CS180 Experiment 3

March 13, 2017

1 A Motivating Example

Notes:

1. Read the file `test-add.c`. How many threads does this code create? What does each thread do?
2. Compile the code using the command `gcc test-add.c`.
3. Run the program using `a.exe 1000 2` and `a.exe 100000 2`. Are the results as expected? Why or why not?

2 Peterson's Solution

Notes:

1. Read textbook page 229 for explanation of the `Peterson's` solution.
2. Study `peterson.c` and `peterson.h`. What is the instruction `mfence` for? Recall the notes and explanation in lecture.
3. `test-add-petersons.c` is an implementation using Peterson's algorithm to ensure mutual exclusion for the motivating example. Compile the code using

   ```
   gcc -o test-add-petersons.exe test-add-petersons.c peterson.c
   ```

   Run the program. Recompile and run the program with the assembly code inlining `mfence` commented. What is the difference in the output?

3 Test and Set Lock

Notes:

1. Read textbook page 231 for explanation of the `TestAndSet` instruction.
2. Study `tsl.c` and `tsl.h`. How is the test and set implemented? You may need to research on the `xchgl` instruction. item `test-add-tsl.c` is an implementation using the algorithm to ensure mutual exclusion for the motivating example. Compile the code using

   ```
   gcc -o test-add-tsl.exe test-add-tsl.c tsl.c
   ```

   Run the program and verify that it works.

4 Semaphore

Notes:

1. Study `test-add-sempahores.c`. What does `sem_init` do? (Hint: use `man sem_init` to find out. Write down anything you don't understand on the man page for discussion with your friend.) Perform similar study for `sem_init` and `sem_post`.

2. Compile the code using

   ```
   gcc -o test-add-semaphores.exe test-add-semaphores.c
   ```

   Run the program and verify that it works.

5 Test-Add Experiment

In this section, we perform to compare the performance of these programs with reference to the iteration of the loop inside each thread.

1. Open the Task Manager window. Next, run the following commands inside a cygwin environment. We assume that the working directory is the directory containing all the executables compiled above. Observe how the CPU usage changes when you run the programs. Note the difference in the "real time" (which is elasped time) reported by the time command. Could you account for the differences between these programs in terms of why they report differenting timings and CPU usage patterns?

   ```
   time ./test-add-semaphores.exe 10000000 2
   time ./test-add-tsl.exe 10000000 2
   time ./test-add-petersons.exe 10000000 2
   ```

6 Using TSL and semaphore to solve the Producer–Consumer Problem

1. Study the code to be found in `simple-producer-consumer-tsl.c` and `simple-producer-consumer-semaphores.c`. The comments within the code should be sufficient for illuminating the working of the code.

2. Compile using the following commands:

   ```
   gcc simple-producer-consumer-tsl.c tsl.c -o \
    simple-producer-consumer-tsl.exe
   gcc simple-producer-consumer-sempahores.c tsl.c -o \
    simple-producer-consumer-semaphores.exe
   ```

3. Run the program with the following arguments in a cygwin environment. Again, note the difference in the elapsed time and the CPU usage stats in the Task Manager window. Could you account for these differences?

   ```
   time ./simple-producer-consumer-tsl.exe 10 40
   time ./simple-producer-consumer-sempahores.exe 10 40
   ```

7 Code Listings

```
#ifndef PETERSON_H
#define PETERSON_H
void get_mutex(int);
void release_mutex(int);
#endif
```

peterson.h

```
#include <stdio.h>
static volatile int interested[2]={0};
static volatile int turn;
static int count=0;
void get_mutex(int pid)
{
    interested[pid]=1;
    int other = pid?0:1;
    turn=pid;
    //__asm__ ("mfence");
    while(turn == pid && interested[other]);
}

void release_mutex(int pid)
{
  interested[pid]=0;
}
```

peterson.c

```
#ifndef TSL_H
#define TSL_H
typedef struct Lock {
    unsigned int locked;
} Lock;

void acquire(Lock *lck);
void release(Lock *lck);
#endif
```

