MONASH University

FIT3143 Tutorial Week 3

Lecturers: ABM Russel (MU Australia) and Vishnu Monn (MU Malaysia)

SHARED MEMORY (PTHREADS)

OBJECTIVES

- The purpose of this tutorial is to introduce Parallel Computing on shared memory
- Understand the concept of POSIX thread

Note: Tutorials are not assessed. Nevertheless, please attempt the questions to improve your unit comprehension in preparation for the labs, assignments, and final assessments.

QUESTIONS/TOPICS

1. Discuss the advantages and disadvantages of Threads.

There are many advantages and some disadvantages (or challenges) when using threads. Although the following points are verbose, these points are for your reference and comprehension.

Advantages:

- **Threads allow parallel programming:** For instance, one thread can be doing a computation, while another thread waits for I/O to complete. They are **cheaper to create in resources**. The **parallel advantage can simplify programming**. For instance, the **producer/consumer problem is easier to solve by coordinating threads**, rather than by a single process that does everything. A third advantage **is context switching between threads is much faster than context switching between processes**.
- **Software portability**: Threaded applications can be developed on serial machines and on parallel machines without changes
- **Latency hiding:** One of the major overheads (disadvantages) in programs is the access latency (delay) for memory access, I/O and communication. **Multiple threads are executing on the same processor**, **thus hiding this latency**. While one thread is waiting for a communication operation, other threads can utilize the CPU, thus **masking associated overhead**.
- **Scheduling and load balancing:** A programmer must express concurrency in a way that minimizes overheads of remote interaction and idling. In many structured applications, the task of allocating equal work to processors is easily accomplished. In unstructured and dynamic applications (e.g. game playing and discrete optimization) this task is more difficult. **Threaded APIs allow the programmer to specify a large number of concurrent tasks and support system-level dynamic mapping of tasks to processor with a view to**

minimizing idling overheads, thus there is no need for explicit scheduling and load balancing.

- **Ease of programming, widespread use**: Threaded programs are easier to write than programs using message passing APIs. With widespread acceptance of POSIX thread API, development tools for POSIX are more widely available and stable.
- **Improved performance and concurrency:** For certain applications, performance and concurrency can be improved by using multithreading and multicontexting together. In other applications, performance can be unaffected or even degraded by using multithreading and multicontexting together. How performance is affected depends on your application.
- **Simplified coding** of remote procedure calls and conversations: In some applications it is easier to code different remote procedure calls and conversations in separate threads than to manage them from the same thread.
- **Threads allow utilization of multiprocessor architectures** to a greater scale and efficiency for task and data parallelism
- **Reduced number of required servers:** Because one server can dispatch multiple service threads, the number of servers to start for your application is reduced. This capability for multiple dispatched threads is especially useful for conversational servers, which otherwise must be dedicated to one client for the entire duration of a conversation.

Disadvantages:

- Sharing the same memory space can be a disadvantage. **For instance, if one thread corrupts the memory space all the other threads in the group will suffer as well**. **The operating system does not protect one thread from another thread like it does for processes.**
- **Difficulty of writing code:** Multithreaded and multicontexted applications are not easy to write. Only experienced programmers should undertake coding for these types of applications.
- **Difficulty of debugging:** It is much harder to replicate an error in a multithreaded or multicontexted application than it is to do so in a single-threaded, single-contexted application. As a result, it is more difficult, in the former case, to identify and verify root causes when errors occur.
- **Difficulty of managing concurrency:** The task of managing concurrency among threads is difficult and has the potential to introduce new problems into an application.
- **Difficulty of testing:** Testing a multithreaded application is more difficult than testing a single-threaded application because defects are often timing-related and more difficult to reproduce.
- **Difficulty of porting existing code:** Existing code often requires significant re-architecting to take advantage of multithreading and multi-contexting. Programmers need to:
    - Remove static variables
    - Replace any function calls that are not thread-safe
    - Replace any other code that is not thread-safe

- Because the completed port must be tested and re-tested, the work required to port a multithreaded and/or multi-context application is substantial

