RETRIEVING THE PERFORMANCE OVERHEAD OF SYNCHRONIZATION MECHANISMS OF VARIOUS POPULAR OPERATING SYSTEMS

Michael Bogner [(a)], Johannes Schütz [(b)], Franz Wiesinger [(c)]

[(a, b, c)] University of Applied Sciences Upper Austria,
Hardware/Software Design & Embedded Systems Design

[(a)] michael.bogner@fh-hagenberg.at, [(b)] johannes.schuetz@fh-hagenberg.at, [(c)] franz.wiesinger@fh-hagenberg.at

ABSTRACT

The importance of multi-core processors increases every day. So multi-threaded programming also becomes more important. Due to data consistency it is necessary to synchronize specific parts of the code. These synchronizing methods cause an overhead during program execution. This paper analyses this overhead based on time on different operating systems. On the one hand, the paper gives a short introduction to the most important synchronization methods, on the other hand a test application is introduced to determine the delay time of each of these methods. All tests are designed to give real world examples of how much overhead is produced. Following the given data of the test application, the delay times of different operating systems are compared to each other. The paper shows that some methods perform better on one system and others perform better on the other systems.

Keywords: synchronization methods, synchronization performance, multi-threaded performance, decision support

1. INTRODUCTION

1.1. Motivation

The development of new processors, such as faster single-core processors and multi-core processors, resulted in new opportunities in software development. On multi-core processors it is now possible to achieve real parallelism of software by running various software components on different cores.

Alongside these new opportunities also new difficulties came. One primary difficulty is the synchronization of software components which run independently. Synchronization refers to controlling the application flow of paralleled software components.

There are certain techniques required to perform synchronization. Because of the various implementations of these techniques, different overhead is produced depending on the way of the implementation.

This paper analyses the overhead on the popular operating systems Windows XP, Windows 7 and Ubuntu 10.04 LTS in each case 32-bit edition.

1.2. Objective

The primary objective is to accomplish a comparison of various synchronization techniques on different operating systems. This is achieved by measuring the delay time by modelling real-world usage of the different techniques and simulating their real-world behaviour in the test environment.

The paper is divided into two main sections. The first section gives an overview to the basics of each synchronization method. Especially the usage of the methods using Win32-API and using POSIX (IEEE 1003.1-2008, 2008) is described.

The second section describes the test scenario, the implementation and the analysis of the results. Both, modelling and simulation of the test scenario is done with industrial applications in mind.

1.3. Related work

A similar approach of analysing synchronization techniques can be found in the paper "A new Look at the Roles of Spinning and Blocking" (Johnson, 2009). There the trade-off between spinning and blocking synchronization is analysed and observed that the trade-off can be simplified by isolating the load control aspects of contention management.

Another approach can be found in the article "Multi-threaded Performance" (Asche, 1996) where strategies for rewriting single-threaded applications to be multi-threaded applications are discussed. It analyses the performance of multi-threaded computations over compatible single-threaded ones in terms of throughput and response.

2. THE BASICS

2.1. Multi-threaded programming

Multi-threaded programming allows the creation of parallel software. Since C++ does not provide mechanisms for multi-threaded applications until C++11 (ISO, 2011), operating system functions have to be accessed. The problem with these functions is that they are implemented differently on various operating systems and also provide different results in regard to performance. C++11 already provides multi-threaded mechanisms on the basis of POSIX-threads, but there are hardly any compilers or commercial software since it was released in August 2011.

Multi-threaded code is program code which is executed simultaneously by the operating system.

On Linux systems these functions are defined by the POSIX standard and implemented in the kernel. On Windows systems they are defined by the Win32-API and also implemented in the kernel.

Because of parallel execution of software components, which also share resources and communicate with each other, the control flow is synchronized. But this synchronized execution also causes problems in the form of deadlocks, race conditions, starvation and live-locks. These problems occur when synchronization methods are used carelessly.

Further discussions on multi-threaded programming can be found in (Akhter & Roberts, 2006), (Williams, 2012) and (Johnson, Athanassoulis, Stoica, & Ailamaki, 2009).