tsl.h

```
#include "tsl.h"



static inline unsigned int TSL(volatile unsigned int* ptr)
{
/*
   xchgl result, *ptr
   xchgl exchanges the values of its two operands, while
   locking the memory bus to exclude other operations.
   You are not required to understand the assembly inlining.
   the xchgl assembly code performs the swap function
   in a single instruction. Swapping between the register
   containing result and the address "ptr" in
   one atomic and indivisable instruction.
*/

    int result;
    asm volatile("lock;"
                 "xchgl %0, %1;"
                 : "=r"(result), "=m"(*ptr)
                 : "0"(1), "m"(*ptr)
                 : "memory");
    return result;
}

void acquire(Lock *lck) {
    while(TSL(&(lck->locked)) != 0)
    ;
}

void release(Lock *lck) {
    lck->locked = 0;
}
```

tsl.c

```
#define _REENTRANT
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <time.h>

int count = 0;

void *add_many(void *arg)
{
    int num_of_rounds = (int) arg;
    int i;
    for(i=0;i<num_of_rounds; i++)
        count++;
}

int main(int argc, char**argv)
{
    int num, flags;
    int seq_or_thd;
    pthread_attr_t a;
    pthread_t tid[2]; //[0] - producer [1] - reader

    if(argc!=3)
    {
        fprintf(stderr, "Usage: %s <num_of_times> <0: sequential OR 1: threads>\n",
            argv[0]);
        exit(-1);
    }

    num = atoi(argv[1]);
    seq_or_thd = atoi(argv[2]);

    if(seq_or_thd)
    {
        //Threads

        flags = PTHREAD_SCOPE_SYSTEM;
        pthread_attr_init(&a);
        pthread_attr_setscope(&a, flags);

        pthread_create(&tid[0], &a, add_many, (void*)num);
        pthread_create(&tid[1], &a, add_many, (void*)num);
        pthread_join(tid[0], NULL);
        pthread_join(tid[1], NULL);
    }
    else
    {
        //Sequential example
        add_many((void*)num);
        add_many((void*)num);
    }

    printf("count is %d\n", count);
}
```

test–add.c

```
1  //Macro for POSIX threads
   #define _REENTRANT
3  //Including the pthread headers
   #include <pthread.h>
5  //Standard stuff
   #include <stdio.h>
7  #include <stdlib.h>
   #include <unistd.h>
9  #include <time.h>

11 #include "tsl.h"

13 int count = 0;
   Lock the_lock;
15
   void *add_many(void *arg)
17 {
       int num = (int)arg;
19     int i;
       for(i=0;i<num; i++)
21     {
           acquire(&the_lock);
23         count++;
           release(&the_lock);
25     }
   }
27
   int main(int argc, char**argv)
29 {
       int num, flags;
31     int seq_or_thd;
       pthread_attr_t a;
33     pthread_t tid[2]; //[0] - producer [1] - reader

35     if(argc!=3)
       {
37         fprintf(stderr, "Usage: %s <num_of_times> <0: sequential OR 1: threads>\n",
               argv[0]);
           exit(-1);
39     }

41     num = atoi(argv[1]);
       seq_or_thd = atoi(argv[2]);
43
       if(seq_or_thd)
45     {
           flags = PTHREAD_SCOPE_SYSTEM;
47         pthread_attr_init(&a);
           pthread_attr_setscope(&a, flags);
49         the_lock.locked = 0;
           pthread_create(&tid[0], &a, add_many, (void*)num);
51         pthread_create(&tid[1], &a, add_many, (void*)num);
           pthread_join(tid[0], NULL);
53         pthread_join(tid[1], NULL);
       }
55     else
       {
57         add_many((void*)num);
           add_many((void*)num);
59     }
       printf("count is %d\n", count);
61 }
```

