

A pragmatic, historically oriented survey
on the universality of synchronization
primitives

University of Oulu

Dept. of Information Processing Science

Pro Gradu

Jouni Leppäjärvi (1251395)

11. May 2008

Summary

We seek to provide a theoretical foundation based on the existing research for evaluating the universalness of synchronization primitives. Ideally, such a criteria should be based on an underlying formal model in terms of which the problem and its solution could be posed. Unfortunately the existing research doesn't seem to provide for this. Still, from a pragmatic point of view, the issue of synchronization is thoroughly dealt with in the operating systems research from the 1960s to the 1970s. Consequently we seek to attack the problem from a pragmatic angle: a set of synchronization primitives can be considered universal, if it can be used to deal with all practical synchronization problems.

The other major theme in this work is the history of multi-threading and synchronization, necessarily including the essence of the history of the electronic digital computer. Our account of computer history culminates on the discussion of the core problem in pragmatic terms: while interrupts and/or multiple CPUs make multi-threading possible, they also result in losing the deterministic behavior that ultimately makes a computer useful. This problem can be solved by introducing mutual exclusion and a wait-signal facility. With these it is possible to restore the determinism of operation. Also, we have found a pragmatic definition of universal synchronization primitives, that is, providing for mutual exclusion and a wait-signal facility. This solution comes at a price though: we have to deal with complications such as deadlocks. With these fundamental issues sorted out we are in a position to discuss implementation issues and to return to our historical theme, the history of multi-threading and synchronization. Finally, we evaluate concrete, well-known synchronization primitives in terms of universality, making use of our newly found criteria.

Preface

Once upon a time in the early 2000s I was in a professional position to ponder a C++ interface for threads and synchronization that could be readily implemented on the most important server platforms in use and, hopefully, on those to come. The thread part of the interface would essentially consist of their creation and termination which seemed trivial enough. The synchronization part was conceptually more difficult, but it seemed feasible to implement something similar to Java's elegant, built in facilities. On the Unix-variants, the implementation could be based on POSIX threads, or pthreads. On the relevant Windows variants it seemed possible to provide a sufficient subset of pthreads functionality based on the native Windows facilities.

It is easy to see that the pthread synchronization facilities are fundamentally identical to those of Java: `synchronized` blocks can be implemented with mutexes, while `wait()` and `notify()` map to an underlying condition variable. The fact the the designers of pthreads and Java respectively, had chosen the the same underlying concept for synchronization was a sufficient assurance that these facilities are universal, that is, sufficient to tackle all practical thread synchronization issues. A further assurance was that these interfaces had not been substantially revised since their introduction, and that apparently no serious critique against them existed or at least was not easy to come by.

I suspected that there existed a sound theoretical foundation here too: after all multi-thread systems had been in production use for ages and hence one would assume that the theoretical aspects had been completely worked out, including the issue of the universality of a set of synchronization primitives; besides, the aforementioned designers had probably based their design on that foundation. At the time I did some research on the subject; while the practical aspects of synchronization seemed to be well covered in professional literature and on the Internet, the theoretical foundation for them seemed elusive. Besides, with scientific papers it seemed to be the case that while one might discover their existence on the Internet, their content would usually not be freely available. So, for the time being these - quite substantial - pragmatic assurances had to suffice.

With this thesis work I have had the opportunity to return to the study of synchronization. As a university student I've had ready access to scientific papers, most often from the on-line databases, where these can be obtained in electronic form, usually free of charge, or, failing that, in paper form through the university library for a reasonable fee. In the process I've realized that such access is a substantial privilege.

As for the theoretical foundation, it quite soon become obvious that a formal treatment of the central issue of universality of synchronization primitives was not likely to be forthcoming; a formal statement of the problem is difficult to obtain. However, the fundamentals of synchronization have been covered in the operating system research from the 1960s to the 1970s in a logical and comprehensive fashion, and nothing radically new in the field would seem to have appeared since. The art of multi-thread programming and synchronization still appears to rest on a pragmatic rather than a rigidly theoretical foundation: the fundamental issue is the potential for indeterminism inherent in the practical hardware capable of multi-threading.

Jouni Leppäjärvi, gradu@leppajarvi.net

Table of Contents

Summary	2
Preface	3
Table of Contents	5
1 Introduction	7
1.1 Goal and motivation.....	7
1.2 Scope.....	8
1.3 Methods.....	9
1.4 Practicalities.....	10
1.5 The organization of this document.....	10
2 A historical preamble	12
2.1 A short prehistory of computers.....	12
2.1.1 Other pioneering work	13
2.2 The EDVAC architecture.....	16
2.2.1 Von Neumann's role	20
2.2.2 The first realizations.....	22
2.3 Interrupt.....	23
2.4 Multiple CPUs.....	24
2.5 Discussion.....	25
2.5.1 Facilitation of multi-threading.....	26
2.5.2 The core problem: race conditions.....	28
3 Synchronization	30
3.1 The fundamental issues.....	30
3.1.1 The universality of synchronization primitives.....	31
3.2 Further problems.....	32
3.3 Implementation.....	34
3.3.1 Atomic operations.....	35
3.3.2 Efficient waiting.....	36
3.4 History	41
3.4.1 A further remark on hardware.....	41
3.4.2 Multi-threading and synchronization.....	41

3.4.3 Fiat lux	44
4 Evaluating concrete synchronization primitives	47
4.1 Semaphore.....	47
4.2 Monitor.....	48
4.3 The POSIX mutex and condition variable.....	51
4.3.1 The POSIX mutex.....	51
4.3.2 The POSIX condition variable.....	52
4.4 Discussion.....	54
5 Conclusion.....	56
References.....	59
Appendix A: reprints and web sources for the references.....	71

Permission to copy and/or distribute without fee all or part of this document is granted provided that the copies are not made or distributed for direct commercial advantage.

Copyright © 2008 Jouni Leppäjärvi

1 Introduction

A *thread*¹ is defined as an independent execution context, where the address space is shared by all the threads present in a system, and the execution of a thread is logically concurrent with that of the other threads. The term *multi-threading* is defined as an aggregate term comprising of the practical implementation of multiple threads and their use by an application programmer. Multi-threading is to be considered a synonym for multiprogramming (Rochester 1955), parallel programming (Gill 1958) and concurrent programming (Codd 1962, 83).

For now the term *synchronization* should be taken quite literally: it is the act of ensuring a desired order of execution among threads. Consequently, *synchronization primitives* are defined as programming abstractions facilitating synchronization. We shall return to these definitions as well as to the questions of what sort of order is to be obtained and why in due time.

1.1 Goal and motivation

The core issue to be resolved by this work is the universality of synchronization primitives. No rigidly formal exposition of this appears to exist within the existing research. As the author is not in a position to suggest one, a pragmatic approach taken: the core issue is rephrased as the question as to *whether a set of synchronization primitives can be considered universal in the sense that such set covers all practical synchronization needs, or, the universality criteria of synchronization primitives*. While the ideal treatment of the subject in terms of an underlying formal model is not possible, the issue of synchronization is thoroughly dealt with in the operating system research from the 1960s to the 1970s, and, as it turns out, this provides ample basis for our approach.

The universality of synchronization primitives is considered a worthy theoretical issue in itself, yet the author has found no existing research addressing the issue directly. As a practical matter, this is deemed to be of substantial importance as the basis of the

¹ (Saltzer 1966, 20) credits the term to V. Vyssotsky.

design of programming interfaces for multi-thread programming and synchronization. In particular, the definition of a portable programming interface for these purposes seems to hinge on this very question.

