CSC 415 – Homework 5 – Producer/Consumer

Aleksandr Kibis

10/28/2014

Part 1:

The first part of this assignment asked to open up Homework 4 and change the counter to be a global variable, plus use a mutex lock in order to increment it then test the difference in process time. I actually already did this in the initial assignment so there was nothing for me to add in order to test any kind of difference.

Part 2:

The final portion of the assignment asked to implement the Producer/Consumer problem using a bounded buffer. I found this assignment to be very much like the previous one except that now there were two groups of threads working with each other to process information contained in the buffer. Both versions of the program were fairly easy to implement, with the exception of the Windows version having a confusing way to initialize semaphores. If it wasn't for the extra values required for windows semaphore initialization, the conversion would have been completely straight forward. Since I am working with Visual Studio when writing Windows code, a lot of problems that I ran into actually had to deal with the IDE and not the Windows API. Since VS doesn't fully support C of any kind (that's what it seems like at least), I had to do a lot of configuration of the project just to remove all the compilation errors. In addition to project configuration, since the base C format is C89, all variables have to be declared at the top of each function, otherwise compilation errors occur.

The run times for each program are definitely in Window's favor, but I don't believe that they're very accurate. While Linux waits for all print statements to run, the Windows Measure-Command gets rid of them completely. This obviously grants it a huge advantage over Linux. My timing results are below.

Consumers	Producers	POSIX (s)	Windows (ms)
1	1	0.002257	0.0819
2	2	0.006795	0.0827
4	4	0.052339	0.0874

