }
    }

    //create consumer threads
    for (i=0;i<numConsumers;i++)
    {
        int rc = pthread_create(&con[i],NULL,consumerFunction,NULL);
        if (rc!=0)
        {
            fprintf(stderr,"[ERROR] could not make consumer thread %d",i+1);
        }
    }

    printf("log of information printed to log.txt\n");

    //block until all threads terminate
    for (i=0;i<numProducers;i++)
        pthread_join(pro[i],NULL);
    for (i=0;i<numConsumers;i++)
        pthread_join(con[i],NULL);
    return EXIT_SUCCESS;
}
```

```
rc = (int) 0
pro = (pthread_t *) 0x100200000
numProducers = (int) 4
i = (int) 3
```

```
(lldb)
```

In the above screen shot, you can see the four extra threads are were created in the left panel.



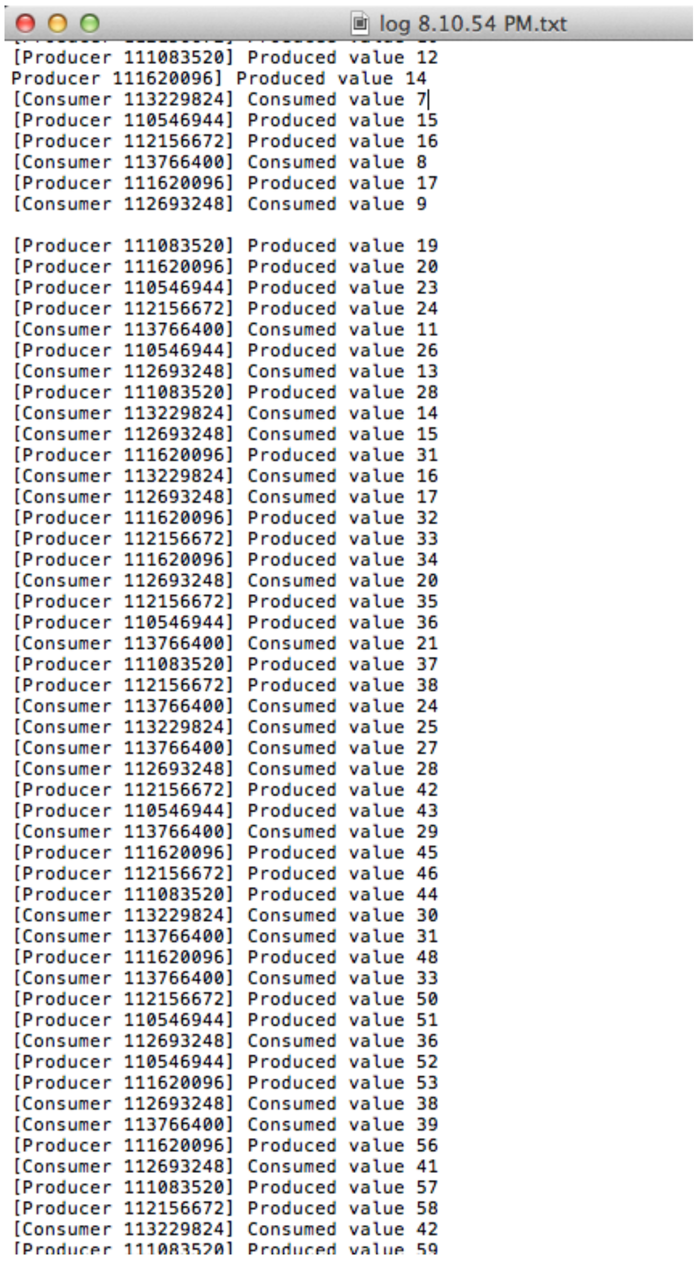
```c
            printf("    ./exe <number of producers> <number of consumers> <buffer size>\n");
            return EXIT_FAILURE;
        }