With the introduction of multi-core processor chips, multiple CPUs, previously confined to the heavy-duty server environments, have already become the norm in even modest PC-class computers; multi-threading is required to make full use of the computational power of such systems. Another important trend is the ever increasing ubiquitousness of embedded systems. These are typically real-time systems where multi-threading is an indispensable implementation technique. Also, as we shall see, the fundamental principles of multi-threading apply in the context of interrupt handling; while in the server and workstation environments such considerations are usually confined to the domain of operating system and device driver implementation, it is common to make direct use of interrupts in embedded systems.

The other a major theme of the present work is the history of multi-thread programming and synchronization, including the essence of the history of the electronic digital computer. In addition of being of value in its own right, the latter is chiefly motivated as a proper preparation on the discussion of the core problem in pragmatic terms: while interrupts and/or multiple CPUs make multi-threading possible, they also result in losing the deterministic behavior that ultimately makes the computer useful. In general, the historical theme is considered well motivated: we shall see that some aspects thereof are poorly known.

1.2 Scope

Our main theme is thread synchronization, where shared memory is used as the medium of inter-thread communication. In particular, we shall consider only a single physical computer installation with one or more CPUs. In case there is more than one CPU we stipulate that they are identical, execute concurrently and have equal access to the memory mapped in the shared address space; in terms of (Flynn 1972) we shall consider SISD / MIMD computer organizations only.

The above statement obviously excludes distributed systems and loosely coupled systems in general. What may not be as obvious is that with the implied single frame of reference we may take a Newtonian (Newton 1687) view of a system, in particular, we can legitimately establish an order of events based on a global clock. This insight is due to (Lamport 1986a, 1986b) where a conceptual framework of synchronization is established based on the specific theory of relativity (Einstein 1905). (Einstein 1954; Halliday & Resnick 1986).

With issues peripheral to the core subject matter, thread synchronization, some latitude in selecting the references has been deemed permissible. For example, to establish the current state of computer architecture and hardware, reputable textbooks have been used instead trying to locate the original research with the simple rationale that the sheer volume of relevant research would have made this an inordinate task. To generalize, it has been considered a proper principle that the further removed an issue is from the main line of investigation, the lower the threshold of using secondary references; on the other hand this means that considerable effort in locating the original research is warranted on issues pertaining to the core subject matter.

1.3 Methods

This thesis can be classified as a qualitative and purely theoretical study; we seek to establish a foundation of our object of study, thread synchronization, by drawing on the existing research. Consequently, the effort is chiefly that of identifying the potentially relevant scientific papers and other publications, making oneself familiar with them and filtering out what is actually pertinent to the object of study, while maintaining a critical, yet unbiased attitude.

As a method we make use of textbooks to establish the 'common perception' on certain subject matters. In the discussion to follow we make some critical comments on the perception thus established. As such this should not be construed as criticism of the individual textbooks themselves. On the contrary, we have chosen these particular textbooks because we believe them to be very well-respected.

1.4 Practicalities

For sample code we shall use C++ (ISO/IEC 1998) as described in (Stroustrup 1997). Our samples are meant for illustration of the concepts discussed, not their real-world implementation. Given this we feel free make use of an ad hoc extension to the language in section 4.2.

As a typographical convention we use italics for emphasis. Above it has been used to make our core theoretical question to stand out. In general, the intent is to alert the reader to the introduction of new terms and concepts. In case a definition is forthcoming later in the text, the use of this typographical device is postponed until then.

1.5 The organization of this document

In the present section we have provided the key definitions, some of them provisional, to be refined in the discussion to follow. We have presented the goal and the scope of this work, and discussed its methods and practicalities. The remainder of this section explains the organization of this document and in so doing provides a summary of the discussion to follow.

Section 2 begins with an exposition of the history of computer hardware. In section 2.1 we present a brief account of the prehistory of computers setting the stage for the discussion of the EDVAC architecture in section 2.2. With this concept as the baseline to further development, sections 2.3 and 2.4 give a brief historical account on the concepts of the interrupt and multiple CPUs. Finally, in section 2.5 we elaborate on these concepts: while the interrupt and/or multiple CPUs are necessary for the implementation of multi-threading, they are also the source of the core problem of race conditions.

Section 3 is an exposition of synchronization, the solution to the core problem exposed in the previous section. In section 3.1 we discuss mutual exclusion, which - in essence - is the solution. To implement mutual exclusion in a practical manner, it is necessary to provide for a wait-signal facility whereby wasteful busy-waiting can be eliminated. The

wait-signal facility is also useful in its own right. With these it is becomes possible to restore the determinism of operation. Also, we have found a pragmatic definition of universal synchronization primitives, that is, providing for mutual exclusion and a wait-signal facility. In section 3.2 we explain how this solution and therefore multi-threading in general comes at a price of further complications, such as deadlock. With these fundamentals established, we are in a position to discuss the underlying implementation considerations in section 3.3. In section 3.4 we return to our historical theme with an exposition of the early history of multi-threading and synchronization, with an emphasis on the history of the individual terms and concepts introduced in the preceding discussion.

In section 4 we evaluate the concrete synchronization primitives the semaphore, the monitor and the POSIX mutex/condition variable in terms of our universality criteria. This time the relevant historical aspects are integrated in the discussion.

In section 5 we conclude this work by presenting a summary of our discussion as a whole. Most importantly, we note our key findings and suggest issues for further research.

2 A historical preamble

In section 2.1 we shall present a brief account of the prehistory of computers setting the stage for the discussion of the EDVAC architecture in section 2.2. With this concept as the baseline to further developments, sections 2.3 and 2.4 give a brief historical account on the introduction of the interrupt and multiple CPUs. Finally, in section 2.5 we elaborate on these concepts: while the interrupt and/or multiple CPUs are necessary for the implementation of multi-threading, they are also the source of the core problem of race conditions.

2.1 A short prehistory of computers

(Webster 1913, 292) defines "computer" as a noun, "one who computes", and compute, the verb as "to determine by calculation". By that definition a computer is a person; Babbage's (1792-1871) dream (Comrie 1946; Wilkes 1975) was automating computing by constructing a machine that could be programmed to carry out laborious calculations, that is, a machine that we have since learned to know as a computer. The technology available in Babbage's time, however, made the construction of such a machine infeasible in practice. (Aiken 1937; Aiken & Hopper 1946a; Comrie 1946; Wilkes 1975).

In (Comrie 1946) it is proclaimed that Babbage's ambitious project had been realized in the Automatic Sequence Controlled Calculator (Aiken & Hopper, G. 1946a; 1946b; 1946c), an electromechanical machine designed and constructed by professor Howard Aiken of Harvard University in collaboration with the IBM Corporation. As scientific publication had been disrupted by the second world war and would not be fully re-established for years (Wilkes 1985, 113), it is no surprise that the work of Konrad Zuse (Desmonde & Berkling 1966; Randell 1982, 159-190; Rojas R. 1997; Stibitz 1966, 814) was not known outside his native Germany. During the same period in which the development of the ASCC took place Zuse independently developed and constructed a series of machines of which the likewise electromechanical, relay based Z3 (Desmonde & Berkling 1966; Rojas 1997) was operational 12 May (Zuse 1993, 62) 1941 (Desmonde & Berkling 1966, 3; Randell 1982, 160; Rojas R. 1997, 5; Schmidhuber 2006, 25; Williams 1998, 10). The ASCC, also known as the Harvard Mark I was

demonstrated in January 1943 and operational in May 1944 (Randell 1982, 191; Williams 1998, 10). It therefore seems that the Z3 was the first concrete embodiment of Babbage's vision of automated computing (Rojas R. 1997, 5; Randell 1982, 160).

