TAMPEREEN TEKNILLINEN YLIOPISTO
TAMPERE UNIVERSITY OF TECHNOLOGY

JUHA LÖFLUND

FAIR THREAD SYNCHRONIZATION

Bachelor of Science Thesis

Examiner: University Lecturer Maarit Harsu
Submitted for review on 7 May 2018

ABSTRACT

JUHA LÖFLUND: Fair thread synchronization
Tampere University of technology
Bachelor of Science Thesis, 21 pages, 6 Appendix pages
May 2018
Bachelor's Degree Programme in Information Technology
Major: Software Engineering

Keywords: threading, mutual exclusion, thread synchronizing

Available computing power in modern computers has been steadily increasing in the form of processor cores. To effectively utilize the available computing power, computed programs must utilize multiple threads. Due to the usage of threads, some form of thread synchronization must also be utilized. However, readymade thread synchronization methods may not always be ideal for all situations. This thesis is examining what are the problems with currently offered solutions, and what is the impact of potential problems.

As can be discovered, some readymade solutions have problems when threads are synchronized according to thread priority. This can lead to a situation where some lower priority thread does not get execution time at all. This thesis introduces a fair thread synchronization technique as a possible solution to the problem, and a case study comparing the fair thread synchronization technique to a readymade solution is carried out. The problem can be clearly seen from the results of the case study. In addition, according to the results, the fair thread synchronization technique can be a utilized as a solution for the problem.

CONTENTS

1. INTRODUCTION 1
2. THREAD SYNCHRONIZATION 2
  2.1 Concurrency and parallelism 2
  2.2 Condition synchronization 3
  2.3 Critical section and mutual exclusion 3
  2.4 Semaphores 4
  2.5 Mutex lock 4
  2.6 Support in high-level programming languages 4
3. SYNCHRONIZING THREADS FAIRLY 6
  3.1 Problem 6
  3.2 *ThreadSynchornizer, a* FIFO-based solution 6
  3.3 Pros and cons of the solution 7
4. CASE STUDY: COMPARING *THREADSYNCHRONIZER* WITH *QWAITCONDITION OF QT* 9
  4.1 Testing methods 9
  4.2 Performance measurements 10
    4.2.1 All threads having same priority 10
    4.2.2 Single thread having a higher priority 11
  4.3 Fairness comparison 11
5. FURTHER APPLICATIONS OF *THREADSYNCHRONIZER* 15
6. CONCLUSIONS 19
REFERENCES 20
APPENDIX A: THREAD SYNCHRONIZATION WITH QT 22
APPENDIX B: THREAD SYNCHRONIZATION WITH C++ 24
APPENDIX C: THREAD SYNCHRONIZATION CLASS WITH QT 26

1. INTRODUCTION

In modern computers, the number of processor cores and parallel threads has been steadily increasing [1]. Current high-end CPU's support up to 36 parallel threads [2] and a typical mainstream CPU usually has four cores and offers eight parallel threads [3]. To fully utilize the available computing power, modern computer programs must be implemented to support multiple concurrent threads [1]. This allows the application to benefit from the available computing power more efficiently, by splitting a single task or an operation into multiple subparts and executing each subpart in parallel [4].

Some form of thread synchronization must be utilized by parallel programs fielding multiple threads. A variety of thread synchronization methods can be employed, depending on set requirements, used operating system and selected programming language. Modern operating systems offer APIs for managing threads and thread execution, but these are not always optimal from programmer's point of view. These API's are especially problematic for the reusability of software components. When these API's are utilized in a software component, that component becomes dependent to this specific operating system. However, operating system independent libraries for managing thread execution are included in modern high-level programming languages such as C++11 [1].

This thesis is looking at how to synchronize N-number of threads in a fair manner easily with high-level programming language. This means that when multiple threads are accessing the same resource, they are synchronized with First-In, First Out –principle, also referred as FIFO.

Thread synchronization at a general level is discussed in Chapter 2. The chapter also takes a brief overlook of the Qt framework, specifically threading related classes. Fair thread synchronization is discussed in Chapter 3 and a solution for fair synchronization is introduced. Chapter 4 is a case study where the solution for the fair thread synchronization introduced in Chapter 3, is compared to an existing solution. Some additional applications for the fair thread synchronization solution, are introduced in Chapter 5. Conclusions of the thesis are discussed in Chapter 6.

2. THREAD SYNCHRONIZATION

2.1 Concurrency and parallelism

A program's concurrency is defined as having more than one thread and each of the threads can be executed in any order, independently of each other. Any given thread can be executed before or after another or all threads can be executed simultaneously. Concurrent execution of two threads is demonstrated in Figure 1. The shown threads are concurrent, but not parallel.

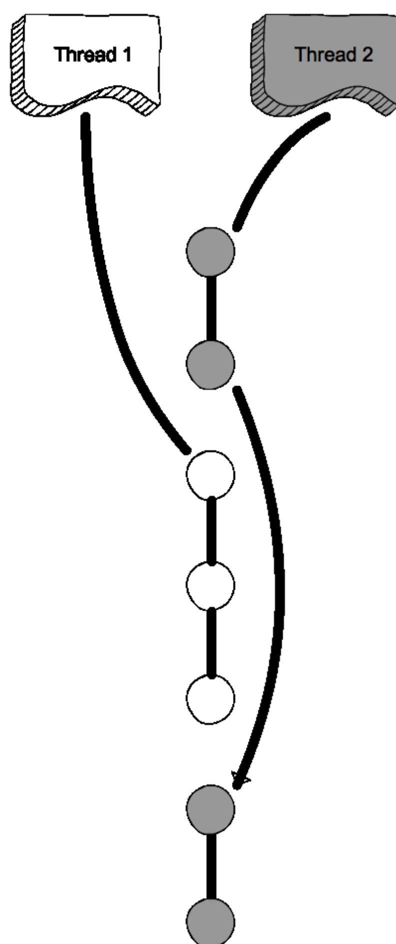

Figure 1 *Concurrent execution of two threads [5]*

On the other hand, parallelism specifically refers to simultaneous execution of different threads as demonstrated in Figure 2. This means that all parallel programs are concurrent, but not all concurrent programs are parallel. Parallel execution of threads always requires that the used computer or similar device, supports multiple parallel threads, i.e. a multi-core processor is used. [6]

