Utilization of Kernel-Level and User-Level Threads in Job Systems

Bachelor Thesis 2

Author: Alessandro Morio
Advisor: FH-Prof. DI Dr. Hilmar Linder

Salzburg, Austria, 19.06.2017

Affidavit

I herewith declare on oath that I wrote the present thesis without the help of third persons and without using any other sources and means listed herein; I further declare that I observed the guidelines for scientific work in the quotation of all unprinted sources, printed literature and phrases and concepts taken either word for word or according to meaning from the Internet and that I referenced all sources accordingly.

This thesis has not been submitted as an exam paper of identical or similar form, either in Austria or abroad and corresponds to the paper graded by the assessors.

Date *Signature*

First Name *Last Name*

Kurzfassung

Diese Bachelorarbeit untersucht wie kernel-level Threads und user-level Threads in Job Systemen angewendet werden können. Zwei Ansätze von Job System Implementierungen werden definiert und untersucht. Messungen von beiden Implementierungen werden durchgeführt, ausgewertet und interpretiert.

Die Organisation und Funktionsweise von Mehrkern-Prozessoren und Multithreading wird untersucht um ein besseres Verständnis davon bereitzustellen wie kernel-level und user-level Threads funktionieren. Das Konzept von Prozessen und Threads wird erläutert um zu verstehen zu geben wie Spiele beziehungsweise Spiel-Engines vom Betriebssystem gehandhabt werden. Das Thema kernel-level und user-level Threads wird im Detail untersucht. Die Implementierungen der zwei Thread Typen und deren Anwendung, in dem Betriebssystems Windows, wird untersucht . Es werden auch verschiedene Multithreading Strategien von Spiel-Engines genannt. Eine dieser Strategien die Prozessoren sehr gut auslastet, und auch von akutellen Spiel-Engines eingesetzt wird, ist das *Job System*. Die Definition eines Job und eines Job System wird dargelegt. Darüber hinaus wird der allgemeine Ansatz und Ablauf eines Job Systems erläutert. Verbesserungen die auf ein grundlegendes Job System angewendet werden können sind beispielsweise *Work Stealing* und *lock-freie Programmierung*.

Um die Auswirkungen von kernel-level und user-level Threads auf Job Systeme zu untersuchen werden zwei Ansätze definiert und implementiert: ein *reiner kernel-level Thread Ansatz* und ein *hybrider Ansatz* der kernel-level und user-level Threads einsetzt. Beide Ansätze werden erläutert und Messungen, bezüglich Zeit und Prozessor Auslastung, werden durchgeführt. Die Ergebnisse werden interpretiert und Verbesserungen werden diskutiert.

Schlüsselwörter: Multithreading, Spiel-Engine, Job System, User-Level Thread, Fiber

Abstract

This thesis examines how kernel-level and user-level threads can be applied to job systems. Two approaches of job system implementations will be defined and examined. Measurements of both implementations are performed, evaluated and interpreted.

The organization and functionality of multi-core processors and multithreading is examined to provide a better understanding how kernel-level and user-level threads work. The concept of processes and threads is explored in order to understand how games, respectively game engines, are handled by an operating system (OS). The topic kernel-level and user-level threads is investigated in detail. The Windows implementations of both thread types, and how they are used, is investigated. Different multithreading strategies for game engines are named. One strategy that utilizes processors very well and current game engines integrate is the *job system*. The definition of a job and job system will be stated. Moreover, the general approach and flow of a job system is examined. Improvements that can be applied to basic job systems is *work stealing* and *lock-free programming*.

In order to examine the impact of kernel-level and user-level threads to job systems two approaches are defined and implemented: a *pure kernel-level thread approach* and a *hybrid approach* using kernel-level and user-level threads. Both approaches are explained and measurements, concerning time and processor utilization, are performed. The results are interpreted and further improvements are discussed.

Keywords: Multithreading, Game Engine, Job System, User-Level Thread, Fiber

Contents

1 Introduction

The first computer games were played on machines that had a CPU with single hardware core which was responsible for calculating the game's logic (Gregory 2014, 340-342, 361). There is a single big loop which processes the game. The instructions of the game were processed sequentially.
Games push hardware to its limits (Gregory 2014, 340-341, 920). A variety of calculations are performed multiple times per second. To create a fluent graphical representation the animation system and the rendering system have to be synchronized at least at 30 to 60 Hz. Other systems like the physics system may have to be updated 120 times per second. The simulation of games become more and more exigent, detailed and complex (Andrews 2015). Hence, the calculations a processor needs to perform, continuously increase.

In 2002 the manufacturers of CPUs were faced with the fact that they were not able to increase the clock rate of CPUs (Ramanathan 2006, 3). The heat emerged from the CPU and power densities could not be handled properly (AMD Corporation 2005, 2-3). Therefore, the manufacturers started producing microprocessors with multiple cores. Due to the fact multi-core CPUs were introduced, software developers began writing code utilizing those newly available hardware cores (Gregory 2014, 361).
AMD and Intel manufacture current high-end desktop multi-core processors. The *AMD Ryzen Threadripper* comes with 16 cores and 32 hardware threads (AMD Corporation 2017). Intel introduces its Core i9 processor family (Intel Corporation 2017a). The *Intel Core i9-7980XE* will have 18 cores and 36 hardware threads. Thus, there are lots of cores that can be utilized in video game simulations.

When game developers start coding games respectively game engines utilizing multiple cores, they are faced with the challenges of multithreaded programming (Gregory 2014, 369-374; Lake 2011, 373). At a coarse level, there are techniques and methods required for writing code for multiple cores. One method for utilizing multiple hardware cores are *fork and join* algorithms. Another method is to assign one thread per game engine subsystem. The subsystems' threads are executing their specific logic and have to be synchronized. At a fine-grained level, programmers need to protect access to shared resources among multiple threads (Stallings 2015, 230-231). Read and write access to shared resources may has to be *atomic*, since read and write operations can consist of multiple instructions.

Making a game and writing code that utilizes hardware in a way that the simulation can be as detailed as possible is high effort. Therefore, most games are build upon a specific *game engine* (Gregory 2014, 11-13). The term 'game engine' was first associated with the game *Doom* in the mid-1990s. The software design of *Doom* had a clear separation between core software components, assets, game world and game logic. Software components could be reused to create a new game. Therefore, one major aspect of a game engine is the re-usability.
Most game engines target a specific genre or purpose (Gregory 2014, 13-32). There are 2D and 3D game engines. There are for example game engines focusing on first-person shooter (FPS), real-time strategy (RTS) or racing games. Each game engine has a different focus on what is important for the implementation and experience of the game.
There also exist game engines which provide a broad spectrum of possible game genre imple-

mentations (Gregory 2014, 13-32). Examples are Unreal Engine[1], Frostbite[2] and Unity[3]. The Unreal Engine was first designed to create FPS games. The current version, Unreal Engine 4, is able to create a variety of 3D games in general. Most of the current broad spectrum game engines are cross-platform including multiple desktop, virtual reality, mobile and console platforms. Some engines like Unity and Unreal can be used for free. However, there are royalty fees[4] [5].

In terms of an operating system (OS) a game is nothing else but a program (Russinovich, Solomon, and Ionescu 2012, 5-14; Stallings 2015, 183-202). A process is a container for resources used when executing the instance of a program. A process consists of at least one thread. A thread is the part of a process which actually scheduled by the OS for the execution of a process. Switching threads can be expensive. The kernel scheduler has to be triggered. Those threads are also known as *kernel-level threads*.
There is another category of threads called *user-level threads* (Stallings 2015, 189-193). These threads are entirely managed by the application respectively a threads library. The OS does not know about them. One advantage of user-level threads over kernel-level threads is the cost of thread switching. The kernel is not involved in those switches. One problem of user-level threads that needs to be considered are blocking system calls. When a user-level thread executes such a system call, not only that user-level thread is blocked, but also all the user-level threads executing on a process or kernel-level thread are. The kernel cannot schedule another user-level thread since the kernel is not aware of them.
Windows provides two mechanisms for using user-level threads: *fibers* and *user-mode scheduling (UMS)* (Russinovich, Solomon, and Ionescu 2012, 13; Microsoft Corporation 2017)[6]. A fiber is a lightweight thread. It has to be manually scheduled. UMS is a mechanism that applications can use to schedule their own threads and perform thread switches in user space.

Some current game engines like Frostbite and Naughty Dog use a specific strategy for utilizing multiple cores: a *job system* (Andersson 2009; Gyrling 2015). Roughly, a job represents a small portion of work (Gregory 2014, 371-372). There are worker threads which do nothing but processing jobs. The job system stores and manages jobs in data structures like queues (Lengyel 2010, 381-386; Lake 2011, 379-383; Andrews 2015) and assigns threads to them. To achieve a good scalability and performance, jobs should be designed small and relatively independent from other jobs in terms of shared resources.

This thesis aims to state the basic structure of a multi-core microprocessor and gives an overview how processes and threads work. The topic kernel-level and user-level threads will be investigated. Concerning user-level threads, the Windows mechanism fibers will be examined in detail. The functionality of a job systems also will be examined. Different approaches how job systems can be improved will be stated. The research question of this thesis is how a hybrid

1. `https://www.unrealengine.com/what-is-unreal-engine-4` Accessed June 16, 2017
2. `https://www.ea.com/frostbite/engine` Accessed June 16, 2017
3. `https://unity3d.com/de/unity` June 16, 2017
4. `https://unity3d.com/de/legal/terms-of-service/software` Accessed 16 June, 2017
5. `https://www.unrealengine.com/eula` June 16, 2017
6. `https://msdn.microsoft.com/de-de/library/windows/desktop/ms681917(v=vs.85).aspx` Accessed May 30, 2017

approach using kernel-level and user-level threads can be applied to a job system and which results, time and processor utilization concerning, compared to an approach with pure kernel-level threads are discovered. Further improvements to job systems utilizing user-level threads will be examined. One basic job system architecture will be defined and implemented with both, kernel-level and user-level threads.

2 Multi-Core Processors and Multithreading

Computer with multi-core processors are state of the art (see section 1 *Introduction*) (Stallings 2015, 65, 101). Multi-core processors are utilized by the OS by distributing multiple processes respectively threads across available cores. Computer hardware, particularly processors, and the concept of processes and threads is examined in this section.

2.1 Computer Hardware: Overview and Processors

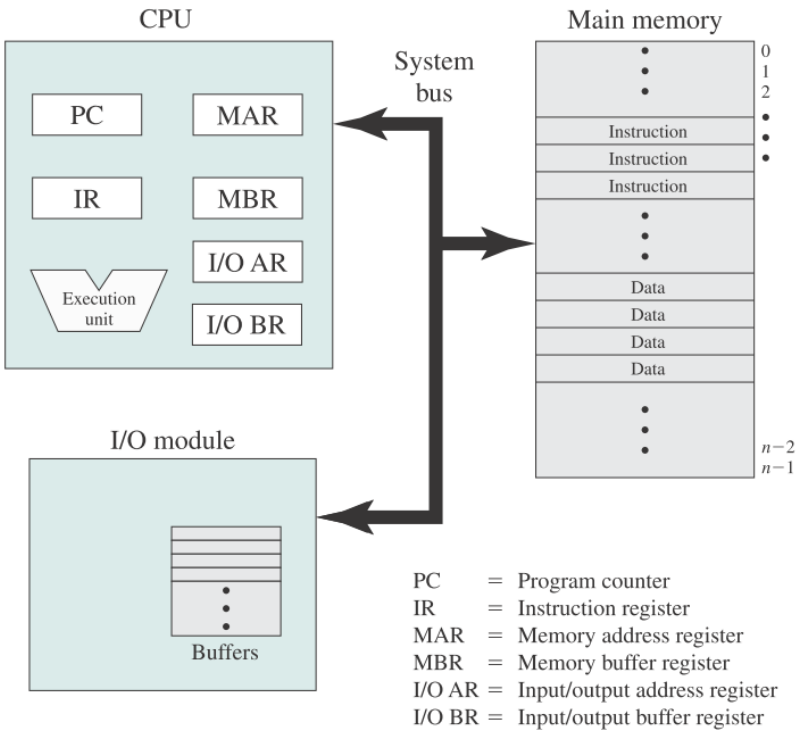


Figure 1: Basic hardware components a computer consists of: CPU, main memory, I/O modules and system bus (Stallings 2015, 38-39).

A computer consists of different interconnected hardware components which the computer uses to execute programs (Stallings 2015, 38-39). At a superficial level there exist the following components:

- Processor
- Main memory
- I/O modules
- System bus

The **processor**, also known as *central processing unit (CPU)*, performs calculations and is responsible for data processing (see figure 1) (Stallings 2015, 38-41). Figure 1 also shows that a CPU contains *registers*. The *instruction register* for example stores bits that specify the next action to be executed by the processor. In **main memory** program instructions and data are stored temporary. **I/O modules** include a variety of different devices like hard disks and displays. The **system bus** interconnects the processor, main memory and I/O modules in order to enable communication between these components.

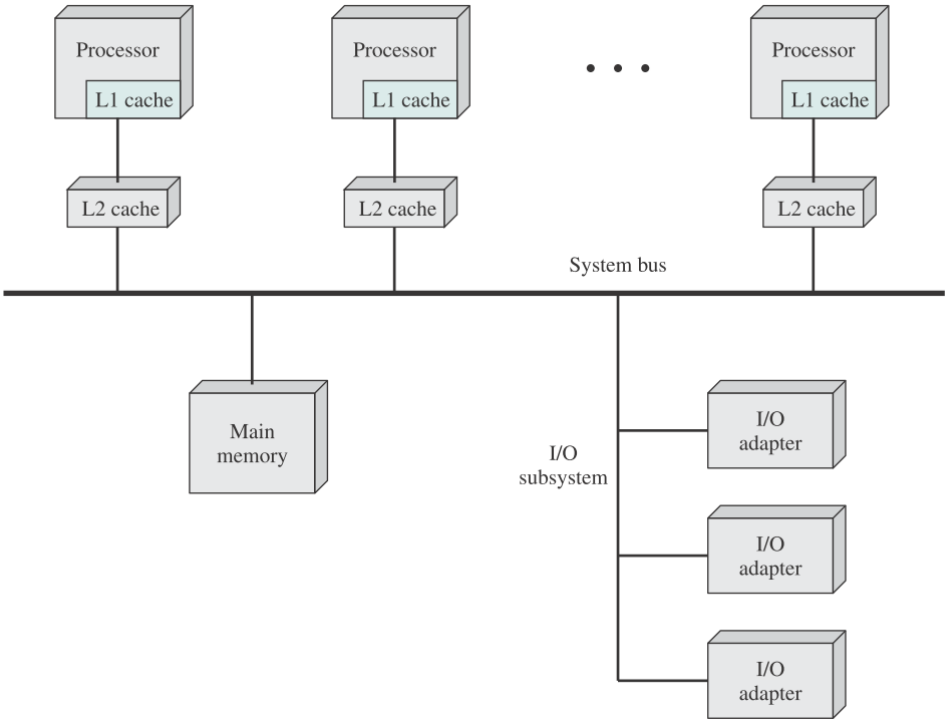


Figure 2: *SMP organization* with multiple processors sharing hardware components (Stallings 2015, 64-65).