During the the second world war the Ballistic Research Laboratory of the US Army Ordnance Department was charged with producing artillery firing tables. Despite having organized some 200 human computers equipped with mechanical calculators and other available computational equipment for the task, the BRL could not keep up with demand in producing the required tables, and was in dire need for a more effective method. This was the problem for which the ENIAC (Goldstine & Goldstine 1946; Stern 1981) was developed and built at the Moore School of Electrical Engineering of the University of Pennsylvania. The ENIAC, conceived and designed chiefly by John Mauchly and J. Presper Eckert, was an electrical machine with some 18000 vacuum tubes. The contract for producing it was signed on June 5, 1943, the machine solved its first problem in November-December 1945 and was officially unveiled on February 16, 1946. (Goldstine & Goldstine 1946; Stern 1981). The electrical construction allowed radically faster operation than an electromechanical one: the ENIAC had a multiplication time of 2.8 milliseconds (Stern 1981, 133), while that of the Z3, slightly faster than the ASCC, was 4-5 seconds (Desmonde & Berkling 1966, 31), that is, ENIAC was faster by more than three orders of magnitude. The speed advantage of electronics was well known before the construction of ENIAC, but so were the reliability figures of vacuum tubes, and using them by the thousands was generally considered to be an impractical proposition. While ENIAC indeed had reliability problems in operation due to tube failure it worked well enough by far to demonstrate the viability of the concept: it was in use until October 1955 (Randell 1982, 299; Rosen 1969, 8). (Stern 1981).

2.1.1 Other pioneering work

A series of relay machines based on binary logic was developed at the Bell Laboratories independently of Zuse's work during the same period of time. The pioneer in this case was George R. Stibitz, who initiated the design and construction of the first machine in the series called "Complex Number Computer" that took place 1937-1940.

This was publicly demonstrated in September 1940 at the meeting of the American Mathematical Society. However, the Complex Number Computer was a hardwired calculator for complex numbers with no provision of program control. The next machine in the series, the "Relay Interpolator" (Cesareo 1946), controlled by a paper tape became operational in September 1943. As Zuse's patent application dated 11 April 1936 (Zuse 1936) includes both the application of the binary system and the use of relay circuits, and the Z3 was operational in 1941 (Desmonde & Berkling 1966, 3; Randell 1982, 160; Rojas 1997, 5; Schmidhuber 2006, 25; Williams 1998, 10) Zuse appears to have anticipated Stibitz in both concept and realization; it is only the "Model V" (Alt 1948a; 1948b), the culmination of the Bell Laboratories series, a general purpose machine that surpasses Zuse's Z3 by including the conditional jump facility the Z3 lacked (Rojas 1997, 7). The first of the two of these machines built was delivered in 1946. The first contemporary publication concerning this series of machines appears to be (Cesareo 1946). (Stibitz 1966; Randell 1982, 241-292).

From the 1930s onward John V. Atanasoff of Iowa State College had been pondering mechanizing the solution of sets of linear equations. In late 1937 he came up with the the concept of a machine based on electronic binary logic (Gustafson 1998, 93). By 1940 he and Clifford E. Berry had constructed a prototype to test the tube circuits performing the arithmetics, an operating memory, or a register file, where individual bits were stored as a charge in a capacitor, and an auxiliary storage based on paper cards perforated and read by electric arcs. The capacitors were installed in rows inside two mechanically rotated drums, with brushes making contact to a row at a time for access and automatic refresh by recharging. This mechanical arrangement severely limited the potential speed of the machine. Due to the second world war having taken everyone away from the project, the development of the machine was abandoned in 1942. At that point the capacitor memory and the tube circuits were in working order, but the auxiliary storage still had reliability problems, and, critically, although Atanasoff had reportedly planned for this, the provision for program control was lacking; the machine was a special purpose calculator with manual switches to control its operation, a numeric keyboard and a separate card reader for input of decimal numbers, and a single mechanical decimal number display for human-readable output.

The only contemporary publication about the machine appears to be Berry's M.Sc. thesis from 1941 dealing with the card reading and punching mechanism used for auxiliary storage. (Gustafson 1998; Metropolis & Worlton 1972, 53; Randell 1982, 294, 315-335; Stern 1981, 33-34).

Zuse was also involved in a pioneering effort in the field of electronic computing. In 1937 he and Helmut Schreyer began investigating vacuum tubes for implementation of binary logic, the foundation of Zuse's concept of a computer (Bauer 1980, 512-513; Desmonde & Berkling 1966, 31). A prototype of a 10-bit arithmetic unit comprising of about 100 tubes was completed in 1942, but this line of development was discontinued in the same year when a plan for a 1500 tube electronic computer was rejected by the German government. (Desmonde & Berkling 1966; Randell 1982, 161, 296).

To this day ENIAC has been "widely thought of as the first electronic digital computer" (Ruttiman 2006, 399). "But that title should arguably be held by the British special-purpose computer Colossus (1943), used during the Second World War in the secret code-breaking centre at Bletchley Park" (Schmidhuber 2006, 25). According to (Sale 1998) "Colossus was the world's first large electronic valve programmable logic calculator" (Sale 1998, 351); (Randell 1977) calls it a "programmable electronic computer" (Randell 1977, 349). The idea of automating the deciphering of the messages encoded with the German Lorenz, or Enigma machines, came from the mathematician Max Newman. Its realization as the Colossus machine was the work of Tommy Flowers. The design started in March 1943, the machine was working by December 1943, and the 1500 vacuum tube Mk1 Colossus was operational in Bletchley Park in January 1944. It was based on binary vacuum tube circuits, in particular, these provided for AND-OR boolean logic. The machine electronically generated the Enigma cipher sequences and systematically applied them on a ciphered message punched on a looped paper tape read at 5000 characters / second. On each pass the machine computed a correlation function for identifying potential matches and then moved on to try another cipher sequence. The machine proved to be reliable and effective; a total of ten were built by the end of the war, the latter with an improved, faster design with 2400 vacuum tubes. The first of these mkII machines was completed in June 1944. The

machines were programmable - or rather configurable - in that the correlation function was set by switches and patch panels; the exact degree of this programmability seems unclear, though. There are no contemporary publications: the very existence of the machines was kept secret by the British government until October 1975, before which time the machines had been destroyed along with their technical documentation. (Randell 1977; Randell 1982, 296-297; Sale 1998).

Priority questions are perhaps naive in the context of independent development of similar ideas occurring approximately the same time, especially considering the individual developments in the wider context of scientific development. The utility and therefore the motivation of Babbage's machine hinged on the numerical methods available at the time, which could only exist in the wider context of mathematics. The ability to actually construct such machines depended on the state of engineering, which is fundamentally based on physics, and through it, again, mathematics. Therefore, the concept and construction of computers was founded on the labors of the countless generations of past scientists; "If I have seen further it is by standing on the shoulders of giants" (Newton 1676). (Babbage 1864; Smith 1970; Williams 1998).