2. With reference to the following code:

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

void *print_message_function( void *ptr );

int main()
{
     pthread_t thread1, thread2;
     char *message1 = "Thread 1";
     char *message2 = "Thread 2";
     int  iret1, iret2;

    /* Create independent threads each of which will execute function */

     iret1 = pthread_create( &thread1, NULL, print_message_function,
(void*) message1);
     iret2 = pthread_create( &thread2, NULL, print_message_function,
(void*) message2);

     printf("Thread 1 returns: %d\n",iret1);
     printf("Thread 2 returns: %d\n",iret2);
     return;
}

void *print_message_function( void *ptr )
{
     char *message;
     message = (char *) ptr;
     printf("%s \n", message);
}
```

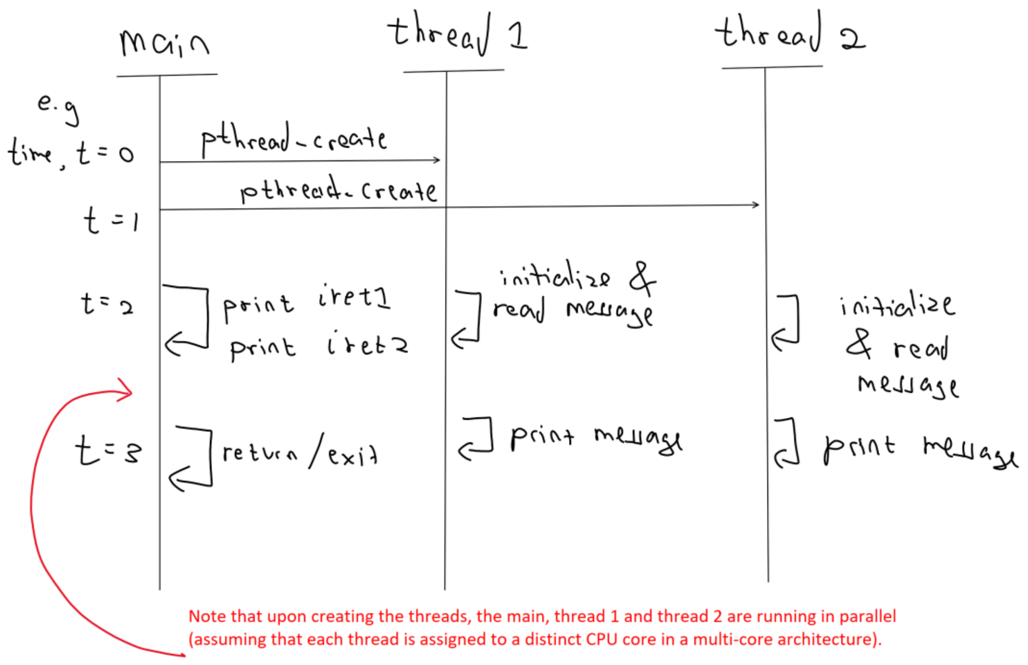
a) Discuss the possible outcomes of the code above.

Depending on how the threads are scheduled by the operating system at runtime:

- Either one or both threads may be able to print the message in the thread function
- Neither thread can print the message in the thread function. Reason being is because the main function does not wait for the threads to complete. That is, the pthread_join() function was not invoked in the main function. Since the main function itself is an application process, it will continue to execute the remaining code after creating the threads. That is, the main function is asynchronous to the threads. Hence, without waiting for the threads to join, the main function could terminate before the threads are able to complete the tasks in the respective thread functions. Since the threads were created under the main application process, once the main process is terminated, the threads will be terminated as well.

b) Explain the reasoning behind your outcomes using a simple thread sequence diagram.

Solution:



Note that upon creating the threads, the main, thread 1 and thread 2 are running in parallel (assuming that each thread is assigned to a distinct CPU core in a multi-core architecture).

Without calling the pthread_join() function, the main() function does not wait for the threads to terminate. Hence, at *t = 3*, there is a risk that the main function could terminate before the threads could complete the print function. Consequently, the threads will be terminated because termination of the main application process will also result in a termination of any running threads which were spawned from the main process.

c) Modify the code above to improve its outcome.

Add pthread_join() into the main() function before the function returns.