Early processors were split up on multiple chips (Stallings 2015, 40). The next evolution of processors is a processor on a single chip. This invention is called *microprocessor*. Microprocessors became the fastest processors for general usage and, furthermore, evolved to multi-core processors. One chip consists of multiple processors which again consist of multiple logical processors. The multiple processors are called *cores* and the logical processors are known as *hardware threads*.

Considering only one processor/core a computer can be thought as a machine that works sequentially (Stallings 2015, 40-41, 62-65). The processor retrieves instructions of a program in main memory (this process is called *fetch*) and executes them in sequence. In reality a computer works parallel. E.g. *instruction pipelining* is used to overlap fetch and execute operations. To maximize parallelism to improve performance computer designers came up with different approaches. One of these is the *symmetric multiprocessor (SMP)*.

A SMP is a single computer system where multiple processors share the same interconnected hardware components (see figure 2) (Stallings 2015, 63-65). Moreover, the processors are equal in terms of functionality (symmetric) and an OS controls the interaction between the multiple processors and their programs. A general SMP organization consists of different core elements: processors, main memory, I/O devices and a shared bus. Each processor has its own control unit, arithmetic logic unit (ALU) and registers. Moreover, each processor can access main memory and I/O devices through a shared bus. Communication between processors is possible through memory or direct signaling.

There are *potential* advantages of a SMP over a system with a single processor (Stallings 2015, 63-64):

- Performance
- Availability
- Incremental growth

The work of a computer can be split between multiple processors for parallel processing. (Stallings 2015, 63-64). If one processor shuts down, the other processors could continue working without the entire system halting. Moreover, additional processors can be added to enhance the processing power.

The processor needs to access main memory in order to fetch instructions and data (Stallings 2015, 57-58). Over the years the processing speed of a CPU increased more than the speed for accessing main memory. Therefore, caches were invented. The memory size of a cache is small but the access speed is fast. There is a cache level hierarchy (see figure 3). The level 1 (L1) cache is the fastest but with very limited cache size. The level 2 (L2) cache is slower than the L1 cache but can store more data and instructions. The level 3 (L3) cache can be shared between multiple cores, has a large memory size but is slower than the L1 and L2 cache.

Current desktop processors are multi-core processors (see section 1 *Introduction*). A multi-core processor unites multiple processors in one chip (see figure 3) (Stallings 2015, 65-66). Each processor has its own registers, ALU and L1 cache (instruction and data). Figure 3 shows the block diagram of the *Intel Core i7-990X*. Each core has its own L1 and L2 cache and a shared L3 cache.

Not only personal computers make use of processors with multiple cores. Game consoles also profit from the parallel processing power when simulating game worlds. Current consoles like Playstation 3, XBOX 360, Playstation 4 and XBOX One come with multi-core processors (Gregory 2014, 362-368).

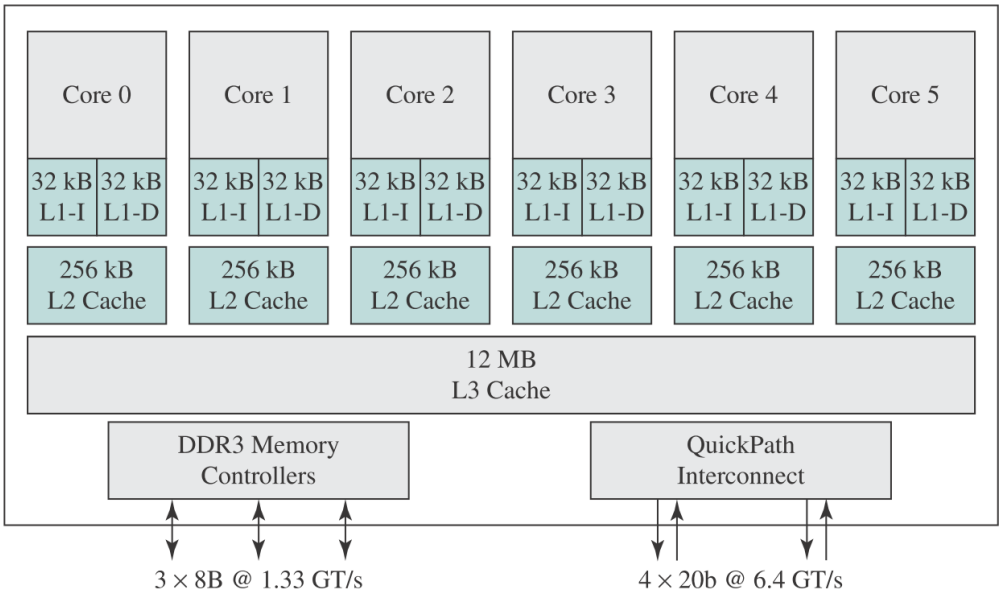


Figure 3: A *multi-core processor* with six cores and multiple caches (Stallings 2015, 65-66).

2.2 Basics of Operating Systems

An *operating system (OS)* is a program that manages application programs and provides an interface to hardware components (Stallings 2015, 77-78). An OS should be convenient to use, utilize system resources efficiently and be extendable/testable in terms of software development.

The single computer hardware components and their cooperation is a complex system (Stallings 2015, 78-79). When an application programmer would have to know and care about all this complexity, writing a new application would take an enormous amount of time. Therefore, an OS provides system programs, also known as utilities or library programs, that implement frequently used functions and convenient interfaces, application programmers can use to develop their user programs (like video games). Thus, one goal of an OS is to hide the complexity of computer hardware. A selection of services that an OS provides to applications and their programmers is the following:

- Program execution
- Access to I/O devices
- File access
- System access
- Application programming interface (API)

Early computers were programmed directly (Stallings 2015, 82-85). There was no OS. The program was written in machine code and the computer executed that program exactly as it was

written. Errors were communicated by specific lights blinking. With the invention of operating systems user programs were no longer allowed to access hardware components directly. Furthermore, altering memory containing the OS and executing privileged instructions, like I/O instructions, is not allowed for user programs. When a user program attempts to execute such instructions, the hardware transferred control to the OS. Due to that fact the concept of *modes of operation* was introduced and the *user mode* and the *kernel mode* come to existence (see figure 4). User programs execute in user mode where some areas in memory are protected and the OS executes in kernel mode. In kernel mode protected memory areas can be accessed and privileged instructions can be executed.

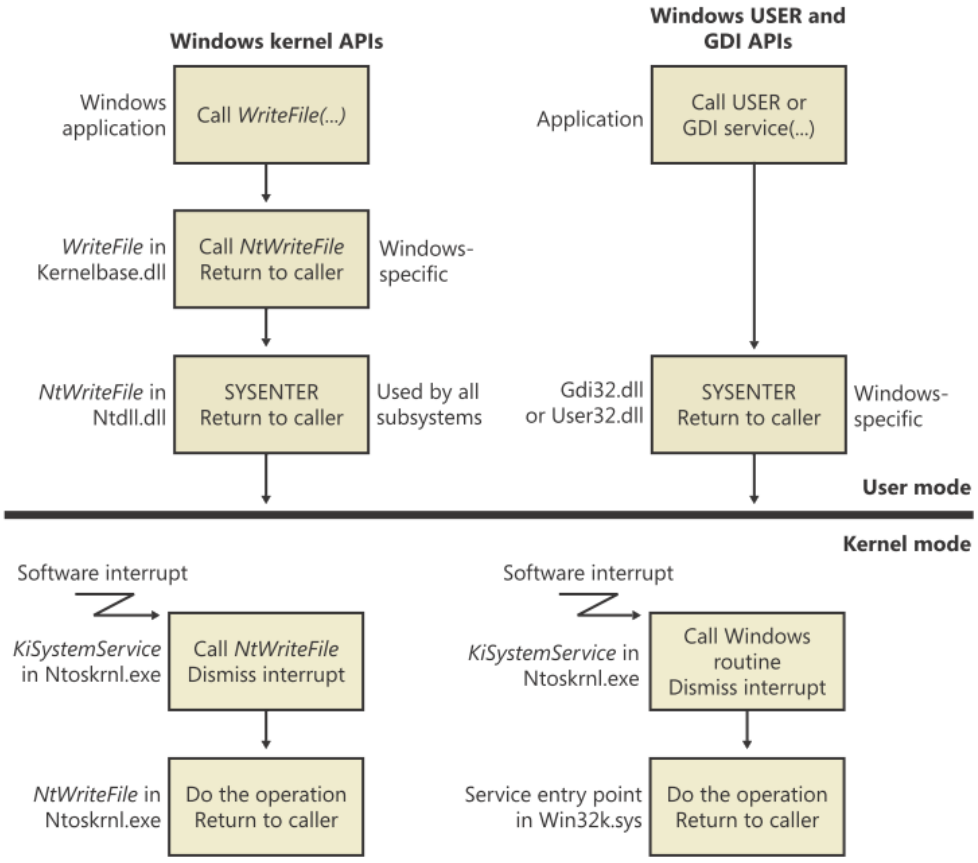


Figure 4: The left part of the image shows how Windows dispatches the *WriteFile* function call (Russinovich, Solomon, and Ionescu 2012, 138). A mode switch from user mode to kernel mode is performed.

The kernel mode provides access to services and functions only callable in kernel mode (Russinovich, Solomon, and Ionescu 2012, 2-4; Stallings 2015, 124-126). Windows refers to those functions as *kernel support functions*. A program in user mode cannot call those functions. There are specific calls a user program can do to access services and functions in kernel mode. Those calls are known as **system calls**. Windows refers to them as *native system services*. Windows and Linux provide many systems calls. System calls can be divided into the following

categories: file system, processes, scheduling, inter-process communication and networking/-socket.
Accessing I/O devices is slow compared to the processing power of a CPU (Stallings 2015, 85-87). An example is reading from and writing to a file. The CPU has to wait until the data is transmitted.

An example how a system call is dispatched in Windows is illustrated in figure 4. On the left side of the figure the function call *WriteFile* is dispatched (Russinovich, Solomon, and Ionescu 2012, 138). First, WriteFile is called in user mode. WriteFile calls the function *NtWriteFile* in user mode which enters kernel mode. In kernel mode the real NtWriteFile function is called to process the I/O request. After the operation is finished, it returns to the caller in user mode.

2.3 The Concept of Processes and Threads

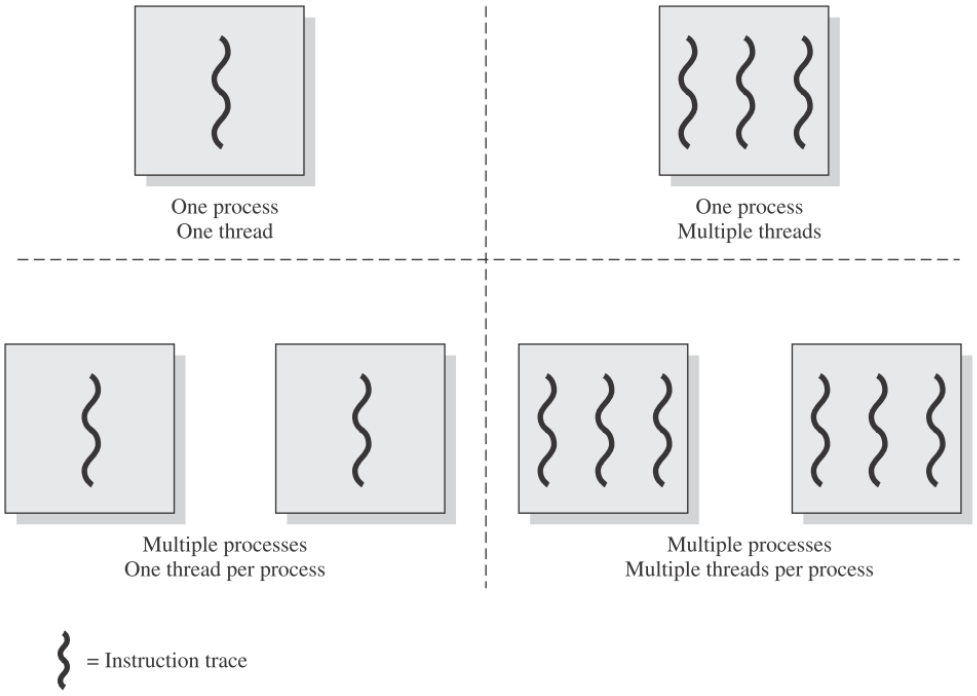


Figure 5: Different models of processes and threads. Windows is an OS that supports multiple processes with multiple threads per process (Stallings 2015, 184).

A process is the instance of a program in execution (Stallings 2015, 138-139). Moreover, a process is much more than just an instance. A process consist of various elements. An excerpt of these elements is:

- Identifier

- Priority
- Program counter
- Registers / memory pointers (stack pointer)
- I/O information
- State

The **identifier** of a process serves the purpose to uniquely identify a process among others (Stallings 2015, 39-41, 138-163; Tanenbaum and Bos 2015, 755). The **priority** of the process determines the scheduling preference relative to other processes. The **program counter** of a process points to the next instruction to be executed by the processor. **Registers** (see figure 1) control the operation of a CPU. Data like the next instruction to be executed is stored in registers. **Memory pointers** of a process point to program code and data in memory which associated with the process. Furthermore, memory pointers also refer to data shared by different processes. Each process also has one or more stacks (last-in-first-out (LIFO) data structure) respectively **stack pointers**. The stack size is initially set. Stacks are for example responsible for storing function call addresses and function parameters. Besides the stack there is also the *heap*. The heap is used to dynamically allocate memory during the execution. Part of the **I/O information** are outstanding I/O requests and accessed files of the process.
The **state** determines whether the process is running or not (Stallings 2015, 138-154, 166). There are different state models with different state types and different transitions between those states. An example of a state model is the *Five-State Process Model* (see figure 6). When a process is created to execute a program, it initially enters the *New* state. The *Ready* state signals the OS that a process is prepared to be executed. When a process is executing its instructions, it is in the *Running* state. A transition from Running to Ready can be triggered when the maximum allowed execution time is over. That is also referred to as *time slice*. Another reason for a transition from Running to Ready is *preemption*. Assume a process is running and is interrupted by the OS to let another process execute its instructions. This procedure is called preemption. The reason for a preemption might be a higher priority level of the other process. A process enters the *Blocked* state when it requests a service which it has to wait for. A *system call* could be one reason (see section 2.2 *Basics of Operating Systems*). If a process terminates, it enters the *Exit* state.