2.2 The EDVAC architecture

During the development of the ENIAC it dawned to its designers that a more powerful machine could be built with far less hardware by refining the design (Wilkes 1985, 108). This was suggested to the BRL, and in September 1944 funds were granted for research and development on this new and improved machine, the EDVAC. (Stern 1981, 58-59). The logical design of the EDVAC (Neumann 1945) is generally known as the *von Neumann architecture*, however, for reasons explained in the following section we refrain from using this term, and use the *EDVAC architecture* instead. The EDVAC architecture is of immense consequence: it is the conceptual and logical blueprint of all electronic computers since, and as such at the core of the definition of a computer in the modern sense (Comer 2005; Paterson & Hennessy 2005; Tanenbaum 2001). In the rest of this section we shall make an attempt to shortly lay out its essence.

A defining aspect of the EDVAC is its electronic implementation; the ENIAC had

proved large scale use of vacuum tubes to be a practical proposition. Vacuum tubes were estimated to be 5000-10000 times faster than relays (Neumann 1945, 4.3, 5.4), the most promising electromechanical components for switching elements. The chief practical consequence of this was a vast expansion in the scope of the problems to whose solution automatic computation could be practically applied; an electronic implementation has been the norm since. (Comer 2005; Neumann 1945; Paterson & Hennessy 2005; Stern 1981; Tanenbaum 2001).

In the EDVAC architecture the hardware is divided into five logical parts: M, I, O, CA and CC. M, the *memory*, is essentially an array identical units of storage each capable of holding the same range of values, where each individual unit is identified by a unique address. I and O are the input and output facilities necessary for exchanging data with the outside world. CA provides the ability to perform arithmetical operations, such as addition and subtraction. CC coordinates the operation of the other units under the control of a program stored in M: data can be moved from M to CA, operations provided by the CA executed on it, and the results thereof stored back in M; I and O are likewise operated under program control by CC to transfer data and programs between the computer and its environment. There may also exist R, external storage, the contents of which may be accessed through I and O. (Neumann 1945, 2.1-2.9). Out of practical necessity the computer must also have facilities for the initiation and control of its operation which may also be used for I and O. The terms *processor* and *CPU* are commonly used for the combination of CC and the CA (Comer 2005; Paterson & Hennessy 2005; Tanenbaum 2001).

An essential feature of the EDVAC architecture is that it specifies a *binary* machine composed from interconnected binary logic components. Such a component is an abstraction of the underlying concrete hardware elements; there is only a limited number of fundamental component types, where the binary output of each type is a logical function of one or more binary inputs, this logical function being characteristic of each specific component type. The use of the binary system results in a construction that is conceptually simple and efficient to implement; in particular, the implementation of arithmetic in CA is vastly simplified compared to that of the of the likewise *digital*,

but decimal system used in the ASCC and the ENIAC, where ten discrete states were needed instead of the two of the binary system. Zuse too had used the binary system in his machines (Desmonde & Berkling 1966, 30; Rojas 1997, 6); this is explicitly mentioned in his patent application dated 11 April 1936 (Zuse 1936, 167), also his logic components are in principle apparently closer to the modern ones (Desmonde & Berkling 1966, 30), although the *E-elements* of the EDVAC design (Neumann 1945, 6.1-6.5) can be implemented in terms of modern logic gates and vice versa. (Comer 2005; Paterson & Hennessy 2005; Neumann 1945).

As a consequence of using the binary system the units of M are made up from a number of binary digits, or bits. Incidentally, the EDVAC design opted for a 32-bit (Neumann 1945, 12.2) *word* as the unit of M and the the arithmetic operations of its CA; in particular, the CA had three 32-bit *registers*, storage local to the CA, two of them used for the arguments and one for the result of arithmetic operations (Neumann 1945, 11.1). Another register is required by the CC to keep track of the current instruction in M (Neumann 1945, 14.4). This is commonly known as the *program counter*, or *PC* for short (Comer 2005, 73-75; Paterson & Hennessy 2005, 80). The concept of registers in the CPU also dates back to Zuse (Rojas 1997, 6-9).

The CC unit provides the program control alluded above by repeating a cycle of fetching a word at a time from consecutive locations of M beginning with some designated location at initialization. Each such word is treated as an *instruction*, that is, it causes a particular operation to take place. Such operations include moving a given word from M to a CA argument register, causing the CA to perform an arithmetic operation with the values in its argument registers so that the resulting value ends up in the result register, moving the word in the result register to given location in M, and causing the CC to fetch its next instruction word from a given location in M instead of the default consecutive location (*jump*). In particular, a variation of the last operation is possible where the target location depends on a previous comparison of two words (*conditional jump*). A comparison is a variation of the regular arithmetical operation of subtraction with the distinction that a special flag is set to the sign of the result of the subtraction (Neumann 1945, 11.3). This flag is subsequently used to select one of two

possible jump targets. (Neumann 1945). The flag may be thought of as an one-bit register; the concept has been since extended to a *flag, or condition code register* with additional bits indicating various conditions (Comer 2005, 123-124; Paterson & Hennessy 2005, 140). The ENIAC had a conditional jump facility (Randell 1982, 299), while the Z3 and ASCC did not (Rojas 1997, 7; Randell 1982, 159, 191), but were controlled by a punched tape, where the instructions followed each other in strict sequence; a *loop* could be effected only by making the control tape into a physical loop (Rojas 1997; Randell 1982).

With the EDVAC the *stored program concept* was introduced, that is, M is used to store both the program and the data. An electromechanical tape reader used to control the Z3 and ASCC (Aiken & Hopper 1946a, 207; Desmonde & Berkling 1966, 31) was an adequate arrangement, as the speed of these machines was limited by other aspects of their electromechanical construction. With vacuum tubes the source of the instruction stream had to be correspondingly fast, if the full performance benefits were to be realized. In the original design of the ENIAC the solution for adapting to different problems was rewiring effected by manual switches and plugged wires². This had been considered adequate for the artillery table calculations, where the need for reprogramming was deemed infrequent. The EDVAC, however, was to be a general purpose machine. As the speed requirement applied equally to program instructions and frequently accessed data, using a fast M for both was an elegant, in retrospect inevitable, solution. (Neumann 1945; Randell 1982, 298-299, 375-381; Metropolis & Worlton 1972, 54; Stern 1981).

In the design of the EDVAC simplicity, in practice reduced tube count, was emphasized over speed; the resulting machine would still be fast, but cheaper and more reliable (Neumann 1945, 5.4-5.6). It was estimated that the other parts, excluding M, could be implemented with well under 1000 vacuum tubes, which would correspond to

² In early 1948 these were replaced by an instruction decoder making use of the existing read-only memory, a battery of mechanical switches ("function tables"), containing the program. Arguably, this makes the ENIAC the first computer to actually implement the stored program concept, albeit in a slightly degenerate form. (Metropolis & Worlton 1972, 54).

less than 8 words of M implemented as tube circuits (Neumann 1945, 12.5). Since the minimum amount memory sufficient for some of the planned uses was 1000 words (Neumann 1945, 12.1-12.4) it was "manifestly impractical" to implement M with tube circuits (Neumann 1945, 12.5). (Neumann 1945). Fortunately two promising technologies for implementing a sufficiently large M were in sight: mercury delay lines and the iconoscope (CRT) (Neumann 1945, 7.6, 12.8; Stern 1981). Of these the former was used in the EDVAC for an M of 1024 32-bit words (Stern 1981, 133). Although some of the memory would be used for the program, capacity for variable data was radically extended: the ENIAC had 20 words of 10 decimal digits (Stern 1981, 133), the Z3 had 64 22-bit floating point words (Desmonde & Berkling 1966, 31; Rojas 1997, 6-7), while the ASCC had 72 words of 26 decimal digits (Aiken & Hopper 1946a, 205). Despite the implementation strategy of primarily minimizing the tube count the performance figures were still favorable: the 2.9 ms multiplication time of the finished machine with 3600 tubes was on par with the ENIAC (Stern 1981, 133). The total tube count projected in (Neumann 1945) was 2000-3000 where most of these would go into integrating the memory to the rest of the system (Neumann 1945, 12.5). Considering that the actual implementation added a redundant CPU for error detection, the estimate was right on target (Stern 1981, 133).

