BIRLA INSTITUTE OF TECHNOLOGY & SCIENCE, PILANI (RAJASTHAN)

CS F422 – Parallel Computing

Lab#1

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:

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

int i;

void thread_func() {
    //  int i = 0;
    while (1) {
              printf("child thread: %d\n", i++);
              // sleep(1);
          }
      }
      int main() {
          pthread_t t1;
          pthread_create(&t1, NULL, thread_func, NULL);
          //int i = 0;
          while (1) {
              printf("main thread: %d\n", i++);
              // sleep(1);
          }
      }
```

Q?

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

Pthread Join:

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


void* function_write();
void* function_read();
FILE* fptr;
pthread_mutex_t mtx = PTHREAD_MUTEX_INITIALIZER;


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


    if ((rc1 = pthread_create(&thread1, NULL, &function_write, (void*)&one))) {
        printf("Thread creation failed: %d\n", rc1);
    }
    pthread_join(thread1, NULL);
    if ((rc2 = pthread_create(&thread2, NULL, &function_read, (void*)&two))) {
        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:

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

void* mutex_function();
pthread_mutex_t mutex1 = PTHREAD_MUTEX_INITIALIZER;
int  counter = 0;

int main() {
    int rc1, rc2;
    int one = 1, two = 2;
    pthread_t thread1, thread2;

    if ((rc1 = pthread_create(&thread1, NULL, &mutex_function, (void*)&one))) {
        printf("Thread creation failed: %d\n", rc1);
    }

    if ((rc2 = pthread_create(&thread2, NULL, &mutex_function, (void*)&two))) {
        printf("Thread creation failed: %d\n", rc2);
    }

    pthread_join(thread1, NULL);
    pthread_join(thread2, NULL);

    exit(0);
}

void* mutex_function(int* param) {
    pthread_mutex_lock(&mutex1);
    counter++;
    printf("I'm in thread id %d, Counter value: %d\n", *param, counter);
    pthread_mutex_unlock(&mutex1);
}
```

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:

```
/*

A program where the producer produces some output and the consumer waits for it.

*/
#include <pthread.h>
#include <stdio.h>

pthread_mutex_t mutex;
pthread_cond_t cond;

int buffer[100];

int loopCount = 5;
int length = 0;

void* producer(void* arg) {
    int i;
    for (i = 0; i < loopCount; i++) {
        pthread_mutex_lock(&mutex);
        buffer[length++] = i;
        printf("Producer length %d\n", length);
        pthread_cond_signal(&cond);
        pthread_mutex_unlock(&mutex);
    }
}

void* consumer(void* arg) {
    int i;
    for (i = 0; i < loopCount; i++) {
        pthread_mutex_lock(&mutex);
        while (length == 0) {
            printf("Consumer waiting...\n");
            pthread_cond_wait(&cond, &mutex);
        }
        int item = buffer[--length];
        printf("Consumer %d\n", item);
        pthread_mutex_unlock(&mutex);
    }
}

int main(int argc, char* argv[]) {

    pthread_mutex_init(&mutex, 0);
    pthread_cond_init(&cond, 0);

    pthread_t pThread, cThread;
    pthread_create(&pThread, 0, producer, 0);
    pthread_create(&cThread, 0, consumer, 0);
    pthread_join(pThread, NULL);
    pthread_join(cThread, NULL);

    pthread_mutex_destroy(&mutex);
    pthread_cond_destroy(&cond);
    return 0;
}
```

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:

```
/*

For false sharing, since the wrong value being accessed is handled by the OS (through a dirty bit), we'll instead observe how constant back-and-forth (ping-pong) memory access slows down our program.

*/

#include <stdio.h>
#include <pthread.h>
#include <time.h>
#include <unistd.h>

int array[100];
#define NUM_ITER 100000000

void* func(void* param) {
    int index = *((int*)param);
    int i;
    for (i = 0; i < NUM_ITER; i++) {
        array[index] += 3;
    }

    return NULL;
}

void subtract_time(struct timespec t1, struct timespec t2, struct timespec* td) {
    td->tv_nsec = t2.tv_nsec - t1.tv_nsec;
    td->tv_sec = t2.tv_sec - t1.tv_sec;
    if (td->tv_sec > 0 && td->tv_nsec < 0) {
        td->tv_nsec += 1000000000;
        td->tv_sec--;
    }
    else if (td->tv_sec < 0 && td->tv_nsec > 0) {
        td->tv_nsec -= 1000000000;
        td->tv_sec++;
    }
}

int main(int argc, char* argv[]) {
    int       first_elem = 0;
    int       bad_elem = 1;
    int       good_elem = 32;
    pthread_t     thread_1;
    pthread_t     thread_2;

    struct timespec start, finish, delta_seq, delta_false, delta_true;
    clock_gettime(CLOCK_REALTIME, &start);
    func(&first_elem);
    func(&bad_elem);
    clock_gettime(CLOCK_REALTIME, &finish);
    subtract_time(start, finish, &delta_seq);

    clock_gettime(CLOCK_REALTIME, &start);
    pthread_create(&thread_1, NULL, func, (void*)&first_elem);
    pthread_create(&thread_2, NULL, func, (void*)&bad_elem);
    pthread_join(thread_1, NULL);
    pthread_join(thread_2, NULL);
    clock_gettime(CLOCK_REALTIME, &finish);
    subtract_time(start, finish, &delta_false);

    /* Just to show that parallel threads in best case *can* improve efficiency */
    clock_gettime(CLOCK_REALTIME, &start);
    pthread_create(&thread_1, NULL, func, (void*)&first_elem);
    pthread_create(&thread_2, NULL, func, (void*)&good_elem);
    pthread_join(thread_1, NULL);
    pthread_join(thread_2, NULL);
    clock_gettime(CLOCK_REALTIME, &finish);
    subtract_time(start, finish, &delta_true);

    printf("%d %d %d\n", array[first_elem], array[bad_elem], array[good_elem]);

    printf("Time taken for seq\t:%d.%.9lds\n", (int)delta_seq.tv_sec, delta_seq.tv_nsec);
    printf("Time taken for false\t:%d.%.9lds\n", (int)delta_false.tv_sec, delta_false.tv_nsec);
    printf("Time taken for true\t:%d.%.9lds\n", (int)delta_true.tv_sec, delta_true.tv_nsec);
    return 0;
}
```