A process is a complex entity (Stallings 2015, 168, 183-184). E.g the costs of a process switch. Therefore, a process is divided into two parts: *resource ownership* and *execution and scheduling*. Resource ownership includes the program copy and data in memory and ownership of files. The execution and scheduling part is responsible for the execution path (instruction trace) of a program and is scheduled by the OS. To distinguish both parts the part execution and scheduling is called *thread*, also known as *lightweight process*, and the part concerning resource ownership is called process.

A thread consists of similar elements as a process does (Stallings 2015, 185, 205):

- Identifier

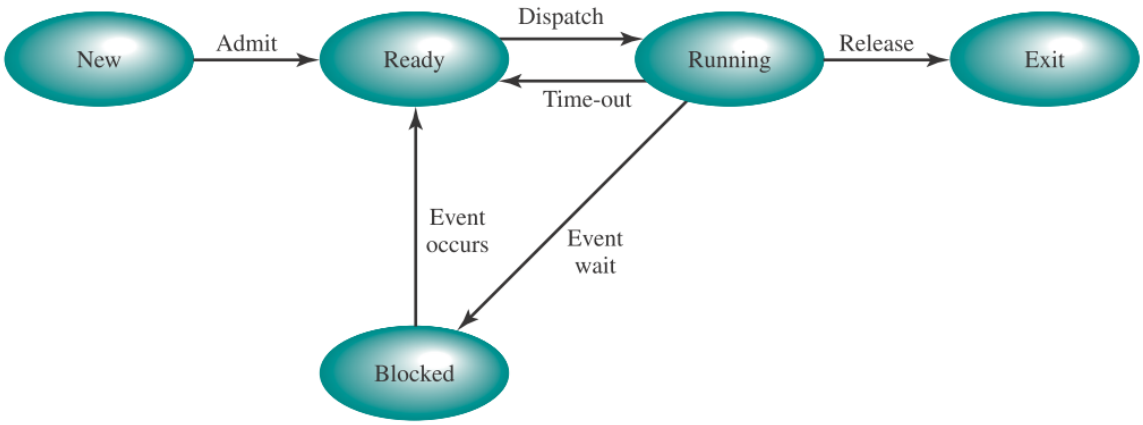


Figure 6: The *Five-State Process Model*. Between the creation and termination a process can be in one of the three states: *Ready*, *Running* or *Blocked* (Stallings 2015, 144-147).

- Priority
- State
- Thread context
- Thread local storage

The **identifier**, **priority** and **state** serve the same basic purpose as in a process (Stallings 2015, 185-194, 201-205). The **thread context** is a set of information that can be saved when a thread is not executing. The thread context includes for example the program counter and stack pointer unique for each thread. The **thread local storage** is used to store variables unique for each thread.
There are two types of threads, *kernel-level threads* and *user-level threads*. Section 3 *Kernel-Level Threads and User-Level Threads* investigates this topic in detail.

Current OS' like Windows support *multithreading* (Stallings 2015, 100, 184). The term multithreading reveals a technique of an OS to divide a single process into multiple threads that run concurrently (see figure 5). On the right side of figure 5 two multithreading models are shown. The OS of a SMP schedules and distributes threads across all available processors (Stallings 2015, 101, 113-114). Windows is such an OS. Multiple threads within one process can execute concurrently on multiple processors.

Using multiple threads instead of multiple processes can be advantageous due to the fact they are more lightweight (Stallings 2015, 101, 186). The time for creation and termination of thread takes less time compared to processes. Furthermore, switching between threads produces less overhead than switching between processes.
In general multithreading is also advantageous when an application has to execute different independent tasks (Stallings 2015, 101, 186-187). Examples are foreground and background work of an application or when the program structure offers modularity executable on threads.

2.4 Concurrency and Multithreading Issues

When writing multithreaded applications developers have to face concurrency related issues that they need to be aware of in order to correctly deal with those issues (Lake 2011, 373; Stallings 2015, 230-231). An excerpt is listed here:

- Atomic operation
- Critical section
- Deadlock
- Mutual exclusion
- Mutex
- Race condition
- Starvation

An **atomic operation** is a sequence of one or more instructions that can only be executed as a whole or not at all (Lake 2011, 373-376; Stallings 2015, 231-249). Atomic operations do not need to be protected by application code. When a thread is in a **critical section** of code, that accesses shared resources, no other thread is able to enter this section. A **deadlock** is a situation where multiple threads are unable to proceed because they wait for each other. **Multual exclusion** means the ability that when one thread is in a critical section in which shared resources are accessed, no other threads are allowed to be in any critical sections that access any of those shared resources. A **mutex**[7] is a programming mechanism that is used to acquire *exclusive* access to data. A lock is acquired. Other attempts by threads to acquire the lock are blocked. The thread which locks the mutex must be the one unlocking it. A **race condition** is a situation in which multiple threads modify and read shared data and the result is not deterministic. **Starvation** occurs when the scheduler ignores a thread for execution although it is able to be executed.

3 Kernel-Level Threads and User-Level Threads

As mentioned in section 2.3 *The Concept of Processes and Threads* there are two general types of thread implementations (Stallings 2015, 189):

- Kernel-Level Threads
- User-Level Threads

7. `http://en.cppreference.com/w/cpp/thread/mutex` Accessed June 18, 2017

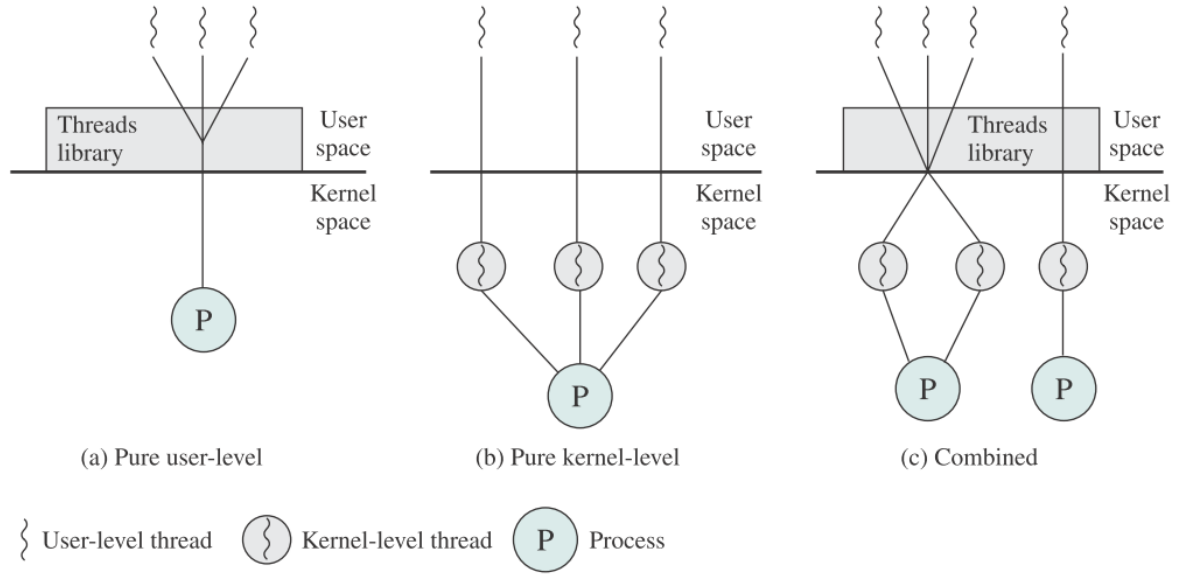


Figure 7: Three types of threading approaches: pure user-level threads, pure kernel-level threads and a combined approach (multiple user-level and kernel-level threads) (Stallings 2015, 189-194).

Kernel-Level Threads

A *kernel-level thread* (see figure 7) is the type of thread which is also known as *lightweight process* (Stallings 2015, 189-193). Kernel-level threads are observed and administrated by the kernel. The application using kernel-level threads is not involved in any direct thread management. The kernel manages processes and their threads. The OS Windows makes use of a kernel-level thread model.

One advantage of kernel-level threads is that multiple threads of one process are able to be executed parallel on multiple hardware cores (Stallings 2015, 193). Blocking threads is another topic that can be handled by the kernel. If one thread of a process is blocked, the kernel can schedule other threads of the same process. Thus, the OS tries to always run a ready thread.

The major disadvantage of kernel-level threads occurs when switching between threads of the same process (Stallings 2015, 193-194). A mode switch from user space to kernel space has to be performed (see section 2.2 *Basics of Operating Systems*). Moreover, when threads switch, also known as *context switch*, the processor needs to save all registers (see section 2.1 *Computer Hardware: Overview and Processors*) of the current running thread and load the registers of the next thread (Stallings 2015, 475-476; Intel Corporation 2017b)[8]. Furthermore, the current thread's data is stored in the cache (see section 2.1 *Computer Hardware: Overview and Processors*). When the next threads accesses different data, the data of the current thread is possibly removed from the cache and the new data is transmitted from main memory to the cache. Receiving data from main memory takes hundreds of cycles. This process of constantly

8. `https://software.intel.com/en-us/node/506127` Accessed June 16, 2017

removing data from cache and fetching data from main memory costs much time.

User-Level Threads

User-level threads (see figure 7) exclusively operate in user space (see section 2.2 *Basics of Operating Systems*) (Stallings 2015, 189-193). The kernel does not know about them. It can not distribute them across multiple processors. This kind of threads are entirely managed by the application respectively a threads library. The data structures, life cycles and scheduling of threads is all handled by the application.

There are some advantages of user-level threads over kernel-level threads (Stallings 2015, 192). Switching between threads does not involve the kernel. All the thread data structures and management logic is within user space. The mode switches from user space to kernel space and reverse do not occur.
Another advantage of user-level threads is the context depended applicable scheduling algorithm (Stallings 2015, 192). Some applications focus on throughput. I.e. to maximize the number of work units completed per unit of time (Stallings 2015, 433). Other applications may target deadlines. The scheduling algorithm needs to maximize the number of deadlines met.

Due to the fact that the kernel is unaware of the threads in user space, the kernel schedules the process or kernel thread, where the user-level threads are running, as one unit. Therefore, only that unit can execute on a processor. A pure user-level threaded approach can not make use of multiple hardware cores.
Another problem are blocking system calls (see section 2.2 *Basics of Operating Systems*) (Stallings 2015, 192-194). When a kernel-level thread blocks, the kernel can schedule another kernel-level thread. However, the kernel is not aware of user-level threads. Therefore, if a user-level thread blocks the kernel can not schedule another user-level thread. Hence, the process running the user-level threads just blocks and has to wait until the blocking call is processed.

Thread Pools

Another topic related to threads is the *thread pool*. A thread pool is a collection of threads that process work of an application (Stallings 2015, 201; Tanenbaum and Bos 2015, 911; Hodgman 2016). The threads are created once and reused throughout the application. That is one major advantage of a thread pool. Thread creation is expensive when frequently and dynamically creating and destroying threads for concurrent parts of the application.
The thread pool is also responsible for the management of its threads (Stallings 2015, 201; Tanenbaum and Bos 2015, 911). Thread pools work with queues of tasks. Tasks are pushed to the thread pool by the application and threads pop tasks from the queue and process them.
The implementation of a thread pool may need to consider blocking threads (Tanenbaum and Bos 2015, 911). When a thread executes a task and blocks for e.g. I/O operation the thread pool wants to schedule other threads. That would lead the thread pool to create further threads and possibly create more threads than hardware threads are available. That situation is called *over-subscription* (see section 3.2 *Game Engines and Parallelization* for more details).

3.1 Implementations of Threads in Windows

Windows provides implementations for both, kernel-level and user-level threads (Russinovich, Solomon, and Ionescu 2012, 12-14). The implementation of a kernel-level thread is referred to as *thread*. The implementation of a user-level thread is called *fiber*. There also exists a mechanism called *user-mode scheduling (UMS)*. A UMS thread is visible to the kernel. However, it can perform a context switch in user mode. This thesis will concentrate on the implementation of fibers.

3.1.1 Implementation of Kernel-Level Threads in Windows

In Windows each process is created with one kernel-level thread (Microsoft Corporation 2017)[9] [10]. That thread is also referred to as *primary/main thread*. The thread itself is the part of a process that is scheduled for execution (see section 2.3 *The Concept of Processes and Threads*). All kernel-level threads one process consists of share its virtual address space and system resources. Windows uses *preemptive multitasking* (see section 2.3 *The Concept of Processes and Threads*) for scheduling threads.

Windows provides functions for working with kernel-level threads (Microsoft Corporation 2017)[10]. An excerpt is stated in listing 1.

```
1  HANDLE WINAPI CreateThread(
2    _In_opt_  LPSECURITY_ATTRIBUTES  lpThreadAttributes,
3    _In_      SIZE_T                 dwStackSize,
4    _In_      LPTHREAD_START_ROUTINE lpStartAddress,
5    _In_opt_  LPVOID                 lpParameter,
6    _In_      DWORD                  dwCreationFlags,
7    _Out_opt_ LPDWORD                lpThreadId
8  );
9
10 DWORD WINAPI ThreadProc(
11   _In_ LPVOID lpParameter
12 );
13
14 DWORD WINAPI GetCurrentThreadId(void);
15
16 VOID WINAPI Sleep(
17   _In_ DWORD dwMilliseconds
18 );
```

Listing 1: Thread function signatures taken from the MSDN (Microsoft Corporation 2017)[10].