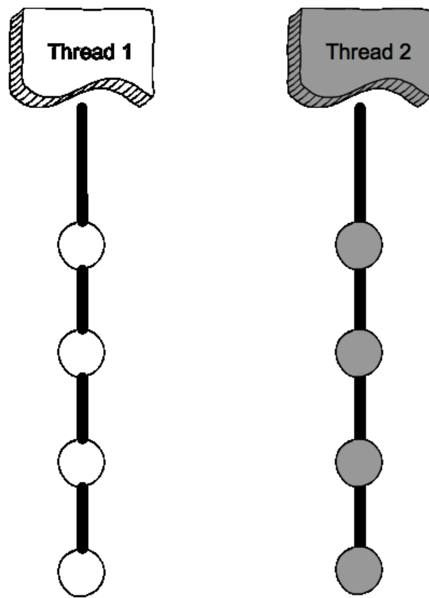

Figure 2 *Simultaneous execution of two threads [5]*

2.2 Condition synchronization

Condition synchronization ensures that a certain predefined condition is met, before an action occurs [4]. In other words, condition synchronization ensures that a thread cannot proceed until a given condition is met [7].

A classic example of condition synchronization is a thread waiting for user input via a keyboard. When a keystroke occurs, the waiting thread is signaled, and it can proceed with its execution. [8]

2.3 Critical section and mutual exclusion

A section of a program, which can be accessed by multiple threads, but only a single thread is permitted to be inside the section at any given time, is defined as a critical section. In the critical section some shared or common resource is usually accessed. [1, 4]

Mutual exclusion is a solution for solving the critical section problem, as mutual exclusion blocks other threads from executing the same section of a program. In other words, mutual exclusion only allows a single thread within a critical section. [1] Synchronization objects such as locks and semaphores provide mutual exclusion [4].

Microsoft Windows API defines some concurrency terminology in a contradictory fashion. The API defines a *Critical Section* [9] and a *Mutex* [10], but the meaning differs from definitions from this section and Section 2.5. A *Critical Section* object is defined in Windows as providing thread synchronization inside a single process. In contrast, in Windows a *Mutex* is defined as providing thread synchronization across processes.

2.4 Semaphores

Semaphore is a synchronization object providing condition synchronization and mutual exclusion. In contrast, locks can only provide mutual exclusion. Depending on a use-case, semaphores can be utilized as synchronization objects or locks. [4]

A counting semaphore permits a certain, predefined number of threads to execute. When a thread is accessing a limited or finite resource, it uses the semaphore to allocate the needed resources to itself. Internally the semaphore reduces the number of available slots. And vice versa, when the acquired resource is relinquished by the thread, the number of slots is incremented by the same amount that was allocated previously. [4] When the semaphore has no free slots, the requesting thread becomes blocked until a slot is available [4]. This pattern of semaphores is utilized in resource allocation, where multiple threads are competing for a limited number of resources [4].

Mutex pattern semaphore is the most commonly used semaphore. The semaphore is initialized to have a single available slot. Mutex pattern semaphore can be utilized to protect critical sections, as only a single thread is permitted to access the shared resource. [4]

2.5 Mutex lock

Lock or mutex lock is a similar synchronization object to a mutex pattern semaphore, utilized to solve the critical section problem. Where a lock can only provide mutual exclusion, mutex pattern semaphore can be utilized to provide conditional synchronization as well. [4]

Additionally, a mutex lock has an owner, which has a significant role when comparing a mutex lock and a mutex pattern semaphore. As the mutex lock has an owner, the lock forces the thread that has locked it to also unlock it. Unlocking the mutex from a different thread is undefined behavior, usually resulting in termination of the program. On the other hand, semaphores can be handled from different threads. [4]

2.6 Support in high-level programming languages

Qt is a cross-platform framework for desktop, embedded and mobile applications. It supports multiple different platforms including Linux, OS X, Windows, Android and many others. Qt is written in C++ and it is not a programming language in its own. Qt framework extends C++, and with an additional pre-compilation step, code written in Qt is generated into native C++ code. [11] Within the scope of this thesis, certain threading related classes provided by Qt Core library are needed [12].

Qt framework provides mutual exclusion with *QMutex* class. The class provides a simple interface for locking and unlocking a lock. Additionally, it provides a method testing the

lock, meaning that if obtaining the lock fails, the method will return after the provided timeout expires. In contrast, normal lock-method will block until the lock has been obtained. *QMutex* is strictly intended for providing mutual exclusion as only the current owner of the mutex can unlock it and unlocking from another thread results in undefined behavior. [13] *QMutexLocker*, a utility class is also provided for easy usage of *QMutex*. When *QMutexLocker* instance is created, the created object obtains the ownership of the mutex and relinquishes the ownership when the *QMutexLocker* is destroyed. *QMutexLocker* also provides methods for manually unlocking and relocking the mutex. [15] Classes with similar functionalities are also offered in C++ [16, 17].

For synchronizing threads, Qt offers *QWaitCondition* class to provide a condition variable. *QWaitCondition* allows inter-thread communication where one thread waits until it receives a notification from another thread. When a thread starts its wait cycle, it provides a locked *QMutex* to the *QWaitCondition*. The provided mutex is unlocked for the duration of the wait and after the thread returns from the wait cycle, the mutex is automatically locked by *QWaitCondition.* This makes *QWaitCondition* very helpful when synchronizing access into a critical section for multiple threads, because the locked mutex grants access to the critical section for the woken thread. [8] Functionality wise *QWaitCondition* is very similar to *std::condition_variable* included in C++ [18].