2.2. Why synchronization?

Due to the previously mentioned problems with multi-threaded programming a synchronization of the control flow is needed.

Synchronization is always needed when parallel reading and writing on memory occurs and when a specific sequence of the control flow has to be guaranteed. Also synchronization is needed when only writing parallel on memory but reading sequential. No synchronization is needed when the memory is written sequential.

In general, synchronization is always necessary when several threads write to a specific memory area.

2.3. POSIX-standard

The POSIX-standard defines a consistent standard system interface for UNIX-systems and Windows-systems. POSIX is the abbreviation for *Portable Operating System Interface for UNIX.* POSIX has been created to enable portable code for various UNIX-systems. POSIX includes the PThread-library for programming multi-threaded applications for UNIX-systems.

2.4. Synchronization mechanisms

There are various synchronization primitives, depending on the operating system and the underlying CPU. Each primitive has its special purpose:

- Critical Sections: atomic areas within a process for exclusive access.
- Events: signal that a certain state of the application occurred.
- Wait-functions: blocked waiting for an event or other signal.
- Mutex: similar to critical sections which work outside process boundaries.
- Semaphore: similar to mutex with an internal counter for handling several threads.
- Spin-lock: similar to critical sections but with short active waiting before passive waiting.
- Interlocked functions: hardware depended atomic operations which directly execute CPU-instructions

Further discussions on these techniques can be found in (Hart, 2010) and (Jones, 2008).

3. TEST SCENARIO

3.1. Introduction

As basis for the tests the Microsoft operating systems Windows XP and Windows 7, as well as the UNIX based free operating system Ubuntu 10.04 LTS are used. Windows XP and Windows 7 have been taken because they are the most commonly used Microsoft operating systems in industry. Ubuntu 10.04 LTS is used because it's one of the most commonly used UNIX based operating systems. The test environment is built with C++ and uses its object oriented capabilities.

3.2. Time measurement

The time measurement is carried out by operating system internal time measurement operations with high accuracy. The measurement uses an accuracy of 1 us which is accurate enough for our real world measurement approach.

Not the absolute time delay is measured, but the delay relative to the current CPU-tick count. This allows a better comparison by disregarding the blur of each operating system because their similar scheduling algorithms.

To keep the source code compatible between Windows and UNIX the Win32-API functions *QueryPerformanceCounter* and *QueryPerformanceFrequency* were implemented using *gettimeofday* (Linux High-Resolution Timer, 2009). The functions have the same interface like the Win32-API functions and are implemented using compiler directives.

3.3. Test flow

Given that the different synchronization techniques primarily differ in their field of application and therefore similar in usage also the test flow is structured similar for each technique.

Basically a test consists of initialization, start of the components, time measurement and analysis.

1. Initialization: The synchronization mechanisms and program components are initialized according to their usage.
2. Execution: The execution and time measurement are carried out in parallel and are repeated to generate a more accurate median.
3. Analysis: The median is calculated and written into an Excel file for further processing.

3.4. Constraints

Given that there are a lot of multi-core processors with various numbers of cores and clock frequencies and the different operating systems, it needs to be ensured that the average result is only exposed to low deviation.

It has to be ensured that the whole program inclusive all its components is executed on only one CPU core. This is needed because the switch of the execution to another CPU core also produces overhead and to avoid inter-core communication influence.

Also it has to be ensured that the overhead of context switches between parallel executed components is minimized. This is done by setting the process priority to the highest priority available.

4. IMPLEMENTATION

4.1. Introduction

This section describes the architecture and implementation of the test application for each synchronization technique.

4.1.1. Software architecture

The architecture is built up from two base classes, one for the test flow and one for threads. Derived from this base classes are all classes needed for testing each synchronization technique. Classes are named after the technique they are used for with the suffixes ‘Test’ and ‘Thread’ to differentiate between the test flow and the threads.

