BIRLA INSTITUTE OF TECHNOLOGY & SCIENCE, PILANI (RAJASTHAN)

CS G532 – High Performance Heterogeneous Computing

Lab#3

Note: Please use programs under *Code_lab1* directory supplied with this sheet. Do not copy from this sheet.

The lab has the following objectives:
Giving practice programs for thread creation, thread join, mutexes, condition variables, barriers

Pthread creation:

In the following program two threads modify a global variable. There is a possibility of corrupting the global variable.

```
1.  #include <pthread.h>
2.  #include <stdlib.h>
3.  #include <stdio.h>
4.
5.  int i;
6.
7.  void thread_func() {
8.      //  int i = 0;
9.      while (1) {
10.         printf("child thread: %d\n", i++);
11.         // sleep(1);
12.     }
13. }
14. int main() {
15.     pthread_t t1;
16.     pthread_create(&t1, NULL, thread_func, NULL);
17.     //int i = 0;
18.     while (1) {
19.         printf("main thread: %d\n", i++);
20.         // sleep(1);
21.     }
22.}
```

Q?

1. Increase number of threads to 3.
2. Is i value consistent? Modify program to use mutexes to protect i variable.

Pthread Join:

```
1· #include <stdio·h>
2· #include <pthread·h>

3·
4· void* function_write();
5· void* function_read();
6· FILE* fptr;
7· pthread_mutex_t mtx = PTHREAD_MUTEX_INITIALIZER;

8·
9· int main() {
10·     int rc1, rc2;
11·     fptr = fopen("·/mutex·txt", "w");
12·     fprintf(fptr, "The Answer to the Ultimate Question of Life, the Universe, and
    Everything is: ??");
13·     fclose(fptr);
14·     pthread_t thread1, thread2;
15·     int one = 1, two = 2;

16·
17·     if ((rc1 = pthread_create(&thread1, NULL, &function_write, (void*)&one))) {
```

```
18·         printf("Thread creation failed: %d\n", rc1);
19·     }
20·             pthread_join(thread1, NULL);
21·     if ((rc2 = pthread_create(&thread2, NULL, &function_read, (void*)&two))) {
22·         printf("Thread creation failed: %d\n", rc2);
23·     }
24·             pthread_join(thread2, NULL);
25·     return 0;
26·}


27·

28·void* function_write(void* param) {
29·     pthread_mutex_lock(&mtx);
30·             fptr = fopen("·/mutex·txt", "a");
31·     fprintf(fptr, "\b\b42·\n");
32·     fclose(fptr);
33·     pthread_mutex_unlock(&mtx);
34·         }


35·

36·void* function_read(void* param) {
37·     pthread_mutex_lock(&mtx);
38·     fptr = fopen("·/mutex·txt", "r");
39·     char dataToRead[50];
40·             while (fgets(dataToRead, 50, fptr) != NULL) {
41·         printf("%s", dataToRead);
42·             }
43·             fclose(fptr);
44·             pthread_mutex_unlock(&mtx);
```

```
45.         }
```

Q?

1. Comment the first pthread_join (Line 20). Does it provide the desired output every time you run it?
2. Comment the second pthread_join (Line 24). Explain the output.
3. Why do we need mutex in function_write and function_read? What happens if they are removed?

(You may need to run the program several times to observe the inconsistencies)

Pthread mutexes:

```
1.  #include <stdio.h>
2.  #include <stdlib.h>
3.  #include <pthread.h>
4.
5.  void* mutex_function();
6.  pthread_mutex_t mutex1 = PTHREAD_MUTEX_INITIALIZER;
7.  int  counter = 0;
8.
9.  int main() {
10.     int rc1, rc2;
11.     int one = 1, two = 2;
12.     pthread_t thread1, thread2;
13.
14.     if ((rc1 = pthread_create(&thread1, NULL, &mutex_function, (void*)&one))) {
15.         printf("Thread creation failed: %d\n", rc1);
16.     }
17.
18.     if ((rc2 = pthread_create(&thread2, NULL, &mutex_function, (void*)&two))) {
19.         printf("Thread creation failed: %d\n", rc2);
20.     }
21.
22.     pthread_join(thread1, NULL);
23.     pthread_join(thread2, NULL);
24.
25.     exit(0);
26.}
27.
28.void* mutex_function(int* param) {
```

```
29·     pthread_mutex_lock(&mutex1);
30·     counter++;
31·     printf("I'm in thread id %d, Counter value: %d\n", *param, counter);
32·     pthread_mutex_unlock(&mutex1);
33·}
```

Q?