QSemaphore provides a general counting semaphore. The class is a generalization of a mutex. It is possible to acquire a semaphore multiple times, whereas a mutex can be locked only once. Additionally, *QSemaphore* offers method for acquisition, which do not block indefinitely like a normal *acquire*-method. [19]

3. SYNCHRONIZING THREADS FAIRLY

3.1 Problem

When synchronizing multiple threads, the different thread priorities may affect how each thread gets synchronized. Threads having a higher priority may always be placed in front of threads having a lower priority. This can lead to situations where some threads will get very little execution time. This can lead to unpredictable delays in application execution, e.g. if UI-thread is pending results from a low-priority background thread, it may take a very long time for the results to become available.

Synchronization objects provided in C++ or Qt do not specify in what order pending threads get execution time [8, 18]. Issues mentioned in the previous paragraph can arise, if thread priority is used to determine when a thread gets execution time.

A simple test program can be used to determine how different synchronization objects behave. This can be achieved by creating multiple threads with various priorities within the test program and synchronizing the created threads with desired synchronization object. Each thread is assigned a unique identifier to distinguish between them. Whenever a thread wakens, the thread's unique identifier is stored (in this case, printed to program output) and the thread goes back to the synchronization object and waits to be signaled again. The test is repeated enough many times to see how the synchronization object handles the pending threads. The aforementioned test program implemented with Qt and utilizing *QWaitCondition* as the synchronization object, is shown in Appendix A.

Running the test program clearly shows that a thread having a higher priority can completely starve out other threads. Based on the results from the test program, it can be hypothesized that the thread having the highest priority is always placed on top of the queue. Even though the specification of *QWaitCondition* does not mention this, practical evidence supports this assumption.

However, different synchronization variables may behave differently, which can be seen when the test program is implemented in C++ and utilizing *std::condition_variable* as the synchronization object. The program is shown in Appendix B. The priorities of different threads do not affect the scheduling order; threads are executed in the same order in each cycle.

3.2 *ThreadSynchronizer, a* FIFO-based solution

One possible solution to the problem shown in the beginning of this chapter, is to implement a thread synchronization mechanism, where each pending thread gets execution

time based how long they have been waiting, effectively ignoring the priorities of each thread. This method of operation can also be described as first-in-first-out or FIFO. This solution ensures fair scheduling of pending threads as each thread will get execution time in the order of arrival.

To overcome the uncertainty of scheduling in various thread synchronization objects, *ThreadSynchronizer* shown in Appendix C utilizes separate synchronization objects for each pending thread. The utilization of separate synchronization objects removes the uncertainty brought by synchronization objects and possible differences in operating system specific implementations of these objects.

When a thread requires synchronization and uses the *Wait*-method of *ThreadSynchronizer* from line 36, the function allocates a new instance of *QWaitCondition* specifically for this function call. A pointer to the created object is stored in the internal queue of *ThreadSynchronizer*. After storing the pointer, the thread begins the wait cycle. *QWaitCondition* is allocated on the stack of the calling thread and thus the object is destroyed when the thread returns from *Wait*. A thread can return from the *Wait*-method by one of two ways:

1) Thread gets waken when *WakeFirst* method in line 62 is called by another thread.
2) The given timeout expires, which can be supplied as an optional parameter in the function call in line 37.

When the *Wake*-method is called, the function obtains a pointer to a *QWaitCondition* for a thread which has been waiting for the longest and wakes the waiting thread by calling the *QWaitCondition*. The function also removes the obtained pointer from the internal queue to ensure that a single thread does not get multiple wake calls.

3.3 Pros and cons of the solution

The solution presented in the previous section introduces added control over thread execution and synchronization. In certain situations, the added control can be beneficial, but it also has downsides. Because an additional layer of code is introduced, and this can affect the performance, as can be seen from performance measurements in Section 4.2. Whether this difference in performance is acceptable depends on application and context. Solution guarantees fair scheduling, and this can be beneficial and even mandatory in some cases.

As shown in Section 3.2, every time the *Wait*-method of *ThreadSynchronizer* is called, an instance of *QWaitCondition* is created and destroyed once the function call returns. These objects could be recycled, either within a single instance of *ThreadSynchronizer* or between all instances of *ThreadSynchronizer* within a process. This would require changing the allocation scheme of *QWaitCondition* from stack allocation to heap allocation, as

the lifecycle of stack allocated objects is much stricter than heap allocated ones [20]. In such a recycling scheme, objects would be fetched from a cache or created if none are available and returned to the cache once the object becomes free. Finding out whether this improvement idea is practical requires careful analysis and performance measurements to support the feasibility. Based on the source codes, *QWaitCondition* has an instance-wide recycling scheme [21], which would support the feasibility of recycling.

As threads are having a priority for a reason, ignoring it might cause problems for high-priority threads. For this reason, the FIFO-based queue utilized in *ThreadSynchronizer* could be replaced by a better solution. One possible solution meriting further research could be a simplified derivative of a scheduling algorithm known as aging [22]. In aging, the priority of a process is gradually increased until it gets execution time [22]. In the simplified version of this algorithm, threads would be sorted by priority in the queue. Additionally, every time a higher priority thread surpasses a lower priority thread, the priority of the lower priority thread is incremented. The approach prevents indefinite thread starvation, while allowing high priority threads faster access to the desired resource.

4. CASE STUDY: COMPARING *THREADSYNCHRONIZER* WITH *QWAITCONDITION OF QT*

This chapter is comparing *ThreadSynchronizer* introduced in Chapter 3 to *QWaitCondition*. The first section is covering testing methods for comparing *ThreadSynchronizer* to *QWaitCondition*. The second section is comparing performance results. The fairness of each is analyzed in the third section.