9. https://msdn.microsoft.com/de-de/library/windows/desktop/ms681917(v=vs.85).aspx Accessed May 30, 2017

10. https://msdn.microsoft.com/de-de/library/windows/desktop/ms684847(v=vs.85).aspx Accessed May 30, 2017

To create an additional kernel-level thread in a process the *CreateThread* function is called (Microsoft Corporation 2017)[11] [12]. The stack size of the thread is defined here and the address to the *ThreadProc* function has to be passed. The *creation flags* parameter determines whether the thread starts its execution directly after it is created or e.g. whether it is suspended. The *ThreadProc* function is a placeholder for the function, defined in the application, which determines the start address for a thread. In Windows the type *LPTHREAD_START_ROUTINE* defines a pointer to this function and reveals the required function signature. *GetCurrentThreadId* returns the thread identifier (DWORD) of the thread calling this function. *Sleep* suspends the thread for the given time interval. When the value zero is passed, the currently executing thread gives up its *time slice* (see section 2.3 *The Concept of Processes and Threads*) and another thread is scheduled for execution if possible.

When an error occurs the *GetLastError* will return an error code specifying the circumstances of the error. This function can be used generally when errors occur using Windows-specific thread, fiber or UMS functions.

3.1.2 Implementation of User-Level Threads in Windows

One implementation of threads operating in user space Windows provides is (Russinovich, Solomon, and Ionescu 2012, 13; Microsoft Corporation 2017)[11]: *Fibers*.

Fibers

A **fiber** is a lightweight thread (Russinovich, Solomon, and Ionescu 2012, 13; Microsoft Corporation 2017)[11] [12] . It has to be scheduled manually. Fibers run in the context of the threads that schedule them. Fibers are invisible to the kernel. Hence, switching fibers do not involve the kernel scheduler. There are several functions and macros for working with fibers, listing 2 lists an excerpt of those.

The *ConvertThreadToFiber* function converts the current thread into a fiber (Microsoft Corporation 2017)[12] [13] [14]. Before creating or scheduling fibers this function has to be called due to the fact that fibers can only be executed by other fibers. This function has an optional parameter which is a pointer to an arbitrary object associated to the fiber. This parameter can be retrieved by the macro *GetFiberData*. One possible practice is to pass a *fiber data structure* holding data related to the fiber. If the function is executed successfully a pointer to the fiber is returned. To create a new fiber, *CreateFiber* is called. A new fiber can only be created by another fiber. This function allocates memory for a new fiber object and sets up a stack. Unlike the *CreateThread*

11. `https://msdn.microsoft.com/de-de/library/windows/desktop/ms681917(v=vs.85).aspx` Accessed May 30, 2017
12. `https://msdn.microsoft.com/de-de/library/windows/desktop/ms684847(v=vs.85).aspx` Accessed May 30, 2017
13. `https://msdn.microsoft.com/en-us/library/windows/desktop/ms682661(v=vs.85).aspx` Accessed May 30, 2017
14. `https://msdn.microsoft.com/en-us/library/windows/desktop/ms686919(v=vs.85).aspx` Accessed May 30, 2017

```
1  LPVOID WINAPI ConvertThreadToFiber(
2    _In_opt_ LPVOID lpParameter
3  );
4
5  LPVOID WINAPI CreateFiber(
6    _In_     SIZE_T                dwStackSize,
7    _In_     LPFIBER_START_ROUTINE lpStartAddress,
8    _In_opt_ LPVOID                lpParameter
9  );
10
11 VOID CALLBACK FiberProc(
12   _In_ PVOID lpParameter
13 );
14
15 PVOID GetCurrentFiber(void); //macro
16
17 PVOID GetFiberData(void); //macro
18
19 VOID WINAPI SwitchToFiber(
20   _In_ LPVOID lpFiber
21 );
22
23 VOID WINAPI DeleteFiber(
24   _In_ LPVOID lpFiber
25 );
```

Listing 2: Fiber function signatures taken from the MSDN (Microsoft Corporation 2017)[12].

function the fiber is not scheduled in any way by this function call. *FiberProc* acts for fibers in the same way the *ThreadProc* function does for threads (see section 3.1.1 *Implementation of Kernel-Level Threads in Windows*). In Windows the type *LPFIBER_START_ROUTINE* defines a pointer to this function and reveals the required function signature. The *SwitchToFiber* function is used to schedule fibers. Here you specify a fiber to be switched to and executed next. When a fiber is no longer needed, it is deleted. The *DeleteFiber* function removes data like the stack and registers of a fiber. If this function is called on the currently executing fiber, its thread is also terminating.

3.2 Game Engines and Parallelization

Games, respectively game engines (see figure 8), require a lot of processing power in order to simulate a detailed and credible game world (Andrews 2015). Therefore, utilizing all possible processing units is one goal a game engine should focus on. The game is split up in multiple parts that are executed concurrently on as many processors as possible. However, that is a challenge. In games many different objects and systems interact with each other and depend on one another (see figure 8).
Parallelizing applications with relatively clear and independent tasks is easier. An application with a graphical user interface (GUI) could have one thread for handling GUI events and a pool

of worker threads executing user commands and calculating expensive operations (Stallings 2015, 186; Hodgman 2016).

Since games have to be as fast as possible the synchronization (see section 2.4 *Concurrency and Multithreading Issues*) overhead must be at a minimum (Andrews 2015). Therefore, a snapshot or copy of commonly accessed data at a certain time is available for every system. That snapshot is called *execution state*. Each system has its own copy of the execution state. Changes to the systems' execution state copy are sent to a *state manager* which processes all incoming changes and broadcasts the updated execution state when all systems finished processing. The data synchronization should be tied to certain time steps and can be or can not be equivalent to a frame.

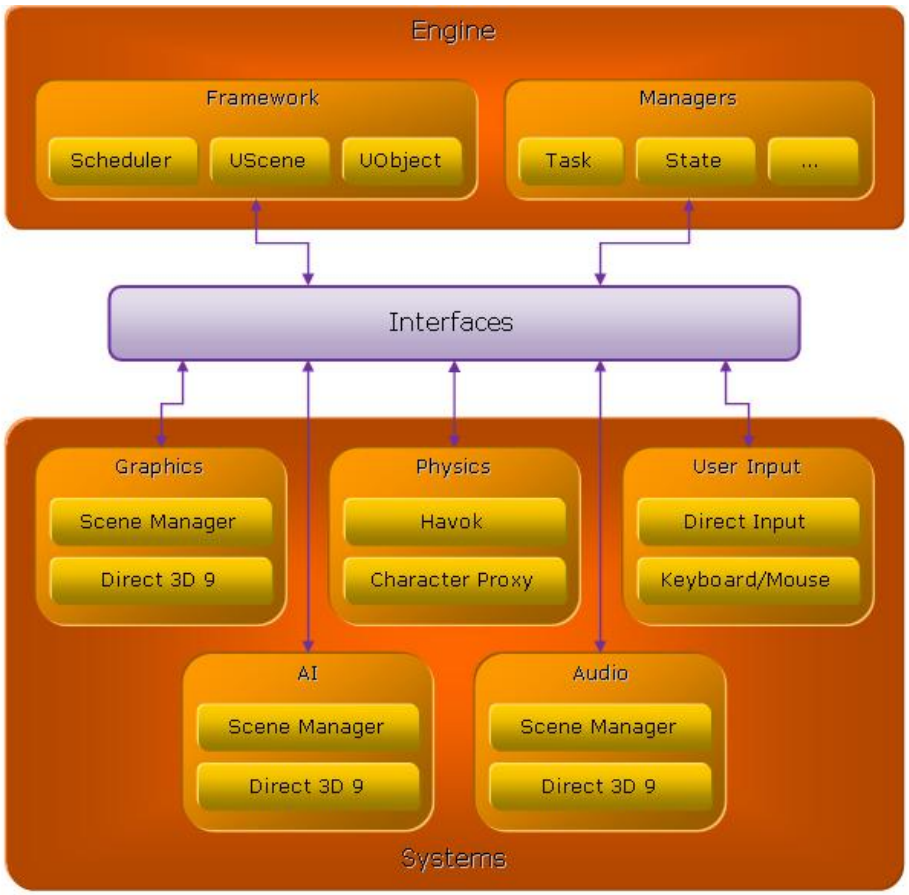


Figure 8: An example game engine diagram (Andrews 2015). The core part of this game engine is the framework and the managers. The game engine can be extended by an arbitrary number of systems representing game processing functionality.

A generalized game engine architecture can be structured as in figure 8 (Andrews 2015). The framework contains i.a. data which is accessible through the execution state. Managers are accessible through the singleton pattern and provide global functionality, which is used across multiple systems. The concrete game logic/functionality is stored in different systems. Due

to the modularity the system interface provides, the concrete systems must communicate with each other and the engine. Accessing shared resources is handled by the state manager and requesting functionality is handled by the service manager.

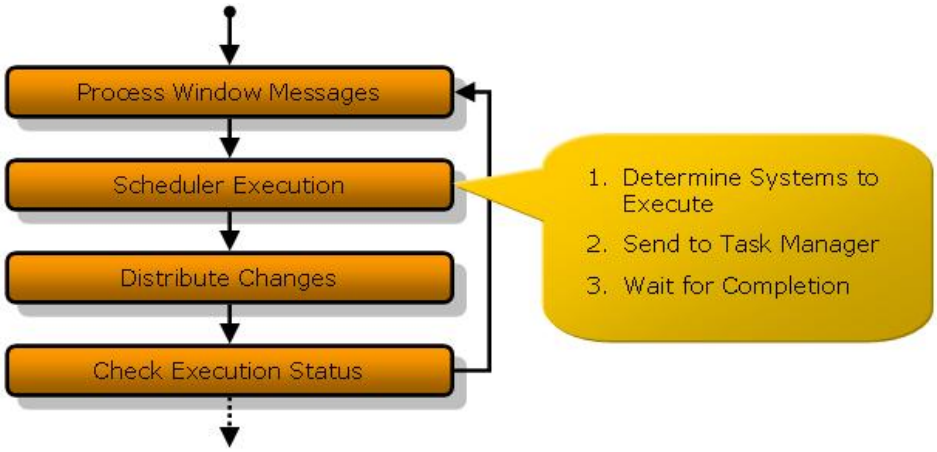


Figure 9: A coarse view of a game loop consisting of four steps (Andrews 2015): *window message processing*, *(task) scheduler execution*, *state change distribution* and *execution status check*.

The framework of a game engine also includes the game loop (see figure 9) (Andrews 2015). The first step is to handle the window the game is running in. Next, the concrete systems and their tasks need to be determined and executed. After the tasks have finished execution, the changes to the execution state are distributed. The final step is to check the execution status to decide to quit the game or perform other actions.

When threading a game engine a basic strategy can be applied: *fork and join* (Gregory 2014, 369; Hodgman 2016). The game engine executes its instructions sequentially until it reaches a point where it can run code concurrently. E.g. a big loop of independent operations. The work is split between multiple threads for processing (that process is called *fork*) and merged after all threads finished working (that is called *join*). After the work is merged the game engine runs again sequentially until the fork and join strategy can be applied again.

One technique to utilize multiple processors in a game engine is to assign one thread per major system (AI, physics, render, etc.) (Granatir and Rodriguez 2010; Stallings 2015, 199-201; Hodgman 2016). Such an approach can be referred to as *coarse threading*. A synchronization of the execution state between all threads is required within a certain time step. Thus, some threads may have to wait for others due to an unbalanced work load of different systems. Moreover, the number of systems and their threads may not fit the number of available hardware threads and therefore lead to *over-* or *under-subscription* (Intel Corporation 2017b)[15]. Under-subscription describes the situation when there are less kernel-level threads than available hardware threads. Thus, the processor is potentially not fully utilized. Over-subscription occurs when there are more kernel-level threads than hardware threads. The OS has to schedule more kernel-level

15. `https://software.intel.com/en-us/node/506100` Accessed June 16, 2017

threads. The number of context switches increases (see section 3 *Kernel-Level Threads and User-Level Threads*). That costs valuable time.

Another approach how to utilize multithreading in a game engine is to implement a *job system* (Andersson 2009; Gregory 2014, 371-72; Andrews 2015; Stallings 2015, 199-201; Hodgman 2016). The game logic/functionality is defined as a set of jobs. Multiple jobs are spread across available processors to be processed in parallel. That approach is also referred to as *fine-grained threading*. More details about job systems and game engines is stated in section 4 *Job Systems*.

Using only coarse-grained parallelism has its limits (Stallings 2015, 199-201). Furthermore, fine-grained parallelism with tasks is a complex challenge. Not every system can be efficiently displayed as multiple tasks. A hybrid solution worked for Valve best. They identified systems which work efficiently when applying coarse-grained parallelism and others which utilize a task-based approach. The concrete implementation of a hybrid approach depends on the game/game engine and must adapted for specific purposes.

4 Job Systems

Current game engines like Frostbite and Naughty Dog (see section 1 *Introduction*) make use of job systems since they are able to utilize multi-core processors very well. Thus, this thesis examines the characteristics and practices of job systems.

Definition: Job and Job System

A *job*, also called *task*, is a set of instructions which represent a relatively small and independent amount of work (Muffat-Méridol 2009a; Lake 2011, 379-383). A job can also be referred to as some data and instructions working with that data (Gregory 2014, 371). A job is the basic element a *job system*, also called *task scheduler* (Sanglard 2013; Reinalter 2017)[16], is working with. The responsibility of a job system is to assign jobs to threads. Those threads are then processing their jobs. The data structure which holds jobs is in most cases a *queue*.

The goal of a job system is to decouple algorithmic parallelism from hardware parallelism (Granatir and Rodriguez 2010; Minadakis 2011a). The work should not be assigned to a constant number of threads. That leads to over-/under-subscription (see section 3.2 *Game Engines and Parallelization*). It should scale with any number of available processors automatically. Hence, the game is split up to independent jobs. E.g. an animation system processing all models would be split to jobs animating a single model.

4.1 General Approach and Flow

Assume there is a system with `N` hardware threads. Furthermore, there is a primary thread (see section 3.1.1 *Implementation of Kernel-Level Threads in Windows*) already executing instruc-

16. `https://blog.molecular-matters.com/2015/08/24/job-system-2-0-lock-free-work-stealing-part-1-basics/` Accessed June 16, 2017

tions. When a job system is initialized it creates worker threads for processing jobs (Muffat-Méridol 2009a; Andrews 2015). In order to avoid over-subscription `N-1` worker threads are created. Therefore, job systems automatically scale with any number of logical cores (Minadakis 2011a).

