

System Software & Multi-User Applications Engineering
M5X01055

Research Report

completed by

Eben Stewart
P115266

for Lecturer

Dr. Tim Bashford

Due

22 June 2018

Contents

Introduction	2
The Operating System.....	2
Multi-programming system.....	2
Recap of OS Concepts.....	3
Processes	5
Threads	7
Inter-Process Communication	8
Summary	9
References	11

Introduction

The Operating System

When a user starts a modern computer, he finds at his disposal numerous devices such as a keyboard, mouse, monitor, printer, as well as other resources necessary to the functioning of the system, such as the computer's memory and hard disk drives etcetera. To print, he presses a software button and information is sent to the printer which whirls into action and prints as instructed. If he's in an office, he may have to choose from two or more printers in a dropdown list on the computer screen. Whichever is chosen, the same print button will start the job rolling.

But behind the scenes, the different printers on offer almost certainly have very different software to control the minutiae of their operation, the movement of their drive belts, the laying down of controlled amounts of the right combination of coloured ink, etc. If the user had to program all of that in for every document printed, printing press typesetters would still be in business. The management of all the components, input/output devices and other resources of computers is handled by the Operating System (OS). It abstracts away the complexity of what actually happens, to make these things easily understandable and operable for the user, presenting him with only the necessary information to get things working as he wishes. Thus, OSs have sections which are protected from the user which oversee the actual resource management and other low-level operations, and sections for managing the interface with the user, which the user is free to access. These are respectively called kernel space and user space. When the OS does work in one or the other of them its in kernel mode or user mode [1, p. 3-6].

Multi-programming system

Much expense goes into designing and manufacturing the Central Processing Unit (CPU) of computers to operate at the highest possible speed. For this speed to be utilised efficiently, the systems which support the CPU must be designed and built to keep the CPU working at as close to its capacity as possible, or as makes business sense.

At its most basic, what the CPU does is fetch each program instruction from memory, decode it to find out what the program is telling the computer to do, execute it and repeat the cycle for the next instruction and the next until the program is finished. Decoding involves finding out what type of instruction it is and its operands. Instruction sets are architecture specific [1] [2].

CPU actions are performed at set intervals called a clock cycle, which are measured in multiples of hertz. This is controlled by the processor clock [3]. Modern desktop and laptop CPUs operate in the range of multiple gigahertz or billions of cycles per second [4] [5] [6].

Instructions typically involve action upon data in memory and the results are again stored in memory. Any delay in accessing data result in wasted CPU cycles. Thus, the working memory that the CPU accesses most frequently, that is most closely integrated into its operations, must work at as close to the speed of the CPU itself as possible [1, p.24] [7].

Memory fast enough to keep up with the CPU is extremely expensive (and must be located very close to or within the CPU to minimise noise due to an effect known as parasitic capacitance [8]) so the smallest amounts possible are used in strategic ways. The fastest bits, inside the processor itself, are the registers. The next fastest are the caches. Usually there are up to three of these. In descending order of speed, expense

and proximity to the CPU, but in ascending order of size, these are termed level 1, 2 and 3 caches [7].

The bulk of the RAM in the computer operates at much slower speeds. It is much cheaper to manufacture and so can be present in large enough quantity to hold all or most of the data that the computer is working on at any one time, across all the various programs that a typical user might have open at once.

The bulk storage, hard disks or tape or to a lesser extent, SSDs, used by the computer is orders of magnitude slower still. Even more significant wasters of CPU time are waiting for user input and for network connections. So, if the CPU completes only one job at a time, there will be times when it is idle waiting for data to be fetched.

Multiprogramming

A solution arrived at early in the history of computing is to have the CPU working to complete more than one job at a time. Although at any one time, the CPU is still only occupied with one job, it can switch to another whenever the running job makes it wait. This is called multi-programming. Such switching is called context switching [1, p.159]. A computer system set up to do this can be called a multi-programming system [1, p.11]. The logic here applies equally to a single CPU system and to multi core/ multiprocessor systems. For any CPU, multiprogramming will use it more efficiently.

To make this work, each separate job must have its own dedicated portion of main memory, partitioned so that other jobs can't interfere with its stored data. The CPU works on each job for a short while, if the job in question is ready to run. It doesn't necessarily switch because the job is waiting for input. Even if the running job doesn't need to wait, the CPU (except set up to run batches of fully automated jobs) switches jobs rapidly, often after tens or hundreds of milliseconds [1, p.86], to give the impression, on a human timescale, of the jobs running simultaneously. CPU time and job running order are doled out according to a scheduling algorithm, of which there are various [9] [10].