4.1 Testing methods

For measuring the differences between the two solutions, using various measurement points, a testing environment was devised. The testing environment was using ten concurrent threads, which were synchronized with both *ThreadSynchronizer* and *QWaitCondition*. The testing environment measured how long a time thread spent in the queue waiting to be awoken.

To verify that threads were waken in the same order as they entered the queue, every time a thread called the test function, a unique identifier was generated. This unique identifier, together with a thread identifier, was stored before entering the queue and after leaving it.

Threads were woken at a predefined interval, which were one, ten and one hundred milliseconds. After a thread was waken and the result stored, it immediately went back to wait in the queue. However, this is really the worst-case situation and may not represent real world situations very well. For this reason, a test execution with a sleep was included. After a thread had completed its single execution cycle, it slept twice that of the wake interval.

Threads accessing a shared resource may also have different priorities. This was also included in the tests by incrementing the priority of a single thread.

To summarize, all the same tests were executed for both *QWaitCondition* and *ThreadSynchronizer*. Each test was executed with the aforementioned wake intervals and included the following items:

1. all threads using same priority, no sleep between access cycles
2. all threads using same priority, sleep between access cycles
3. one thread having higher priority, no sleep between access cycles
4. one thread having higher priority, sleep between access cycles.

To reduce randomness, the duration of each test item was set to 6000 ms and multiplied with the wake interval. Each test was also repeated ten times.

4.2 Performance measurements

This section is looking at how *ThreadSynchronizer*, listed in Appendix C, is performing when compared to *QWaitCondition*. In this context, performance is measured by how long a time thread has to wait to get execution time and how consistent the wait time is.

4.2.1 All threads having same priority

When all threads have the same priority, the difference between the two solutions is quite small as can be seen by comparing average time spent in queue, as depicted in Table 1. However, when calculating average deviation, more meaningful differences can be seen between the two solutions with longer wake intervals. Average deviation reveals at how consistent intervals threads are waken.

Table 1. *Time spent waiting for execution on average (averages of all threads).*

Wake interval (ms)	Average time in queue (ms)		Average deviation (ms)	
	QWaitCondition	***ThreadSynchronizer***	***QWaitCondition***	***ThreadSynchronizer***
1	19.999	20.008	0.019	0.033
10	110.073	110.037	0.145	0.074
100	1010.157	1010.042	0.298	0.039

Theoretically, the average time a thread spends waiting in the queue, should be close to the wake interval. However, it can be seen from the above table that this not so. Difference between calculated and reality is especially large when a thread is wakened every millisecond. This difference is likely due to accumulated overhead. The overhead can be estimated:

$$O = \frac{A - T * C}{C} = \frac{A}{C} - T,$$

where O denotes the overhead, A means the average time in queue, T means the wake interval, and C means the number of threads. Applying the above formula, the approximate overhead can be calculated as 1.0 ms.

4.2.2 Single thread having a higher priority

When upgrading the priority of a single thread to a higher one, the results for *ThreadSynchronizer* stay virtually the same as in Section 4.2.1. However, results change significantly for *QWaitCondition*. The thread having higher priority starves the other threads completely.

Table 2. *Time spent waiting for execution on average (averages of all threads), with additional sleep.*

Wake interval (ms)	Average time in queue (ms)		Average deviation (ms)	
	QWaitCondition	**ThreadSynchronizer**	***QWaitCondition***	**ThreadSynchronizer**
1	16.984	17.029	15.980	0.050
10	88.981	89.026	87.960	0.037
100	808.377	809.014	797.205	0.090

When the same test is executed with the sleep described in Section 4.1, averages for both solutions are close together as seen in Table 2. However, in the same table, the average deviations for *QWaitCondition* are very different. The likely explanation is that *QWaitCondition* stores pending threads in a priority-based queue, as seen from Qt's source codes [21].

Because of priority-based queueing, a thread having the highest priority is always placed as the first item in the queue, thus getting more execution time, which can be seen in Table 3. Thread number 10 has higher priority than the other threads and the average time it spent in the queue waiting for execution time is far less compared to the others.

Table 3. *Thread-wise time spent waiting for execution, on average, with additional sleep using QWaitCondition.*

Wake interval (ms)	Average time in queue (ms)		Average deviation (ms)	
	Threads 1-9	**Thread 10**	**Threads 1-9**	**Thread 10**
1	32.965	1.005	0.117	0.008
10	176.934	1.015	0.401	0.031
100	1597.156	3.082	32.267	4.018

4.3 Fairness comparison

As described in Section 4.1, the generated unique identifier is used to verify in what order the threads are entering and exiting the queue. The data also allows to calculate how much available resource a single thread consumes when compared to other threads. Table 6 demonstrates the recorded data for *QWaitCondition* at 100 ms wake interval. The first

and the second columns show the thread number and the generated identifier when a thread is entering the queue. The latter two columns show the same information when a thread is exiting the queue. When threads are exiting the queue in the same order as they entered, each row has the same information in columns one and two as in columns three and four as shown in Table 4. However, when a thread has been surpassed in the queue, the identifier indicating when the thread entered the queue, is listed below the row when the thread entered the queue. Similarly, when a thread is surpassing others in the queue, the thread's exit point is listed above the row when it originally entered the queue. Within this section, all tables demonstrating thread synchronization order contain only a small section of the original data as the overall amount of data is too large.

When all threads were having the same priority, no differences can be seen between *QWaitCondition* and *ThreadSynchronizer*, and both can be stated as fair in this regard. Findings are similar as in Section 4.2.1. Regardless of the used wake interval, the synchronization order for both solutions is similar to results in Table 4 for *ThreadSynchronizer* at 100 ms wake interval.