A straight-forward approach stores all jobs in one queue (Muffat-Méridol 2009a; Andrews 2015; Reinalter 2017)[17]. All threads can push and pop jobs on that queue. Therefore, the access to the queue must be locked with a synchronization primitive like a *mutex* (see section 2.4 *Concurrency and Multithreading Issues*). A mutex allows only one thread at a time to enter a critical section.
Due to the fact all jobs are stored in one queue the applied scheduling policy is first-in-first-out (FIFO), also known as first-come-first-served (FCFS) (Stallings 2015, 435-439). That means the first job entered the queue will be the first job to be processed. The time spent waiting in a queue to be processed equals the sum of execution duration of all jobs enqueued before.

```
1  typedef void (*JobFunction)(Job*, const void*);
2
3  struct Job
4  {
5    JobFunction function;
6    Job* parent;
7    int32_t unfinishedJobs; // atomic
8    ...
9  };
```

Listing 3: Job struct definition (Reinalter 2017)[17].

```
1  namespace tbb {
2          class task {
3          protected:
4              task();
5
6          public:
7              virtual ~task() {}
8
9              virtual task* execute() = 0;
10             ...
```

Listing 4: Task class definition (Intel Corporation 2017b)[20].

There are two basic approaches how a job can be implemented:

- Function pointer/callback function and associated data (Andersson 2009; Lengyel 2010,

17. https://blog.molecular-matters.com/2015/08/24/job-system-2-0-lock-free-work-stealing-part-1-basics/ Accessed June 16, 2017

381-390; Minadakis 2011a; Gyrling 2015; Reinalter 2017)[18]

- Class with virtual execution function and associated data (Muffat-Méridol 2009b; Lake 2011, 379f; Intel Corporation 2017b)[19] [20]

As shown in listing 3 a job can be implemented as a struct with mainly a function pointer and job specific data (Reinalter 2017)[18]. In this case, the job function has two parameters: the job itself and a pointer to data the job is working with. Furthermore, the variable *unfinishedJobs* determines whether the job is finished or not. When the job is created, the value is atomically (see section 2.4 *Concurrency and Multithreading Issues*) incremented by a function. When the job is finished, the value is atomically decremented by a function.
Listing 4 presents an excerpt of Intel's *Threading Building Blocks (TBB) task* class (Intel Corporation 2017b)[20]. The class has a virtual *execute* function which requires to be overridden. In order to create a new task a concrete task class is created which derives from task and implements the execute function.

When using a job system to process the game, the systems (see section 3.2 *Game Engines and Parallelization*) must describe their game logic/functionality as jobs and dependencies between jobs (Minadakis 2011b; Andrews 2015).
In order to create a suitable job for the job system some considerations about the job design need to be made. An imprecise statement is that a job should not be too big and not too small (Minadakis 2011b; Sanglard 2013). The minimum complexity of a job should not result in a job switching overhead and the maximum complexity should not cross a border where jobs can no longer distributed efficiently across multiple workers.
Therefore, well sized and independent jobs utilize multi-core processors most, since they can be spread across all available workers and do not have to wait on other jobs (Gregory 2014, 371-372).
Avoiding locks (see section 2.4 *Concurrency and Multithreading Issues*) or blocking calls (see section 2.2 *Basics of Operating Systems*) in jobs is another job design advice. Locking or blocking call result in a blocked worker thread and may cause cache flush (Granatir and Rodriguez 2010).

One goal of a job is to be independent from other jobs. A game is a very complex simulation and sets of rules (Gregory 2014, 8-13, 845-850; Andrews 2015). Therefore, dependencies between different jobs do exist. E.g. before a scene can be rendered, all animations need to be calculated first (Minadakis 2011b). That means the rendering system depends on the animation system and has to wait for it. The game can be represented by a graph of jobs (see figure 10). That *job graph* could represent one frame of a game. When a thread calls a wait function, the thread stops its execution until the wait condition becomes true. Thus, the thread blocks and the utilization of hardware threads decreases when the number of threads respects over-subscription (see section 3.2 *Game Engines and Parallelization*).

18. `https://blog.molecular-matters.com/2015/08/24/job-system-2-0-lock-free-work-stealing-part-1-basics/` Accessed June 16, 2017
19. class CInternalTask in nulstein/nulstein/TaskScheduler.h
20. `https://software.intel.com/en-us/node/506299` Accessed June 16, 2017

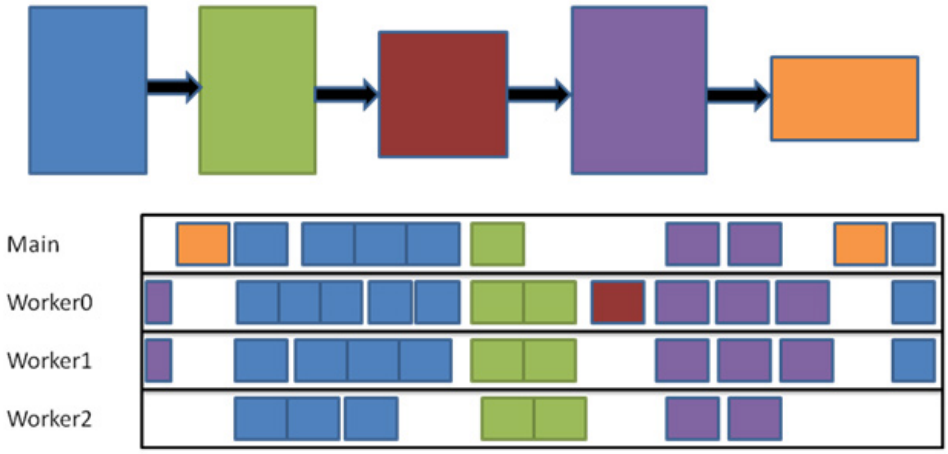


Figure 10: An example job dependency graph (Minadakis 2011b). The big blocks represent parent jobs which are related (colorized) to their child jobs (little blocks). In this example the parent jobs depend on one another.

One approach how jobs can be designed in order to reduce contention of the primary thread job queue is to initially create few jobs which again create sub-jobs which again can be pushed within a workers job queue without locking (Andrews 2015).

4.2 Job System Improvements

Compared to other multithreading approaches like one thread per subsystem and fork-and-join algorithms (see section 3.2 *Game Engines and Parallelization*) a simple job system does utilize hardware threads very well (Muffat-Méridol 2009a; Andrews 2015). However, there are some improvements that can be applied to a simple job system that further enhance performance.

Multiple Queues and Work Stealing

Instead of using one common queue for all jobs one job queue per thread can be implemented (Muffat-Méridol 2009a). One common queue has much contention due to the fact that N threads access that queue. At first the only active thread is the primary thread and the worker threads sleep and wait for a signal that jobs are available. Then the primary thread pushes jobs to its queue and signals worker threads that jobs may be available. The worker threads then pop jobs from the primary thread queue and start executing them. Since jobs can create further jobs, these are enqueued to the queue of the executing thread which results in no locks on worker thread queues.

This approach can result in unbalanced job queues (Muffat-Méridol 2009a). That can cause starvation of a thread. To overcome this situation the concept of *work stealing* can be used (Reinalter 2017)[21]. When a thread has no job in its queue it determines another thread and tries

21. https://blog.molecular-matters.com/2015/08/24/job-system-2-0-lock-free-work-stealing-part-1-basics/ Accessed June 16, 2017

to steal the bottom half of jobs from that queue. That requires a lock. However, that lock is only used when the job queue is empty. Moreover, only two threads access such a queue. In fact when the worker threads are initialized and wait for jobs to be processed they do steal jobs from the primary thread.

Lock-Free Queues

One improvement to a simple job system with a global queue was to implement multiple queues and work stealing. That reduces the amount of contention. However, there are still critical sections which must be locked.
There exists a programming technique called *lock-free* or *lockless* programming (Microsoft Corporation 2017)[22]. The essence of lock-free programming is that shared resources between multiple threads can be safely accessed without using locking mechanics. However, lock-free programming is very complex. A lot of knowledge about hardware and compiler is required.
In order to utilize lock-free programming in a job system a *lock-free job queue* would enhance performance since there are many small jobs which are continuously created, enqueued and dequeued (Reinalter 2017)[23].

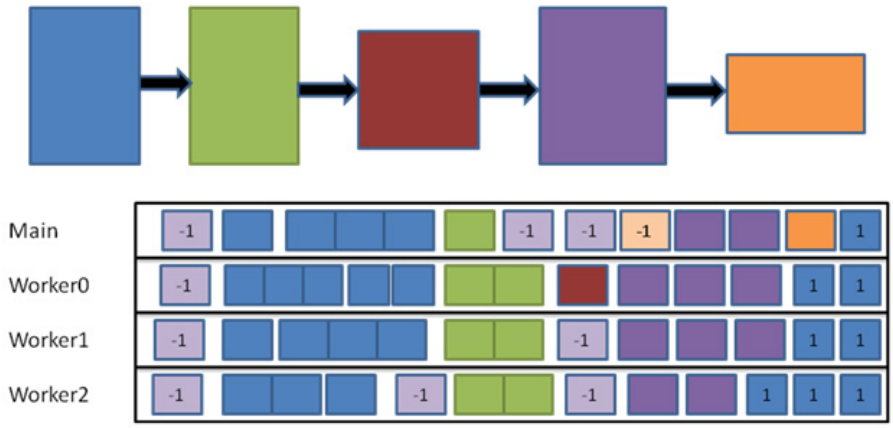


Figure 11: This figure shows overlapped job dependency graphs (Minadakis 2011b). When working with frames the job graphs of the previous frame is marked with `-1` and the job graph of the next frame is marked with `1`.

Job Graph: Overlapped Frames

There is a strategy which avoids blocking a thread entering a wait function (Minadakis 2011b). The thread can help to process other available jobs until the wait condition is true. This scenario only applies when the dependency graph allows other independent jobs to be processed. When

22. `https://msdn.microsoft.com/en-us/library/windows/desktop/ee418650(v=vs.85).aspx` Accessed June 16, 2017
23. `https://blog.molecular-matters.com/2015/09/25/job-system-2-0-lock-free-work-stealing-part-3-going-lock-free/` Accesed June 16, 2017

there are no possible jobs of the current frame available, then jobs of other frames can be processed. This technique is called *overlapped dependency graph execution* (see figure 11).

There are further possibilities how job systems can be improved. Examples are custom scheduling algorithms, job memory management and memory efficient job queues (Sanglard 2013; Andrews 2015; Stallings 2015, 433-435; Reinalter 2017)[24].

4.3 Game Engines and Job Systems: Examples

A game engine consists of many sub systems (Gregory 2014, 33, 340-341, 916). These systems have a responsibility and process/do a specific work. In order to implement a job system the game logic/functionality the systems execute need to be split up into independent portions of work (Granatir and Rodriguez 2010; Minadakis 2011a).

Assume there is an *animation system* (Minadakis 2011b). The animation system animates all models in the game. There is one big loop which iterates over all models and performs all animations per frame. Since big loops are a good indicator for parallization (Granatir and Rodriguez 2010), the loop is split up in jobs. Each resulting job would have the responsibility to animate only one model. Therefore, there would be as many jobs as models to be animated.

Another system that can be represented by jobs is the *AI system* (Granatir and Rodriguez 2010). Assume the AI system has an array of AI objects and iterates over the array every update step. The AI may need information about their surroundings to determine their next decision. Therefore, the AI would access the current execution state (see section 3.2 *Game Engines and Parallelization*) and retrieve data. Read-only access to data is practicable since that can be managed without locks. If data has to be written, outsourcing that logic in other jobs should be considered.

4.4 Pros and Cons

There are clear advantages for using a job system (Intel Corporation 2017b)[25] instead of directly working with threads. When using threads, the programmer need to consider utilizing the hardware cores. On Windows each logical thread corresponds to a physical thread (see section 3 *Kernel-Level Threads and User-Level Threads*). When there are not as many threads as available processors, *under-subscriptions* (see section 3.2 *Game Engines and Parallelization*) occurs. The application will not use all available processors. When there are more logical threads than hardware cores *over-subscription* takes place. Another reason for working with jobs instead of threads is that jobs are more lightweight in terms of initialization and termination. A job is basically a small set of instructions that gets executed by a thread. A thread consists for instance of a state and context data (see section 2.3 *The Concept of Processes and Threads*) that has to be managed. Another advantage of using a job system is the custom scheduling algorithm that can

24. `https://blog.molecular-matters.com/2015/09/08/job-system-2-0-lock-free-work-stealing-part-2-a-specialized-allocator/` Accessed June 16, 2017
25. `https://software.intel.com/en-us/node/506100` Accessed June 16, 2017

be implemented. The OS typically uses fair scheduling algorithms like Round-Robin. In terms of an OS that is reasonable. When writing a job system for a game engine the goal is to dispatch jobs in an efficient way. *Load balancing* is a duty the job system administrates. It distributes jobs equally over threads. When working with threads, load balancing has to be done manually. To sum up, the major advantage of using tasks over threads is that it relieves the developers. They can focus on designing tasks and do not have to care about multithreading.

One challenge of job systems is the correct and efficient implementation of a job system. That is lots of work. The implementation of correct locking needs to be done. If the job system is not working correctly, the entire game engine is not working correctly.
Another possible challenge for the programmers is thinking is tasks. An existing game or game engine needs to be split into tasks (see section 3.2 *Game Engines and Parallelization*). The effort to be made to split a system up in multiple tasks and defining dependencies needs to be considered.

5 Implementation of Job Systems utilizing Threads

After the topics threads and job systems were examined, kernel-level and user-level threads will be applied to job systems.
In order to examine the utilization of kernel-level and user-level threads in job systems two approaches of job systems with different kinds of thread usage strategies were implemented. Measurements concerning time and processor utilization usage will be performed. The results of the two approaches will be presented and compared to each other. Further improvements how the implementations could perform better will be stated.

The basic job system architecture is in the two implementations the same. Jobs are stored in FIFO queues and are executed as fast as possible. The job system is used to execute jobs of pseudo game systems. There are different *systems* (see section 3.2 *Game Engines and Parallelization*), responsible for the game's functionality, which push jobs to the scheduler. The scheduler is responsible for distributing and assigning jobs to threads.

Note: The entire source code of the implementations can be downloaded following the link in the appendix.

5.1 Job Management

A job is implemented as struct holding a function pointer and associated job-related data (see listing 5 and section 4 *Job Systems*). The job holds a pointer to arbitrary data which can be accessed within the job callback function, since the job itself is passed to the function. There also is a pointer to a parent job and an atomic work counter which is used for job dependencies (see section 4.1 *General Approach and Flow*).

