CIS 3207 Lab 3 -- Producer / Consumer

Dan McGinnis -- October 29, 2013

The purpose of this lab is to have multiple producer and consumer threads writing and reading into an circular array. Access to the array is to be controlled through semaphores and mutex locks.

To run the file on Linux:

```
make
./pc x y
python3 test_harness.py log_file
```

To run the file on Windows (after building in Visual Studio):

```
win_pc.exe x y
```

where `x` and `y` are the number of producer and consumer threads to be created respectively.

The program takes two integers as input. The first is the number of producer threads. The second is the number of consumer threads. The program checks these input values to make sure they are reasonable. Currently that range is set from 2 to 1024. If the numbers are outside of that range, the program notifies the user and exits right away. I have set the buffer size to be 1000 cells long, so it is important to keep in mind that with very low numbers for producer of consumer, it will take a while for the threads to meet near the middle of the array.

The consumer thread starts at the beginning of the array and works upward. The consumer starts at the back of the array and works down. Sempahores are used to prevent the consumer from reading if there isn't any data and the producer from inserting if there isn't any room in the array. A mutex is used to protect the actual read or write to the array and the update of the counter variable. All the data associated with this program and the sempahores and mutex are stored in a struct. This struct is initialized in the main function and passed to each of the threads. Because the threads require that a void data type be passed in the struct is recast in the producer and consumer functions.

I have included functionality to wait for the threads to exit. After this occurs the files are safely closed and the mutex is destroyed before the program is terminated. Currently the flow of the program never reaches this code, because there is no mechanism to safely wind down the threads. I attempted to implement this in a way that ensured the producers and consumers did equal work without turning the program into a serial operation, but was unable to do so correctly. Because this functionality is not called for in the lab spec, I have removed the code that attempted to do this.

I developed this program and did all the testing on OS X and Ubuntu. Once it was properly working, I ported the program to Windows using Visual Studio 2012. Because the port required minor changes to API calls in the program and a slight re-ordering of declarations, I only performed minimal testing on the Windows platform to verify that the program was functioing properly. I have included a diff of the files for the linux and Windows versions, annotated as needed. This Readme file was written in Markdown, converted to HTML5, and "printed" as a PDF.

Testing

I tested this program in several ways. The first route I persued was manual testing of the output. I initally had all the output that now goes to the log writing to the screen. I lowered the size of the buffer and increased the time delay so that it was reasonable to observe the operation of the program as it ran. Once I was satisified that it was working properly, I reverted the program to writing into the log file and ran it for larger numbers of threads. I read through the log file to makes sure that the program appeared to be working properly. Additionally, while I was attempting to implement the wind down function to gracefully end the program, I observed the thread count displayed by running the `top` program on a second terminal. Finally, I implemented the previously mentioned python based test harness. This allowed me to run the program with a much bigger buffer and larger producer and consumer threads and still make sure that the reads and writes were happening as expected. I tested this program on both 64bit OS X 10.8 and a 32bit vm running the latest long term release of Ubuntu. Testing on the Ubuntu VM showed a limitation of roughly 381 threads per process. Other testing on a 64 bit version of OS X showed the program to be function with a thread count several orders of magnitude greater. The program produces two log files in the current directory, one called `human_log_file` and the other called `log_file`. The `human_log_file` contains all of the relevant data and displays it in a friendly format. The other `log_file` just contains the values written and read to the buffer. Becaue the `human_log_file` is meant for human consumption it is opened in append mode. Since that would interfere with the programtic testing, the `log_file` is opened in write mode and overwritten every time the program is run. The number 0, which represents an empty buffer is not recorded when read. The purpose of this second file is to allow a programatic inspection of the programs output. I have included a python 3 program called `test_harness.py`. This program will run through the `log_file` and count the number of times each number is seen. Ideally each number would be seen twice, once for a write and once for a read. The program actually tests for a multiple of 2 to allow for the fact that the payload is a randomly generated number and could occur more than once. However, because the program runs until terminated, there is not a way to safely wind down the program and guarentee that the producer and consumer threads have done equal work. As a result, the `test_harness.py` will report a success if less that 1% of the numbers found are unique.The test harness will print out a message indicating success or failure. In the test harness file, there are additional print statements that could be uncommented to provide more feedback that could be compared to the `human_log_file` for further debugging. Because I belive the program to be functioning properly, this output is silenced for now.