test–add–tsl.c

```
#define _REENTRANT
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <time.h>
#include "peterson.h"

int count = 0;

struct num_and_thread_id
{
    int num;
    int id;
};

void *add_many(void *arg)
{
    struct num_and_thread_id *s_pointer = (struct num_and_thread_id *) arg;
    int i;
    for(i=0;i<s_pointer->num; i++)
    {
        get_mutex(s_pointer->id);
        count++;
        release_mutex(s_pointer->id);
    }
}

int main(int argc, char**argv)
{
    int num, flags;
    int seq_or_thd;
    pthread_attr_t a;
    pthread_t tid[2]; //[0] - producer [1] - reader

    if(argc!=3)
    {
        fprintf(stderr, "Usage: %s <num_of_times> <0: sequential OR 1: threads>\n",
            argv[0]);
        exit(-1);
    }

    num = atoi(argv[1]);
    seq_or_thd = atoi(argv[2]);

    if(seq_or_thd)
    {
        //Threads
        struct num_and_thread_id param, param2;

        flags = PTHREAD_SCOPE_SYSTEM;
        pthread_attr_init(&a);
        pthread_attr_setscope(&a, flags);

        param.num = num;
        param.id = 0;

        pthread_create(&tid[0], &a, add_many, (void*)&param);

        param2.num = num;
        param2.id = 1;

        pthread_create(&tid[1], &a, add_many, (void*)&param2);
        pthread_join(tid[0], NULL);
        pthread_join(tid[1], NULL);
    }
    else
```

```
67      {
            struct num_and_thread_id param;
69          param.num = num;
            param.id = 0;
71          //Sequential example
            add_many((void*)&param);
73          add_many((void*)&param);
        }
75
        printf("count is %d\n", count);
77  }
```

test–add–petersons.c

```c
#define _REENTRANT
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <time.h>
#include <semaphore.h>


int count = 0;

sem_t common_sem;

void *add_many(void *arg)
{
    int num = (int)arg;
    int i;
    for(i=0;i<num; i++)
    {
        sem_wait(&common_sem);
        count++;
        sem_post(&common_sem);
    }
}

int main(int argc, char**argv)
{
    int num, flags;
    int seq_or_thd;
    pthread_attr_t a;
    pthread_t tid[2]; //[0] - producer [1] - reader

    if(argc!=3)
    {
        fprintf(stderr, "Usage: %s <num_of_times> <0: sequential OR 1: threads>\n",
            argv[0]);
        exit(-1);
    }

    num = atoi(argv[1]);
    seq_or_thd = atoi(argv[2]);

    sem_init(&common_sem, 0, 1);

    if(seq_or_thd)
    {
        flags = PTHREAD_SCOPE_SYSTEM;
        pthread_attr_init(&a);
        pthread_attr_setscope(&a, flags);


        pthread_create(&tid[0], &a, add_many, (void*)num);
        pthread_create(&tid[1], &a, add_many, (void*)num);

        pthread_join(tid[0], NULL);
        pthread_join(tid[1], NULL);
    }
    else
    {
        //Sequential example
        add_many((void*)num);
        add_many((void*)num);
    }

    printf("count is %d\n", count);
}
```