```
int main()
{
     pthread_t thread1, thread2;
     char *message1 = "Thread 1";
     char *message2 = "Thread 2";
     int  iret1, iret2;

     iret1 = pthread_create( &thread1, NULL, print_message_function,
     (void*) message1);
     iret2 = pthread_create( &thread2, NULL, print_message_function,
     (void*) message2);

     printf("Thread 1 returns: %d\n",iret1);
     printf("Thread 2 returns: %d\n",iret2);

     pthread_join( thread1, NULL);
     pthread_join( thread2, NULL);
     return;
}
```

3. With reference to the following code:

a) Code which includes usage of **pthread_mutex**

```
#include <stdio.h>
#include <pthread.h>
#include <stdlib.h>
#include <time.h>
#define NUMTHREADS 4

pthread_mutex_t gMutex;
int g_sum = 0;

void *threadFunc(void *arg)
{
      int myVal = rand_r(time(NULL));
      pthread_mutex_lock( &gMutex );
      g_sum += myVal;
      pthread_mutex_unlock( &gMutex );
}

int main() {

      pthread_t hThread[NUMTHREADS];

      pthread_mutex_init( &gMutex, NULL );
      for (int i = 0; i < NUMTHREADS; i++)
            pthread_create(&hThread[i],NULL,threadFunc,NULL);

      for (int i = 0; i < NUMTHREADS; i++)
            pthread_join(hThread[i]);

      printf ("Global sum = %f\n", g_sum);
}
```

b) Code which does not include usage of a mutex

```
#include <stdio.h>
#include <pthread.h>
#include <stdlib.h>
#include <time.h>
#define NUMTHREADS 4

int g_Array[NUMTHREADS];

void *threadFunc(void *arg)
{
      int threadNo = *((int*)arg);
      int myVal = rand_r(time(NULL));
      g_Array[threadNo] = myVal;
}

int main() {

      pthread_t hThread[NUMTHREADS];
      int threadNum[NUMTHREADS];

      for (int i = 0; i < NUMTHREADS; i++){
            threadNum[i] = i;
            pthread_create(&hThread[i],NULL,threadFunc,&threadNum[i]);
```

```
        }

        for (int i = 0; i < NUMTHREADS; i++){
              pthread_join(hThread[i]);
              printf ("Thread %d generated value: %d\n", i, g_Array[i]);
        }
}
```

c) Analyze the code in parts (a) and (b) above. Both the code in (a) and (b) should compile and produce a correct output. Notice that for both code snippets, the threads are writing into global variables. However, the code snippet in (a) used a mutex but the code snippet in (b) did not use a mutex. Why is this the case?

In (a), multiple threads are writing to a single integer in memory. That is, each thread would access the same address in memory when adding the local value into the global variable. Hence, in this context, there is a risk of a race condition and the actual value in the global sum variable may not represent a correct summation. Hence, to ensure a thread safe operation, a mutex is used to ensure that only one thread can write into the global integer variable at any given point of time.

In (b), at first glance, it may appear that multiple threads are writing to the same shared global variable (i.e., array), which would risk a race condition. However, notice that each thread is writing to a different index of the array. None of the threads would write to the same array index. Hence, in this case, multiple threads are writing to memory at the same time, but at different locations of memory (based on different indices of the array). Hence, in this context, there is no risk of race condition and hence there is no need to use a mutex here.

4. With reference to the following code:

```
#include <stdio.h>
#include <unistd.h>
#include <pthread.h>
#define NUM_THREADS 2

// Global variables
pthread_mutex_t g_Mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t g_Cond = PTHREAD_COND_INITIALIZER;
int g_Val = 0; // Global shared variable