Multiprocessing

Modern computers of the complexity of a smartphone and higher contain multicore processors that actually do run more than one process concurrently. The problem then becomes more complicated as there are two or more compute cores, each with their own jobs running that must be kept operating as efficiently as possible though multiprogramming. Multiprocessing does not actually necessitate a qualitatively different approach. Both multiprogramming and multiprocessing produce concurrency, real or simulated [11, p.207].

Structure of This Paper

First, a basic recap of basic computer operations and components, explaining in more detail concepts introduced above as well as others to lay the groundwork for a detailed explanation of processes, their scheduling and how they communicate with each other, threads and how these differ from fully fledged processes. Issues of communication between these abstractions of separate entities will be discussed.

Recap of OS Concepts

Fetch Decode and Execute Cycle

Early computers carried out the steps of the fetch execute cycle sequentially for each instruction before starting on the next one. Modern computers, of course, range in capacity from very limited embedded systems through to supercomputers. All but the

most limited have systems in place to speed up the fetch execute cycle by having dedicated hardware for each step and working on more than one step of the cycle at a time. The most straightforward of these systems and most illustrative of the concept is called a pipeline.

This consists of a fetch unit, decode unit and execute unit with instructions passing from the first to the second to the last in a linear fashion. To process a series of instructions: $m, m+1, m+2 \dots m+n$, first m is loaded into the fetch unit. When it passes from the fetch to the decode unit, the next instruction $m+1$ is loaded into the fetch unit. When m passes to the execute unit, $m+1$ takes its place in the decode unit and $m+2$ enters the fetch unit, and so on in assembly line fashion [1, p. 21] [12].

There have been improvements in pipeline design, such as the various iterations of the RISC pipeline, which breaks the cycle up into five steps of equal length so that no step has to wait for another to finish [12]. There are also longer pipelines as well as more sophisticated setups like the superscalar CPU. This last has the facility to fetch, then decode two or more instructions at the same time. The results go into a common holding buffer and then are selected from it by specialised execute units, generally one each for integer arithmetic, floating point arithmetic and Boolean operations. Things get complicated in this design in that there is the potential for instructions to be executed out of order [1, p.22], but elaboration upon this is outside of the scope of this paper.

Registers and Caches

The general registers are where the data actually being worked on at that moment by the CPU are stored. There are also special registers to store things necessary to the execution of the job at hand. These include: the Program Counter, which holds the address of the next instruction to be executed, the Stack Pointer, which holds the address of the top of the system stack, the Program Status Word (PSW), which holds a variety of other information necessary to the immediate working of the CPU upon the job at hand [1, p. 21]. Next are the levels one, two and three caches. These are located outside the processor but in close proximity to it. Level one cache being fastest, smallest, closest and most expensive and level three being slowest cheapest etc of the three.

Processes

What is a Process?

Up until now the word 'job' has been used to mean what the CPU is working on. A more technical term for a program being run or which is all set up to be run is 'process' [1, p.85] [13]. A computer program is a set of instructions for the CPU to execute. It is a plan for some action or actions. When that plan is acted upon, is set in motion, a process is created. If it is successfully being executed by the CPU, it is in a running state. Processes have two facets, the management of resources and the thread of execution.

Reference has been made above to various resources needed by the CPU to be able to execute a series of instructions. When the scheduler switches the context from one process to another, the state of the process being switched out must be saved so that the thread of its execution can begin again where it left off once it regains the CPU as if it had never been interrupted.

Things like the contents of the registers including the program counter, stack pointer and PSW etc, the pointer to the memory reserved for it, called its 'address space', pointer to the program code, information about its priority in relation to other processes, its state, (more on this ahead), pointers to files it has open, its process ID and much more must be saved and exchanged for the state of the process that is about to be run. This state (of one process) is saved in a data structure called a Process Control Block [11]. The Operating system maintains an array of these called a Process table. It has an entry for the Process Control Block of every process existing at any one time [1, p. 94].

State

Stalling [11] describes a model where a process can be in one of five possible states. If it is 'new', it is freshly created but not yet part of the pool of executable processes. If it is in the 'exit' state, it's no longer part of the pool of executable processes because it has terminated naturally or aborted for whatever reason. A process that is neither new nor exiting can be either 'running', 'ready' or 'blocked'.