To summarize, the essence of the EDVAC architecture is its logical structure: at the top level the division to the processor, the memory and the i/o-facilities, at the level directly above of the hardware implementation the composition of the system of interconnected binary logic components. The machine operates under program control, where the program is stored in the memory shared with the data. The binary logic components are of electronic implementation; the use of vacuum tubes resulted in performance that made a vast number of novel applications possible. Subsequently the advances in electronics have sustained a similar effect of increasing performance and capacity resulting in continuously expanding sphere of application. (Comer 2005, 47-48; Neumann 1945; Paterson & Hennessy 2005; Tanenbaum 2001, 6-18).

2.2.1 Von Neumann's role

In 1944 the development of the ENIAC attracted the attention of the eminent scientist

(Aspray 1990) John von Neumann, who was involved in the US nuclear weapons program at Los Alamos; he saw that the ENIAC could be used for the extensive calculations needed at Los Alamos, which at the time were carried out by human computers using mechanical calculators (Feynman 1992, 125-132; Metropolis 1987). Von Neumann also realized that such a machine would have immense value as a scientific and engineering tool in general. Beginning in September 1944 he started to make periodic visits to the Moore School to learn more of the ENIAC and to collaborate on the development of EDVAC. (Aspray 1990; Stern 1980; 1981).

Von Neumann composed a memorandum, (Neumann 1945) on the EDVAC design discussions he participated at the Moore school titled "First Draft of a Report on the EDVAC", in itself a masterful description of the logical design of the planned machine. Although the EDVAC, like the ENIAC, was also a BRL project started during the war, in principle subject to military security, the memorandum got rather widely, even internationally (Aspray 1990, 41; Wilkes 1985, 108) distributed. The problem with the draft report and its de facto publication was "... that it bore von Neumann's name only" [, which] "led to a grave injustice being done to Eckert and Mauchly, since many people have assumed that the ideas in the draft report were all von Neumann's own. It is now clear that Eckert and Mauchly had advanced a substantial way in their thinking before von Neumann was given permission to visit the ENIAC project" (Wilkes 1985, 109). In particular, the stored program concept had been conceived during the preliminary design of the EDVAC, predating von Neumann's involvement (Metropolis & Worlton 1972, 55; Stern 1980, 354). (Aspray 1990; Metropolis & Worlton 1972; Stern 1980; 1981).

While the effective publication of (Neumann 1945) can be considered an accident, von Neumann "did not attempt to correct the impression that he was the sole author" (Stern 1980, 356). Indeed, (Burks & al. 1946) by Burks, Goldstine and von Neumann, the first officially published scientific paper on the EDVAC architecture fails to give credit to Eckert and Mauchly. The underlying reason for this appears to have been personal animosity that developed between von Neumann and the two of Eckert and Mauchly as a result of a patent dispute. (Aspray 1990; Stern 1980; 1981).

In March 1946 Eckert and Mauchly refused to sign an agreement assigning patent rights relating to the EDVAC to their employer, the University of Pennsylvania; they resigned and proceeded to claim the patent rights for themselves, and set up a company to exploit them commercially. Von Neumann resented this as he saw that the development of the electronic computer should continue as a primarily academic pursuit. He also filed patent claims and submitted (Neumann 1945) as his disclosure. In April 1947 it was determined that in the legal sense (Neumann 1945) was a publication that would predate formal patent applications by more than a year, and so the ideas expressed in it were not patentable, but in the public domain. This was a serious blow to the business aspirations of Eckert and Mauchly. Von Neumann was pleased with the result, in particular, because by this time he was in charge of a new computer project at his home base, the Institute of Advanced Study. (Burks & al. 1946) and subsequent publications resulting from this project served to cement the idea of the EDVAC architecture as "von Neumann architecture". With von Neumann's untimely death of cancer in 1957 the posterity was left without his help in settling the issue. (Aspray 1990; Stern 1980; 1981).

The chief reason for including this section in our present discussion is that although the above has been known for quite some time now, the unmitigated use of the term "von Neumann architecture" still continues. To make our point concrete: such use is still to be found in some of the finest modern textbooks of our field such as (Comer 2005) and (Silberschatz & al. 2005).

2.2.2 The first realizations

The EDVAC itself was not the first computer to implement the stored program concept in practice. Due to Eckert and Mauchly leaving and taking some other project personnel with them the completion of the machine was delayed: the central processor was not accepted by the Army Ordnance until 1949 (Stern 1981, 92), and the machine was not completed until 1952 (Metropolis & Worlton 1972, 54).

The first machine to implement the stored program concept in practice was the

Manchester University SSEM, Small Scale Experimental Machine or "The Baby", a prototype machine with the most rudimentary computing facilities for testing the attached 32x32 bit electrostatic Williams tube (CRT) memory. It ran its first programs in June-August 1948. (Lavington 1978, 5; 1993, 45; Napper 1998, 367-368; Williams & Kilburn 1948).

The first practical implementation of the EDVAC architecture is generally considered to be the EDSAC, or Electronic Delay Storage Automatic Calculator (Metropolis & Worlton 1972, 54; Randell 1982, 379-380; Rosen 1969, 9; Wilkes 1990; Wilkes & al. 1951): it executed its first program in May 1949 (Randell 1982, 380) and was publicly demonstrated on 22 June 1949 (Renwick 1949). The machine was built with some 3000 vacuum tubes (Randell 1982, 380), and it had a 512x17 bits (Wilkes & al. 1951, 3) mercury delay line memory (Wilkes 1990; Wilkes & Renwick 1949). An addition to the original EDVAC design (Neumann 1945) were bit manipulation instructions: left and right shifting and "collate", or logical AND. (Wilkes 1990; Wilkes & al. 1951; Wilkes & Renwick 1949).

The first commercially available computer was the Ferranti Mark I (Napper 1998; Lavington 1978; 1993). The first unit was delivered in February 1951 (Lavington 1978, 5; Napper 1998, 373). The Ferranti Mark I was the commercial version of the Manchester Mark I (Napper 1998; Lavington 1978; 1993), a result of continued development in Manchester University originating from the SSEM. The Ferranti Mark I comprised of some 4000 vacuum tubes and a combination of 256 40-bit words of Williams Tube and 4-16k words of drum memory. The Mark I Autocode programming language, available from March 1954 automated the drum transfers so that a single logical memory space approaching the available drum capacity was available to the programmer. (Lavington 1978, 6-7).

2.3 Interrupt

In 1954 the DYSEAC computer (Elbourn & Witt 1953; Leiner & Alexander 1954; Leiner 1954; Weik 1961, 234-235), and, with it the *interrupt* was introduced (Knuth 1997, 231; Smotherman 1989; 2006; 2007). An interrupt is a processor's reaction to an

event at the hardware level such that "at the earliest convenient moment (but never sooner than after completing the current instruction) the central processor interrupts the execution of the current program (in such a neat way that the interrupted computation can be resumed at a later moment as if nothing had happened) and starts executing an interrupt program instead ... " (Dijkstra 1971, 116). In general, the event triggering an interrupt may be completely asynchronous with the operation of the processor.