To avoid memory allocations at runtime, which cause memory fragmentation, and provide some management functions concerning jobs a *JobManager* is implemented (see listing 6) (Reinalter

```
9  struct Job;
10
11 typedef void(*JobCallback)(Job*);
12
13 struct Job
14 {
15   Job()
16     : m_Data(nullptr), m_Parent(nullptr), m_WorkCounter(0)
17   {}
18
19   void* m_Data;
20   JobCallback m_Callback;
21   Job* m_Parent;
22   std::atomic_int m_WorkCounter;
23 };
```

Listing 5: Job.h (see source files of the author)

2017)[26]. The *JobManager* allocates the memory for all jobs in advance and provides functions for retrieving a new job and finishing completed jobs. After a callback function is executed by a worker, the *FinishJob* function is called to decrease the work counter (see listing 5) of the job itself. When the work counter reaches zero, the job is completed. As other manager classes the *JobManager* is implemented with the singleton pattern (see section 3.2 *Game Engines and Parallelization*).

Jobs can be related (see section 4.1 *General Approach and Flow*). A major job may have an arbitrary number of child jobs which belong to its parent (see figure 10). A child job is created by calling the *CreateChildJob* function of the *JobManager* (see listing 6). This function sets the parent of the child job and atomically (see section 2.4 *Concurrency and Multithreading Issues*) increases the work counter of the parent by one. All the jobs, parent and children, are processed by different worker threads. When a job function is finished executing, the *FinishJob* function is called. When the work counter of the job reaches zero and it also has a parent, then the work counter of the parent is also decreased by one. In that manner it can be determined by the parent job whether it and all its children completed the execution.

Since jobs may depend on each other the *JobManager* provides the *WaitForJob* function (see listing 6). Assume there is job *A* and job *B*. Due to the game rules job *B* is only allowed to be executed after job *A* finished processing its logic. However, jobs are pushed to queues and processed by worker threads depending on the number of jobs in the queue. To implement a dependency between job *A* and job *B* the *WaitForJob* function is used. The function waits until the target work counter is reached and then exits the function. In most cases the value of the target work counter is zero since that value signals a complete job. When a job is pushed after the *WaitForJob* function is called then it depends on the given job. This approach also works with parent and child jobs.

26. https://blog.molecular-matters.com/2015/09/08/job-system-2-0-lock-free-work-stealing-part-2-a-specialized-allocator/ Accesed June 16, 2017

```
7  class JobManager
8  {
9  public:
10   ~JobManager();
11
12   static JobManager* GetInstance();
13
14   Job* CreateJob();
15
16   Job* CreateChildJob(Job* parent);
17
18   void FinishJob(Job* job);
19
20   void WaitForJob(Job* job, int targetWorkCounter);
21   ...
```

Listing 6: JobManager.h (see source files of the author)

5.2 Scheduler and Worker Implementations

In order to examine the research question of this thesis two different scheduler/worker approaches are implemented. A pure user-level thread approach will not be examined due to the fact only one hardware thread would be utilized (see section 3 *Kernel-Level Threads and User-Level Threads*). The two approaches are listed here:

Assumption: There are `N` hardware threads and one primary thread is executing the game loop.

- `N-1` kernel-level worker threads
- `N-1` kernel-level scheduler threads with `X` user-level worker threads

Each implementation consists of a scheduler and workers. The scheduler is responsible for the initialization of workers and their threads. Moreover, the scheduler provides a function for pushing jobs to. Workers mainly process jobs. The exact flow depends on the implementation.

5.2.1 Pure Kernel-Level Thread Approach

N-1 Kernel-Level Worker Threads

This pure kernel-level thread approach consists of two elements: *KernelThreadScheduler* and *KernelThreadWorker*.

The KernelThreadScheduler is the class game systems push jobs (the thread executing this instruction trace is the primary thread) to (see listing 7). The KernelThreadScheduler determines the KernelThreadWorker with the minimum sized job queue and adds a job to that queue. In order to access the KernelThreadScheduler the singleton pattern is applied: The *GetInstance* function is used to globally retrieve the only instance of the scheduler. The scheduler holds

```
8  class KernelThreadScheduler
9  {
10 public:
11   ~KernelThreadScheduler();
12
13   static KernelThreadScheduler* GetInstance();
14
15   void Initialize();
16
17   void ActivateWorkers();
18
19   bool PushJob(Job* job);
20
21   bool AreAllWorkerThreadsReadyForProcessing() const;
22
23   ...
```

Listing 7: KernelThreadScheduler.h (see source files of the author)

an array of KernelThreadWorker objects. The *Initialize* function initializes `N-1` KernelThreadWorker objects. Furthermore, the KernelThreadScheduler offers functions to activate workers and to check the ready status of worker threads.

The KernelThreadWorker is responsible for processing jobs (see listing 8). Each KernelThreadWorker is initialized by the KernelThreadScheduler. Per KernelThreadWorker one kernel-level thread is created. The stack size and address to its processing function is passed to the *CreateThread* function (see section 3.1.1 *Implementation of Kernel-Level Threads in Windows*). The current stack size of a KernelThreadWorker's kernel-level thread is 512KB. After the kernel-level thread is created it enters the specified function and waits to be activated. As soon as all KernelThreadWorkers are ready all of them are activated by the KernelThreadScheduler shortly before the game's systems start creating and pushing jobs to the KernelThreadScheduler.

The processing of jobs starts in the loop of the KernelThreadWorker (see listing 9). The current implementation just waits for jobs to be pushed. If jobs are available, the worker tries to pop a job from its job queue. That requires a lock. If the job was successfully popped the job is executed and the *FinishJob* function of the *JobManager* is called to complete the job (see section 5.1 *Job Management*).

5.2.2 Hybrid Approach: Kernel-Level and User-Level Threads

`N-1` Kernel-Level Scheduler Threads with `X` User-Level Worker Threads

The basic hybrid approach with kernel-level and user-level threads is composed of three components: *FiberScheduler*, *FiberSchedulerThread* and *FiberWorker*.

The interface of the FiberScheduler is very similar to the KernelThreadScheduler (see listing 7). The FiberScheduler has for example functions for initialization, pushing jobs and activating its

```
6  class KernelThreadWorker
7  {
8  public:
9    KernelThreadWorker();
10   ~KernelThreadWorker();
11
12   ...
13
14   void Initialize(KernelThreadScheduler* scheduler);
15
16   DWORD ThreadProc();
17
18   bool IsReadyForProcessing() const;
19
20   void Activate();
21
22   Job* FrontJob();
23
24   bool PushJob(Job* job);
25
26   bool PopJob(Job** OUT_Job);
27
28   size_t JobCount();
29
30   size_t JobCount_Unsafe() const;
31
32   ...
```

Listing 8: KernelThreadWorker.h (see source files of the author)

threads. However, instead of holding an array of KernelThreadWorkers it has an array of FiberSchedulerThreads. $N-1$ FiberSchedulerThread objects are initialized by the FiberScheduler and jobs are pushed to those objects according to the minimum sized job queue.

A FiberSchedulerThread is a kernel-level thread that is responsible for managing user-level threads and assigning jobs to them. The concrete user-level thread implementation used in this application is the Windows mechanism *fibers* (see section 3.1.2 *Implementation of User-Level Threads in Windows*). A FiberSchedulerThread mainly consists of an array of FiberWorker objects, a queue of ready FiberWorkers and a queue of jobs (see listing 10).
The initialization function of a FiberSchedulerThread creates a kernel-level thread with a specified stack size (512KB) and start address (see section 3.1.1 *Implementation of Kernel-Level Threads in Windows*). As soon as the kernel-level thread is created it *converts* itself to a fiber in order to be able to create and schedule/manage other fibers (see section 3.1.2 *Implementation of User-Level Threads in Windows*). Afterwards, its FiberWorkers are initialized. When all FiberWorkers are initialized and pushed to its queue of ready FiberWorkers the FiberSchedulerThread waits to be activated.
After the FiberSchedulerThread enters its job processing loop it waits for jobs to be pushed to its job queue (see listing 11). If jobs are available and FiberWorkers are ready a job is tried to

```
35 std::unique_lock<std::mutex> guard(m_scheduleMutex);
36 m_scheduleCondition.wait(guard);
37 while (true)
38   {
39     while (JobCount_Unsafe() <= 0)
40     {
41       ...
42       //Improvement: work stealing
43       Sleep(1);
44     }
45
46     Job* job = nullptr;
47     if (PopJob(&job))
48     {
49       job->m_Callback(job);
50       JobManager::GetInstance()->FinishJob(job);
51     }
52   }
```

Listing 9: KernelThreadWorker.cpp (see source files of the author)

```
1 FiberWorker m_fibers[MAX_FIBERS_PER_SCHEDULER_THREAD];
2
3 std::queue<FiberWorker*> m_readyFibers;
4
5 std::queue<Job*> m_jobs;
```

Listing 10: FiberSchedulerThread.h (see source files of the author)

be popped. That process requires a lock. If a job is successfully popped, it is assigned to the first ready FiberWorker. Then the fiber is switched from the fiber of the FiberSchedulerThread to the fiber of the FiberWorker. The FiberWorker starts processing the job.

The FiberWorker represents the Windows user-level thread implementation *fiber* (see section 3.1.2 *Implementation of User-Level Threads in Windows*) with a job processing ability.
When the *Initialize* function of the FiberWorker is called a new fiber is created (see section 3.1.2 *Implementation of User-Level Threads in Windows*). The stack size (16KB), start address and associated fiber object is passed to the *CreateFiber* function.
The FiberSchedulerThread switches to a FiberWorker when a job should be processed (see listing 11). The fiber of the FiberWorker is then executing its *Proc* function (see listing 12). The job is executed and completed with the *FinishJob* function of the JobManager. Afterwards the FiberWorker is pushed to its FiberSchedulerThreads ready FiberWorker queue. Then the fiber is switched back to the fiber of the FiberSchedulerThread.

```
44 std::unique_lock<std::mutex> guard(m_scheduleMutex);
45 m_scheduleCondition.wait(guard);
46 while (true)
47 {
48   while (JobCount_Unsafe() <= 0)
49   {
50       ...
51       //Improvement: work stealing
52     Sleep(1);
53   }
54
55   /*
56   Important: First pop ready worker, then Job!
57   */
58
59   if (m_readyFibers.size() > 0)
60   {
61     FiberWorker* fiber = m_readyFibers.front();
62     Job* job = nullptr;
63     if (PopJob(&job))
64     {
65       fiber->SetJob(job);
66       m_readyFibers.pop();
67       m_currentExecutingFiber = fiber;
68       SwitchToFiber(fiber->m_FiberPointer);
69     }
70   }
71
72 }
```

Listing 11: FiberSchedulerThread.cpp (see source files of the author)

5.3 Challenges of the Implementation

Writing a multithreaded application, in terms of kernel-level threads, is a challenge. Read and write access to shared resources need to be protected, data races and other concurrency issues may occur (see section 2.4 *Concurrency and Multithreading Issues*). Furthermore, when writing a user-level scheduler, user-level thread management within one kernel-level thread and possibly over multiple kernel-level threads needs to be implemented. The developer needs to consider problems which he or she never dealt with before.

First of all the implementation consists of multiple job queues. The access to those queues need to be protected due to the fact multiple threads push and pop jobs on that queues. In order to safely read or write data of a queue the executing thread has to enter a critical section (see section 2.4 *Concurrency and Multithreading Issues*). A straight-forward approach would wrap each access to a job queue with a critical section. However, that is not always necessary. An example is the waiting loop for jobs in listing 9 and listing 11. The read access to the job queue is not protected. I.e. the job count may not be correct. However, that does not matter since the *PopJob* function has locked access to the queue and the correct execution of the logic

```
38 void FiberWorker::Proc(void * p)
39 {
40   while (true)
41   {
42     if (m_CurrentJob && m_hasWork)
43     {
44       m_CurrentJob->m_Callback(m_CurrentJob);
45       JobManager::GetInstance()->FinishJob(m_CurrentJob);
46     }
47     m_hasWork = false;
48     m_SchedulerThread->PushReadyFiber(this);
49     SwitchToFiber(m_SchedulerThread->m_schedulerFiberPointer);
50   }
51 }
```

Listing 12: FiberWorker.cpp (see source files of the author)

depends on the *PopJob* function. The waiting loop serves the purpose to execute the locked *PopJob* function as less frequent as possible to spare expensive instructions. Therefore, the programmer always should consider whether a critical section is necessary or not.

Another challenge is the understanding of how kernel-level threads and user-level threads work together. First of all kernel-level threads do run concurrently on multiple cores. On Windows they use preemptive scheduling (see section 3.1.1 *Implementation of Kernel-Level Threads in Windows*). Within one kernel-level thread multiple user-level threads can be scheduled. Fibers (see section 3.1.2 *Implementation of User-Level Threads in Windows*) use cooperative scheduling. A specific fiber has to be switched to by the application.
Switching fibers within the same kernel-level thread that created them can be done by simply calling the *SwitchToFiber* (see section 3.1.2 *Implementation of User-Level Threads in Windows*) function. However, switching fibers among multiple kernel-level threads is a challenge. Assume the scenario in which a fiber, running on kernel-thread *A*, tries to switch to a fiber currently executed by kernel-thread *B*. That can cause problems (Microsoft Corporation 2017)[27]. Therefore, a strategy needs to be implemented that handles such situations.

5.4 Measurements and Results

Important Note

The implemented job systems are applied to a pseudo game. Hence, the implemented systems of the pseudo game define and push job with specific characteristics. Implementing a full functional game and a suitable job system goes beyond the scope of this thesis. Concrete job design and their dependencies would have influence on the implementation of the job system. Therefore, the measurements and results are limited to the characteristics of the jobs. The focus of the performed measurements is job throughput.

27. https://msdn.microsoft.com/de-de/library/windows/desktop/ms686350(v=vs.85).aspx Accessed June 18, 2017

Job Design

Six pseudo systems were implemented for performing the measurements, each defining and creating jobs with specific characteristics:

- SystemA
- SystemB
- SystemC
- ExpensiveSystemX
- ExpensiveSystemY
- ExpensiveSystemZ

System A to C represent systems which perform CPU or memory related work. **SystemA** creates jobs which perform numerous square root calculations. **SystemB** creates a variable of type *std::string* and appends strings to that variable. **SystemC** represents a very small job. Only few calculations are performed.
The expensive systems X to Z represent systems which perform expensive or potentially blocking system calls (see section 2.2 *Basics of Operating Systems*). **ExpensiveSystemX** opens a file, reads some data and closes the file. **ExpensiveSystemY** creates a directory and removes it again. **ExpensiveSystemZ** opens a file and closes it immediately.