1. Comment the lines which invoke the mutex variables. See the output in the file afterwards. Is it the desired output? *(You may need to run the program several times to observe the inconsistency)*
2. Rewrite the program to work with a larger number of threads. Specify the number of threads in a `#define` block. How will you specify the thread id (for printf)?

Pthread condition variables:

```
1·  /*
2·
3·  A program where the producer produces some output and the consumer waits for it·
4·
5·  */
6·  #include <pthread·h>
7·  #include <stdio·h>
8·
9·  pthread_mutex_t mutex;
10· pthread_cond_t cond;
11·
12· int buffer[100];
13·
14· int loopCount = 5;
15· int length = 0;
16·
17· void* producer(void* arg) {
18·     int i;
19·     for (i = 0; i < loopCount; i++) {
20·         pthread_mutex_lock(&mutex);
21·         buffer[length++] = i;
22·         printf("Producer length %d\n", length);
23·         pthread_cond_signal(&cond);
```

```
24·         pthread_mutex_unlock(&mutex);
25·     }
26·}
27·
28·void* consumer(void* arg) {
29·     int i;
30·     for (i = 0; i < loopCount; i++) {
31·         pthread_mutex_lock(&mutex);
32·         while (length == 0) {
33·             printf("Consumer waiting···\n");
34·             pthread_cond_wait(&cond, &mutex);
35·         }
36·         int item = buffer[--length];
37·         printf("Consumer %d\n", item);
38·         pthread_mutex_unlock(&mutex);
39·     }
40·}
41·
42·int main(int argc, char* argv[]) {
43·
44·     pthread_mutex_init(&mutex, 0);
45·     pthread_cond_init(&cond, 0);
46·
47·     pthread_t pThread, cThread;
48·     pthread_create(&pThread, 0, producer, 0);
49·     pthread_create(&cThread, 0, consumer, 0);
50·     pthread_join(pThread, NULL);
51·     pthread_join(cThread, NULL);
52·
53·     pthread_mutex_destroy(&mutex);
54·     pthread_cond_destroy(&cond);
55·     return 0;
56·}
```

Q?

1. What will happen if we don't have the mutex?
2. Try to extend this program by having 2 consumers or 2 producers.

False sharing:

Cache coherence introduces another problem for shared-memory programming: false sharing. When one core updates a variable in one cache line, and another core wants to access another variable in the same cache line, it will have to access main memory, since the unit of cache coherence is the cache line.

Refer to the program "pth_mat_vect_rand_split.c". This program takes number of threads and dimensions of the matrix. Randomly fills values.

Q?

1. Run the program for a single thread with dimensions 8000000x8, 8000x8000, 8x8000000. Note the time taken for each run. What are your observations. In all three cases, number of computations are same. What makes the time difference?
2. Run the program for 2 threads for all three cases of dimensions.
3. Run the program for 4 threads for all three cases of dimensions. Why 8x8000000 is taking more time than that of 2 threads?

Read-write locks:

The following program uses primitives like mutexes and condition variables to create a composite construct known as read-write lock. Read write lock should allow any number of readers together but only one writer at any time.

```
1. struct mylib_rwlock_t{
2.     int readers;
3.     int writer;
4.     pthread_cond_t readers_proceed;
5.     pthread_cond_t writer_proceed;
6.     int pending_writers;
7.     pthread_mutex_t read_write_lock;
8. };
9.
10. void mylib_rwlock_init(mylib_rwlock_t* l) {
```

```
11·     l->readers = l->writer = l->pending_writers = 0;
12·     pthread_mutex_init(&(l->read_write_lock), NULL);
13·     pthread_cond_init(&(l->readers_proceed), NULL);
14·     pthread_cond_init(&(l->writer_proceed), NULL);
15·}
16·
17·void mylib_rwlock_rlock(mylib_rwlock_t* l) {
18·     /* if there is a write lock or pending writers, perform condition
19·     wait·· else increment count of readers and grant read lock */
20·
21·     pthread_mutex_lock(&(l->read_write_lock));
22·       while ((l->pending_writers > 0) || (l->writer > 0)) {
23·            pthread_cond_wait(&(l->readers_proceed), &(l->read_write_lock));
24·       }
25·       l->readers++;
26·       pthread_mutex_unlock(&(l->read_write_lock));
27· }
28·
29·
30· void mylib_rwlock_wlock(mylib_rwlock_t* l) {
31·     /* if there are readers or writers, increment pending writers
32·       count and wait· On being woken, decrement pending writers
33·       count and increment writer count */
34·
35·       pthread_mutex_lock(&(l->read_write_lock));
36·       while ((l->writer > 0) || (l->readers > 0)) {
37·            l->pending_writers++;
38·            pthread_cond_wait(&(l->writer_proceed),
39·                 &(l->read_write_lock));
40·       }
41·     l->pending_writers--;
42·       l->writer++;
43·       pthread_mutex_unlock(&(l->read_write_lock));
```

```
44. }
45.
46.
47. void mylib_rwlock_unlock(mylib_rwlock_t* l) {
48.     /* if there is a write lock then unlock, else if there are
49.     read locks, decrement count of read locks. If the count
50.     is 0 and there is a pending writer, let it through, else
51.    if there are pending readers, let them all go through */
52.
53.     pthread_mutex_lock(&(l->read_write_lock));
54.     if (l->writer > 0)
55.         l->writer = 0;
56.     else if (l->readers > 0)
57.         l->readers--;
58.     pthread_mutex_unlock(&(l->read_write_lock));
59.     if ((l->readers == 0) && (l->pending_writers > 0))
60.         pthread_cond_signal(&(l->writer_proceed));
61.    else if (l->readers > 0)
62.         pthread_cond_broadcast(&(l->readers_proceed));
63. }
```