Consequently, the interrupt may occur at any machine instruction boundary causing the main program to be preempted by the execution of the interrupt program, or *interrupt handler*, after which the execution of the main program is resumed at the machine instruction following the instruction boundary at which the presence of the triggering hardware condition was detected; the interrupt handler code is thus effectively injected into the instruction stream of the CPU. To do this without disturbing the interrupted computation it is necessary to preserve the machine state relevant to the interrupted computation over the execution of the interrupt handler (Gill 1958, 7; Mersel 1956, 52). (Leiner & Alexander 1954; Leiner 1954; Dijkstra 1971).

The rationale for introducing the interrupt is that the processor may, in principle, react instantly to an event external to it without constantly checking for its occurrence in the program, but letting the hardware do the checking automatically. In particular, the interrupt mechanism provides a convenient means for making use of the processor while an input/output operation is in progress on an external device. (Dijkstra 1971; Gill 1958, 5; Knuth 1997, 231; Leiner & Alexander 1954; Leiner 1954).

On commercial computers an interrupt mechanism was first built into a single Univac 1103A per customer request for use in wind tunnel experiments (Mersel 1956; Codd 1962, 82). The IBM 709 (Greenstadt 1957) had an interrupt system as a design feature; it was announced in 1957, and the first unit was delivered in 1958. (Bashe & al. 1981; Rosen 1969, 14-15).

2.4 Multiple CPUs

The concept of multiple CPUs sharing a common memory was introduced with the Univac-LARC (Eckert 1956; 1959; Lukoff & al. 1959; Weik 1961, 958-961), which

had the option of two main processors in its design (Eckert 1956, 17). Only two of these machines were built (Rosen 1969, 26), and neither of them had the optional second CPU (Gray 1999a), so according to (Enslow 1974, 36) the first concrete multi-CPU computer was the Burroughs 825 (Anderson & al. 1962; Thompson & Wilkinson 1963). "The first D825 was delivered to the Naval Research Laboratory in August 1962 and was operational by November" (Gray 1999a). Of the some fifty units delivered (Gray 1999a) most went to the US military and it is perhaps for this reason that no publications on these machines and their applications other than those already mentioned seem to exist; it is considered fair to say that multiprocessor systems were introduced into the mainstream with the IBM 360 series models 65 and 67 (Blaauw 1964; Blaauw & Brooks 1964) and the Univac 1108-II (Stanga 1967; Sperry Rand Corporation 1965; 1970) in the later part of the 1960s (Gray 1999b; Padegs A. 1981; Rosen 1969 32-33; Rosen 1972).

The rationale for introducing multiple CPUs was twofold: making more computational power available per an installation and increased reliability of systems thus equipped. The latter meant that a system could remain in service, albeit with reduced computational power, as long as a single functional CPU remained in operation. This last point was considered especially important in military applications. (Anderson & al. 1962; Codd 1962, 85, 122; Thompson & Wilkinson 1963).

2.5 Discussion

In the following sections we discuss the relevance of above hardware developments to multi-threading. In section 2.5.1 we'll see how the interrupt and multiple CPUs make multi-threading possible in the first place. In section 2.5.2 we expose the resulting core problem: race conditions. The main point is that facilitating for multi-threading on the hardware level and race conditions are inextricably intertwined. This will serve as a basis of our discussion of synchronization in section 3.

"In the very old days, machines were strictly sequential, they were controlled by what was called 'a program' but could be called very adequately 'a sequential program'.

Characteristic for such machines is that when the same program is executed twice -

with the same input data, if any - both times the same sequence of actions will be evoked. In particular: transport of information to or from peripherals was performed as a program-controlled activity of the central processor." (Dijkstra 1971, 115). To elaborate: the input and output for a computer could be a stack of punched cards. The input cards would be read by the program operating the card reader unit and likewise the output would be produced by the program operating the card punch. The crucial feature of the program controlled input and output here is that it happens in perfect synchronism with the execution of the program, that is, the program decides when it is ready to read another card or produce one; the program always waits until the respective operations of reading or punching a card are completed before proceeding with its computation or further input or output operations. In other words, in absence of a hardware failure, such a machine is a completely deterministic one, and this is - in fact - what makes it useful in the first place.

2.5.1 Facilitation of multi-threading

By our definition of a thread as an independent execution context we can rephrase the description of an interrupt in terms of threads. The interrupt handler executes in the context of a temporary thread existing for the duration of the said execution.

Alternatively we can say that for each interrupt there exists a thread waiting for the interrupt event to occur at which point the thread wakes up and calls the interrupt handler to resume its dormant state upon return.

The essence of interrupt handling is storing the state of the interrupted thread, then processing the interrupt, and finally restoring the state of the interrupted thread and resuming its execution. Multi-threading on a single CPU system is implemented using the interrupt by arranging for a *context switch*: within the interrupt handler the state of the current thread is stored into suitable a data structure specific to the current thread, and, when the interrupt processing is over, the previously saved state of a different thread is restored and its execution resumed from the point at which the state was saved. With a provision to generate periodic hardware interrupt signals it is thus possible to create a *virtual processor* for each thread as needed, where the incremental cost of each thread is merely the storage needed for the state of a thread. (Dijkstra

1967; Gill 1958; Salzer 1966; Silberschatz & al. 2005; Tanenbaum 2001).

Multiple concurrently executing CPUs obviously give rise to multiple concurrent threads. Interrupts are used in both single and multi-CPU systems to create virtual CPUs on top of the physical ones and to arrange for a thread to remain dormant in wait for an external event, such as the completion of a pending input/output operation. Strictly concurrent execution is possible with multiple CPUs only; in a single-CPU system only one machine instruction and thus only thread can be in execution at any given time. However, in both single and multi-CPU systems the machine instruction streams corresponding to the execution of different threads may be interleaved in time. Furthermore, from the point of view of the programmer, such interleaving occurs on a random basis. In the general case the programmer has no control of the external events causing interrupts potentially resulting in a context switch, and in any case altering the execution timing of the interrupted thread. Because the threads can share a CPU or a memory cell only by taking turns in its use, the execution timing of an individual thread also depends on that of the other threads. Thus, in the general case, we have a complex, dynamic system of cyclic feedback loops with essentially random timings and consequently random interleaving of machine instructions from different threads. (Comer 2005; Dijkstra 1965a; 1971; Paterson & Hennessy 2005; Silberschatz & al. 2005; Tanenbaum 2001).

To return to the point that threads can share a memory cell only by taking turns in its use, we first note that this is necessarily so with a single CPU as only one memory operation can be in progress at any given time. Even with multiple CPUs the threads must similarly take turns. This is so because of technical and logical reasons (Dijkstra 1971, 116). In a typical multi-CPU machine the technical reason is that the CPUs share a common *bus*: essentially a set of wires connecting the CPUs and the memory. Also, for technical reasons the unit of the sharing is not just a single memory cell, that is, a binary digit, but a group of memory cells typically corresponding to the machine word; such an arrangement results in better performance as multiple memory cells are accessed in parallel. However, the fundamental consideration here is the logical one: the result of two or more simultaneous write operations on a single cell of memory is

ultimately undefined as only one of the values to be written can in fact be stored. This consideration obviously extends on any larger unit of memory cells accessed simultaneously as a group. (Comer 2005; Paterson & Hennessy 2005; Tanenbaum 2001).