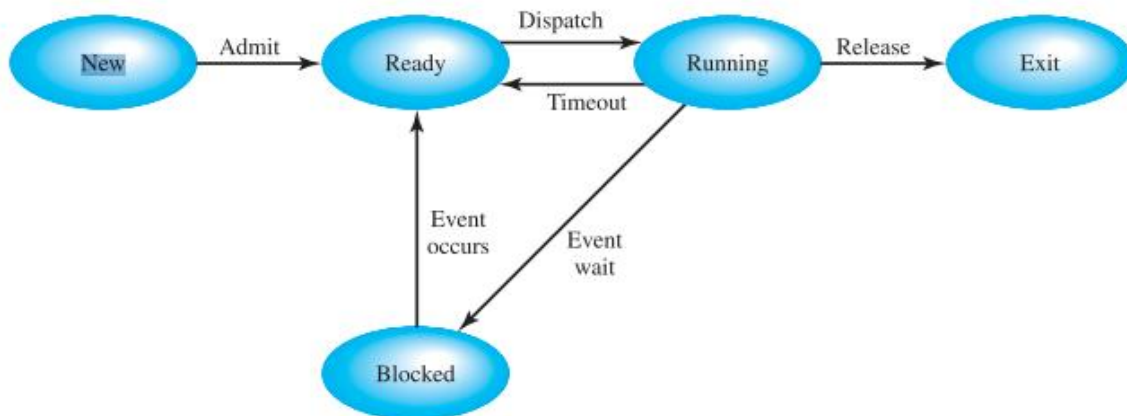


Figure 1 Stallings Five State Process Model [13, p.117]

A running process is just that; it is currently being executed by the CPU and it has all the resources it needs for that to go ahead. A ready process has what it needs to run, but the scheduler has deemed that it is not at that moment to have the CPU. A blocked process cannot run because it's waiting for something to happen. It might be waiting on input from a disk read, from a network connection or from user input. Until it enters the ready state it won't be picked to run by the scheduler [11].

Scheduling

The criteria for how and when to perform a context switch and to which of the available processes, are encoded in a scheduling algorithm. Scheduling algorithms assign processes priorities according to these criteria and decide what order to run them in. Different scheduling algorithms do this in different ways according to the needs of the system designers [14]. They are designed with certain goals in mind.

Scheduling algorithms are broadly categorised to be either interactive, real-time or batch. Interactive algorithms are preemptive. This means that the scheduler will interrupt processes for context switching. Batch algorithms are non-preemptive, which means that once a process is started it will run uninterrupted until it is done or blocks for input. Real-time algorithms are used in systems designed for a specific purpose where accurate timekeeping is essential such as factory machine or aviation control systems [1, p. 164]. They are outside the scope of this paper.

There are goals that are common to all three categories, such as fairness, which means making sure that CPU time is fairly shared out between processes, enforcement of the policies the system is governed by (usually business rules) and the need to keep all parts of the system equally busy as far as possible. Other goals vary according to the category of the algorithm [1, p. 154].

Batch scheduling has the longest history, stemming from the days before the advent of multiprogramming and context switching. It involves scheduling jobs which do not need user input, to be run from beginning to end without context switching not caused by blocking. Decisions on the running order can still increase efficiency of CPU use somewhat. Lack of the overhead implicit in context switching helps here [14]. Batch scheduling is still relevant and in use for many business purposes such as payroll processing and such. Common goals for batch scheduling are to get as many jobs done as possible in the shortest amount of time, to keep the CPU as busy as possible and to keep turnaround time to a minimum [1, p. 154]. Because it doesn't involve context switching, it does not seem relevant to a discussion of multiprogramming.

Algorithms for batch scheduling include 'First Come First Serve' and 'Shortest Job First'. First Come First Serve is self-explanatory and simple to implement with a queue data structure to decide the running order. When a process blocks and is switched out, it goes to the back of the queue. It is unfortunately characterised by high average wait times. This is because with no pre-emption, processes which have long periods of uninterrupted computation and infrequent I/O do not often block and will greatly delay processes that do very short bursts of computation interleaved with lots of I/O calls [1, pp. 156-157].