There is another category of jobs that can be implemented with the current implementation of the JobManager (see section 5.1 *Job Management*): jobs which depend on each other. For this measurement no system with job dependencies is implemented due to the fact that the *WaitForJob* function of the JobManager would stall the primary thread. This measurement focuses on job throughput.

Measurement Tools and Preparation

The measurements take place on the notebook *Acer Aspire E 17 E5-773G-5424* with the following specifications: *Intel Core i5-6200U* 2x 2.3 GHz Turbo Boost up to 2.8 GHz (4 hardware threads) and 8 GB DDR3 PC3-12800 (800MHz).

The tools used for measurements were *Sysinternals Process Explorer v16.21* and *Intel VTune Amplifier XE 2017 Update 2*.

The Process Explorer is used to measure the following data:

- Context switches (see section 3 *Kernel-Level Threads and User-Level Threads*)
- CPU cycles
- I/O reads

Intel VTune Amplifier performs the following measurements:

- Total execution time
- Time for job creation and processing

- Hotspot detection

For each implementation, pure kernel-level approach and hybrid approach, ten measurements are performed. During the measurements all foreground applications are closed and as less background processes as possible are executed. Since there is a variability in processor utilization minimum and maximum values of results are noted. There is important information and configuration which impacts on the measurements. The information and configuration partially depends on the implementation.

General information and configuration:

- Program architecture: 32bit
- Release build
- One primary kernel-level thread is creating and pushing jobs in the game loop
- The number of jobs is 600000. 100000 per system.

Pure kernel-level approach specific:

- There are three kernel-level threads processing jobs

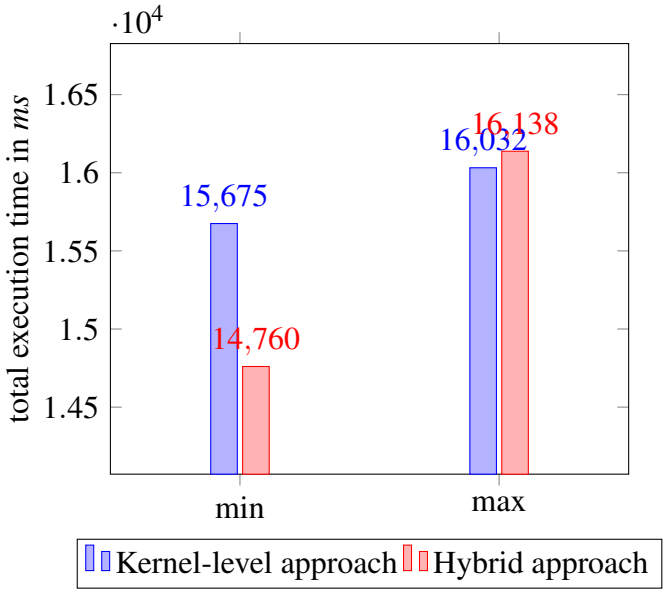
Hybrid approach specific:

- There are three kernel-level threads scheduling and executing user-level threads
- Per kernel-level thread there are 64 fibers processing jobs

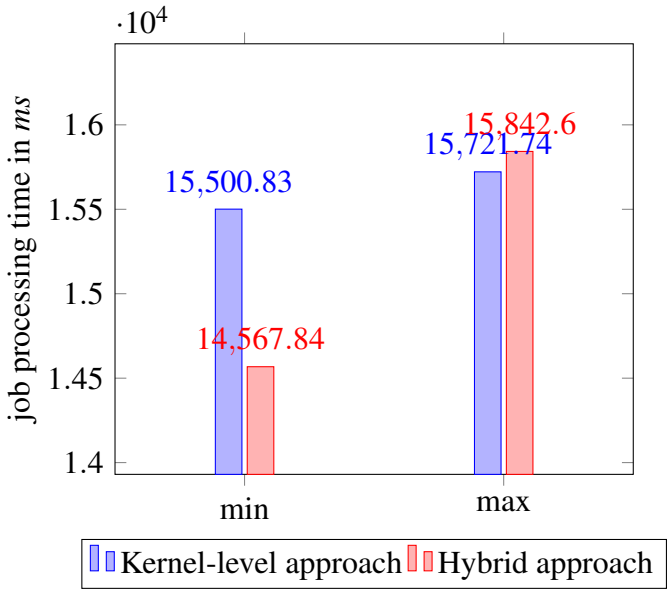
Results

Note: The concrete measurement data can be downloaded following the link in the appendix.

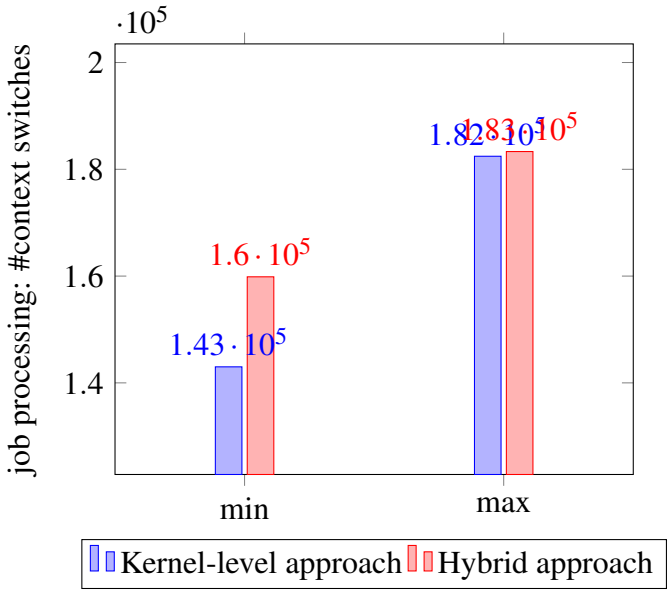
The measurements were performed and the following results were recorded:



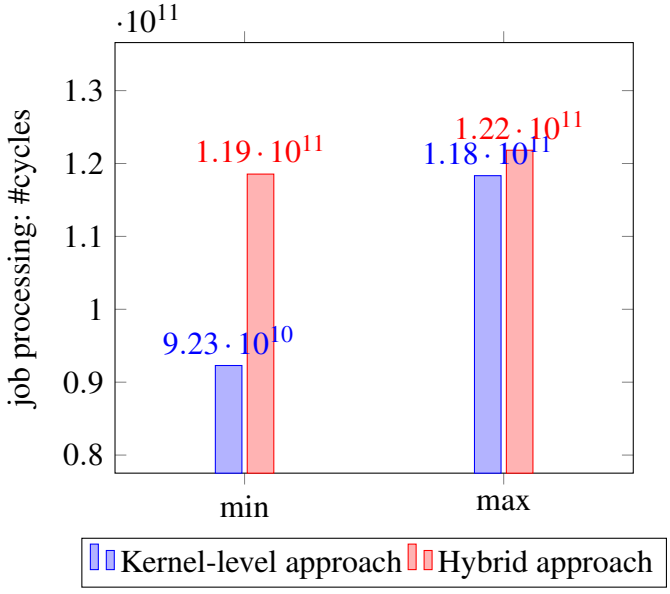
The maximum total execution time of the implementations differs about 100ms. The minimum total execution time of the hybrid approach is circa one second faster. However, there is more range between the results of the hybrid approach. The kernel-level approach has less variability between the minimum and maximum results.



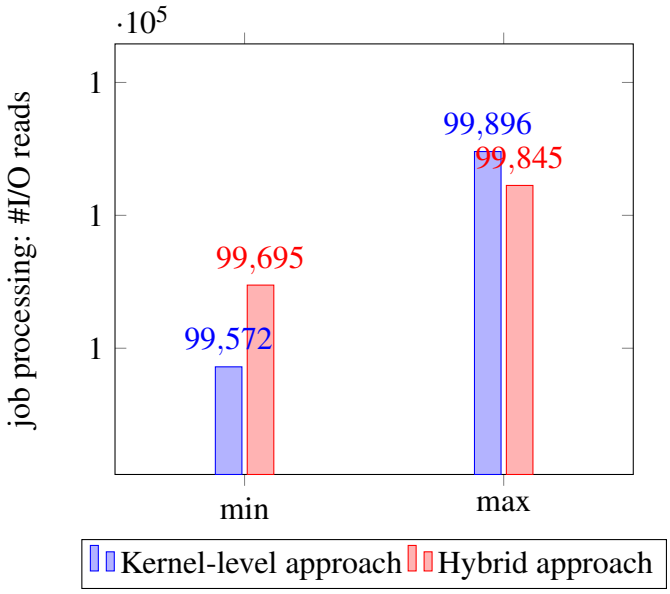
The results for the time for job creation and processing is similar to the results of the total execution time of the implementations.



The number of context switches during the job creation and processing presents a clear result. The kernel-level approach measurements show that the minimum and maximum results is less than the corresponding results of the hybrid approach. Thus, there is an overhead in the hybrid approach implementation.



The number of cycles during the job creation and processing is similar to the results of the context switch measurements. The kernel-level approach measurements show that the minimum and maximum results is less than the corresponding results of the hybrid approach. Hence, there is an overhead in the hybrid approach implementation.



The ExpensiveSystemX performs I/O reads. This measurement shows that the number of I/O reads varies around 300 reads.

Unordered top hotspots of the pure kernel-level approach:

- close
- mkdir
- open
- Sleep

In terms of the kernel-level approach the top hotspots include **close**, **mkdir** and **open**. These calls are system calls and potentially blocking (see section 2.2 *Basics of Operating Systems*). The **Sleep** function call is also identified as a hotspot. The current implementation calls the Sleep function when it e.g. waits for jobs. Section 5.5 *Possible/Further Improvements* examines how that can be improved.

Unordered top hotspots of the hybrid approach:

- open
- Sleep
- std::string::append
- SwitchToFiber

In terms of the hybrid approach the top hotspots include **open** and **Sleep**. These calls are also hotspots of the kernel-level approach and already explained. Another hotspot is the **std::string::append** function. The measurements of the number of context switches and cycles show that the hybrid approach has an overhead. Fibers have to be explicitly switch to. Intel VTune Amplifier identified the **SwitchToFiber** function as a hotspot of the hybrid approach. That would also explain the high numbers of context switches and cycles.

Seeing all measurements in context there is an overhead in the hybrid implementation due to switching fibers. Top hotspots are potentially blocking system calls and Sleep function calls due to the current implementation.

5.5 Possible/Further Improvements

Improvements: Algorithms and Data Structures

The current implementation creates `N-1` kernel-level threads for processing jobs. The primary thread mainly adds jobs to the queue. When working with frames (see section 4.1 *General Approach and Flow* and section 4.2 *Job System Improvements*) the primary thread could also process jobs when it completed adding jobs for the current frame. That would support the processing of jobs.

Some jobs depend on each other (see section 5.1 *Job Management*). Therefore, the *JobManager* provides the *WaitForJob* function. The current implementation of this function contains a loop that checks whether the work counter reaches the target value. Thus, the thread executing this function can be considered idle. Instead of just waiting for the work counter to become the desired value, available jobs can be processed (Reinalter 2017)[28]. The waiting time can be used efficiently.

All jobs are pushed the scheduler (see section 5.2 *Scheduler and Worker Implementations*). The scheduler then identifies the worker job queue with the least jobs and pushes a job to it. Workers pop jobs from their queues to process them. Hence, the access to the worker job queue must be locked. One approach to improve this issue is *work stealing* in combination with job queue management (see section 4.2 *Job System Improvements*): Each thread has its own job queue. Worker threads steal jobs from the primary thread's queue and push new jobs to their own queue to avoid locking mechanisms on their worker queue. Work stealing from other worker job queues requires a lock. However, work stealing occurs less frequently than pushing jobs.

Using lock-free job queues (see section 4.2 *Job System Improvements*) would further improve performance since no locks would be used.

Improvement: Kernel-Level and User-Level Threads

The plain utilization of user-level threads in job systems does not have any benefits (see section 5.4 *Measurements and Results*). On the contrary, it generates a slight overhead. The required

28. `https://blog.molecular-matters.com/2015/08/24/job-system-2-0-lock-free-work-stealing-part-1-basics/` Accessed June 16, 2017

memory for the user-level thread data structures is higher in contrast to a pure kernel-level thread approach with $N-1$ worker threads. Moreover, there is a user-level thread switching overhead that does not occur in the kernel-level thread approach. In order to justify the utilization of user-level threads in job systems an advanced approach must be implemented.

There are advanced approaches of how kernel-level user-level threads can be used in order to achieve a better utilization of processors. Assume there are N hardware threads available on a system. Assume the hybrid approach of this thesis is processing jobs. There is one primary kernel-level thread and $N-1$ kernel-level worker threads with X user-level threads per kernel-level thread. Assume all kernel-level worker threads execute a user-level thread which processes a job with a blocking system call (see section 2.2 *Basics of Operating Systems*). At this moment the entire job system halts. There is no progress. In order to overcome this situation the job system could provide more kernel-level threads than hardware threads are available (Stallings 2015, 193-196). These additional *assistant kernel-level threads* would be responsible for processing blocking system calls and other expensive operations. Since user-level threads use cooperative scheduling (see section 3.1.2 *Implementation of User-Level Threads in Windows*), they can be moved to another kernel-level threads before execution expensive operations. After the user-level threads finished executing the expensive operation, they are moved back to the kernel-level worker threads. This approach would justify the utilization of user-level threads in job systems.

6 Conclusion

This thesis examined how kernel-level threads and user-level threads can be applied to a job system. First the basics of computer hardware components and operating systems were stated to provide a foundation to understand how multithreading works. Next kernel-level threads and user-level threads were introduced: how they work and their characteristics. Job systems were examined and how they are used in game engines. Both topics, threads and job systems, were combined and implemented. Two implementations were examined, stated and compared in terms of time and memory.

Current games and game engines run on operating systems like Windows. Operating systems provide an interface face to the hardware components of a computer like a processor, main memory and I/O devices. The evolution of processors lead to multi-core processors. A program in execution is called process. A process consist of various elements. An example is the identifier and the state. Furthermore, processes consists of one or more threads. Each thread in a process also has an identifier and a state and can run on different processors. Thus, a program can run concurrently. There are some challenges with multithreaded programs: locking critical sections and objects, data races, dead locks etc.

Due to the fact that games push hardware to its limits, the computer respectively the processor cores needs to be utilized as much as possible. Therefore, multithreading a game engine is a way to achieve more performance. There are two basic types of threads: user-level threads and kernel-level threads. Kernel-level threads are visible to the operating system. The OS distributes

them on processors for execution. User-level threads are implemented in the application respectively a threads library. The OS does not know about them and can not distribute them across multiple processors. Windows provides a implementation of user-level threads called fibers.