POSIX

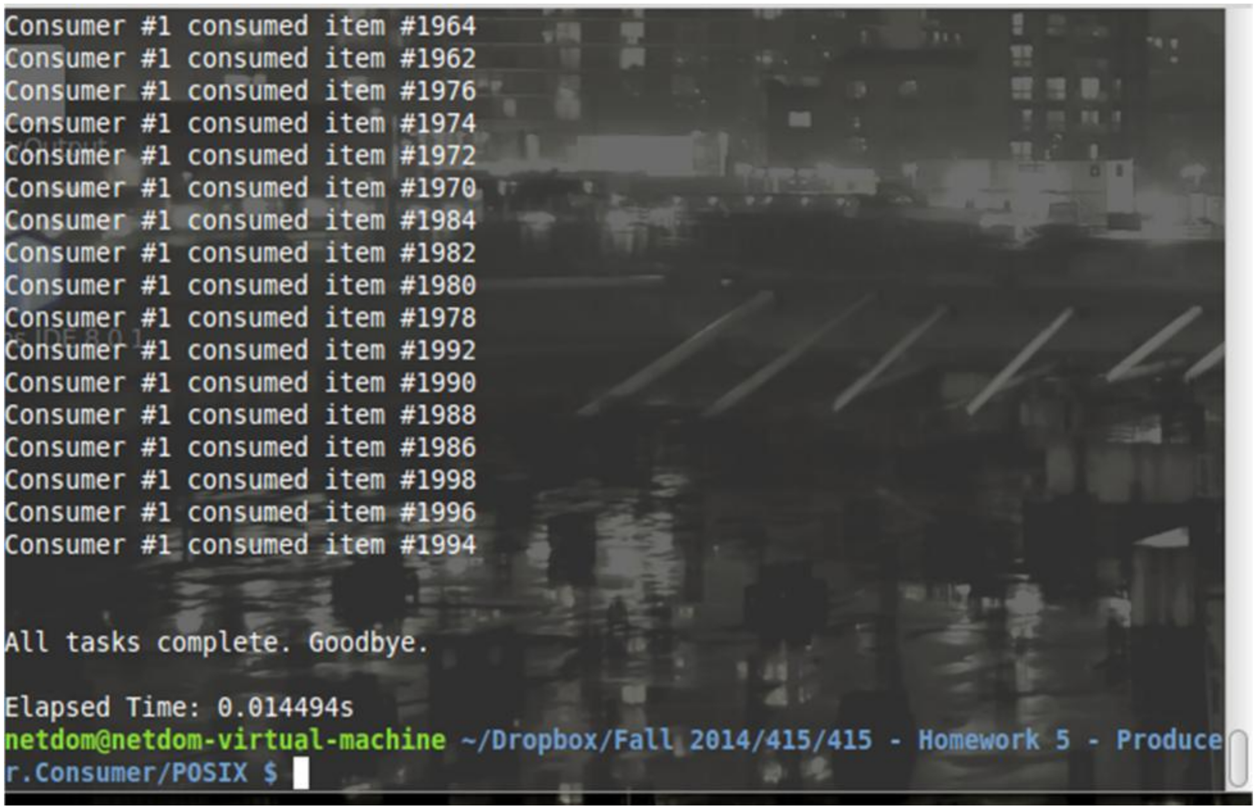
Code

```
/*
 * File:    cannibal.c
 * Author:  Aleksandr Kibis
 * Date:    10/26/2014
 *
 * Compile: gcc -o run cannibal.c -lpthread
 * Run:     ./run <int bufSize> <int numProducers> <int numConsumers> <int itemsPerProducer>
 */

#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>
#include <time.h>

#define EXIT_SUCCESS 0
#define EXIT_FAILURE 1

int threadCount, bufSize, itemsPerProducer;
static int index = 0;
int* buf;
sem_t empty;
sem_t full;
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;

void* producer(void* id);
void* consumer(void* id);

struct threadAttr {
    int id;
};

/*
 *
 */
int main(int argc, char** argv) {

    clock_t t1, t2;

    t1 = clock();

    // exit if incorrect usage
    if (argc != 5){
        printf("Usage: ./run <buf size> <# of producers> <# of consumers> 
<items/producer>\n");
        exit(EXIT_FAILURE);
    }

    // grab command line args
    bufSize = atoi(argv[1]);
    int producerCount = atoi(argv[2]);
    int consumerCount = atoi(argv[3]);
    itemsPerProducer = atoi(argv[4]);
    threadCount = producerCount;

    // print line args
    printf("Buffer Size: %d\n"
            "Number of Producers: %d\n"
            "Number of Consumers: %d\n"
            "Items Per Producer: %d\n",
            bufSize, producerCount, consumerCount, itemsPerProducer);

    // init buffer
    buf = malloc(bufSize * sizeof (int));
```

```c
// init sync objects
sem_init(&empty, 0, bufSize);
sem_init(&full, 0, 0);

// create threads
int i, retval;
pthread_attr_t pta;
pthread_attr_init(&pta);
pthread_t prods[producerCount];
pthread_t cons[consumerCount];
struct threadAttr *producers;
struct threadAttr *consumers;

for (i = 0; i < producerCount; i++) {
    producers = malloc(sizeof (struct threadAttr));
    (*producers).id = i;
    retval = pthread_create(&prods[i], &pta, producer, (void*) producers);
    //printf("Producer Thread %d\n", i);
    if (retval < 0) {
        printf("Error in creating producer thread.\n");
        exit(EXIT_FAILURE);
    }
}

for (i = 0; i < consumerCount; i++) {
    consumers = malloc(sizeof (struct threadAttr));
    (*consumers).id = i;
    retval = pthread_create(&cons[i], &pta, consumer, (void*) consumers);
    //printf("Consumer Thread %d\n", i);
    if (retval < 0) {
        printf("Error in creating consumer thread.\n");
        exit(EXIT_FAILURE);
    }
}


// wait for threads

for (i = 0; i < producerCount; i++) {
    //printf("Wait Producer %d\n", i);
    retval = pthread_join(prods[i], NULL);
    //printf("Producer Joined %d\n", i);
