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:

```
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:

```
1. /*
2.
3. 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.
4.
5. */
6.
7. #include <stdio.h>
8. #include <pthread.h>
9. #include <time.h>
10.      #include <unistd.h>
11.
12.      int array[100];
13.      #define NUM_ITER 100000000
14.
15.      void* func(void* param) {
16.          int index = *((int*)param);
17.          int i;
18.          for (i = 0; i < NUM_ITER; i++) {
19.              array[index] += 3;
20.          }
21.
22.          return NULL;
23.      }
24.
25.      void subtract_time(struct timespec t1, struct timespec t2, struct
   timespec* td) {
26.          td->tv_nsec = t2.tv_nsec - t1.tv_nsec;
27.          td->tv_sec = t2.tv_sec - t1.tv_sec;
28.          if (td->tv_sec > 0 && td->tv_nsec < 0) {
29.              td->tv_nsec += 1000000000;
30.              td->tv_sec--;
31.          }
32.          else if (td->tv_sec < 0 && td->tv_nsec > 0) {
33.              td->tv_nsec -= 1000000000;
34.              td->tv_sec++;
35.          }
36.      }
37.
38.      int main(int argc, char* argv[]) {
```

```
39.           int       first_elem = 0;
40.           int       bad_elem = 1;
41.           int       good_elem = 32;
42.           pthread_t     thread_1;
43.           pthread_t     thread_2;
44.
45.           struct timespec start, finish, delta_seq, delta_false,
   delta_true;
46.           clock_gettime(CLOCK_REALTIME, &start);
47.           func(&first_elem);
48.           func(&bad_elem);
49.           clock_gettime(CLOCK_REALTIME, &finish);
50.           subtract_time(start, finish, &delta_seq);
51.
52.           clock_gettime(CLOCK_REALTIME, &start);
53.           pthread_create(&thread_1, NULL, func, (void*)&first_elem);
54.           pthread_create(&thread_2, NULL, func, (void*)&bad_elem);
55.           pthread_join(thread_1, NULL);
56.           pthread_join(thread_2, NULL);
57.           clock_gettime(CLOCK_REALTIME, &finish);
58.           subtract_time(start, finish, &delta_false);
59.
60.           /* Just to show that parallel threads in best case *can*
   improve efficiency */
61.           clock_gettime(CLOCK_REALTIME, &start);
62.           pthread_create(&thread_1, NULL, func, (void*)&first_elem);
63.           pthread_create(&thread_2, NULL, func, (void*)&good_elem);
64.           pthread_join(thread_1, NULL);
65.           pthread_join(thread_2, NULL);
66.           clock_gettime(CLOCK_REALTIME, &finish);
67.           subtract_time(start, finish, &delta_true);
68.
69.           printf("%d %d %d\n", array[first_elem], array[bad_elem],
   array[good_elem]);
70.
71.           printf("Time taken for seq\t:%d.%.9lds\n",
   (int)delta_seq.tv_sec, delta_seq.tv_nsec);
72.           printf("Time taken for false\t:%d.%.9lds\n",
   (int)delta_false.tv_sec, delta_false.tv_nsec);
73.           printf("Time taken for true\t:%d.%.9lds\n",
   (int)delta_true.tv_sec, delta_true.tv_nsec);
74.           return 0;
75.       }
```

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.

End of lab1