test–add–semaphores.c

```
#define _REENTRANT
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <time.h>
#include <semaphore.h>
#include "tsl.h"
#define BUFFER_SIZE 256

/*
    Global Variables:
    1. Of course not supposed to have global variables in any "real" code.
    2. For purposes of demo only.
    3. produced_buffer is the "shared buffer" between all producers and consumers.
       Producers write into it, Consumers read from it.
    4. Two locks:
        a. space_count_lock - ensure mutual exclusive access to space_count variable
           .
        b. produced_count_lock - ensure mutual exclusive access to produced_count
           variable.
        c. buffer_lock - ensure mutual exclusive access to produced_buffer.
    5. consumed_buffer - a shared buffer among the consumers for storing whatever
                         they obtained from the buffer.
    6. spacecount is the variable keeping track of the number of produced items in
        the
       buffer not consumed.
*/
int produced_count=0, space_count=BUFFER_SIZE;
int produced_buffer[BUFFER_SIZE];
int *consumed_buffer;
Lock space_count_lock, produced_count_lock;
Lock buffer_lock;

/*
    Consumer thread function.
    1. out - the index in the produced_buffer that can be consumed.
    2. consumed_in - the index in the consumed_buffer that can be written to.
    3. out and consumed_in are "shared" by all consumer threads.
       Note the static declaration.
    4. consumed, num and i are local variables. num is the number of times
       the loop will iterate.
*/

void *consumer(void *arg)
{
    static unsigned char out=0;
    static unsigned consumed_in = 0;
    int consumed;
    int num = (int) arg;
    int i;

    for(i=0; i< num ; i++)
    {
        /*
            Read the produced_count variable.
            Spin if produced_count is 0;
            (i.e., buffer is empty).
            Otherwise, decrement produced_count and move into
            consumption.
        */
        while(1)
        {
            int temp;
            acquire(&produced_count_lock);
            temp=produced_count;
            if(temp>0)
```

```
65                  produced_count--;
                release(&produced_count_lock);
67              if(temp!=0)
                    break;
69          }
            /*
71              Get mutual exclusive access to produced_buffer
            */
73          acquire(&buffer_lock);
            consumed = produced_buffer[out++];
75          consumed_buffer[consumed_in++] = consumed;
            release(&buffer_lock);
77
            /*
79              Get mutual exclusive access to space_count
                and increment it because there's one more
81              space available now.
            */
83          acquire(&space_count_lock);
            space_count++;
85          release(&space_count_lock);
        }
87  }

89  /*
        Producer thread function.
91      1. in - the index in the produced_buffer that can be written to.
        2. produced_num - the currently produced number.
93      3. in and produced_num are "shared" by all consumer threads.
           Note the static declaration.
95      4. num and i are local variables. num is the number of times
           the loop will iterate.
97      5. Note that because produced_num is put within a mutual exclusive
           section, it will increase from  1 to (number of producers threads * num).
99  */

101 void *producer(void *arg)
    {
103     static unsigned char in = 0;
        static int produced_num = 0;
105     int num = (int) arg;
        int i =0;
107
        for(i=0; i< num ; i++)
109     {
            /*
111             Read the space_count variable.
                Spin if space_count is 0;
113             (i.e., buffer is full).
                Otherwise, increment space_count and
115             enter production.
