**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. int i;
5. void thread\_func() {
6. // int i = 0;
7. while (1) {
8. printf("child thread: %d\n", i++);
9. // sleep(1);
10. }
11. }
12. int main() {
13. pthread\_t t1;
14. pthread\_create(&t1, NULL, thread\_func, NULL);
15. //int i = 0;
16. while (1) {
17. printf("main thread: %d\n", i++);
18. // sleep(1);
19. }
20. }

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. void\* function\_write();
4. void\* function\_read();
5. FILE\* fptr;
6. pthread\_mutex\_t mtx = PTHREAD\_MUTEX\_INITIALIZER;
7. int main() {
8. int rc1, rc2;
9. fptr = fopen("./mutex.txt", "w");
10. fprintf(fptr, "The Answer to the Ultimate Question of Life, the Universe, and Everything is: ??");
11. fclose(fptr);
12. pthread\_t thread1, thread2;
13. int one = 1, two = 2;
14. if ((rc1 = pthread\_create(&thread1, NULL, &function\_write, (void\*)&one))) {
15. printf("Thread creation failed: %d\n", rc1);
16. }
17. pthread\_join(thread1, NULL);
18. if ((rc2 = pthread\_create(&thread2, NULL, &function\_read, (void\*)&two))) {
19. printf("Thread creation failed: %d\n", rc2);
20. }
21. pthread\_join(thread2, NULL);
22. return 0;
23. }
24. void\* function\_write(void\* param) {
25. pthread\_mutex\_lock(&mtx);
26. fptr = fopen("./mutex.txt", "a");
27. fprintf(fptr, "\b\b42.\n");
28. fclose(fptr);
29. pthread\_mutex\_unlock(&mtx);
30. }
31. void\* function\_read(void\* param) {
32. pthread\_mutex\_lock(&mtx);
33. fptr = fopen("./mutex.txt", "r");
34. char dataToRead[50];
35. while (fgets(dataToRead, 50, fptr) != NULL) {
36. printf("%s", dataToRead);
37. }
38. fclose(fptr);
39. pthread\_mutex\_unlock(&mtx);
40. }

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. void\* mutex\_function();
5. pthread\_mutex\_t mutex1 = PTHREAD\_MUTEX\_INITIALIZER;
6. int counter = 0;
7. int main() {
8. int rc1, rc2;
9. int one = 1, two = 2;
10. pthread\_t thread1, thread2;
11. if ((rc1 = pthread\_create(&thread1, NULL, &mutex\_function, (void\*)&one))) {
12. printf("Thread creation failed: %d\n", rc1);
13. }
14. if ((rc2 = pthread\_create(&thread2, NULL, &mutex\_function, (void\*)&two))) {
15. printf("Thread creation failed: %d\n", rc2);
16. }
17. pthread\_join(thread1, NULL);
18. pthread\_join(thread2, NULL);
19. exit(0);
20. }
21. void\* mutex\_function(int\* param) {
22. pthread\_mutex\_lock(&mutex1);
23. counter++;
24. printf("I'm in thread id %d, Counter value: %d\n", \*param, counter);
25. pthread\_mutex\_unlock(&mutex1);
26. }

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. A program where the producer produces some output and the consumer waits for it.
3. \*/
4. #include <pthread.h>
5. #include <stdio.h>
6. pthread\_mutex\_t mutex;
7. pthread\_cond\_t cond;
8. int buffer[100];
9. int loopCount = 5;
10. int length = 0;
11. void\* producer(void\* arg) {
12. int i;
13. for (i = 0; i < loopCount; i++) {
14. pthread\_mutex\_lock(&mutex);
15. buffer[length++] = i;
16. printf("Producer length %d\n", length);
17. pthread\_cond\_signal(&cond);
18. pthread\_mutex\_unlock(&mutex);
19. }
20. }
21. void\* consumer(void\* arg) {
22. int i;
23. for (i = 0; i < loopCount; i++) {
24. pthread\_mutex\_lock(&mutex);
25. while (length == 0) {
26. printf("Consumer waiting...\n");
27. pthread\_cond\_wait(&cond, &mutex);
28. }
29. int item = buffer[--length];
30. printf("Consumer %d\n", item);
31. pthread\_mutex\_unlock(&mutex);
32. }
33. }
34. int main(int argc, char\* argv[]) {
35. pthread\_mutex\_init(&mutex, 0);
36. pthread\_cond\_init(&cond, 0);
37. pthread\_t pThread, cThread;
38. pthread\_create(&pThread, 0, producer, 0);
39. pthread\_create(&cThread, 0, consumer, 0);
40. pthread\_join(pThread, NULL);
41. pthread\_join(cThread, NULL);
42. pthread\_mutex\_destroy(&mutex);
43. pthread\_cond\_destroy(&cond);
44. return 0;
45. }