Program details

The `prod_con.h` file

- Lists all the included files
- Defines the global variables needed
- Defines the struct that holds all the relevant program data
- Lists the headers of all the programs functions.

The `main.c` file

- Checks user input for a reasonable number of threads
- Opens the log files or terminates the program upon failure to open files.
- Populates the struct with data for the program to begin running
- Initiates the timer, the random number generator, the semaphore, and the mutex
- Fires off the producer and consumer threads and waits

The `module.c` file for Linux

- Contains the code for the four functions needed for the program to function. Documentation from the functions themselves is included below.

int mod (int a, int b)

```
*
* This function provides modular operations for C. Because the C standard
*  defines the % operator as the remainder function, calling for a negative
*  number mod a positive number will result in a negative result.
*  i.e. -1 % 10 returns -9
* This function checks the output and if it is negative will add the mod value
*  to the result. mod(-1, 10) returns 9.
* This function is very heavily based on the following post on Stack Overflow:
* http://stackoverflow.com/a/4003287/738842
*
* Input:
*      Two integers
*
* Output:
*      a positive integer representing a modulo b
*
* Modifies:
*      none
*
* Assumptions:
*      b is a positive integer
*
*/
```

struct timeval time_stamp(struct timeval start, struct timeval current)

```
*
* This function takes two microsecond values and returns the difference. The
*  function takes into account that the value may "roll over" before the
*  difference is calculated. This function is based very heavily on the
*  following post on Stack Overflow: http://stackoverflow.com/a/10487325/738842
*
* Input:
*      Two timeval structs.
*
* Output:
*      A timeval struct representing the difference between the two input
*          in microseconds [one millionth of a second].
*
* Modifies:
*      none
*
* Assumptions:
*      There is valid data in both input variables.
*      The time gap being calculated is of a size that is reasonably
*          represented in microseconds [ i.e. less an several minutes].
*/
```

void *producer(void *indata)

```
*
* This function accepts a void pointer to a struct as the input. The struct
*      is recast as type t_data before and work is done on it.
*      After waiting some random time, a random integer is generated and the
*      semaphores and mutex are checked respectively. Once the data structure
*      is locked, the random number payload is written into the next
*      available cell in the data structure. The counter is updated and the
*      locks and semphores are released.
*
* Input:
*    indata: a pointer to a void data type. This is a requirement of pthreads.
*          once inside the function the input is recast as type t_data.
*
* Output:
*    writes output to logfiles
*
* Modifies:
*    The buffer[] array of the struct is filled with random numbers.
*
* Assumptions:
*    buffer[] has already been allocated.
*    That the cells of buffer[] have been zeroed.
*
* Note:
*  Because this function is intended to be called as several dozen threads, the exact
*      order of it's behavior is not guarenteed.
*
*
*/
```

void consumer(void *indata)

```
*
* This function accepts a void pointer to a struct as the input. The struct
*      is recast as type t_data before and work is done on it.
*      After waiting some random time, a random integer is generated and the
*      semaphores and mutex are checked respectively. Once the data structure
*      is locked, the random number payload is written into the next
*      available cell in the data structure. The counter is updated and the
*      locks and semphores are released.
*
```

```
* Input:
*    indata: a pointer to a void data type. This is a requirement of pthreads.
*          once inside the function the input is recast as type t_data.
*
* Output:
*    writes output to logfiles
*
* Modifies:
*    The buffer[] array of the struct is filled with random numbers.
*
* Assumptions:
*    buffer[] has already been allocated.
*    That the cells of buffer[] have been zeroed.
*
* Note:
*  Because this function is intended to be called as several dozen threads, the exact
*      order of it's behavior is not guarenteed.
*
*
*/
```

The `module.c` file for Windows

- Contains the code for the four functions needed for the program to function. Documentation from the functions that differ is included below.

unsigned __stdcall producer(void *indata)

```
*
* This function accepts a void pointer to a struct as the input. The struct
*      is recast as type t_data before and work is done on it.
*      After waiting some random time, a random integer is generated and the
*      semaphores and mutex are checked respectively. Once the data structure
*      is locked, the random number payload is written into the next
*      available cell in the data structure. The counter is updated and the
*      locks and semphores are released.
*
* Input:
*    indata: a pointer to a void data type. This is a requirement of pthreads.
*          once inside the function the input is recast as type t_data.
*
* Output:
*    writes output to logfiles
*
* Modifies:
*    The buffer[] array of the struct is filled with random numbers.
*
* Assumptions:
*    buffer[] has already been allocated.
*    That the cells of buffer[] have been zeroed.
*
* Note:
*  Because this function is intended to be called as several dozen threads, the exact
*      order of it's behavior is not guarenteed.
*
*
*/
```

