Abstract

This lab aims to run the producer and a consumer at the same time (concurrently) while managing the buffer that both cons and pros are using with the help of pthreads.

1 Introduction

The lab was requiring the implementation of the producer consumer problem and to add certain pthreads in order to run them concurrently without facing problems such as race condition. The consumer consumes (dequeues) from the queue representing the buffer and must wait (Sleep) while the queue (buffer) is empty. However, the producer produces (enqueues) in the queue and must wait (sleep) when the queue is full.

2 Implementation

We added a mutex lock allow synchronization between producer and consumer (pthread$_{m}utex_{l}ock(fifo- > lock);$)
Then we added another pthread (pthread$_{c}ond_{w}ait(fifo- > notfull, fifo- > lock)$ which waits for a signal from the consumer if the buffer is full
Another pthread was added before terminating, (pthread$_{c}ond_{s}ignal(fifo- >$ notEmpty);) that makes all producers signal to all consumers
we added a pthread (pthread$_{c}ond_{w}ait(fifo- > notEmpty, fifo- > lock))which$ indicates that if buffer is empty then wait for a signal from producer.
Another pthread we added (pthread$_{c}ond_{s}ignal(fifo- > notEmpty)similarto$ the previous one but here we are sending a signal from a consumer to all other consumers to make sure they all terminated.
we used struct timeval stop, start; to measure the time of execution in milliseconds.
gettimeofday(start,NULL); to start the clock. gettimeofday(stop, NULL) to end the clock.
We gave the producers and consumers varibales and gave them random values from 0 to 10 for the sake of testing.

```
#include <pthread.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <time.h>
#include <sys/time.h>

/*
 * Define constants for how big the shared queue should be and how
 * much total work the produceers and consumers should perform
 */

#define QUEUESIZE 5
#define WORKMAX 30

/*
 * These constants specify how much CPU bound work the producer and
 * consumer do when processing an item. They also define how long each
 * blocks when producing an item. Work and blocking are implemented
 * int he do_work() routine that uses the msleep() routine to block
```

```
 * for at least the specified number of milliseconds.
 */
#define PRODUCER_CPU   25
#define PRODUCER_BLOCK 10
#define CONSUMER_CPU   25
#define CONSUMER_BLOCK 10

/****************************************************
 *   Shared Queue Related Structures and Routines    *
 ****************************************************/
typedef struct {
  int buf[QUEUESIZE];   /* Array for Queue contents, managed as circular queue */
  int head;             /* Index of the queue head */
  int tail;             /* Index of the queue tail, the next empty slot */

  int full;             /* Flag set when queue is full  */
  int empty;            /* Flag set when queue is empty */

  pthread_mutex_t *lock;     /* Mutex protecting this Queue's data */ //
  pthread_cond_t  *notFull;   /* Used by producers to await room to produce*/
  pthread_cond_t  *notEmpty;  /* Used by consumers to await something to consume*/
} queue;

/*
 * Create the queue shared among all producers and consumers
 */
queue *queueInit (void)
{
  queue *q;

  /*
   * Allocate the structure that holds all queue information
   */
  q = (queue *)malloc (sizeof (queue));
  if (q == NULL) return (NULL);

  /*
   * Initialize the state variables. See the definition of the Queue
   * structure for the definition of each.
   */
  q->empty = 1;
  q->full  = 0;

  q->head  = 0;
  q->tail  = 0;

  /*
   * Allocate and initialize the queue mutex
   */
  q->lock = (pthread_mutex_t *) malloc (sizeof (pthread_mutex_t));
  pthread_mutex_init (q->lock, NULL);

  /*
   * Allocate and initialize the notFull and notEmpty condition
   * variables
   */
```

```
  q->notFull = (pthread_cond_t *) malloc (sizeof (pthread_cond_t));
  pthread_cond_init (q->notFull, NULL);