Shortest Job First aims to run jobs in ascending order of their run time. This is good for getting as many jobs done as possible in the shortest amount of time. It is only doable in systems without user input where run time is computable beforehand, such as batch systems. 'Priority Based Scheduling' runs the most important processes first based upon assigned priority [14]. This is also an important business goal.

Interactive algorithms are for use in environments where there are users interacting with the software, thus introducing the high probability of jobs waiting for user input. Interactive algorithms use preemption to interleave work on two or more jobs at a time. As discussed above, this achieves more efficient use of the CPU, but it also keeps users from having the experience of waiting for other jobs to finish before the job they're interested in can run. Interactive systems have users waiting for results, so the perception of a quick response time is an important scheduling goal. Examples of

interactive algorithms are: Round Robin, and Priority Based Scheduling. There are many more.

Round Robin scheduling assigns each process a time interval and preempts it when time is up. This is very fair [14]. The idea of Priority Based Scheduling is the same for interactive scheduling as for Batch.

Threads

Processes can have more than one thread of execution. These are treated by the scheduler in a similar way to fully fledged processes. They have their own thread stacks to store their resources when context switched, but these resources are much lighter than those of processes. Consequently, they are much quicker and less expensive of system resources to create and destroy than processes.

Complex processes often benefit from having multiple threads to control the many activities they must carry out at the same time. The capacity to carry out various activities within a complex process concurrently can be greatly advantageous. For example, a word processor could have one thread performing periodic automatic saves, while another constantly adjusts the layout as bits are added to a document in the middle, or the entire document is made one font size larger and another constantly updates the screen [1, pp. 97-100].

Or a chat message server could create a dedicated thread to handle each message received or sent. Due to the possibility of handling a great many messages at the same time, with the high probability of blocking for network input, concurrency in message handling is desirable both for performance reasons and to simplify the coding of the structure of the program. In addition, since there is always the possibility of messages being lost by the network, decoupling their execution from that of the rest of the process by assigning its handling to a thread that lives only for the passing of that single message helps simplify the coding of error handling.

Threads share the address space of their parent process. This means they must manage shared resources to a much greater degree than processes. But since they are created to cooperate by the same programmer or programming team, the advantages of this can be leveraged and the potential pitfalls avoided. The creation and destruction of threads is much less resource intensive for the system and so they can be treated as more ephemeral than processes. While a process running an instance of Mozilla Firefox might last for hours or days, some threads might be created and destroyed many times during that time span. Other threads within that process may last as long as the process itself.

Threads can be created and managed in the kernel, in user space or a combination of both. There are various advantages to kernel threads and other advantages and pitfalls to user threads. Kernel threads are more expensive to create and destroy but can more easily be woken and put to sleep by system calls. User space threads are created and destroyed much more cheaply, but there is no ready mechanism for sleeping and waking them.

Posix threads, or pthreads, are a standardised, portable implementation of threads for which a highly convenient library exists. Pthread library procedures such as `pthread_create()`, `pthread_join()`, `pthread_exit`, `pthread_mutex_lock` (and `unlock`) etc. allow simple, versatile coding of threaded programs [15] [16].

Inter-Process Communication

Processes often need to communicate with each other. The output of one is frequently used by another. There are many ways that cooperation between processes can be advantageous and/or necessary. But because processes are not necessarily written by the same programmers or written to closely cooperate with each other, there are many ways that they can interfere with each other's proper functioning.

There are two main mechanisms by which processes can communicate: shared memory and message passing. Processes have their own address space which is protected from use by other processes, but it is not difficult to set up variables of primitive or complex types in regions of shared memory. These are then accessible by two or more processes and form the basis of shared information and cooperative working, if all is set up well [17] [18].

Atomicity

An atomic operation is one which is indivisible. In practice this means that it only takes one step of the CPU to perform or that it is protected from preemptive interruption by some mechanism or another [19] [11, p. 289].

Race Conditions

Data corruption can happen when non-atomic operations upon a shared variable or variables are interrupted by a preemptive context switch to another process which acts upon the same variables. If the second process changes the variable in ways that are unrecognised by the first process, it is called a race condition. To avoid this, it is necessary to find some way to stop more than one process at a time from reading or writing to shared data even when there is a context switch or switches.

This is a very complex subject and there is no space to go through all the possible solutions, flawed or workable. One or two examples will suffice to illustrate the sorts of problems there are and possible approaches to their solution.