unsigned __stdcall consumer(void *indata)

```
*
* This function accepts a void pointer to a struct as the input. The struct
*      is recast as type t_data before and work is done on it.
*      After waiting some random time, the semaphores and mutex are checked
*      respectively. Once the data structure is locked, the value in the current
*      cell (if it isn't 0, indicating nothing has been written) is removed and
*      a NULL character is written into the cell. The counter is updated and the
*      locks and semphores are released.
*
* Input:
*    indata: a pointer to a void data type. This is a requirement of pthreads.
*          once inside the function the input is recast as type t_data.
*
* Output:
*    writes output to logfiles
*
* Modifies:
*    The buffer[] array of the struct is filled with random numbers.
*
* Assumptions:
*    buffer[] has already been allocated.
*    That the cells of buffer[] have been zeroed.
*
* Note:
*  Because this function is intended to be called as several dozen threads, the exact
*      order of it's behavior is not guarenteed.
*
*/
```

Linux/Windows Differences

The linux file is listed first and indicated by the <. Windows is represented by >.

The changes in these files fell into two catagories.

1. Changes because of using the Windows API as opposed to the Posix API.
2. Changes from the C11 standard, which is supported in Linux, to the C89 standard for Windows.

diff -B prod*con.h win*pc/win*pc/prod*con.h

```
< #include <sys/time.h>
< #include <pthread.h>
< #include <semaphore.h>
< #include <unistd.h>
---
> #include <windows.h>
> #include <time.h>
> #include <process.h>
> #include <malloc.h>

< struct timeval start;                     <--Timer Functionality works the same but
< struct timeval current;                       stores data in a different data structure.
< struct timeval temp_time;
---
> SYSTEMTIME start;
> SYSTEMTIME current;
> SYSTEMTIME temp_time;

<     pthread_mutex_t mutex;                <--Pthreads and Semaphores for Linux
<     sem_t empty;
<     sem_t full;
---
>     HANDLE mutex;                         <--Handles in Windows
>     HANDLE empty;
>     HANDLE full;

< void *producer(void *indata);
< void *consumer(void *indata);
---
> unsigned __stdcall producer(void *indata);
> unsigned __stdcall consumer(void *indata);
```

diff -B main.c win*pc/win*pc/main.c

```
---
>
>   static t_data lab_3;                    <-- C89 forces declaration at the top of the function.
>   int num_pro_threads;
>   int num_con_threads;
>   int i;
>   time_t clk;
>   HANDLE pro_threads[1024];
>   HANDLE con_threads[1024];
>     /* Windows wouldn't let me use a variable that wasn't set at run time. So I
>      *  have set the thread array to be the max size of allowable input. If the
>      *  user enters thread numbers lower than 1024, the remainder of the array
>      *  will remain unused. This is not the most efficient method, but this
>      *  program isn't meant for production.
>      */
>
>   if (argc < 3)

<     int num_pro_threads = atoi(argv[1]);
<     int num_con_threads = atoi(argv[2]);
---
>
>     num_pro_threads = atoi(argv[1]);
>     num_con_threads = atoi(argv[2]);

<
<     static t_data lab_3 = {.tail = MAX_SIZE-1, .head = 0, .counter = 0};   <--- This is the feature I enjoy most in C11.
---
>
>   lab_3.tail = MAX_SIZE-1;
>   lab_3.head = 0;
>   lab_3.counter = 0;

<     gettimeofday(&start, NULL);
<     pthread_mutex_init (&lab_3.mutex, NULL);
<     sem_init(&lab_3.empty, 0, MAX_SIZE);
<     sem_init(&lab_3.full, 0, 0);
---
>     GetSystemTime(&start);
>     lab_3.mutex = CreateMutex(NULL, false, NULL);
>     lab_3.empty = CreateSemaphore(NULL, 0, MAX_SIZE, NULL);
>     lab_3.full = CreateSemaphore(NULL, 0, 0, NULL);

<     srandom(time(NULL));
<
<     pthread_t pro_threads[num_pro_threads];
<     pthread_t con_threads[num_con_threads];
<
<     int i = 0;
---
>     srand((unsigned)time(NULL));

<         pthread_create(&con_threads[i], NULL, consumer, &lab_3);
---
>         unsigned thread_id;
>       con_threads[i] = (HANDLE)_beginthreadex(NULL, 0, consumer, &lab_3, 0, &thread_id);
```

```
<         pthread_create(&pro_threads[i], NULL, producer, &lab_3);
<
---
>         unsigned thread_id;
>       pro_threads[i] = (HANDLE)_beginthreadex(NULL, 0, producer, &lab_3, 0, &thread_id);

<     (void) sleep(1);
---
>     (void) Sleep(1);

<         pthread_join(pro_threads[i], NULL);
---
>         WaitForSingleObject(pro_threads[i], NULL);

<         pthread_join(con_threads[i], NULL);
---
>         WaitForSingleObject(con_threads[i], NULL);

<     time_t clk = time(NULL);
---
>     clk = time(NULL);

<     pthread_mutex_destroy(&lab_3.mutex);
---
>     CloseHandle(&lab_3.mutex);
```