  q->notEmpty = (pthread_cond_t *) malloc (sizeof (pthread_cond_t));
  pthread_cond_init (q->notEmpty, NULL);

  return (q);
}

/*
 * Delete the shared queue, deallocating dynamically allocated memory
 */
void queueDelete (queue *q)
{
  /*
   * Destroy the mutex and deallocate its memory
   */
  pthread_mutex_destroy (q->lock);
  free (q->lock);

  /*
   * Destroy and deallocate the condition variables
   */
  pthread_cond_destroy (q->notFull);
  free (q->notFull);

  pthread_cond_destroy (q->notEmpty);
  free (q->notEmpty);

  /*
   * Deallocate the queue structure
   */
  free (q);
}

void queueAdd (queue *q, int in)
{

  /*
   * Put the input item into the free slot
   */
  q->buf[q->tail] = in;
  q->tail++;

  /*
   * wrap the value of tail around to zero if we reached the end of
   * the array. This implements the circularity of the queue inthe
   * array.
   */
  if (q->tail == QUEUESIZE)
    q->tail = 0;

  /*
   * If the tail pointer is equal to the head, then the enxt empty
   * slot in the queue is occupied and the queue is FULL
   */
```

```
  if (q->tail == q->head)
    q->full = 1;

  /*
   * Since we just added an element to the queue, it is certainly not
   * empty.
   */
  q->empty = 0;

  return;
}

void queueRemove (queue *q, int *out)
{
  /*
   * Copy the element at head into the output variable and increment
   * the head pointer to move to the next element.
   */
  *out = q->buf[q->head];
  q->head++;

  /*
   * Wrap the index around to zero if it reached the size of the
   * array. This implements the circualrity of the queue int he array.
   */
  if (q->head == QUEUESIZE)
    q->head = 0;

  /*
   * If head catches up to tail as we delete an item, then the queue
   * is empty.
   */
  if (q->head == q->tail)
    q->empty = 1;

  /*
   * since we took an item out, the queue is certainly not full
   */
  q->full = 0;

  return;
}

/*****************************************************
 *   Producer and Consumer Structures and Routines    *
 *****************************************************/
/*
 * Argument struct used to pass consumers and producers thier
 * arguments.
 *
 * q     - arg provides a pointer to the shared queue.
 *
 * count - arg is a pointer to a counter for this thread to track how
 *         much work it did.
 *
 * tid   - arg provides the ID number of the producer or consumer,
```

```
 *         whichis also its index into the array of thread structures.
 *
 */
typedef struct {
  queue *q;
  int   *count;
  int    tid;
} pcdata;

int memory_access_area[100000];


/*
 * Sleep for a specified number of milliseconds. We use this to
 * simulate I/O, since it will block the process. Different lengths fo
 * sleep simulate interaction with different devices.
 */
void msleep(unsigned int milli_seconds)
{
  struct timespec req = {0}; /* init struct contents to zero */
  time_t          seconds;

  /*
   * Convert number of milliseconds input to seconds and residual
   * milliseconds to handle the cse where input is more than one
   * thousand milliseconds.
   */
  seconds        = (milli_seconds/1000);
  milli_seconds  = milli_seconds - (seconds * 1000);

  /*
   * Fill in the time_spec's seconds and nanoseconds fields. Note we
   * multiply millisconds by 10^6 to convert to nanoseconds.
   */
  req.tv_sec  = seconds;
  req.tv_nsec = milli_seconds * 1000000L;

  /*
   * Sleep for the required period. The first parameter specifies how
   * long. In theory this thread can be awakened before the period is
   * over, perhaps by a signal. If so the timespec specified by the
   * second argument is filled in with how much time int he original
   * request is left. We use the same one. If this happens, we just
   * call nanosleep again to sleep for what remains of the origianl
   * request.
   */
  while(nanosleep(&req, &req)==-1) {
    printf("restless\n");
    continue;
  }

}

/*
 * Simulate doing work.
 */
```

```
void do_work(int cpu_iterations, int blocking_time)
{
  int i;
  int j;
  int local_var;

  local_var = 0;