Table 4. *Thread synchronization order with ThreadSynchronizer at 100 ms wake interval.*

	Entering queue		Exiting queue
Thread number	**Unique identifier**	**Thread number**	**Unique identifier**
1	2215933149	1	2215933149
2	45564477	2	45564477
3	2280578173	3	2280578173
4	611728441	4	611728441
5	3833064745	5	3833064745
6	1779456281	6	1779456281
7	545730938	7	545730938
8	1880964532	8	1880964532
9	1996980587	9	1996980587
10	878211633	10	878211633

When a single thread is having a higher priority than other threads, bigger and more meaningful differences are visible for *QWaitCondition*. In this scenario, the single thread having the higher priority will complete starve out other threads, similarly as in section 4.2.2. The starvation can also be seen from Table 5. Initially, all threads enter the queue, but only a single thread gets any execution time and all other threads become complete starved. But as the stated in Section 4.1, this is not a fully realistic scenario, and this is compensated with a sleep time described in the same section.

Table 5. *Thread starvation with QWaitCondition.*

Entering queue		Exiting queue	
Thread number	**Unique identifier**	**Thread number**	**Unique identifier**
1	3108412803	10	3869406454
2	1271899378	10	1893594946
3	3812422854	10	3996173184
4	1521659203	10	902101116
5	1280725400	10	4190281462
6	4034391223	10	1900172198
7	2190315840	10	4034412022
8	442166512	10	3304833470
9	2577950124	10	3203927030
10	3869406454	10	3005580724
10	1893594946	10	4028571215
10	3996173184	10	661271208
10	902101116	10	632818809
10	4190281462	10	719981133

With described sleep time using *QWaitCondition*, the thread having the higher priority no longer complete starves out the other threads, but still consumes about 50 % of available resources. The remaining 50 % is equally divided between the remaining threads. Table 6 demonstrates this behavior. When the test is stated, each thread enters the queue for the first time and for the first full test cycle, threads are in order. However, when looking at the exit order of the threads, thread number ten is listed in every second row. After initial test cycle, this can also be seen when threads are entering the queue. As can be expected from the design of *ThreadSynchronizer*, no differences were visible between different threads.

Table 6. *Thread synchronization order with QWaitCondition at 100 ms wake interval.*

Entering queue		Exiting queue	
Thread number	**Unique identifier**	**Thread number**	**Unique identifier**
2	517121028	10	888715374
3	64217383	2	517121028
1	3683312599	10	173490974
4	2936409884	3	64217383
5	965584088	10	4150017827
6	949489883	1	3683312599
7	3623746235	10	1162338631
8	2933268189	4	2936409884
9	3296582518	10	2782934254
10	888715374	5	965584088
10	173490974	10	2177001493

2	1456985834	6	949489883
10	4150017827	10	3678744266
3	589854715	7	3623746235
10	1162338631	10	3202488071
1	1798102511	8	2933268189
10	2782934254	10	2462570411
4	896185401	9	3296582518
10	2177001493	10	2794752915
5	2398519705	2	1456985834
10	3678744266	10	2333037547
6	969774479	3	589854715
10	3202488071	10	632389621
7	2118486544	1	1798102511
10	2462570411	10	628342185
8	1780033093	4	896185401
10	2794752915	5	2398519705
9	3446071464	10	2535960579
10	2333037547	6	969774479

5. FURTHER APPLICATIONS OF *THREADSYNCHRONIZER*

Problems highlighted in Chapter 3 for priority-based thread synchronization can be an issue in other synchronization objects as well. If thread synchronization needs to be controlled beyond synchronization primitives provided by the programming language, *ThreadSynchronizer* can be utilized to implement various other thread synchronization objects.

A simplified version of a counting semaphore utilizing *ThreadSynchronizer* is shown in Program 1. The constructor of the class takes the initial number of free slots as a parameter. The given value is used to initialize the instance of *CountingSemaphore*. When a resource is required to be allocated to a thread, *Acquire* method in line 9 is called. The method checks if any slots are available and allocates a slot if one is available. When no slots are available, the calling thread is blocked until a slot is freed by another thread. Thread can relinquish the allocated slot by calling *Release* method in line 26. The method frees the slot and signals the first waiting thread that a slot has been freed. Section 2.4 contains more information about counting semaphores.

```
    #include "ThreadSynchronizer.h"
 2
    class CountingSemaphore final
 4  {
    public:
 6      CountingSemaphore(const unsigned int availableSlots) :
                                  m_slots(availableSlots) {}
 8
        void Acquire()
10      {
            // Utilize RAII-pattern QMutexLocker class
12          // to obtain the ownership of the mutex
            QMutexLocker locker(&m_lock);
14          if (m_slots > 0)
            {
16              // Slot available, acquire it
                m_slots--;
18          }
            else if (m_synchronizer.Wait(locker))
20          {
                // Slot released by another thread, acquired it
22              m_slots--;
            }
24      }

26      void Release()
        {
28          QMutexLocker locker(&m_lock);
            // Release the slot by incrementing the number of free slots
```

```
30          m_slots++;

32          // Signal a pending thread that a slot is now free
            m_synchronizer.WakeFirst();
34      }

36  private:
        unsigned int m_slots;
38      QMutex m_lock;
        ThreadSynchronizer m_synchronizer;
40  };
```

Program 1. *Example of a simple counting semaphore implemented utilizing ThreadSynchronizer.*

The example program is oversimplified, and it does not have all the needed runtime checks, for example *Release* function in line 26 can be called more times than *Acquire*. But despite the shortcomings of the example program, it is fully working and could be utilized as a counting semaphore as currently implemented.