Q?

1. What inferences can you draw from the output?

2. How does the OS handle false sharing? Is it being handled in this program?
3. Try changing the number of iterations going on in func() (L13 & 18) to see if the effect persists at different orders of magnitude.
4. Try to change the bad_elem (L40) variable to see how far the cache line might extend for your architecture.

Read-write locks:

```
struct mylib_rwlock_t{
    int readers;
    int writer;
    pthread_cond_t readers_proceed;
    pthread_cond_t writer_proceed;
    int pending_writers;
    pthread_mutex_t read_write_lock;
};

void mylib_rwlock_init(mylib_rwlock_t* l) {
    l->readers = l->writer = l->pending_writers = 0;
    pthread_mutex_init(&(l->read_write_lock), NULL);
    pthread_cond_init(&(l->readers_proceed), NULL);
    pthread_cond_init(&(l->writer_proceed), NULL);
}

void mylib_rwlock_rlock(mylib_rwlock_t* l) {
    /* if there is a write lock or pending writers, perform condition
    wait.. else increment count of readers and grant read lock */

    pthread_mutex_lock(&(l->read_write_lock));
    while ((l->pending_writers > 0) || (l->writer > 0)) {
                        pthread_cond_wait(&(l->readers_proceed),   &(l->read_write_lock));
    }
    l->readers++;
    pthread_mutex_unlock(&(l->read_write_lock));
}


void mylib_rwlock_wlock(mylib_rwlock_t* l) {
    /* if there are readers or writers, increment pending writers
    count and wait. On being woken, decrement pending writers
    count and increment writer count */

    pthread_mutex_lock(&(l->read_write_lock));
    while ((l->writer > 0) || (l->readers > 0)) {
        l->pending_writers++;
        pthread_cond_wait(&(l->writer_proceed),
            &(l->read_write_lock));
    }
    l->pending_writers--;
    l->writer++;
    pthread_mutex_unlock(&(l->read_write_lock));
}


void mylib_rwlock_unlock(mylib_rwlock_t* l) {
    /* if there is a write lock then unlock, else if there are
    read locks, decrement count of read locks. If the count
    is 0 and there is a pending writer, let it through, else
    if there are pending readers, let them all go through */

    pthread_mutex_lock(&(l->read_write_lock));
    if (l->writer > 0)
        l->writer = 0;
    else if (l->readers > 0)
        l->readers--;
    pthread_mutex_unlock(&(l->read_write_lock));
    if ((l->readers == 0) && (l->pending_writers > 0))
        pthread_cond_signal(&(l->writer_proceed));
    else if (l->readers > 0)
        pthread_cond_broadcast(&(l->readers_proceed));
}
```

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:

```
#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++) {
        pthread_create(&threadArr[i], NULL, &wait_thread, &threadIdArr[i]);
    }

    for (int j = 0; j < NTHREAD; ++j) {
        pthread_join(threadArr[j], NULL);
    }

    return 0;
}

void* wait_thread(void* param) {
    int threadId = *(int*)param;
    int sleepTime = (threadId + 1) * 2;
    printf("Thread %d will perform computation for %ds.\n", threadId, sleepTime);
    sleep(sleepTime);
    t[threadId] = (threadId + 1) * 10;
    mylib_barrier(&myBarrier, NTHREAD);

    pthread_mutex_t printMutex = PTHREAD_MUTEX_INITIALIZER;
    pthread_mutex_lock(&printMutex);
    printf("At threadId %d, value of: ", threadId);
    for (int i = 0; i < NTHREAD; i++) {
        printf("t%d = %d, ", i, t[i]);
    }
    printf("\n");
    pthread_mutex_unlock(&printMutex);

    return NULL;
}


void mylib_init_barrier(mylib_barrier_t* b) {
    b->count = 0;
    pthread_mutex_init(&(b->count_lock), NULL);
    pthread_cond_init(&(b->ok_to_proceed), NULL);
}

void mylib_barrier(mylib_barrier_t* b, int num_threads) {
    pthread_mutex_lock(&(b->count_lock));
    b->count++;
    if (b->count == num_threads) {
        b->count = 0;
        pthread_cond_broadcast(&(b->ok_to_proceed));
    }
    else
        while (pthread_cond_wait(&(b->ok_to_proceed), &(b->count_lock)) != 0);
    pthread_mutex_unlock(&(b->count_lock));
}
```

Q?

1. Run the program. Do you observe the desired output?
2. Try changing the parameter NTHREAD to NTHREAD – 1 to see undesirable output.

Have fun!

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