Q?

1. In ./pthread_rwlock.c you have been provided with a set of functions and a skeleton program to find the minimum out of a set of random values given to various threads. Use rwlock to find the minimum value of all. (Hint: Use rwlock on global_min, whose value is updated by each thread if it's greater than the thread's value)

Barriers:

Barrier constraints all threads to reach a point and wait until all threads reach there and then proceed. The following program uses primitives like mutexes and condition variables to create a composite construct barrier.

```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<pthread.h>

void* wait_thread(void* param);
typedef struct mylib_barrier_t mylib_barrier_t;
void mylib_init_barrier(mylib_barrier_t* b);
void mylib_barrier(mylib_barrier_t* b, int num_threads);

#define NTHREAD 2


struct mylib_barrier_t {
    pthread_mutex_t count_lock;
    pthread_cond_t ok_to_proceed;
    int count;
};

mylib_barrier_t myBarrier;
int t[NTHREAD];

int main(int argc, char const* argv[]) {
    pthread_t threadArr[NTHREAD];
    int threadIdArr[NTHREAD];
    for (int j = 0; j < NTHREAD; j++) {
        threadIdArr[j] = j;
    }

    mylib_init_barrier(&myBarrier);

    int i = 0;
    for (i = 0; i < NTHREAD; i++) {
```

```
34·         pthread_create(&threadArr[i], NULL, &wait_thread,
   &threadIdArr[i]);
35·     }
36·
37·     for (int j = 0; j < NTHREAD; ++j) {
38·         pthread_join(threadArr[j], NULL);
39·     }
40·
41·     return 0;
42·}
43·
44·void* wait_thread(void* param) {
45·     int threadId = *(int*)param;
46·     int sleepTime = (threadId + 1) * 2;
47·     printf("Thread %d will perform computation for %ds·\n", threadId,
   sleepTime);
48·     sleep(sleepTime);
49·     t[threadId] = (threadId + 1) * 10;
50·     mylib_barrier(&myBarrier, NTHREAD);
51·
52·     pthread_mutex_t printMutex = PTHREAD_MUTEX_INITIALIZER;
53·     pthread_mutex_lock(&printMutex);
54·     printf("At threadId %d, value of: ", threadId);
55·     for (int i = 0; i < NTHREAD; i++) {
56·         printf("t%d = %d, ", i, t[i]);
57·     }
58·     printf("\n");
59·     pthread_mutex_unlock(&printMutex);
60·
61·     return NULL;
62·}
63·
64·
```

```
65·void mylib_init_barrier(mylib_barrier_t* b) {
66·     b->count = 0;
67·     pthread_mutex_init(&(b->count_lock), NULL);
68·     pthread_cond_init(&(b->ok_to_proceed), NULL);
69·}
70·
71·void mylib_barrier(mylib_barrier_t* b, int num_threads) {
72·     pthread_mutex_lock(&(b->count_lock));
73·     b->count++;
74·     if (b->count == num_threads) {
75·          b->count = 0;
76·          pthread_cond_broadcast(&(b->ok_to_proceed));
77·     }
78·     else
79·          while                    (pthread_cond_wait(&(b->ok_to_proceed),
   &(b->count_lock)) != 0);
80·     pthread_mutex_unlock(&(b->count_lock));
81·}
```

Q?

1. Run the program. All both threads following barrier?
2. Try changing the parameter NTHREAD to NTHREAD – 1 to see undesirable output.
3. Write a program which computes sum of n numbers using p threads. Call the above barrier function to make all pthreads wait before printing the result.

----- End of lab1 -----