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. void mylib\_rwlock\_init(mylib\_rwlock\_t\* l) {
10. l->readers = l->writer = l->pending\_writers = 0;
11. pthread\_mutex\_init(&(l->read\_write\_lock), NULL);
12. pthread\_cond\_init(&(l->readers\_proceed), NULL);
13. pthread\_cond\_init(&(l->writer\_proceed), NULL);
14. }
15. void mylib\_rwlock\_rlock(mylib\_rwlock\_t\* l) {
16. /\* if there is a write lock or pending writers, perform condition
17. wait.. else increment count of readers and grant read lock \*/
18. pthread\_mutex\_lock(&(l->read\_write\_lock));
19. while ((l->pending\_writers > 0) || (l->writer > 0)) {
20. pthread\_cond\_wait(&(l->readers\_proceed), &(l->read\_write\_lock));
21. }
22. l->readers++;
23. pthread\_mutex\_unlock(&(l->read\_write\_lock));
24. }
25. void mylib\_rwlock\_wlock(mylib\_rwlock\_t\* l) {
26. /\* if there are readers or writers, increment pending writers
27. count and wait. On being woken, decrement pending writers
28. count and increment writer count \*/
29. pthread\_mutex\_lock(&(l->read\_write\_lock));
30. while ((l->writer > 0) || (l->readers > 0)) {
31. l->pending\_writers++;
32. pthread\_cond\_wait(&(l->writer\_proceed),
33. &(l->read\_write\_lock));
34. }
35. l->pending\_writers--;
36. l->writer++;
37. pthread\_mutex\_unlock(&(l->read\_write\_lock));
38. }
39. void mylib\_rwlock\_unlock(mylib\_rwlock\_t\* l) {
40. /\* if there is a write lock then unlock, else if there are
41. read locks, decrement count of read locks. If the count
42. is 0 and there is a pending writer, let it through, else
43. if there are pending readers, let them all go through \*/
44. pthread\_mutex\_lock(&(l->read\_write\_lock));
45. if (l->writer > 0)
46. l->writer = 0;
47. else if (l->readers > 0)
48. l->readers--;
49. pthread\_mutex\_unlock(&(l->read\_write\_lock));
50. if ((l->readers == 0) && (l->pending\_writers > 0))
51. pthread\_cond\_signal(&(l->writer\_proceed));
52. else if (l->readers > 0)
53. pthread\_cond\_broadcast(&(l->readers\_proceed));
54. }

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.

1. #include<stdio.h>
2. #include<stdlib.h>
3. #include<unistd.h>
4. #include<pthread.h>
5. void\* wait\_thread(void\* param);
6. typedef struct mylib\_barrier\_t mylib\_barrier\_t;
7. void mylib\_init\_barrier(mylib\_barrier\_t\* b);
8. void mylib\_barrier(mylib\_barrier\_t\* b, int num\_threads);
9. #define NTHREAD 2
10. struct mylib\_barrier\_t {
11. pthread\_mutex\_t count\_lock;
12. pthread\_cond\_t ok\_to\_proceed;
13. int count;
14. };
15. mylib\_barrier\_t myBarrier;
16. int t[NTHREAD];
17. int main(int argc, char const\* argv[]) {
18. pthread\_t threadArr[NTHREAD];
19. int threadIdArr[NTHREAD];
20. for (int j = 0; j < NTHREAD; j++) {
21. threadIdArr[j] = j;
22. }
23. mylib\_init\_barrier(&myBarrier);
24. int i = 0;
25. for (i = 0; i < NTHREAD; i++) {
26. pthread\_create(&threadArr[i], NULL, &wait\_thread, &threadIdArr[i]);
27. }
28. for (int j = 0; j < NTHREAD; ++j) {
29. pthread\_join(threadArr[j], NULL);
30. }
31. return 0;
32. }
33. void\* wait\_thread(void\* param) {
34. int threadId = \*(int\*)param;
35. int sleepTime = (threadId + 1) \* 2;
36. printf("Thread %d will perform computation for %ds.\n", threadId, sleepTime);
37. sleep(sleepTime);
38. t[threadId] = (threadId + 1) \* 10;
39. mylib\_barrier(&myBarrier, NTHREAD);
40. pthread\_mutex\_t printMutex = PTHREAD\_MUTEX\_INITIALIZER;
41. pthread\_mutex\_lock(&printMutex);
42. printf("At threadId %d, value of: ", threadId);
43. for (int i = 0; i < NTHREAD; i++) {
44. printf("t%d = %d, ", i, t[i]);
45. }
46. printf("\n");
47. pthread\_mutex\_unlock(&printMutex);
48. return NULL;
49. }
50. void mylib\_init\_barrier(mylib\_barrier\_t\* b) {
51. b->count = 0;
52. pthread\_mutex\_init(&(b->count\_lock), NULL);
53. pthread\_cond\_init(&(b->ok\_to\_proceed), NULL);
54. }
55. void mylib\_barrier(mylib\_barrier\_t\* b, int num\_threads) {
56. pthread\_mutex\_lock(&(b->count\_lock));
57. b->count++;
58. if (b->count == num\_threads) {
59. b->count = 0;
60. pthread\_cond\_broadcast(&(b->ok\_to\_proceed));
61. }
62. else
63. while (pthread\_cond\_wait(&(b->ok\_to\_proceed), &(b->count\_lock)) != 0);
64. pthread\_mutex\_unlock(&(b->count\_lock));
65. }

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 -----**