  for (j = 0; j < cpu_iterations; j++ ) {
    for (i = 0; i < 1000; i++ ) {
      local_var = memory_access_area[i];
    }
  }

  if ( blocking_time > 0 ) {
    msleep(blocking_time);
  }
}

void *producer (void *parg)
{
  queue  *fifo;
  int     item_produced;
  pcdata *mydata;
  int     my_tid;
  int    *total_produced;

  mydata = (pcdata *) parg;

  fifo           = mydata->q;
  total_produced = mydata->count;
  my_tid         = mydata->tid;

  /*
   * Continue producing until the total produced reaches the
   * configured maximum
   */
  while (1) {
    /*
     * Do work to produce an item. Tthe get a slot in the queue for
     * it. Finally, at the end of the loop, outside the critical
     * section, announce that we produced it.
     */
    do_work(PRODUCER_CPU, PRODUCER_BLOCK);

    /*
     * If the queue is full, we have no place to put anything we
     * produce, so wait until it is not full.
     */

     pthread_mutex_lock (fifo->lock); //mutex lock for synchronizing the communication betwee

    while (fifo->full && *total_produced != WORK_MAX) {

      printf ("prod %d:  FULL.\n", my_tid);
      pthread_cond_wait(fifo->notFull, fifo->lock); //if buffer full then wait for signal from
```

```
    }

    /*
     * Check to see if the total produced by all producers has reached
     * the configured maximum, if so, we can quit.
     */
    if (*total_produced >= WORK_MAX) {
      break;
    }

    /*
     * OK, so we produce an item. Increment the counter of total
     * widgets produced, and add the new widget ID, its number, to the
     * queue.
     */
    item_produced = (*total_produced)++;
    queueAdd (fifo, item_produced);

    /*
     * Announce the production outside the critical section
     */
    printf("prod %d:  %d.\n", my_tid, item_produced);
    if (*total_produced<WORK_MAX)
    {
    pthread_cond_signal (fifo->notEmpty); //when item is produced signal to consumer //
    pthread_mutex_unlock (fifo->lock); //
    }

    else // if items produced reach max capacity then the producers job is over and break
    {
        break;
    }

  }

  printf("prod %d:  exited\n", my_tid);
  pthread_cond_signal(fifo->notEmpty); //before terminating all producers signal to consumers
  pthread_mutex_unlock (fifo->lock);
  return (NULL);
}

void *consumer (void *carg)
{
  queue  *fifo;
  int     item_consumed;
  pcdata *mydata;
  int     my_tid;
  int    *total_consumed;

  mydata = (pcdata *) carg;

  fifo           = mydata->q;
  total_consumed = mydata->count;
  my_tid         = mydata->tid;

  /*
```

```
   * Continue producing until the total consumed by all consumers
   * reaches the configured maximum
   */
  while (1) {
    /*
     * If the queue is empty, there is nothing to do, so wait until it
     * si not empty.
     */

     pthread_mutex_lock (fifo->lock); //mutex lock for synchronizing the communication betwee

    while (fifo->empty && *total_consumed != WORK_MAX) {
      printf ("con %d:   EMPTY.\n", my_tid);
      pthread_cond_wait (fifo->notEmpty, fifo->lock); //if buffer is empty then wait for a sig
    }

    /*
     * If total consumption has reached the configured limit, we can
     * stop
     */



    if (*total_consumed >= WORK_MAX) {
      break;
    }

    /*
     * Remove the next item from the queue. Increment the count of the
     * total consumed. Note that item_consumed is a local copy so this
     * thread can retain a memory of which item it consumed even if
     * others are busy consuming them.
     */
    queueRemove (fifo, &item_consumed);
    (*total_consumed)++;


    /*
     * Do work outside the critical region to consume the item
     * obtained from the queue and then announce its consumption.
     */
    do_work(CONSUMER_CPU,CONSUMER_CPU);
    printf ("con %d:   %d.\n", my_tid, item_consumed);
    pthread_cond_signal(fifo->notFull);
    pthread_mutex_unlock (fifo->lock);

  }

  printf("con %d:   exited\n", my_tid);
  pthread_cond_signal(fifo->notEmpty); //a signal from a consumer to other consumers to make s