    if (retval < 0) {
        printf("Error in joining producer thread.");
        exit(EXIT_FAILURE);
    }
}

for (i = 0; i < consumerCount; i++) {
    //printf("Wait Consumer %d\n", i);
    retval = pthread_join(cons[i], NULL);
    //printf("Consumer Joined %d\n", i);
    if (retval < 0) {
        printf("Error in joining consumer thread.");
        exit(EXIT_FAILURE);
    }
}

// exit
printf("\n\nAll tasks complete. Goodbye.\n");
```

```c
    t2 = clock();
    printf("\nElapsed Time: %.6fs\n", (t2 - t1) / (double) CLOCKS_PER_SEC);

    // free memory
    free(producers);
    free(consumers);
    free(buf);

    return (EXIT_SUCCESS);
}

// produces items and fills buffer
void* producer(void* threadAttr) {
    int counter = 0;
    int item;

    struct threadAttr *prodAttr = (struct threadAttr*) threadAttr;
    int id = (*prodAttr).id;

    while (1) {

        // enter synchronized state
        sem_wait(&empty);
        pthread_mutex_lock(&lock);

        if (index < bufSize) {
            // create unique item number based on thread, id, and iteration
            item = counter * threadCount + id;
            // write to buffer
            buf[index] = item;
            if (index > bufSize) {
                //printf("Error\n");
                break;
            }
            index++;
            counter++;
        }
        pthread_mutex_unlock(&lock);
        sem_post(&full);

        if (counter == 1000) {
            //printf("Max reached.\n");
            break;
        }
//        if (id == 3){
//            printf("item: %d\n", item);
//        }
    }
    return NULL;
}

// consumes items within the buffer
void* consumer(void* threadAttr) {
    int itemsConsumed;
    int item;
    struct threadAttr *consAttr = (struct threadAttr*) threadAttr;
    // get thread id
    int id = (*consAttr).id;

    // loop until all items have been consumed
    while (1) {
        sem_wait(&full);
        pthread_mutex_lock(&lock);


        // *consume* item
        item = buf[index - 1];
        printf("Consumer #%d consumed item #%d\n", id, item);
        index--;
        itemsConsumed++;
        //printf("Consumer: %d\titemsConsumed: %d\n", id, itemsConsumed);
        if (index < 0) {
            //printf("Error in Consumer\n");
            break;
        }

        pthread_mutex_unlock(&lock);
        sem_post(&empty);

        if (itemsConsumed == 1000) {
            //printf("Consumer %d: All items consumed.\n", id);
            break;
        }
//          if(id == 0 )
//              printf("%d\n", itemsConsumed);
    }
    return NULL;
}
```