With a RAII-pattern [14] utility class, the usage of *CountingSemaphore* can be made more convenient as demonstrated by Program 2. The counting semaphores *Acquire* method is called in the constructor of the convenience class as seen in line 11. Similarly, the destructor calls release when the object is destroyed. The convenience class also offers methods for manually releasing and reacquiring the slot. Either *Release* or *ReAcquire* methods do nothing if called multiple times. *ReAcquire* method may block the calling thread if all free slots have been allocated.

```
    #include "CountingSemaphore.h"
2   #include <atomic>

4   class CountingSemaphoreUtility
    {
6   public:
         CountingSemaphoreUtility(CountingSemaphore& semaphore):
8                                   m_semaphore(semaphore),
                                    m_isReleased(false)
10      {
            m_semaphore.Acquire();
12      }

14      ~CountingSemaphoreUtility()
        {
16          Release();
        }
18
        void Release()
20      {
            bool isReleased = false;
22          // If Release is not yet called (m_isReleased is false),
            // compare_exchange_weak sets m_isReleased to true
24          // and returns true
            if (m_isReleased.compare_exchange_weak(isReleased, true))
```

```
26          {
                m_semaphore.Release();
28          }
        }
30
    void ReAcquire()
32  {
            bool isReleased = true;
34          // If Release has been called (m_isReleased is false),
            // compare_exchange_weak sets m_isReleased to false
36          // and returns true
            if (m_isReleased.compare_exchange_weak(isReleased, false))
38          {
                m_semaphore.Acquire();
40          }
    }
42
    private:
44      CountingSemaphore& m_semaphore;
        std::atomic<bool> m_isReleased;
46  };
```

Program 2. *RAII-pattern [14] convenience class for counting semaphore.*

A RAII-pattern utility class is especially useful when it is utilized in a function with multiple return points or when exceptions can be raised. The RAII-pattern guarantees that the acquired resource is released when the utility class instance goes out of scope, regardless whether this happens due a raised exception or a return statement. [14] Program 3 demonstrates the problem with traditional approach by using the *CountingSemaphore* directly. The example function in the program has two return points, either if the first function call returns false in line 11, or at the very end of the example function. However, if either of the function used by *MyExampleFunction* raises an exception, the acquired resource is never released. Using a utility class solves these problems and slightly simplifies the code as shown in Program 4.

```
    #include "CountingSemaphore.h"
 2
    // Statically allocated CountingSemaphore
 4  static CountingSemaphore s_countingSemaphore;

 6  void MyExampleFunc()
    {
 8      // Acquire one slot/resource etc.
        s_countingSemaphore.Acquire();
10
        if (!FunctionOne())
12      {
            // Release the acquired resource.
14          s_countingSemaphore.Release();
            return;
16      }

18      FunctionTwo();
```

```
20      // Release the acquired resource.
        s_countingSemaphore.Release();
22  }
```

Program 3. *Example of direct resource allocation*

This convenience class allows to create stack allocated objects, which are automatically destroyed when the object goes out of scope. The usage of *CountingSemaphoreUtility* is demonstrated in Program 4. The program contains a statically allocated instance of *CountingSemaphore,* as shown in line 5. That instance is then utilized by *CountingS-emaphoreUtility* in line 10 to allocate a resource for the duration of the function call.

```
    #include "CountingSemaphore.h"
 2  #include "CountingSemaphoreUtility.h"

 4  // Statically allocated CountingSemaphore
    static CountingSemaphore s_countingSemaphore;
 6
    void MyExampleFunc()
 8  {
        // Acquire one slot/resource etc.
10      CountingSemaphoreUtility wrapper(s_countingSemaphore);

12      if (!FunctionOne())
        {
14          // ‘wrapper’ -object goes out of scope and is destroyed.
            return;
16      }

18      FunctionTwo();
    } // ‘wrapper’ -object goes out of scope and is destroyed.
20    // CountingSemaphore::Release is called by the destructor of
      // CountingSemaphoreUtility
```

Program 4. *Example usage of CountingSemaphoreUtility*

6. CONCLUSIONS

As the readymade thread synchronization methods and techniques provided by different programming languages may not always be ideal for all situations, the foundation of this thesis was to find a solution for fair thread synchronization. Thread synchronization can be especially problematic when threads are synchronized according to thread priority and when some threads are having a higher priority than the other threads. In extreme cases, this can result in complete starvation of the lower priority threads, as demonstrated by the case study in Section 4.2.2. A possible solution to this problem is introduced in Section 3.2. However, as can be seen from the performance results in the case study in Section 4.2 for the solution, the added logic can introduce some minor performance reduction, although the difference is minimal at worst.

In normal circumstances, thread synchronization methods and techniques offered by a programming language are usually adequate as well as straightforward and easiest to implement. However, when the need for more specialized solutions arises, the solution suggested in this thesis can be beneficial.

An improvement idea for enhancing the functionality of *ThreadSynchronizer* by replacing the FIFO-based thread synchronization with an applied version of a scheduling algorithm known as aging, is introduced in Section 3.3. The applied algorithm takes thread priorities into account, allowing high priority threads faster access to the desired resource. Additionally, the approach also prevents the indefinite thread starvation of lower priority threads. The details of the algorithm are described in Section 3.3. As the algorithm takes thread priorities better into account, it might be a better overall solution when compared to the current implementation of *ThreadSynchronizer*. However, the additional logic required to implement the algorithm might cause performance degradation. Further research into the feasibility of this algorithm is merited.

REFERENCES

[1] M. Gregoire, N.A. Solter, S.J. Kleper, Professional C++, 2nd ed. Wrox, Hoboken, 2011, 1106 p.

[2] Intel® Core™ i9-7980XE Extreme Edition Processor, Intel Corporation, web page. Available (accessed 1.2.2018): https://www.intel.com/content/www/us/en/products/processors/core/x-series/i9-7980xe.html.

[3] Intel® Core™ i7-8700K Processor, Intel Corporation, web page. Available (accessed 1.2.2018): https://www.intel.com/content/www/us/en/products/processors/core/i7-processors/i7-8700k.html.