  pthread_mutex_unlock(fifo->lock);
  return (NULL);
}

/**********************************************
 *   Main allocates structures, creates threads,   *
```

```
 *   waits to tear down.                                *
 ****************************************************/
int main (int argc, char *argv[])
{
  pthread_t *con;
  int        cons;
  int       *concount;

  queue     *fifo;
  int        i;

  pthread_t *pro;
  int       *procount;
  int        pros;

  pcdata    *thread_args;

  /*
   * Check the number of arguments and determine the numebr of
   * producers and consumers
   */

   struct timeval stop, start; //used to measure the time of execution in microseconds

   gettimeofday(&start,NULL); //start of clock


  if (argc != 3) {
    printf("Usage: producer_consumer number_of_producers number_of_consumers\n\n");
    //exit(0);
  }

  srand(time(0));

  pros = rand()%10; //generates a random number of producers between 1 and 10
  cons = rand()%10; //generates a random number of consumers between 1 and 10

  if (pros!=0 && cons!=0) //making sure we have at least one producer and at least one consume
  {

  printf ("Number of Producers: %d\n", pros); //displaying the number of Producers
  printf("Number of Consumers: %d\n\n", cons); //displaying number of Consumers

  /*
   * Create the shared queue
   */
  fifo = queueInit ();
  if (fifo == NULL) {
    fprintf (stderr, "main: Queue Init failed.\n");
    exit (1);
  }

  /*
   * Create a counter tracking how many items were produced, shared
   * among all producers, and one to track how many items were
   * consumed, shared among all consumers.
```

```
 */
procount = (int *) malloc (sizeof (int));
if (procount == NULL) {
  fprintf(stderr, "procount allocation failed\n");
  exit(1);
}

concount = (int *) malloc (sizeof (int));
if (concount == NULL) {
  fprintf(stderr, "concount allocation failed\n");
  exit(1);
}

/*
 * Create arrays of thread structures, one for each producer and
 * consumer
 */
pro = (pthread_t *) malloc (sizeof (pthread_t) * pros);
if (pro == NULL) {
  fprintf(stderr, "pros\n");
  exit(1);
}

con = (pthread_t *) malloc (sizeof (pthread_t) * cons);
if (con == NULL) {
  fprintf(stderr, "cons\n");
  exit(1);
}

/*
 * Create the specified number of producers
 */
for (i=0; i<pros; i++){
  /*
   * Allocate memory for each producer's arguments
   */
  thread_args = (pcdata *)malloc (sizeof (pcdata));
  if (thread_args == NULL) {
    fprintf (stderr, "main: Thread_Args Init failed.\n");
    exit (1);
  }

  /*
   * Fill them in and then create the producer thread
   */
  thread_args->q     = fifo;
  thread_args->count = procount;
  thread_args->tid   = i;
  pthread_create (&pro[i], NULL, producer, thread_args);
}

/*
 * Create the specified number of consumers
 */
for (i=0; i<cons; i++){
  /*
```

```
     * Allocate space for next consumer's args
     */
    thread_args = (pcdata *)malloc (sizeof (pcdata));
    if (thread_args == NULL) {
      fprintf (stderr, "main: Thread_Args Init failed.\n");
      exit (1);
    }

    /*
     * Fill them in and create the thread
     */
    thread_args->q     = fifo;
    thread_args->count = concount;
    thread_args->tid   = i;
    pthread_create (&con[i], NULL, consumer, thread_args);
  }

  /*
   * Wait for the create producer and consumer threads to finish
   * before this original thread will exit. We wait for all the
   * producers before waiting for the consumers, but that is an
   * unimportant detail.
   */
  for (i=0; i<pros; i++)
    pthread_join (pro[i], NULL);
  for (i=0; i<cons; i++)
    pthread_join (con[i], NULL);

  /*
   * Delete the shared fifo, now that we know there are no users of
   * it. Since we are about to exit we could skip this step, but we
   * put it here for neatness' sake.
   */
  queueDelete (fifo);

  gettimeofday(&stop, NULL); //end of clock

  printf("\nTime: %lu", (stop.tv_sec-start.tv_sec)*1000000 + stop.tv_usec - start.tv_usec); //

  }

  return 0;
}
```