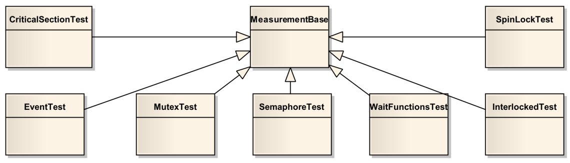

Figure 1: Test Classes

Figure 1 shows the class hierarchy of the test classes. Each test class implements methods for controlling the test flow and interpreting the test results.

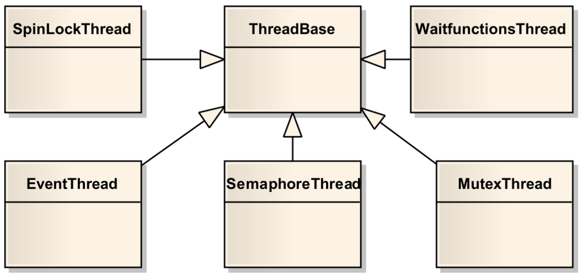

Figure 2: Thread Classes

Figure 2 shows the class hierarchy of the thread classes. The base class is used for controlling the typical thread flow such as creating, starting or suspending. The derived classes implement the specific methods for each synchronization technique and the control flow. The threads are implemented as fire-and-forget threads so there is no need to stop and delete them. They are used for calculating the overhead of the various synchronization techniques.

Because the POSIX-standard doesn’t provide an implementation for manual-reset-events, they are self-implemented using a conditional variable and a mutex. There are methods and a structure realized which implement the functionality.

4.1.2. Test flow

Figure 3 shows the typical test flow with the help of the base classes. The user creates a new test class and initializes it. The test class then creates the corresponding thread class and starts the testing. The testing is repeated as often the as *MEASURES* declares. After the tests are completed the average overhead is calculated and written to a file for further processing.

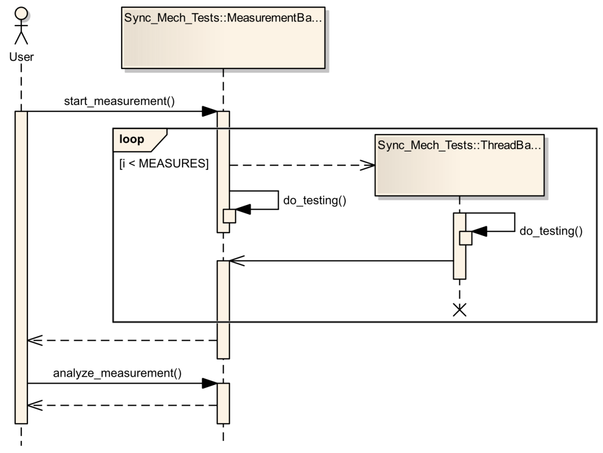

Figure 3: Test flow

4.1.3. Choosing processor core and process priority

As described in section 3.4 the overhead of context switches and switching to another core needs to be reduced so the results don’t get falsified.

To reduce the number of context switches the process priority is set to real-time. To block core switching the process affinity mask has to be set. It doesn’t matter on which core the program is executed as long as it is only one core.

Listing 1 and Listing 2 show the Windows implementation and respectively the UNIX implementation of setting process priority and affinity mask.

Listing 1: Windows Process Priority and Affinity Mask (without error handling)

```
HANDLE h_process = GetCurrentProcess();
// set process priority to high
SetPriorityClass(h_process, REALTIME_PRIORITY_CLASS);
// set process affinity mask to only use core 0
SetProcessAffinityMask(h_process, AFFINITY_MASK);
```

Listing 2: UNIX Process Priority and Affinity Mask (without error handling)

```
cpu_set_t mask;
CPU_ZERO(&mask);
CPU_SET(0, &mask);
```

```
// set process priority to high
setpriority(PRIO_PROCESS,0,-20);
//set process affinity mask to only use core 0
sched_setaffinity(0, sizeof(mask), &mask);
```

4.2. Synchronization techniques

All tests are designed to give real world examples of how much overhead is produced by the various techniques and not just laboratory values.

4.2.1. Critical sections

The test flow is shown in Figure 4. The delay time measured is the delay from entering and respectively leaving the critical section. This way of measurement is done because critical sections are used for short sections only so there is not much overhead.