Output



```
Consumer #1 consumed item #1964
Consumer #1 consumed item #1962
Consumer #1 consumed item #1976
Consumer #1 consumed item #1974
Consumer #1 consumed item #1972
Consumer #1 consumed item #1970
Consumer #1 consumed item #1984
Consumer #1 consumed item #1982
Consumer #1 consumed item #1980
Consumer #1 consumed item #1978
Consumer #1 consumed item #1992
Consumer #1 consumed item #1990
Consumer #1 consumed item #1988
Consumer #1 consumed item #1986
Consumer #1 consumed item #1998
Consumer #1 consumed item #1996
Consumer #1 consumed item #1994


All tasks complete. Goodbye.

Elapsed Time: 0.014494s
netdom@netdom-virtual-machine ~/Dropbox/Fall 2014/415/415 - Homework 5 - Produce
r.Consumer/POSIX $
```

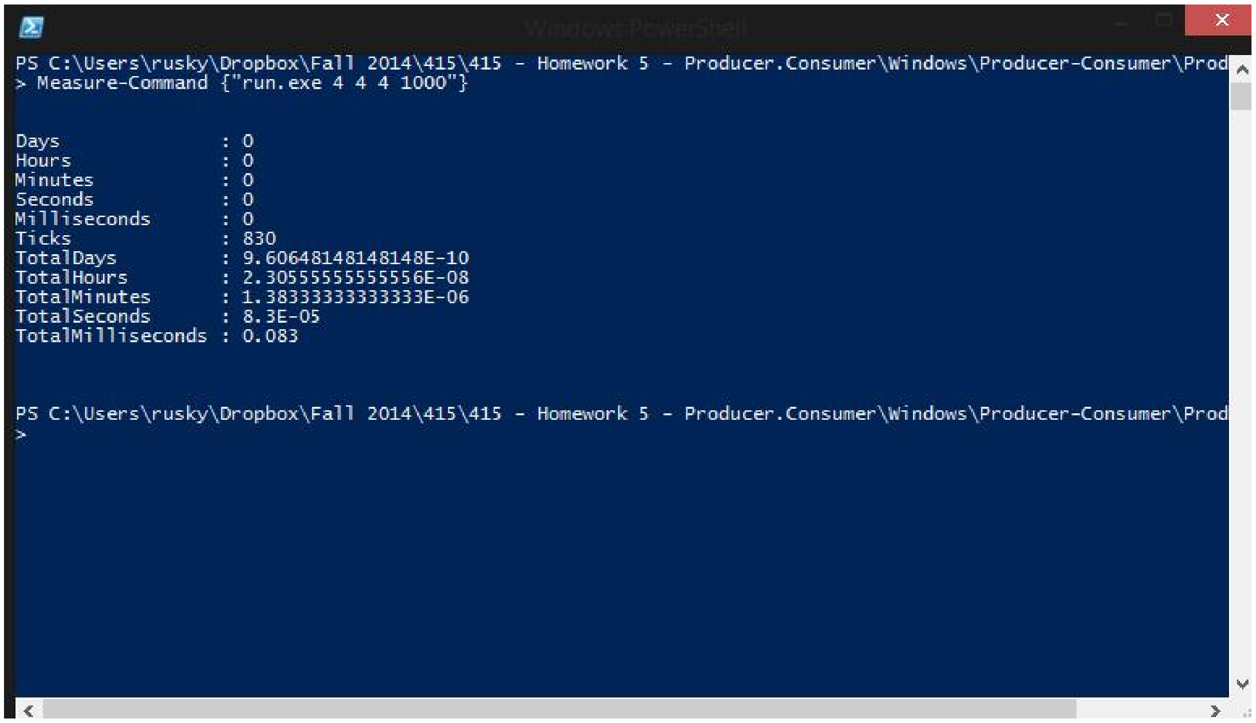
Windows

Code

```c
/*
* File:       cannibal32.c
* Author:     Aleksandr Kibis
* Date:       10/28/2014
*
* Compile:  cl -o run cannibal32.c
* Run:        run.exe <int bufSize> <int # prods> <int # cons> <int items/prod>
*/

#include <stdio.h>
#include <stdlib.h>
#include <Windows.h>

#define EXIT_SUCCESS 0
#define EXIT_FAILURE 1

static int index = 0;

int threadCount, bufSize, itemsPerProducer, producerCount, consumerCount;
int* buf;

HANDLE lock, empty, full;
HANDLE prods[4];
HANDLE cons[4];

DWORD WINAPI producer(LPVOID id);
DWORD WINAPI consumer(LPVOID id);

struct threadAttr {
    int id;
};

/*
 *
 */
int main(int argc, char** argv) {

    int i;
    struct threadAttr *producers;
    struct threadAttr *consumers;
       DWORD threadID;

    // exit if incorrect usage
    if (argc != 5){
        printf("Usage: ./run <buf size> <# of producers> <# of consumers>
<items/producer>\n");
        exit(EXIT_FAILURE);
    }

    // grab command line args
    bufSize = atoi(argv[1]);
    producerCount = atoi(argv[2]);
    consumerCount = atoi(argv[3]);
    itemsPerProducer = atoi(argv[4]);
    threadCount = producerCount;

    // print line args
    printf("Buffer Size: %d\n"
            "Number of Producers: %d\n"
            "Number of Consumers: %d\n"
            "Items Per Producer: %d\n",
            bufSize, producerCount, consumerCount, itemsPerProducer);

    // init buffer
    buf = (int*)malloc(bufSize * sizeof(int));

    // init sync objects
       empty = CreateSemaphore(NULL, bufSize, bufSize, NULL);
       full = CreateSemaphore(NULL, 0, bufSize, NULL);
       lock = CreateMutex(NULL, FALSE, NULL);


    // create threads

    for (i = 0; i < producerCount; i++) {
    producers = (struct threadAttr*)malloc(sizeof (struct threadAttr));
        (*producers).id = i;
              //printf("i: %d\n", (*producers).id);
              prods[i] = CreateThread(NULL, 0, producer, producers, 0, &threadID);
        //printf("Producer Thread %d\n", i);
    }

    for (i = 0; i < consumerCount; i++) {
    consumers = (struct threadAttr*)malloc(sizeof (struct threadAttr));
        (*consumers).id = i;
              cons[i] = CreateThread(NULL, 0, consumer, consumers, 0, &threadID);
        //printf("Consumer Thread %d\n", i);
    }


    // wait for threads

    for (i = 0; i < producerCount; i++) {
        //printf("Wait Producer %d\n", i);