2.5.2 The core problem: race conditions

To illustrate the effect of random interleaving of machine instructions from different threads a simple example adapted from (Codd 1962, 124) follows.

```
int i;  
...  
--i;
```

Given the above code fragment, consider the following sequence of events: thread A executing the code reaches the decrement operation, reads the value of the variable *i* from memory and decrements the value. Before thread A completes the operation by writing the value of *i* back to memory, thread B executing the same code reaches the point of the decrement operation and executes all its three steps of reading the memory, decrementing the value and writing the decremented value back to memory. After this thread A completes its decrement operation by writing its copy of the decremented value back to memory. The net result is that the value of the variable *i* in memory is decremented by one only although the decrement operation was executed twice. This is due to the interleaving of the machine operations as described above; if the two threads had executed their decrement operations without the component machine operations overlapping in time, the result would have been that the value of *i* in memory would have been decremented by two. In other words, this simple computation may result in two outcomes, that is, the result is not deterministic.

The consequences of the simple example above can be generalized to the principle that the results of computations may not be deterministic when one or more variables are shared by two or more threads and at least one of the threads modifies the said variable(s) (Dijkstra 1965a, 64). Such a situation where the potential for indeterminate operation exists is known as a *race condition*. The existence of race

conditions is unacceptable, because it is their determinism that makes computers and software useful in the first place. The interleaving of machine operations from different threads, however, is a part of the implementation - in fact, the very definition - of multi-threading. Therefore, race-conditions must be eliminated, or else multi-threading is to be discarded as a technique of practical merit.

With race conditions the very fabric of logical causality breaks down as operations on the logical level are broken down in the corresponding machine operations; the causal relationship between the logical operations is invalidated due to the arbitrary interleaving of their component machine operations. We owe this insight to (Lamport 1986a; 1986b).

A race condition exists in a multi-thread system, when information necessary for deterministic operation is temporarily unavailable, having become private to some thread for the time being. That is, the critical information exists only in the CPU registers, stack and/or any other storage available to the said thread only. This can be considered a definition, or - perhaps - a theorem.

3 Synchronization

In this section we discuss synchronization, the solution to the core problem exposed in the previous section. In section 3.1 we introduce mutual exclusion, which - in essence - is the solution. However, to implement mutual exclusion in a practical manner, it is necessary to provide for a wait-signal facility whereby wasteful busy-waiting can be eliminated. The wait-signal facility is also useful in its own right. With these it is possible to restore the determinism of operation. Also, we have found a pragmatic definition of universal synchronization primitives, that is, providing for mutual exclusion and a wait-signal facility. In section 3.2 we explain how this solution and therefore multi-threading in general comes at the price of additional complications such as deadlocks. With these fundamentals established, we are in a position to discuss the underlying implementation considerations of synchronization in section 3.3. In section 3.4 we return to our historical theme with an exposition of the early history of multi-threading and synchronization, in particular, the history of the individual terms and concepts introduced in the preceding discussion.

3.1 The fundamental issues

The effects of arbitrary interleaving of machine operations can be compensated and the resulting indeterminism avoided with the provision of *mutual exclusion* which is defined as an arrangement by which it is possible to prevent more than one thread at a time entering a specified section of code, where the 'specified section of code' is a *critical section*. "The purpose of the critical sections is to resolve any ambiguity in the inspection and modification of the" [shared] "variables" (Dijkstra 1965a, 64) even when such variables make up arbitrarily complex data structures; code enclosed in a critical section becomes an indivisible operation from the point of view of the threads other than the one in the critical section. A further observation of practical merit is that mutual exclusion can be used with other resources than shared memory that, by their nature, need protection from overlapping access by multiple processes, such as a computer terminal or a line printer. (Codd 1962; Dijkstra 1965a; 1967; 1971).

The *wait-signal facility* is defined as an arrangement where a thread can wait, removed from execution, until it is signaled it to continue. In general this is required to replace

busy waiting which must be avoided in multi-threading: not only are processor cycles that could be used for productive work by other threads wasted, but in a multi-CPU setup so are bus cycles which slows down the other threads, even if they are executing on different processors. (Dijkstra 1965a, 26-28). In particular, the wait-signal facility is needed, if a critical section is already occupied by a thread when another thread tries to enter it. The latter thread must wait until the critical section becomes available; it will be signaled to continue by the implementation of mutual exclusion in the context of the thread leaving the critical section. (Codd 1962; Dijkstra 1965a; 1971).

We are now in a position to define *synchronization* as the act of 1) ensuring an order of execution among threads such that the computations performed produce deterministic results, while 2) eliminating wasteful busy waiting. A *synchronization primitive* is defined as an entity with a programming interface that facilitates synchronization.

3.1.1 The universality of synchronization primitives

A *set of universal synchronization primitives* is defined as a set of synchronization primitives that can be used to deal with all practical synchronization needs. A *universal synchronization primitive* is defined as the sole member of a single member set of universal synchronization primitives.

"But since the order of operations in time is not completely specified, the output of a concurrent computation may be a time-dependent function of its input unless special precautions are taken. ... In contrast, the output of a sequential process can always be reproduced when its input is known." (Brinch Hansen 1973, 30-31). Here, "a sequential process" should be considered a synonym for a thread in a single-thread system.

"Brinch Hansen and Hoare, who call the interface module a monitor, proposed this scheme. As it ensures that shared variables are accessed under mutual exclusion only, it makes sequential programming verification rules applicable to multiprograms adhering to the scheme's discipline." (Wirth 1977, 578).

Given the above we argue that it is the difference between a single-thread system and a

multi-thread one, in particular, the problems arising from this difference that is the crux of the issue of the universality of synchronization primitives: if these problems can be completely compensated by using a set of synchronization primitives, such a set of synchronization primitives can be used to deal with all practical synchronization needs and can therefore be considered universal. Consequently, a set of synchronization primitives is universal, if it provides mutual exclusion and the wait-signal facility as these are sufficient to compensate the effects of arbitrary interleaving of machine instructions and to avoid wasteful busy waiting.

3.2 Further problems

The concept of deadlock can be found in (Dijkstra 1965a) as "the deadly embrace" (Dijkstra 1965a, 73-75). A simple example of this is a programming error where a critical section is not properly released and subsequent attempts to enter it result in waiting indefinitely. In general, a *deadlock* is a situation where a set of threads removed from execution waits for each other in a cyclic dependence: "each process is waiting for a condition which can only be satisfied by one of the others; but since each process expects one of the others to resolve the conflict, they are all unable to continue." (Brinch Hansen 1973, p. 122). Here, a "process" should be considered a synonym for a thread.

"After you after you blocking" (Dijkstra 1965b) or a *livelock* (Ashcroft 1975, 135) is a situation where a group of running threads are in a state of cyclic wait-signal dependence. A simple example of this is two threads alternating between waiting and signaling the other thread. Livelock can be considered an active form of deadlock: the threads locked in the cycle of mutual waiting and signaling are effectively unable to continue with useful work yet they continuously consume machine cycles.

Deadlocks and livelocks can be avoided by correct application of mutual exclusion and the wait-signal facility, hence no additional arrangements like mutual exclusion or wait-signal facility are needed to compensate for them. With deadlocks an important practical technique for their avoidance is to arrange acquiring nested critical sections in a fixed order (Hoare 1971, 65). This insight is credited to Dijkstra in (Hoare 1971, 70).

The validity of this method, that is, the absence of deadlock can be proved by induction (Brinch Hansen P. 1973, 126-127).