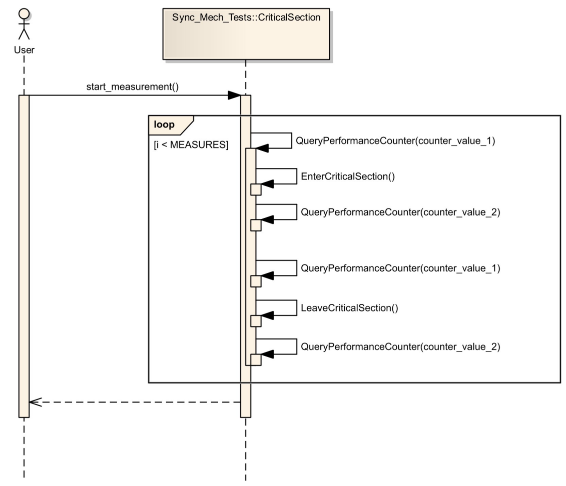
User
Sync_Mech_Tests::CriticalSection
start_measurement()
loop
[i < MEASURES]
QueryPerformanceCounter(counter_value_1)
EnterCriticalSection()
QueryPerformanceCounter(counter_value_2)
QueryPerformanceCounter(counter_value_1)
LeaveCriticalSection()
QueryPerformanceCounter(counter_value_2)

Figure 4: Measuring Critical Sections

4.2.2. Events

The test flow is shown in Figure 5. After starting the test a thread is created which initially waits for an event to continue execution. In the test method this event is signalled and the thread continues its execution. After signalling the event the test method waits for an event signalled by the thread. After continuing execution the thread sets the event and finishes its work. The delay of events is calculated by measuring the time from signalling the event to recognizing the signalled event. With this method of measurement not only the execution time of the technique is measured but also the overhead produced by context switches which gives a real world example of the overhead.

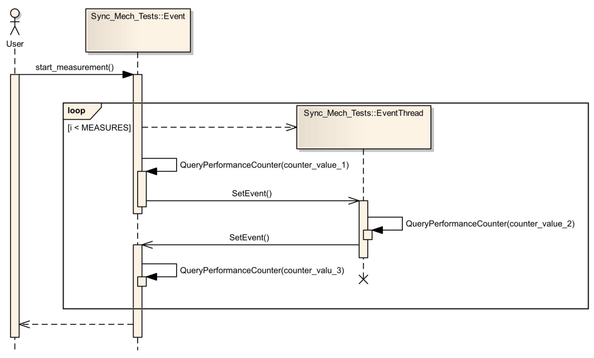
User
Sync_Mech_Tests::Event
start_measurement()
loop
[i < MEASURES]
Sync_Mech_Tests::EventThread
QueryPerformanceCounter(counter_value_1)
SetEvent()
QueryPerformanceCounter(counter_value_2)
SetEvent()
QueryPerformanceCounter(counter_valu_3)

Figure 5: Measuring Events

4.2.3. Wait-functions

Because there are no directly equivalent functions in the POSIX-standard this tests measures the delay of recognizing the exiting of a thread. And also wait-functions from Win32-API which are used to wait for signals of mutex, semaphores and events are measures in the corresponding tests.

Figure 6 shows the test flow of measuring the wait-functions. After the test is started a thread is created and started and the test method waits for it to complete. The time is measured after the thread was started and after the thread completed execution.

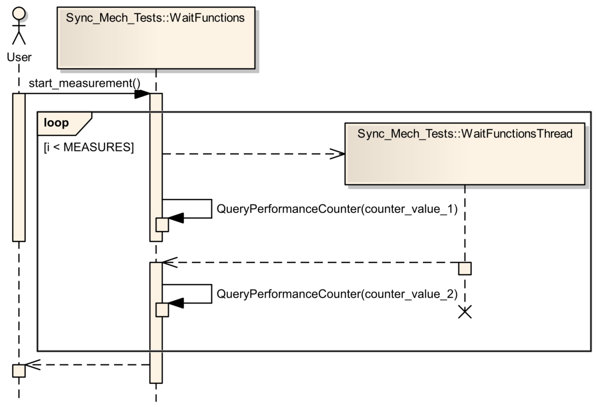
User
Sync_Mech_Tests::WaitFunctions
start_measurement()
loop
[i < MEASURES]
Sync_Mech_Tests::WaitFunctionsThread
QueryPerformanceCounter(counter_value_1)
QueryPerformanceCounter(counter_value_2)

Figure 6: Measuring Wait-functions

4.2.4. Mutex

The test flow of testing mutex is similar to that of testing critical sections with the difference that there is a thread to communicate with.