            */
117         while(1)
            {
119             int temp;
                acquire(&space_count_lock);
121             temp=space_count;
                if(temp>0)
123                 space_count--;
                release(&space_count_lock);
125             if(temp!=0)
                {
127                 break;
                }
129         }
            /*
131             Get mutual exclusive access to produced_buffer
            */
```

```
133         acquire(&buffer_lock);
            produced_buffer[in++] = ++produced_num;
135         release(&buffer_lock);

137         /*
                Get mutual exclusive access to produced_count
139             and increment it because there's one more
                produced item now.
141         */
            acquire(&produced_count_lock);
143         produced_count++;
            release(&produced_count_lock);
145     }
    }
147
    /*
149     The main function.
    */
151

153 int main(int argc, char**argv)
    {
155     int num, flags;
        int num_of_consumers, num_of_producers;
157     int i;
        pthread_attr_t a;
159     pthread_t *producer_thds, *consumer_thds;

161     /*
            Error message indicating usage of the function.
163     */


165
        if(argc!=3)
167     {
            fprintf(stderr, "Usage: %s <num_of_times> <num of consumer-producer thds>\n",
                 argv[0]);
169         exit(-1);
        }
171
        /*
173         Obtaining the arguments of the program.
        */
175
        num = atoi(argv[1]);
177     num_of_consumers = atoi(argv[2]);
        num_of_producers = atoi(argv[2]);
179
        /*
181         POSIX thread initialization.
            setting flag to PTHREAD_SCOPE_SYSTEM ensure that the threads
183         created are kernel-level threads i.e., visible to the
            kernel.
185     */

187     flags = PTHREAD_SCOPE_SYSTEM;
        pthread_attr_init(&a);
189     pthread_attr_setscope(&a, flags);

191     /*
            Initializing the locks.
193         0 means not locked at the beginning.
            Read tsl.c for more information.
195     */

197     space_count_lock.locked=0;
        produced_count_lock.locked=0;
199     buffer_lock.locked=0;
```

```
201     /*
            Allocating the memory buffer for consumed_buffer.
203         Basically, the total produced buffer size would be
            number of producers*num.
205     */
    consumed_buffer = (int *) malloc(num_of_producers * num * sizeof(int));
207

209     /* Initializing the thread id array*/
    producer_thds = (pthread_t *)malloc(num_of_producers * sizeof(pthread_t));
211     consumer_thds = (pthread_t *)malloc(num_of_consumers * sizeof(pthread_t));

213     /*Creating all the producer and consumer threads*/
    for(i=0;i<num_of_producers;i++)
215         pthread_create(&producer_thds[i], &a, producer, (void*)num);
    for(i=0; i<num_of_consumers; i++)
217         pthread_create(&consumer_thds[i], &a, consumer, (void*)num);

219     /*wait for all producer and consumer threads to complete*/
    for(i=0;i<num_of_producers;i++)
221         pthread_join(producer_thds[i], NULL);
    for(i=0;i<num_of_consumers; i++)
223         pthread_join(consumer_thds[i], NULL);

225     /*check consumed_buffer */
    for(i=1; i<num_of_producers *num; i++)
227     {
            if(consumed_buffer[i]!=consumed_buffer[i-1]+1)
229         {
                printf("Result is wrong at index consumed_buffer[%d].\n", i);
231             printf("consumed_buffer[%d] is %d.\n", i-1, consumed_buffer[i-1]);
                printf("consumed_buffer[%d] is %d.\n", i, consumed_buffer[i]);
233             return -1;
            }
235     }
    printf("Result is correct!\n");
237     free(consumer_thds);
    free(producer_thds);
239     free(consumed_buffer);
    return 0;
241 }
```