[4] R.H. Carver, K. Tai, Modern Multithreading: Implementing, Testing, and Debugging Multithreaded Java and C++/Pthreads/Win32 Programs, John Wiley and Sons, 2005, 465 p.

[5] P. Pitkämäki Parallel vs concurrent in Node.js, web page. Available (accessed 2.4.2018): https://bytearcher.com/articles/parallel-vs-concurrent/.

[6] D. Buttlar, J. Farrell, B. Nichols, PThreads Programming, O'Reilly Media, 1996, 267 p.

[7] M.L. Scott, Programming Language Pragmatics, 2nd ed. Morgan Kaufmann Publishers Inc, San Francisco, 2005, 915 p.

[8] QWaitCondition Class | Qt Core 5.5, The Qt Company Ltd, web page. Available (accessed 04.03.2018): http://doc.qt.io/archives/qt-5.5/qwaitcondition.html#wakeOne.

[9] Critical Section Objects (Windows), Microsoft Corporation, web page. Available (accessed 18.02.2018): https://msdn.microsoft.com/en-us/library/windows/desktop/ms682530(v=vs.85).aspx.

[10] Mutex Objects (Windows), Microsoft Corporation, web page. Available (accessed 18.2.2018): https://msdn.microsoft.com/en-us/library/windows/desktop/ms684266(v=vs.85).aspx.

[11] About Qt - Qt Wiki, The Qt Company Ltd, web page. Available (accessed 2.4.2018): http://wiki.qt.io/About_Qt.

[12] Threading Classes | Qt 5.10, The Qt Company, web page. Available (accessed 2.4.2018): https://doc.qt.io/qt-5/thread.html.

[13] QMutex Class | Qt Core 5.10, web page. Available (accessed 6.4.2018): http://doc.qt.io/qt-5/qmutex.html.

[14] RAII - cppreference.com, web page. Available (accessed 2.4.2018): http://en.cppreference.com/w/cpp/language/raii.

[15] QMutexLocker Class | Qt Core 5.10, The Qt Company Ltd, web page. Available (accessed 6.4.2018): http://doc.qt.io/qt-5/qmutexlocker.html.

[16] lock_guard - C++ Reference, web page. Available (accessed 6.4.2018): http://www.cplusplus.com/reference/mutex/lock_guard/.

[17] mutex - C++ Reference, web page. Available (accessed 6.4.2018): http://www.cplusplus.com/reference/mutex/mutex/.

[18] condition_variable - C++ Reference, web page. Available (accessed 04.03.2018): http://www.cplusplus.com/reference/condition_variable/condition_variable/.

[19] QSemaphore Class | Qt Core 5.5, The Qt Company Ltd, web page. Available (accessed 6.4.2018): http://doc.qt.io/archives/qt-5.5/qsemaphore.html.

[20] Dobry Tep, Kinariwala Bharat, Programming in C, University of Hawai'i, 1993, 598 p.

[21] QWaitCondition_win.cpp, The Qt Company Ltd, web page. Available (accessed 18.02.2018): http://code.qt.io/cgit/qt/qt.git/plain/src/corelib/thread/qwaitcondition_win.cpp.

[22] A. Silberschatz, P. Baer Galvin, G. Gagne, Operating System Concepts, 9th ed. John Wiley & Sons, Inc, 2013, 919 p.

APPENDIX A: THREAD SYNCHRONIZATION WITH QT

```cpp
#include <QCoreApplication>
#include <QThread>
#include <QWaitCondition>
#include <QMutex>
#include <QDebug>
#include <QVector>

// Thread storage
QVector<QSharedPointer<QThread>> threads;

// Synchronization objects
QWaitCondition condition;
QMutex mutex;

bool runTests = true;

QVector<QPair<QThread::Priority, QString>> test_data = []
{
    QVector<QPair<QThread::Priority, QString>> data;
    data.append(qMakePair(
                    QThread::NormalPriority,
                    QString("Thread_1_NormalPriority")));
    data.append(qMakePair(
                    QThread::NormalPriority,
                    QString("Thread_2_NormalPriority")));
    data.append(qMakePair(
                    QThread::NormalPriority,
                    QString("Thread_3_NormalPriority")));
    data.append(qMakePair(
                    QThread::LowPriority,
                    QString("Thread_4_LowPriority")));
    data.append(qMakePair(
                    QThread::HighPriority,
                    QString("Thread_5_HighPriority")));
    return data;
}();

class Thread: public QThread
{
public:
    Thread(): QThread() {}

    ~Thread() override
    {
        condition.wakeAll();
        QThread::wait();
    }
```

```
48
   protected:
50     virtual void run() override
       {
52         while (runTests)
           {
54             // Obtain the ownership of the mutex
               QMutexLocker lock(&mutex);
56
               // Wait until signaled
58             condition.wait(lock.mutex());

60             // Print object name
               qDebug() << QObject::objectName();
62         }
       }
64 };

66 QSharedPointer<QThread> Create(QThread::Priority priority,
                                  QString name)
68 {
       QSharedPointer<Thread> pTester(new Thread());
70     pTester->setObjectName(name);
       pTester->start(priority);
72     return pTester;
   }
74
   void Test_QWaitCondtion()
76 {
       // Create the threads
78     for (auto testData : test_data)
       {
80         // Create a thread
           threads << Create(testData.first, testData.second);
82     }

84     // Start testing
       auto testIterations = 100;
86     for (auto i = 1; i <= testIterations; ++i)
       {
88         // Wake one thread from QWaitCondition
           condition.wakeOne();
90
           // Sleep for a 100ms before waking the next thread
92         QThread::msleep(100);
       }
94
       // Stop & destroy the threads
96     runTests = false;
       threads.clear();
100 }
```