Figure 7 shows the test flow. After starting the test a mutex is created in blocked mode and a thread is created and started. The thread opens the mutex and waits for it to be released. After the thread owns the mutex the test method waits for it to be released. The overhead is calculated by measuring the time from releasing the mutex in the test method and respectively in the thread and getting to own the mutex in the thread and respectively the test method.

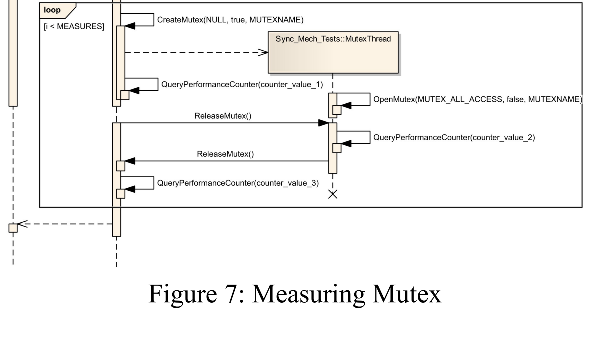
User
Sync_Mech_Tests::Mutex
start_measurement()
loop
[i < MEASURES]
CreateMutex(NULL, true, MUTEXNAME)
Sync_Mech_Tests::MutexThread
QueryPerformanceCounter(counter_value_1)
OpenMutex(MUTEX_ALL_ACCESS, false, MUTEXNAME)
ReleaseMutex()
QueryPerformanceCounter(counter_value_2)
ReleaseMutex()
QueryPerformanceCounter(counter_value_3)

Figure 7: Measuring Mutex

4.2.5. Semaphore

The test flow of the semaphore test, shown in Figure 8, works corresponding to the mutex test with the difference that more threads are used.

4.2.6. Spin-locks

The test flow of the spin-locks test is corresponding to that of testing critical sections with the only difference being spin-locks in testing instead of critical sections. Because of this nearly equivalent test flow there is now explicit figure given to illustrate it.

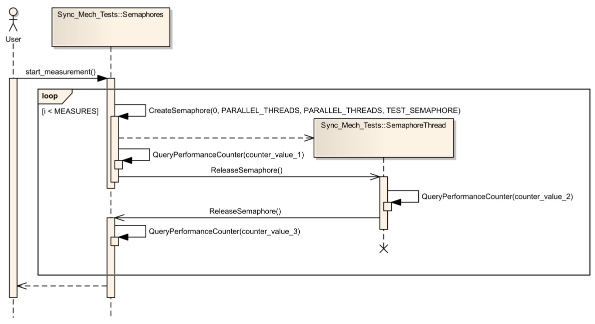
User
Sync_Mech_Tests::Semaphores
start_measurement()
loop
[i < MEASURES]
CreateSemaphore(0, PARALLEL_THREADS, PARALLEL_THREADS, TEST_SEMAPHORE)
Sync_Mech_Tests::SemaphoreThread
QueryPerformanceCounter(counter_value_1)
ReleaseSemaphore()
QueryPerformanceCounter(counter_value_2)
ReleaseSemaphore()
QueryPerformanceCounter(counter_value_3)

Figure 8: Measuring Semaphore

4.2.7. Interlocked functions

The test flow of measuring interlocked functions, shown in Figure 9, is very simple. Each function is executed and its execution time is measured.