        //create buffer and file
        buffer = (struct element*)calloc(buffer_size,sizeof(struct element));
        fd = open("log.txt",O_WRONLY|O_CREAT|O_TRUNC,S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP | S_IROTH);
        max_buffer_size=buffer_size;
        count=0;
        in=0;
        out=0;

        //thread arrays for producer and consumer
        pthread_t* pro=(pthread_t*)(calloc(numProducers,sizeof(pthread_t)));
        pthread_t* con=(pthread_t*)(calloc(numConsumers,sizeof(pthread_t)));

        //create producer threads
        for (i=0;i<numProducers;i++)
        {
            int rc = pthread_create(&pro[i],NULL,producerFunction,NULL);
            if (rc!=0)
            {
                fprintf(stderr,"[ERROR] could not make producer thread %d",i+1);
            }
        }

        //create consumer threads
        for (i=0;i<numConsumers;i++)
        {
            int rc = pthread_create(&con[i],NULL,consumerFunction,NULL);
            if (rc!=0)
            {
                fprintf(stderr,"[ERROR] could not make consumer thread %d",i+1);
            }
        }

        printf("log of information printed to log.txt\n");

        //block until all threads terminate
        for (i=0;i<numProducers;i++)
            pthread_join(pro[i],NULL);
        for (i=0;i<numConsumers;i++)
```

```
i = (int) 3
con = (pthread_t *) 0x100200020
```

```
(lldb)
```

Once a break point after the second loop is hit, it is clear the extra three consumer threads are all created and running as well.

A screen shot of my log file is below:



```
[Producer 111083520] Produced value 12
Producer 111620096] Produced value 14
[Consumer 113229824] Consumed value 7
[Producer 110546944] Produced value 15
[Producer 112156672] Produced value 16
[Consumer 113766400] Consumed value 8
[Producer 111620096] Produced value 17
[Consumer 112693248] Consumed value 9

[Producer 111083520] Produced value 19
[Producer 111620096] Produced value 20
[Producer 110546944] Produced value 23
[Producer 112156672] Produced value 24
[Consumer 113766400] Consumed value 11
[Producer 110546944] Produced value 26
[Consumer 112693248] Consumed value 13
[Producer 111083520] Produced value 28
[Consumer 113229824] Consumed value 14
[Consumer 112693248] Consumed value 15
[Producer 111620096] Produced value 31
[Consumer 113229824] Consumed value 16
[Consumer 112693248] Consumed value 17
[Producer 111620096] Produced value 32
[Producer 112156672] Produced value 33
[Producer 111620096] Produced value 34
[Consumer 112693248] Consumed value 20
[Producer 112156672] Produced value 35
[Producer 110546944] Produced value 36
[Consumer 113766400] Consumed value 21
[Producer 111083520] Produced value 37
[Producer 112156672] Produced value 38
[Consumer 113766400] Consumed value 24
[Consumer 113229824] Consumed value 25
[Consumer 113766400] Consumed value 27
[Consumer 112693248] Consumed value 28
[Producer 112156672] Produced value 42
[Producer 110546944] Produced value 43
[Consumer 113766400] Consumed value 29
[Producer 111620096] Produced value 45
[Producer 112156672] Produced value 46
[Producer 111083520] Produced value 44
[Consumer 113229824] Consumed value 30
[Consumer 113766400] Consumed value 31
[Producer 111620096] Produced value 48
[Consumer 113766400] Consumed value 33
[Producer 112156672] Produced value 50
[Producer 110546944] Produced value 51
[Consumer 112693248] Consumed value 36
[Producer 110546944] Produced value 52
[Producer 111620096] Produced value 53
[Consumer 112693248] Consumed value 38
[Consumer 113766400] Consumed value 39
[Producer 111620096] Produced value 56
[Consumer 112693248] Consumed value 41
[Producer 111083520] Produced value 57
[Producer 112156672] Produced value 58
[Consumer 113229824] Consumed value 42
[Producer 111083520] Produced value 59
```

As larger values are being produced, consumers are registering smaller values. Running the program for just a few seconds can create a very large text file.