A way forward is to define a section of a process' code where it acts upon shared data, as a critical section. Parts of its code where it acts only upon data in its own address space are then considered non-critical. It is then necessary to find a way to prevent other processes from accessing their critical sections when one process is in its critical section. This is called mutual exclusion and it involves locking access to shared resources in certain circumstances [11, p. 212].

One way to do this is to stop a processor's scheduler from performing a context switch when a process signals it has entered its critical section. If no other processes can run concurrently, none of them can enter their critical section and thus a race condition cannot happen. A huge problem with this is that it gives a user process effective control over what should be an autonomous kernel process, namely the scheduler. There is potentially huge disruption to the functioning of the OS. Even if this was not problematic in a single processor system, in a multiprocessing one it fails since the second (or third or eighth) processor's scheduler will not have its interrupts disabled [1, p.122].

There are various ways of approaching this problem and many of them are unworkable or problematic for various reasons. Any solution which relies on the behaviour of the CPU(s) or scheduler(s) will most likely face the same objections as the proposed solution above. So, the focus of the solution should be independent of these. Since any non-atomic operation could be unpredictably interrupted by one scheduler or another (if a multiprocessing system), a workable solution must rely upon an atomic operation to

somehow lock access to their critical regions by other processes once one process has grabbed the lock, so to speak.

Some solutions involve processes looping constantly to check if certain conditions have occurred. This is called busy waiting. If carried out for long periods or by many concurrent processes or threads, it is very wasteful of CPU cycles. When those conditions are met, for example if a lock has been released by a process (or thread (see below)) leaving its critical region, the loop is broken, and execution can continue as appropriate, perhaps by the process entering its own critical region now that the coast is clear. This is called a spin lock and it is very wasteful of CPU cycles. Better solutions involve sleeping and waking through processes signalling each other to do so in a cooperative manner [1, p.124].

Semaphores and their simpler variants, mutexes, are effective ways to enforce mutual exclusion and eliminate the risk of race conditions and associated problems [1, pp. 129] [20]. The pthread library includes a very good implementation of mutexes for use in interthread communication [15] [21]

But solutions can give rise to new problems. The ability to lock resources, if used injudiciously can give rise to a difficult problem called the deadlock .

This is a situation where two or more processes (less likely for threads because of greater planned cooperation) permanently block each other somewhat like automotive gridlock at a busy intersection. It can happen when each process is blocked awaiting a trigger from another process that itself is blocked, if the unblocking of the second process directly or indirectly depends on a trigger from the first process which it cannot give, of course, because it is blocked. The trigger might be a signal or the release of a shared resource. This mutual dependency failure can be extended to include any number of processes [11, p.263]. This is a huge subject which will not be delved into here except to name another synchronisation primitive called a Monitor which are a solution for deadlock in that through their use a thread can wait for a resource in its critical section while other threads still have access to that resource. This works through the use of condition variables which there is no time here to explain. A huge problem with monitors is that very few programming languages support them [1, pp. 137-144].

Message Passing

Messages can be passes back and forth by processes and threads using UNIX sockets. These are like network signals and sockets. There no shared access to common resources, thus avoiding all the problems that that entails. However, like network message passing there is always the possibility of transmission failure. This can be guarded against through the use of syn/ack pairs where receipt of a message is always acknowledged by a return message. Then success of transmission is at least known if not guaranteed and can be compensated for by a system involving retransmission of a message. Identification of a retransmitted message as such so that a duplicate can be ignored instead of misidentified as a new original message is necessary and can be implemented in a variety of ways, perhaps through the use of message IDs [1, p. 144].

Summary

The need to maximise the use of the CPU to achieve efficient, quick execution of programs and therefore highly useable computer systems, especially in the face of the astronomical cost of the highest speed memory. Much thought and expense has gone into strategies to juggle resources to that end. The abstraction of multiprogramming, yielding virtual concurrency of operations coupled with the more recent advent of

multiprocessing, yielding actual concurrency, enables the leverage of the incredible computing power of the modern CPUs present even in modest, consumer level desktop and laptop PCs.

Multiprogramming is enabled by the ubiquitous abstractions of processes which allow simplified management of resources and by their threads of execution. Complexity of program execution management is made even more manageable by the further abstraction of threads of execution to sort of mini-processes but with their own desirable characteristics, such as shared address space, lightweight, speedy creation and destruction and trusted, close cooperation that cannot be achieved in often competing processes where security must always be an issue.