simple–producer–consumer–tsl.c

```
#define _REENTRANT
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <time.h>
#include <semaphore.h>
#define BUFFER_SIZE 256

/*
    Global Variables:
    1. Of course not supposed to have global variables in any "real" code.
    2. For purposes of demo only.
    3. produced_buffer is the "shared buffer" between all producers and consumers.
       Producers write into it, Consumers read from it.
    4. Three semaphores:
        a. empty - 0 when buffer is empty, BUFFER_SIZE when buffer is full.
                   Initialized to 0. Because buffer is empty at the beginning.
        b. buffer_mutex - ensure mutual exclusion in access to shared buffer.
        c. full - 0 when buffer is full, BUFFER_SIZE when buffer is empty.
                  Initialized to BUFFER_SIZE. Because buffer is full at the
                      beginning
    5. consumed_buffer - a shared buffer among the consumers for storing whatever
                         they obtained from the buffer.
*/
int produced_buffer[BUFFER_SIZE];
sem_t empty, buffer_mutex, full;
int *consumed_buffer;


/*
    Consumer thread function.
    1. out - the index in the produced_buffer that can be consumed.
    2. consumed_in - the index in the consumed_buffer that can be written to.
    3. out and consumed_in are "shared" by all consumer threads.
       Note the static declaration.
    4. consumed, num and i are local variables. num is the number of times
       the loop will iterate.

    Refer to slides for explanation of the consumer semaphore solution.
*/

void *consumer(void *arg)
{
    static unsigned char out=0;
    static unsigned int consumed_in = 0;
    int consumed;
    int num = (int) arg;
    int i;

    for(i=0; i< num ; i++)
    {
        sem_wait(&empty);

        sem_wait(&buffer_mutex);
        consumed = produced_buffer[out++];
        consumed_buffer[consumed_in++] = consumed;
        sem_post(&buffer_mutex);
        sem_post(&full);
    }
}

/*
    Producer thread function.
    1. in - the index in the produced_buffer that can be written to.
    2. produced_num - the currently produced number.
    3. in and produced_num are "shared" by all consumer threads.
```

```
67          Note the static declaration.
        4. num and i are local variables. num is the number of times
69          the loop will iterate.
        5. Note that because produced_num is put within a mutual exclusive
71          section, it will increase from  1 to (number of producers threads * num).
        Refer to slides for explanation of the producer semaphore solution.
73  */

75  void *producer(void *arg)
    {
77      static unsigned char in = 0;
        static int produced_num = 0;
79      int num = (int) arg;
        int i =0;
81
        for(i=0; i< num ; i++)
83      {
            sem_wait(&full);
85          sem_wait(&buffer_mutex);
            produced_buffer[in++] = ++produced_num;
87          sem_post(&buffer_mutex);
            sem_post(&empty);
89      }
    }
91

93  /*
        The main function.
95  */

97  int main(int argc, char**argv)
    {
99      int num, flags;
        int num_of_consumers, num_of_producers;
101     int i;
        pthread_attr_t a;
103     pthread_t *producer_thds, *consumer_thds;

105     /*
            Error message indicating usage of the function.
107     */

109     if(argc!=3)
        {
111         fprintf(stderr, "Usage: %s <num_of_times> <num of consumer-producer-threads>\
                n", argv[0]);
            exit(-1);
113     }

115     /*
            Obtaining the arguments of the program.
117     */

119     num = atoi(argv[1]);
        num_of_consumers = atoi(argv[2]);
121     num_of_producers = atoi(argv[2]);

123     /*
            POSIX thread initialization.
125         setting flag to PTHREAD_SCOPE_SYSTEM ensure that the threads
            created are kernel-level threads i.e., visible to the
127         kernel.
        */
129
        flags = PTHREAD_SCOPE_SYSTEM;
131     pthread_attr_init(&a);
        pthread_attr_setscope(&a, flags);
133
```

```
    /*
135         Initializing the sempahores.
            prototype of sem_init:
137             sem_init(address of semaphore variable, flag, initial_value);
            The flag when 1 indicates that this semaphores can be
139         shared with other processes. For this demo, we only need it
            to 0.
141     */

143     sem_init(&full, 0, BUFFER_SIZE);
        sem_init(&empty, 0, 0);
145     sem_init(&buffer_mutex, 0, 1);

147     /*
            Allocating the memory buffer for consumed_buffer.
149         Basically, the total produced buffer size would be
            number of producers*num.
151     */
        consumed_buffer = (int *) malloc(num_of_producers * num * sizeof(int));
153
        /* Initializing the thread id array*/
155     producer_thds = (pthread_t *)malloc(num_of_producers * sizeof(pthread_t));
        consumer_thds = (pthread_t *)malloc(num_of_consumers * sizeof(pthread_t));
157
        /*Creating all the producer and consumer threads*/
159     for(i=0;i<num_of_producers;i++)
            pthread_create(&producer_thds[i], &a, producer, (void*)num);
161     for(i=0; i<num_of_consumers; i++)
            pthread_create(&consumer_thds[i], &a, consumer, (void*)num);
163
        /*wait for all producer and consumer threads to complete*/
165     for(i=0;i<num_of_producers;i++)
            pthread_join(producer_thds[i], NULL);
167     for(i=0;i<num_of_consumers; i++)
            pthread_join(consumer_thds[i], NULL);
169
        /*check consumed_buffer */
171     for(i=1; i<num_of_producers *num; i++)
        {
173         if(consumed_buffer[i]!=consumed_buffer[i-1]+1)
            {
175             printf("Result is wrong at index consumed_buffer[%d].\n", i);
                printf("consumed_buffer[%d] is %d.\n", i-1, consumed_buffer[i-1]);
177             printf("consumed_buffer[%d] is %d.\n", i, consumed_buffer[i]);
                return -1;
179         }
        }
181     printf("Result is correct!\n");
        free(consumer_thds);
183     free(producer_thds);
        free(consumed_buffer);
185
        return 0;
187 }
```

simple–producer–consumer–semaphore.c