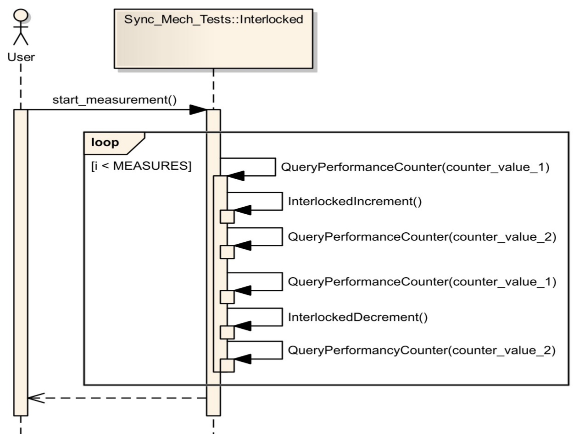
User
Sync_Mech_Tests::Interlocked
start_measurement()
loop
[i < MEASURES]
QueryPerformanceCounter(counter_value_1)
InterlockedIncrement()
QueryPerformanceCounter(counter_value_2)
QueryPerformanceCounter(counter_value_1)
InterlockedDecrement()
QueryPerformancyCounter(counter_value_2)

Figure 9: Measuring Interlocked Functions

5. ANALYSIS

5.1. General Findings

Through the evaluation of the test results insights could be gained on how the performance of synchronization techniques differs on different operating systems. When viewing the test results it is important to differ between synchronization techniques, which are influenced by the operating system and those without. Influenced by the operating system are mutex, semaphore, events and wait-functions. Without influence are critical sections, spin-locks and interlocked functions.

If synchronization does not depended on the operating system, less overhead can clearly be recognized. In general, for process-internal synchronization critical sections should be used and for calculations interlocked functions if available. Spin-locks are particularly well suited for synchronization of small areas which are divided among multiple processor cores, but produce more overhead if the number of critical areas exceeds the number of processor cores.

If synchronization depends on the operating system, wait-functions cause the least overhead because they only wait for a certain signal, usually in a blocked manner. Because of the operating system influenced token system of mutexes they produce more overhead than critical sections. Semaphores produce similar overhead to mutex but have even more impact due to the internal counter. The overhead of events exists of operating system influence and the expense to signal the wait-function to continue execution.

Table 1 shows the results of the tests on Windows 7, Windows XP and Ubuntu 10.04 LTS. As it can be seen clearly, synchronization techniques without the influence of the operating system are by far, the fastest. Values of 1us indicate that the measurement is near or beyond its precision, which doesn't mind as the high values are important in real-world applications. That confirms the knowledge that for process internal synchronization critical sections and for calculations interlocked functions should be used. Also recognizable is that wait-functions produce nearly the same overhead regardless of whether they are waiting on one or more signals. Mutex, semaphore and events produce very different overhead on the several operating systems; this will be illustrated in the next section.

Looking at the results of operating system influenced synchronization techniques it can be said, that semaphores should be avoided when possible. If synchronization is needed outside of process boundaries use events or mutex under Windows but try to avoid events under UNIX.

Table 1: Results Windows XP, Windows 7, Ubuntu 10.04 LTS (time in us)

Synchronization technique	Win 7	Win XP	Ubuntu
EnterCriticalSection	1	1	1
Interlocked Decrement	1	1	1
Interlocked Increment	1	1	1
LeaveCriticalSection	1	1	1
Lock Spinlock	1	1	1
ReleaseMutex	21	59	14
ReleaseSemaphore	39	27	11
SetEvent	32	20	51
Unlock Spinlock	1	1	1
WaitForMultipleObjects	18	20	19
WaitForSingleObject	18	19	15
Waiting for Mutex	24	26	42
Waiting for Semaphore	145	105	106

5.2. Differences between Windows XP, Windows 7 and Ubuntu 10.04 LTS

In a direct comparison of windows and UNIX it can be seen that in general the mechanisms require less effort under UNIX than under Windows.