There are multiple ways how game engines can be parallelized. A straight forward approach is to assign one kernel-level thread per system. Another approach whose goal it is to utilize hardware threads as much as possible is the job system. Besides the primary thread there are kernel-level threads that exclusively process jobs. The systems of the game engine mainly create jobs and push them to job queues. Job systems can be improved in multiple ways.

Research Question and Implementation

The research question of this thesis is how kernel-level threads and user-level thread can be applied to job systems and which results, time and processors utilization concerning, are discovered.

In order to examine the research question of this thesis two approaches of job systems utilizing threads were implemented:

- Pure kernel-level thread approach
- Hybrid approach (kernel-level threads and user-level threads)

Job management is one major part of the job system implementation. A job consists of a function pointer, data the job is working with and job dependency related data. Jobs are administrated by a job manager. The job manager is responsible for the creation and completion of jobs. Moreover, the job manager provides functions to establish dependencies and relations between jobs.

Besides the job management, schedulers and workers were implemented. The pure kernel-level thread implementation consists of two elements: the KernelThreadScheduler and the KernelThreadWorker. The KernelThreadScheduler manages KernelThreadWorkers and assigns jobs to them. Each KernelThreadWorker manages a kernel-level thread which exclusively processes jobs.
The hybrid implementation consists of three elements: the FiberScheduler, FiberSchedulerThread and the FiberWorker. The FiberScheduler manages FiberSchedulerThreads and assigns jobs to them. The FiberSchedulerThread creates a kernel-level thread which is used to schedule and execute fibers. Moreover, the FiberSchedulerThread manages an array of FiberWorkers. A FiberWorker represents a fiber and job processing functionality.

Different measurements were performed. The total execution time and the time for creating and processing jobs was measured. Furthermore, the number of context switches and cycles was recorded. Top hotspots were identified. System calls, e.g. I/O reads, are expensive. Sleep function calls of the current implementation occur frequently when for example a worker currently has no jobs in its queue. There also is a clear overhead in the hybrid implementation. Switching between fibers causes more work for the processor. Without further improvements the usage of user-level threads has no benefits.

Outlook: Improvements and Related Topics

In order to further improve the implementations examined by this thesis the section 4.2 *Job System Improvements* and section 5.5 *Possible/Further Improvements* provide initial information. Work stealing, lock-free programming and the implementation of overlapping frames are three possibilities.
Concerning user-level threads there are also improvements that can be implemented. One possible improvement that would justify the usage of user-level threads in job systems is the usage of more kernel-level threads than hardware threads available in the system. These additional kernel-level threads would process blocking system calls and expensive operations. Section 5.5 *Possible/Further Improvements* examines that approach in detail.
In order to further develop and research the topics and implementations of this thesis both approaches could be equipped with additional kernel-level threads for processing expensive or blocking operations. The concrete handling of moving such operations to these additional kernel-level has to be investigated.

There exist further topics related to this thesis. Intel's TBB library was only superficially investigated by this thesis. Intel TBB provides more features than those mentioned in this thesis (Intel Corporation 2017b)[29]. Another implementation that supports developers multithreading their applications is the Grand Central Dispatch (Stallings 2015, 107-108). The Grand Central Dispatch is a thread pool (see section 3 *Kernel-Level Threads and User-Level Threads*) mechanism implemented in Max OS X and iPhone iOS that executes tasks on threads concurrently. Furthermore, it provides extra threads for blocking on I/O.
Scheduling algorithms were barely mentioned. There are scheduling algorithms for various purposes (Stallings 2015, 433-435). Priority queuing is also often part of scheduling algorithms. These algorithms can for example focus on turnaround time, deadlines, throughput or fairness. An excerpt of scheduling policies is: FCFS, Round Robin and Highest Response Ratio Next. Scheduling algorithms is a topic that can further be researched and applied to the scheduler of a job system.
Game engines utilize multi-core processors with different strategies (see section 3.2 *Game Engines and Parallelization*). There are game engines whose source code is available online. Epic Games provides access to the Unreal Engine 4 source code via GitHub: `https://www.unrealengine.com/ue4-on-github`. After the access to Epic Games GitHub repository is unlocked the source code can be found here: `https://github.com/EpicGames/UnrealEngine`. The Unreal Engine makes use of multithreading[30]. Moreover, the Unreal Engine makes use of tasks and task graphs[31]. One example that is implemented is with tasks is a parallel for loop[32]. The source code of this engine can be downloaded and the used task system can be modified and improved for further research.

29. `https://software.intel.com/en-us/node/506045` Accessed June 16, 2017
30. `https://github.com/EpicGames/UnrealEngine/blob/master/Engine/Source/Runtime/Core/Public/HAL/ThreadManager.h` Accessed June 14, 2017
31. `https://github.com/EpicGames/UnrealEngine/blob/master/Engine/Source/Runtime/Core/Public/Async/TaskGraphInterfaces.h` Accessed June 14, 2017
32. `https://github.com/EpicGames/UnrealEngine/blob/master/Engine/Source/Runtime/Core/Public/Async/ParallelFor.h` Accessed June 14, 2017

To sum up, kernel-level and user-level threads can be utilized in job systems to enhance the processor utilization of a game engine. The implementation of a high-performance job system needs lots of effort and many improvements are possible. The usage of kernel-level threads, without over-subscription, utilizes the multi-core processors very well. In order to justify the utilization of user-level threads in job systems further improvements has to be implemented and more research has to be done.

Acronyms

ALU Arithmetic logic unit

API Application programming interface

CPU Central processing unit

FCFS First-come-first-served

FPS First-person shooter

FIFO First-in-first-out

GUI Graphical user interface

LIFO Last-in-first-out

RTS Real-time strategy

SMP Symmetric multiprocessor

OS Operating system

TBB Threading Building Blocks

UMS User-mode scheduling

List of Figures

Listings

List of Tables

References

AMD Corporation. 2005. *MULTI-CORE PROCESSORS - THE NEXT EVOLUTION IN COMPUTING.* Technical report. AMD Corporation. http://static.highspeedbackbone.net/pdf/AMD_Athlon_Multi-Core_Processor_Article.pdf.

———. 2017. *AMD Unveils Expanding Set of High-Performance Products and Technologies Propelling Next Phase of Growth.* Accessed May 30, 2017. http://www.amd.com/en-us/press-releases/Pages/amd-unveils-expanding-2017may16.aspx.

Andersson, Johan. 2009. "Parallel Futures of a Game Engine." In *Intel Dynamic Execution Environment Symposium 2009.* Accessed May 15, 2017. https://www.slideshare.net/repii/parallel-futures-of-a-game-engine-2478448.

Andrews, Jeff. 2015. *Designing the Framework of a Parallel Game Engine.* Technical report. Intel Corporation. Accessed May 12, 2017. https://software.intel.com/en-us/articles/designing-the-framework-of-a-parallel-game-engine.

Granatir, Orion R., and Omar A. Rodriguez. 2010. "Don't Dread Threads." In *Game Developers Conference (GDC) 2010.* Accessed May 15, 2017. http://www.gdcvault.com/play/1012189/Don-t-Dread.

Gregory, Jason. 2014. *Game engine architecture.* Second edition. Boca Raton: CRC Press. ISBN: 1466560010.

Gyrling, Christian. 2015. "Parallelizing the Naughty Dog Engine Using Fibers." In *Game Developers Conference (GDC) 2015.* Accessed May 15, 2017. http://www.gdcvault.com/play/1022186/Parallelizing-the-Naughty-Dog-Engine.

Hodgman, Brooke. 2016. "Parallel Game Engine Design." In *Game Connect Asia Pacific (GCAP) 2016.* Accessed May 14, 2017. https://www.youtube.com/watch?v=JpmK0zu4Mts.

Intel Corporation. 2017a. *Intel® Core™ i9-7980XE Extreme Edition Processor.* Accessed May 30, 2017. http://www.intel.com/content/www/us/en/products/processors/core/x-series/i9-7980xe.html.

———. 2017b. *Intel® Threading Building Blocks Documentation.* Accessed June 16, 2017. https://software.intel.com/en-us/tbb-documentation.

Lake, Adam. 2011. *Game programming gems 8.* Boston, Mass.: Course Technology. ISBN: 9781584507024.

Lengyel, Eric. 2010. *Game engine gems.* Sudbury, MA: Jones and Bartlett Publishers. ISBN: 9780763778880.

Microsoft Corporation. 2017. *MSDN: Learn to Develop with Microsoft Developer Network.* Accessed June 16, 2017. https://msdn.microsoft.com/.

Minadakis, Yannis. 2011a. "Efficient Scaling in a Task-Based Game Engine." In *Game Developers Conference (GDC) 2011*. Accessed May 15, 2017. http://gdcvault.com/play/1014644/Efficient-Scaling-in-a-Task.

———. 2011b. *Using Tasking to Scale Game Engine Systems*. Technical report. Intel Corporation. Accessed May 15, 2017. https://software.intel.com/en-us/articles/using-tasking-to-scale-game-engine-systems/.

Muffat-Méridol, Jérôme. 2009a. *Do-it-yourself Game Task Scheduling — Intel® Software*. Technical report. Intel Corporation. Accessed June 16, 2017. https://software.intel.com/en-us/articles/do-it-yourself-game-task-scheduling/.

———. 2009b. *Nulstein source code*. Intel Corporation. Accessed June 16, 2017. https://software.intel.com/sites/default/files/m/d/4/1/d/8/nulstein.zip.

Ramanathan, R. M. 2006. *Intel®Multi-Core Processors: Making the Move to Quad-Core and Beyond*. Technical report. Intel Corporation.

Reinalter, Stefan. 2017. *Molecular Musings: Development blog of the Molecule Engine*. Accessed June 16, 2017. https://blog.molecular-matters.com/.

Russinovich, Mark E., David A. Solomon, and Alex Ionescu. 2012. *Windows Internals Part 1*. 6th ed. Redmond, Wash.: Microsoft Press. ISBN: 0735671303.

Sanglard, Fabien. 2013. *Doom3 BFG Source Code Review: Multi-threading*. Accessed June 16, 2017. http://fabiensanglard.net/doom3_bfg/threading.php.

Stallings, William. 2015. *Operating systems: Internals and design principles*. Global edition, eighth edition. Always learning. Boston et al.: Pearson. ISBN: 978-1-292-06135-1.

Tanenbaum, Andrew S., and Herbert Bos. 2015. *Modern operating systems*. Global edition, fourth edition. Boston: Pearson. ISBN: 013359162X.

Appendix

Link to the author's **source files** and concrete **measurement** data:
`https://github.com/animet/BA2_JobSystems_Threads`

Archived Websites and Web Resources

Note: Only web sites which allow crawlers could be archived. Furthermore, links that refer to video stream web site are not archived because the video file cannot be archived.

Footnotes

`https://web.archive.org/web/20170617142230/https://www.unrealengine.com/what-is-unreal-engine-4`

`https://web.archive.org/web/20170617142346/https://www.ea.com/frostbite/engine`

`https://web.archive.org/web/20170617142421/https://unity3d.com/de/unity`

`https://web.archive.org/web/20170617142505/https://unity3d.com/de/legal/terms-of-service/software`

`https://web.archive.org/web/20170617142538/https://www.unrealengine.com/eula`

`https://web.archive.org/web/20170618070311/http://en.cppreference.com/w/cpp/thread/mutex`

`https://web.archive.org/web/20170617142636/https://msdn.microsoft.com/de-de/library/windows/desktop/ms681917(v=vs.85).aspx`

`https://web.archive.org/web/20170617142822/https://msdn.microsoft.com/de-de/library/windows/desktop/ms684847(v=vs.85).aspx`

`https://web.archive.org/web/20170617142922/https://msdn.microsoft.com/en-us/library/windows/desktop/ms682661(v=vs.85).aspx`

`https://web.archive.org/web/20170617143002/https://msdn.microsoft.com/en-us/library/windows/desktop/ms686919(v=vs.85).aspx`

`https://web.archive.org/web/20170617143345/https://msdn.microsoft.com/en-us/library/windows/desktop/ee418650(v=vs.85).aspx`

`https://web.archive.org/web/20170618171303/https://msdn.microsoft.com/de-de/library/windows/desktop/ms686350(v=vs.85).aspx`

`https://web.archive.org/web/20170617142722/https://software.intel.com/en-us/node/506127`

https://web.archive.org/web/20170617143041/https://software.intel.com/en-us/node/506100

https://web.archive.org/web/20170617143248/https://software.intel.com/en-us/node/506299

https://web.archive.org/web/20170618094540/https://software.intel.com/en-us/node/506045

https://web.archive.org/web/20170617143155/https://blog.molecular-matters.com/2015/08/24/job-system-2-0-lock-free-work-stealing-part-1-basics/

https://web.archive.org/web/20170618091218/https://blog.molecular-matters.com/2015/09/08/job-system-2-0-lock-free-work-stealing-part-2-a-specialized-allocator/

https://web.archive.org/web/20170618103845/https://blog.molecular-matters.com/2015/09/25/job-system-2-0-lock-free-work-stealing-part-3-going-lock-free/

Info: There are no links of the GitHub repository of Epic Game's Unreal Engine archived due to the fact an unlocked access is required.

References

https://web.archive.org/web/20170617143807/http://static.highspeedbackbone.net/pdf/AMD_Athlon_Multi-Core_Processor_Article.pdf

https://web.archive.org/web/20170617143845/http://www.amd.com/en-us/press-releases/Pages/amd-unveils-expanding-2017may16.aspx

https://web.archive.org/web/20170617144402/https://www.slideshare.net/repii/parallel-futures-of-a-game-engine-2478448

https://web.archive.org/web/20170617145124/https://software.intel.com/en-us/articles/designing-the-framework-of-a-parallel-game-engine

https://web.archive.org/web/20170617144855/https://software.intel.com/en-us/tbb-documentation

https://web.archive.org/web/20170617145028/https://msdn.microsoft.com/de-de/default.aspx

https://web.archive.org/web/20170617145232/https://software.intel.com/en-us/articles/using-tasking-to-scale-game-engine-systems

https://web.archive.org/web/20170617145521/https://software.intel.com/en-us/articles/do-it-yourself-game-task-scheduling

https://web.archive.org/web/20170617145716/https://blog.molecular-matters.com/

https://web.archive.org/web/20170617145807/http://fabiensanglard.net/doom3_bfg/threading.php