              WaitForSingleObject(prods[i], INFINITE);
        //printf("Producer Joined %d\n", i);
    }

    for (i = 0; i < consumerCount; i++) {
        //printf("Wait Consumer %d\n", i);
              WaitForSingleObject(cons[i], INFINITE);
        //printf("Consumer Joined %d\n", i);
    }

    // exit
    printf("\n\nAll tasks complete. Goodbye.\n");

    // free memory
       CloseHandle(lock);
    free(buf);

    return (EXIT_SUCCESS);
}

// produces items and fills buffer
DWORD WINAPI producer(LPVOID threadAttr) {
    int itemsProduced = 0;
    int item;
```

```c
        int id;

    struct threadAttr *prodAttr;
        prodAttr = (struct threadAttr*) threadAttr;

        id = (*prodAttr).id;
        //printf("ID: %d\n", id);
    while (1) {

        // enter synchronized state
        WaitForSingleObject(empty, INFINITE);
        WaitForSingleObject(lock, INFINITE);

        if (index < bufSize) {
            // create unique item number based on thread, id, and iteration
            item = itemsProduced * threadCount + id;
            // write to buffer
            buf[index] = item;
            if (index > bufSize) {
                //printf("Error\n");
                break;
            }
            index++;
            itemsProduced++;
        }
        ReleaseMutex(lock);
        ReleaseSemaphore(full, 1, NULL);

        if (itemsProduced == 1000) {
            //printf("Max reached.\n");
            break;
        }
//        if (id == 3){
//            printf("item: %d\n", item);
//        }
    }
    return EXIT_SUCCESS;
}

// consumes items within the buffer
DWORD WINAPI consumer(LPVOID threadAttr) {
    int itemsConsumed = 0;
    int item;
        int id;
    struct threadAttr *consAttr;
        consAttr = (struct threadAttr*) threadAttr;
    // get thread id
    id = (*consAttr).id;



    // loop until all items have been consumed
    while (1) {
        WaitForSingleObject(full, INFINITE);
        WaitForSingleObject(lock, INFINITE);


        // gobble item
        item = buf[index - 1];
        printf("Consumer #%d consumed item #%d\n", id, item);
        index--;
        itemsConsumed++;
        //printf("Consumer: %d\titemsConsumed: %d\n", id, itemsConsumed);
        if (index < 0) {
           //printf("Error in Consumer\n");
            break;
               }

        ReleaseMutex(lock);
        ReleaseSemaphore(empty, 1, NULL);

        if (itemsConsumed == 1000) {
            //printf("Consumer %d: All items consumed.\n", id);
            break;
        }
    }
    return EXIT_SUCCESS;
}
```

Output



```
PS C:\Users\rusky\Dropbox\Fall 2014\415\415 - Homework 5 - Producer.Consumer\Windows\Producer-Consumer\Prod
> Measure-Command {"run.exe 4 4 4 1000"}


Days              : 0
Hours             : 0
Minutes           : 0
Seconds           : 0
Milliseconds      : 0
Ticks             : 830
TotalDays         : 9.60648148148148E-10
TotalHours        : 2.30555555555556E-08
TotalMinutes      : 1.38333333333333E-06
TotalSeconds      : 8.3E-05
TotalMilliseconds : 0.083



PS C:\Users\rusky\Dropbox\Fall 2014\415\415 - Homework 5 - Producer.Consumer\Windows\Producer-Consumer\Prod
>
```