APPENDIX B: THREAD SYNCHRONIZATION WITH C++

```
#include <iostream>
#include <thread>
#include <mutex>
#include <vector>
#include <condition_variable>
#include <windows.h> // SetThreadPriority
#include <string>
#include <memory> // std::unique_ptr

// Container to hold the threads
std::vector<std::unique_ptr<std::thread>> threads;

// Thread synchronization objects
std::mutex mutex;
std::condition_variable condition;

// Simple boolean to control the execution of threads
bool runTests = true;

// Test data
typedef int PRIORITY;
typedef std::string ThreadName;
std::vector<std::pair<PRIORITY, ThreadName>> test_data = []
{
    std::vector<std::pair<PRIORITY, ThreadName>> data;

    data.push_back(
            std::make_pair(
                THREAD_PRIORITY_NORMAL,
                ThreadName("Thread_1_NormalPriority")));
    data.push_back(
            std::make_pair(
                THREAD_PRIORITY_NORMAL,
                ThreadName("Thread_2_NormalPriority")));
    data.push_back(
            std::make_pair(
                THREAD_PRIORITY_NORMAL,
                ThreadName("Thread_3_NormalPriority")));
    data.push_back(
            std::make_pair(
                THREAD_PRIORITY_BELOW_NORMAL,
                ThreadName("Thread_4_BelowNormalPriority")));
    data.push_back(
            std::make_pair(
                THREAD_PRIORITY_ABOVE_NORMAL,
                ThreadName("Thread_5_AboveNormalPriority")));
    return data;
```

```
48  }();

50  void DoTask(ThreadName threadName)
    {
52      while (runTests)
        {
54          // Obtain the ownership of the mutex
            std::unique_lock<std::mutex> lock(mutex);
56          // Wait
            condition.wait(lock);
58          std::cout << threadName << std::endl;
        }
60  }

62  std::unique_ptr<std::thread> Create(PRIORITY priority,
                                        ThreadName name)
64  {
        std::unique_ptr<std::thread> thread(new std::thread(DoTask, name));
66
        // Set the thread priority
68      SetThreadPriority((HANDLE)thread->native_handle(), priority);
        return thread;
70  }

72  void Test_std_condition_variable()
    {
74      // Create the threads
        for (unsigned int i = 0; i < test_data.size(); ++i)
76      {
            auto testItem = test_data.at(i);
78          threads.push_back(Create(testItem.first,
                                     testItem.second));
80      }

82      auto testIterations = 100;
        for (auto i = 1; i <= testIterations; ++i)
84      {
            // Signal one thread
86          condition.notify_one();

88          // Sleep a for 100ms before signaling the next thread
            Sleep(100);
90      }

92      // Stop the threads
        runTests = false;
94      condition.notify_all();

96      for (unsigned int i = 0; i < threads.size(); ++i)
        {
98          // Wait until the execution has finished
            threads.at(i)->join();
100     }
    }
```

APPENDIX C: THREAD SYNCHRONIZATION CLASS WITH QT

```
   #include <QList>
 2 #include <QWaitCondition>
   #include <QMutexLocker>
 4
   //! \brief Provides fair thread synchronization
 6 //! \details Threads are synchronized around a single QMutex
   //! In other-words, the same mutex must be throughout
 8 //! the lifespan of ThreadSynchronizer -instance
   //! The mutex must be locked before using any of the provided functions
10 //! \author Juha Löflund
   class ThreadSynchronizer final
12 {
       // Disable copying of ThreadSynchronizer
14     // by deleting copy constructor and assignment operator
       Q_DISABLE_COPY(ThreadSynchronizer)
16
   public:
18     ~ThreadSynchronizer()
       {
20         // Check that no threads are pending
           Q_ASSERT_X(m_waitingThreads.isEmpty(),
22                    Q_FUNC_INFO,
                      "Destroying while threads are pending");
24     }

26     //! \brief Waits until timeout or Wake -command
       //! \param[in] pLock Locked QMutex
28     //! \param[in] maxWaitTime Maximum wait time in milliseconds
       //! Function will wait indefinitely with default value
30     //! \details More details, see
       //!          http://doc.qt.io/qt-5/qwaitcondition.html#wait
32     //! Locked QMutexLocker will unlocked during the wait
       //! and it will be returned to the same locked state.
34     //! \return True, if thread was wakened before timeout
       //! \author Juha Löflund
36     inline bool Wait(QMutex* pLock,
                        const ulong maxWaitTime = ULONG_MAX)
38     {
           // Create an instance of QWaitCondition
40         // for the calling thread
           QWaitCondition waitCondition;
42
           // Add it to the queue and start waiting
44         m_waitingThreads.append(&waitCondition);
           if (waitCondition.wait(pLock, maxWaitTime))
46         {
               return true;
```

```
48          }

50          // QWaitCondtion::wait() timed out,
            // remove the pointer of 'waitCondition'
52          // from ‘m_waitingThreads’
            // It is possible that the thread times out
54          // just as it is being waken.
            // In this case, nothing is removed
56          // Interpret this as a success by returning true
            return m_waitingThreads.removeAll(&waitCondition) == 0;
58      }

60      //! \brief Wakes the first thread in the queue
        //! \author Juha Löflund
62      inline void WakeFirst()
        {
64          if (!m_waitingThreads.isEmpty())
            {
66              m_waitingThreads.takeFirst()->wakeOne();
            }
68      }

70      //! \brief Checks if any thread(s) are waiting
        //! \return True, if any thread is queued
72      //! \author Juha Löflund
        inline bool HasThreadsWaiting() const
74      {
            return !m_waitingThreads.isEmpty();
76      }

78      //! \brief Checks how many threads are waiting
        //! \return Number of waiting threads
80      //! \author Juha Löflund
        inline quint32 WaitingThreadsCount() const
82      {
            return m_waitingThreads.size();
84      }

86  private:
        QList<QWaitCondition*> m_waitingThreads;
88  };
```