// Thread 1 Callback Function
void *Thread1Func(void *pArg)
{
      int* p = (int*)pArg;
      int myNum = *p; // myNum = 0

      sleep(1); // Intentionally slow down the execution of Thread1Func
      pthread_mutex_lock(&g_Mutex);
      printf("Thread Id: %d. Global value before updated by Thread2Func:
%d\n", myNum, g_Val);

      pthread_cond_wait(&g_Cond, &g_Mutex);
      printf("Thread Id: %d. Global value after updated by Thread2Func:
%d\n", myNum, g_Val);

      g_Val += 1;
```

```
	pthread_mutex_unlock(&g_Mutex);

	return NULL;
}

// Thread 2 Callback Function
void *Thread2Func(void *pArg)
{
	int* p = (int*)pArg;
	int myNum = *p; // myNum = 1

	pthread_mutex_lock(&g_Mutex);
	g_Val += 2;
	printf( "Thread Id: %d. Global value increased by 2!\n", myNum);
	pthread_mutex_unlock(&g_Mutex);
	pthread_cond_signal(&g_Cond);

	return NULL;
}

// Main program
int main()
{
	pthread_t hThread[NUM_THREADS]; // Stores the POSIX thread IDs
	int threadNum[NUM_THREADS]; // Pass a unique thread ID
	int i = 0;

	// Initialize the mutex & condition variable
	pthread_mutex_init(&g_Mutex, NULL);
	pthread_cond_init(&g_Cond, NULL);

	// Create both threads
	threadNum[0] = 0;
	pthread_create(&hThread[0], NULL, Thread1Func, &threadNum[0]);

	threadNum[1] = 1;
	pthread_create(&hThread[1], NULL, Thread2Func, &threadNum[1]);


	// Wait for all threads to finish
	for(i = 0; i < NUM_THREADS; i++)
	{
		pthread_join(hThread[i], NULL);
	}
	printf( "Global Value: %d\n", g_Val);

	// Clean up
	pthread_cond_destroy(&g_Cond);
	pthread_mutex_destroy(&g_Mutex);
	return 0;
}
```

For the code above to work, `Thread1Func` should execute earlier to ensure that it calls `pthread_cond_wait` before `Thread2Func` calls `pthread_cond_signal`. However, notice that a `sleep` function is used in `Thread1Func` which would delay its execution.

a) Consequently, what would be the outcome if the code above was compiled and executed? Briefly discuss the concept of lost and spurious signals when using condition variables for POSIX threads.

Given that both **Thread1Func** and **Thread2Func** are running in parallel, **Thread2Func** would proceed to call **pthread_cond_signal**. However, the **sleep()** function in **Thread1Func** would delay the call of **pthread_cond_wait**. Therefore, the signal from **Thread2Func** will be lost. Once **Thread2Func** wakes up from the **sleep()** function and calls **pthread_cond_wait**, it will remain in a wait state indefinitely because , **Thread2Func** has invoked **pthread_cond_signal** and exited. This in turn would cause a program failure (i.e., the program will freeze or hang).

b) If the sleep function in **Thread1Func** is to remain, modify **Thread2Func** to ensure that **pthread_cond_signal** is not lost (Note: Adding another sleep function in **Thread2Func** is not the appropriate solution here).

```
// Thread 2 Callback Function
void *Thread2Func(void *pArg)
{
   int* p = (int*)pArg;
   int myNum = *p; // myNum = 1

   sleep(0.1); // 100ms sleep

   pthread_mutex_lock(&g_Mutex);
   g_Val += 2;
   printf( "Thread Id: %d. Global value increased by 2!\n", myNum);
   pthread_mutex_unlock(&g_Mutex);
   pthread_cond_signal(&g_Cond);


   // Possible solution if Thread 1 Callback function is delayed
   pthread_mutex_lock(&g_Mutex);
       while(g_Val < 3)
       {
           pthread_mutex_unlock(&g_Mutex);
           pthread_cond_signal(&g_Cond);
           sleep(0.1);
           pthread_mutex_lock(&g_Mutex);
       }
   pthread_mutex_unlock(&g_Mutex);

   return NULL;
}
```