diff -B module.c win*pc/win*pc/module.c

```
<    int ret = a % b;
---
>    int ret;
>    ret = a % b;

< void *producer(void *indata)
---
> unsigned __stdcall producer(void *indata)

<     t_data *data = indata;                        <--The data comes in as void, but C11 doesn't require
---                                                     as casting.
>     t_data *data = (t_data*) indata;
>   int temp;
>   int payload;

<         sleep(random()/1000000000);                             //generates a sleep from 0 to 21
<         int payload = random();                                 //produce item
---                                                 <--Random returns a 32 bit integer.
>                                                   <--Rand returns a value less than 30000
>       temp = rand();
>       Sleep(temp/100);                                    //generates a Sleep from 0 to 21
>
>       payload = rand();                                   //produce item

<             sem_wait(&data->empty);
---
>             WaitForMultipleObjects(MAX_SIZE, &data->empty, FALSE, INFINITE);

<         pthread_mutex_lock(&data->mutex);
---
>         WaitForSingleObject(&data->mutex, INFINITE);

<                 pthread_mutex_unlock(&data->mutex);
<                 sem_post(&data->full);
<                 return NULL;
---
>                 ReleaseMutex(&data->mutex);
>                 ReleaseSemaphore(&data->full, 1, NULL);
>                 return 0;

<             gettimeofday(&current, NULL);
<             temp_time = time_stamp(start, current);
---
>             GetSystemTime(&current);
>             temp_time.wMilliseconds = start.wMilliseconds - current.wMilliseconds;

<             fprintf(human_log_file, "%13d was written to the data structure at %d microseconds.\n", payload, temp_time.tv_usec);
---
>             fprintf(human_log_file, "%13d was written to the data structure at %d microseconds.\n", payload, temp_time.wMilliseconds);

<         pthread_mutex_unlock(&data->mutex);
<         sem_post(&data->full);
---
>         ReleaseMutex(&data->mutex);
>         ReleaseSemaphore(&data->full, 1, NULL);

<     return NULL;                              <--Different return types for Linux and Windows.
---
>     return 0;

< void *consumer(void *indata)
---
> unsigned __stdcall consumer(void *indata)

<     t_data *data = indata;
---
>    t_data *data = (t_data*) indata;
```

```
>     int temp;

<         int temp = 0;
<         sleep(random()/1000000000);                     //generates a sleep from 0 to 21
---
>
>         temp = 0;
>         Sleep(rand()/100);                      //generates a Sleep from 0 to 21

<             sem_wait(&data->empty);
---
>             WaitForMultipleObjects(MAX_SIZE, &data->empty, FALSE, INFINITE);

<         pthread_mutex_lock(&data->mutex);
---
>         WaitForSingleObject(&data->mutex, INFINITE);

<

<             gettimeofday(&current, NULL);
<             temp_time = time_stamp(start, current);
---
>             GetSystemTime(&current);
>             temp_time.wMilliseconds = start.wMilliseconds - current.wMilliseconds;

<             fprintf(human_log_file, "%13d was removed from the data structure at %d microseconds.\n", temp, temp_time.tv_usec);
---
>             fprintf(human_log_file, "%13d was removed from the data structure at %d microseconds.\n", temp, temp_time.wMilliseconds);

<         pthread_mutex_unlock(&data->mutex);
<         sem_post(&data->full);
---
>         ReleaseMutex(&data->mutex);
>         ReleaseSemaphore(&data->full, 1, NULL);

<     return NULL;
---
>     return 0;
```