However, this abstraction of a plethora of separate entities leads to issues of communication between them or Inter-Process Communication. This can be handled using either shared resources or message passing. Various issues of each, strategies for their resolution and implementation have been discussed superficially.

References

- [1] Andrew S. Tanenbaum and Herbert Bos, *Modern Operating Systems*, 4th ed. India: Pearson India Education Services Pvt. Ltd., 2017.
- [2] 'Fetch Execute Cycle | The CPU & The Fetch-Execute Cycle Definition', *Teach Computer Science*. [Online]. Available: <https://teachcomputerscience.com/fetch-execute-cycle/>. [Accessed: 17-Jun-2018].
- [3] 'Difference Between Oscillator and Crystal | Difference Between'. [Online]. Available: <http://www.differencebetween.net/technology/difference-between-oscillator-and-crystal4/>. [Accessed: 20-Jun-2018].
- [4] 'Intel® Core™ X-Series Processors'. [Online]. Available: <https://www.intel.co.uk/content/www/uk/en/products/processors/core/x-series.html>. [Accessed: 20-Jun-2018].
- [5] 'Welcome to AMD | Processors | Graphics and Technology | AMD'. [Online]. Available: <https://www.amd.com/en>. [Accessed: 20-Jun-2018].
- [6] 'Products Archive - Raspberry Pi'. [Online]. Available: <https://www.raspberrypi.org/products/>. [Accessed: 20-Jun-2018].
- [7] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*. Elsevier Science, 2011.
- [8] P. Larsson, 'Parasitic resistance in an MOS transistor used as on-chip decoupling capacitance', *IEEE Journal of Solid-State Circuits*, vol. 32, no. 4, pp. 574–576, Apr. 1997.
- [9] K. L. Krause, V. Y. Shen, and H. D. Schwetman, 'Analysis of Several Task-Scheduling Algorithms for a Model of Multiprogramming Computer Systems', *J. ACM*, vol. 22, no. 4, pp. 522–550, Oct. 1975.
- [10] C. L. Liu and J. W. Layland, 'Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment', *J. ACM*, vol. 20, no. 1, pp. 46–61, Jan. 1973.
- [11] W. Stallings, *Operating Systems: Internals and Design Principles*, 7th ed. Upper Saddle River, NJ, USA: Prentice Hall Press, 2011.
- [12] 'Pipelining'. [Online]. Available: <https://cs.stanford.edu/people/eroberts/courses/soco/projects/risc/pipelining/index.html>. [Accessed: 19-Jun-2018].
- [13] Rashid Bin Muhammad, 'Operating Systems Notes', *University of Kent*. [Online]. Available: <http://personal.kent.edu/~rmuhamma/OpSystems/Myos/cpuScheduling.htm>. [Accessed: 20-Jun-2018].
- [14] 'Operating System Scheduling algorithms', *tutorialspoint*. [Online]. Available: https://www.tutorialspoint.com/operating_system/os_process_scheduling_algorithms.htm. [Accessed: 17-Jun-2018].
- [15] Alfred Park, 'Multithreaded Programming (POSIX pthreads Tutorial)', *Multithreaded Programming*. [Online]. Available: <https://randu.org/tutorials/threads/>. [Accessed: 25-Apr-2018].
- [16] 'pthreads(7) - Linux manual page'. [Online]. Available: <http://man7.org/linux/man-pages/man7/pthreads.7.html>. [Accessed: 25-Apr-2018].
- [17] 'Interprocess communication (IPC)'. [Online]. Available: <https://www.w3schools.in/operating-system-tutorial/interprocess-communication-ipc/>. [Accessed: 22-Jun-2018].
- [18] 'Inter Process Communication - GeeksforGeeks'. [Online]. Available: <https://www.geeksforgeeks.org/inter-process-communication/>. [Accessed: 22-Jun-2018].
- [19] 'Standard ECMA-335'. [Online]. Available: <https://www.ecma-international.org/publications/standards/Ecma-335.htm>. [Accessed: 22-Jun-2018].
- [20] C. M. Davidson, 'A Note on Concurrent Programming Control', *IEEE Transactions on Software Engineering*, vol. SE-13, no. 7, pp. 865–866, Jul. 1987.
- [21] 'pthread_mutex_lock(3): lock/unlock mutex - Linux man page'. [Online]. Available: https://linux.die.net/man/3/pthread_mutex_lock. [Accessed: 29-Apr-2018].