On all systems critical sections, spin-locks and interlocked functions only produce such a small overhead that no difference is recognizable. The same can be seen by looking at wait-functions, which produce nearly the same overhead on the several operating systems. A big difference can be seen when looking at mutex, which produces more than twice the overhead when it is released on Windows XP than on Windows 7 or Ubuntu. Acquiring a mutex produces more overhead on Ubuntu than on Windows XP or Windows 7 which produce similar overhead. Another big difference can be seen at semaphores. Acquiring a semaphore on Windows XP or Ubuntu produces nearly the same overhead but produces a lot more overhead on Windows 7. This could be explained due to the internal implementation of the semaphore counter. In general, acquiring a semaphore produces the most overhead of all synchronization techniques. Releasing a semaphore produces very different overhead on all operating systems. On Windows XP the overhead is twice as much as on Ubuntu and on Windows 7 three times as much overhead. Also events produce different overhead on each operating system. The least overhead is produced on Windows XP, a little more overhead is produced on Windows 7 but on Ubuntu the overhead is more than twice the overhead produced on Windows XP. This can be explained by looking at the implementation of the manual-reset-event on Ubuntu which uses a mutex and a conditional variable, so the overhead of two mechanisms is included in this test.

In general, it can be seen beside a few exceptions that Ubuntu operating system produces less overhead than both Windows operating systems. With the Windows operating systems it is more complicated, because some mechanisms produce less overhead on Windows XP and some on Windows 7.

A visual representation of these differences can be seen in Figure 10.

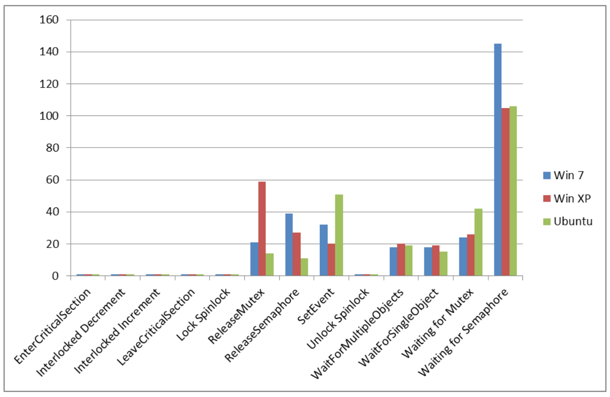

Figure 10: Difference between Windows and Linux

CONCLUSION

The performed tests have shown what average overhead is expected on the various operating systems. Also it was pointed out that UNIX comparing all mechanism produces less overhead than Windows operating systems. It can be seen that each operating system has its strengths and weaknesses in the implementation of synchronization techniques.

Based on these measurements it can now be shown which operating systems are the better option for each synchronization technique, provided a free selection is an option. In the field of synchronization Linux would be in almost every field the better option, except for events which have less overhead under Windows than under Linux.

On UNIX-systems it cannot be assumed, despite the POSIX-standard that the overhead on average is the same, since different UNIX-derivatives also have different kernel implementations. But in general it can be assumed that different Linux-distributions with the same kernel produce the same overhead.

Due to the different performance of synchronization techniques it is important to analyse in advance which mechanisms will be needed to not slowing down a multi-threaded application unnecessarily.

REFERENCES

Akhter, S., Roberts, J., 2006. *Multi-Core Programming: Increasing Performance through Software Multi-threading.* Intel Press.

Asche, R. R., 1996. *Multithreaded Performance*. Available from: http://msdn.microsoft.com/en-us/library/ms810437.aspx [accessed April 2012]

Hart, J. M., 2010. *Windows System Programming.* Addison-Wesley Professional.

IEEE 1003.1-2008., 2008. 1003.1-2008 - IEEE Standard for Information Technology - Portable Operating System Interface (POSIX(R)). IEEE.

ISO, 2011. *ISO/IEC 14882:2011, Information technonology - Programming languages – C++*. Available from: www.iso.org [accessed April 2012]

Johnson, R., et al. 2009. *A new look at the roles of spinning and blocking.* In Proceedings of the Fith International Workshop on Data Management on New Hardware (DaMoN ’09). ACM, New York

Jones, M. T., 2008. *GNU/Linux Application Programming.* Charles River Media.

Williams, A., 2012. *C++ Concurrency in Action: Practical Mutlithreading.* Available from: http://www.manning.com/williams/ [accessed April 2012]