3 Experimental Platform

We used Visusal Studio to write and execute the code.

4 Results

Number of Producers: 2
Number of Consumers: 6
con 3: EMPTY.
con 0: EMPTY.
con 1: EMPTY.

con 2: EMPTY.
con 4: EMPTY. con 5: EMPTY. prod 1: 0. prod 0: 1. con 3: 0. con 3: 1. con 3: EMPTY. prod 1: 2. con 1: 2. con 1: EMPTY. prod 0: 3. prod 1: 4. con 2: 3. con 2: 4. con 2: EMPTY. prod 1: 5. prod 0: 6. con 0: 5. con 0: 6. con 0: EMPTY. con 3: EMPTY. prod 1: 7. prod 0: 8. con 1: 7. con 1: 8. con 1: EMPTY. con 2: EMPTY. prod 0: 9. prod 1: 10. con 5: 9. con 5: 10. con 5: EMPTY. con 3: EMPTY. prod 0: 11. prod 1: 12. con 4: 11. con 4: 12. con 4: EMPTY. con 1: EMPTY. prod 0: 13. prod 1: 14. con 0: 13. con 0: 14. con 0: EMPTY. con 3: EMPTY. prod 0: 15. prod 1: 16. con 2: 15. con 2: 16. con 2: EMPTY. con 4: EMPTY. prod 0: 17. prod 1: 18. con 0: 17. con 0: 18. con 0: EMPTY. con 5: EMPTY. con 3: EMPTY. prod 0: 19. prod 1: 20. con 1: 19. con 1: 20. con 1: EMPTY. prod 0: 21. con 0: 21. con 0: EMPTY. con 2: EMPTY. prod 1: 22. prod 0: 23. con 4: 22. con 4: 23. con 4: EMPTY. prod 1: 24. prod 0: 25. con 0: 24. con 0: 25. con 0: EMPTY. con 5: EMPTY. prod 1: 26. prod 0: 27. con 3: 26. con 3: 27. con 3: EMPTY. con 4: EMPTY. prod 1: 28. prod 0: 29. prod 0: exited con 5: 28. con 5: 29. con 5: exited con 1: exited con 3: exited prod 1: exited con 4: exited con 0: exited con 2: exited
Time: 830552
Number of Producers: 4
Number of Consumers: 9
con 0: EMPTY. con 3: EMPTY. con 2: EMPTY. con 5: EMPTY. con 7: EMPTY. con 8: EMPTY. con 1: EMPTY. con 6: EMPTY. con 4: EMPTY. prod 0: 0. prod 2: 1. prod 1: 2. con 2: 0. con 2: 1. con 2: 2. con 2: EMPTY. con 3: EMPTY. prod 0: 3. con 5: 3. con 5: EMPTY. con 0: EMPTY. prod 3: 4. prod 2: 5. prod 0: 6. con 8: 4. con 8: 5. con 8: 6. con 8: EMPTY. prod 2: 7. prod 0: 8. prod 3: 9. con 7: 7. con 7: 8. con 7: 9. con 7: EMPTY. con 4: EMPTY. prod 2: 10. prod 0: 11. con 3: 10. con 3: 11. con 3: EMPTY. prod 3: 12. con 0: 12. con 0: EMPTY. con 1: EMPTY. con 6: EMPTY. con 5: EMPTY. prod 2: 13. con 8: 13. con 8: EMPTY. con 2: EMPTY. prod 1: 14. prod 3: 15. prod 0: 16. con 3: 14. con 3: 15. con 3: 16. con 3: EMPTY. prod 2: 17. prod 1: 18. con 1: 17. con 1: 18. con 1: EMPTY. con 7: EMPTY. con 4: EMPTY. con 0: EMPTY. prod 0: 19. prod 1: 20. prod 2: 21. con 8: 19. con 8: 20. con 8: 21. con 8: EMPTY. con 5: EMPTY. prod 0: 22. con 2: 22. con 2: EMPTY. prod 3: 23. con 3: 23. con 3: EMPTY. prod 1: 24. con 1: 24. con 1: EMPTY. prod 0: 25. con 7: 25. con 7: EMPTY. prod 3: 26. prod 2: 27. con 0: 26. con 0: 27. con 0: EMPTY. prod 0: 28. con 8: 28. con 8: EMPTY. prod 1: 29. prod 1: exited con 5: 29. con 5: exited con 6: exited con 3: exited prod 0: exited con 4: exited con 0: exited prod 2: exited con 2: exited con 1: exited con 8: exited con 7: exited prod 3: exited
Time: 823320

5 Conclusion

To sum up, the problem of race conditions was solved by adding the needed pthreads by managing the producers and consumers that are running concurently on the same buffer, by making them wait and send signals.
Repository link: https://github.com/JicksonHD/$CSC_447_LAB1 - Parallele-$