In an implementation each thread may be assigned a *priority*, so that, under contention, a thread with a higher priority is assigned to a CPU in preference of other threads (Codd 1962). While this arrangement is of considerable practical merit, it is also the source of further problems.

(Tanenbaum 2001) first mentions *starvation* as "a situation ... in which ... the programs continue to run indefinitely, but fail to make any progress" (Tanenbaum 2001, 126). This appears similar to what we have called a livelock above. Later it is described as "a problem closely related to deadlock ..." [, where the application of a resource allocation policy] "may lead to some processes never getting service even though they are not deadlocked" (Tanenbaum 2001, 184). "Starvation can be avoided by using a first-come, first-serve, resource allocation policy" (Tanenbaum 2001, 185), that is, by abolishing the use of priority. In other words, "a major problem with priority scheduling is indefinite blocking, or starvation" (Silberschatz & al. 2005, 163). Again, the term "process" as used above should be considered a synonym for a thread. In addition to making the point that the application of priority may give rise to further problems, the above leads to the observation that the concept of starvation may not be entirely well-defined. However, it appears fair to say that starvation is a condition where a thread is deprived of a needed resource to the extent that it can't fulfill its purpose. Consequently avoiding starvation is a global system design issue, rather than a problem that can be dealt within the scope of synchronization primitives.

A high priority thread must wait for a low priority thread when about to enter a critical section currently occupied by the low priority thread. In practice this is not problem, as long as the critical sections are short in duration. However, if it so happens that a medium priority thread is assigned to a CPU in preference of the low priority thread, we have a situation in which the high priority thread will not be assigned to a CPU until the medium priority thread gives it up, which may take an arbitrarily long time. Such a condition is known as *priority inversion*. The solution to this problem is *priority*

inheritance, that is, arranging that the priority of the thread in the critical section is equivalent or higher to that of any thread waiting to enter the critical section. The concepts are readily generalized to resource allocation in general. This is a somewhat moot point, though, since mutual exclusion is the basis of resource allocation in the context of multi-threading, and so solving the problem for mutual exclusion effectively solves it in the general case as well. (Lampson & Redell 1980, 112-113; Sha & al. 1990).

3.3 Implementation

In this section we will sketch how synchronization primitives may be implemented. To do so we need a concrete example, and therefore introduce the *mutex* (Dijkstra 1965a, 51). Below a mutex is presented as a partial C++ class declaration:

```
class mutex
{
public:
    void lock();
    void unlock();
private:
    bool locked;
    ...
};
```

A mutex is a synchronization primitive that provides for mutual exclusion where the critical section to be protected by the mutex is enclosed between invocations of the `lock()` and `unlock()` operations:

```
mutex m;
...
m.lock();

// critical section
...
m.unlock();
```

While a thread is within the critical section (owns the mutex) other threads calling `lock()` on the same mutex must wait until the thread within the critical section calls `unlock()` on the mutex (releases the mutex). The `locked` instance variable indicates this condition. Given this specification a first attempt of an implementation could be:

```

void mutex::mutex(void) : locked(false)
{
}

void mutex::lock(void)
{
    while(locked);
    locked = true;
}

void mutex::unlock(void)
{
    locked = false;
}

```

This, of course, is unsatisfactory on two accounts. Firstly, it may happen that two (or more) threads enter `lock()` sufficiently simultaneously that they all find `locked` false and proceed to set it true, in which case mutual exclusion is not achieved. Secondly, waiting is implemented as a busy waiting loop.

3.3.1 Atomic operations

It seems that we are now in an impasse: we need mutual exclusion to implement mutual exclusion. Fortunately we can expect the hardware to provide this. In a single-CPU system we can simply disable interrupts (and thereby a potential context switch) to make testing and setting `locked` in an indivisible, or *atomic operation*: only one execution of an atomic operation may be in progress at any given time. To make use of this arrangement, we declare a class as follows:

```

namespace sys
{
    class atomic
    {
    public:
        static void begin(void)
        {
            disable_interrupts();
        }

        static void end(void)
        {
            enable_interrupts();
        }
    };
};

```

Above `disable_interrupts()` and `enable_interrupts()` stand for the platform

specific operations to disable (or, rather, postpone) and enable interrupts: there is usually a pair of machine instructions for this purpose. This method of obtaining mutual exclusion is due to (Loopstra 1959b, 343). The use of the `sys` namespace is intended to convey the idea that a facility is system specific. We will subsequently use it for the same purpose without further comment.

With `sys::atomic` we can revise `mutex::lock()` as follows:

```
void mutex::lock(void)
{
    sys::atomic.begin();

    while (lock == true)
    {
        sys::atomic.end();
        sys::atomic.begin();
    }

    lock = true;

    sys::atomic.end();
}
```

The implementation is now logically correct assuming the presence of a virtual processor for each thread implemented using a timer interrupt, but it is still impractical as it uses busy waiting; in fact, the somewhat convoluted arrangement of enabling and then immediately disabling interrupts in the wait loop makes it possible to snatch the CPU away by an interrupt. Assuming that the variables of type `bool` can be inspected and modified in a single memory operation, `unlock()` can be left as it is, due to the technical and logical reasons, or fundamental properties of memory, as discussed in section 2.5.1. Another necessary assumption made above is that every reference to a variable results in memory access, that is, there is no need for the `volatile` qualifier to force the compiler to generate code with this property (or, in fact, it is as if every variable was declared with the `volatile` qualifier). In what follows we will continue to make these assumptions without further comment.

3.3.2 Efficient waiting

Now we are to deal with the second shortcoming in our initial attempt to construct a

mutex: the need for efficient waiting. In this respect, our revised `lock()` is still not satisfactory. For this, we need the underlying implementation of threads to provide us with some facilities, as follows:

```
namespace sys
{
    class thread_id
    {
        ...
    };

    void suspend(void);
    void resume(const thread_id tid);
    thread_id self(void);

    class thread_set
    {
    public:
        void put(thread_id tid);
        bool get(thread_id& tid);
    private:
        ...
    };
};
```

Above, `sys::thread_id` is an abstract type for a thread identifier. The method `sys::suspend()` causes the calling thread to be removed from execution until `sys::resume()` is called with the thread identifier of the thread which previously called `sys::suspend()`. Also, `sys::atomic.end()` is invoked as a part of `sys::suspend()`.

A thread can obtain its own thread identifier by calling `sys::self()`. The class `sys::thread_set` represents a set of threads; a thread can be added to the set with `sys::thread_set::put()` and a previously added thread removed from the set with `sys::thread_set::get()`. In case the set is empty, `sys::thread_set::get()` returns `false`, otherwise `true` with the argument set to the thread removed from the set. As with `sys::thread_id` we don't care about the implementation details of `sys::thread_set`. With these and the previously introduced `sys::atomic` class we can now revise the mutex implementation as follows:

```

class mutex
{
public:
    void lock()
    {
        sys::atomic.begin();

        while (locked == true)
        {
            waiting.add(sys::self());
            sys::suspend();
            sys::atomic.begin();
        }

        locked = true;

        sys::atomic.end();
    }

    void unlock()
    {
        sys::thread_id next;

        sys::atomic.begin();

        locked = false;

        if (waiting.get(next) == true)
            sys::resume(next);

        sys::atomic.end();
    }

private:
    bool locked;
    sys::thread_set waiting;
};

```

In `lock()` we first test the `locked` flag. In case it is already set we add the calling thread to the set of threads waiting for this mutex and cause it to be removed from execution with `sys::suspend()`. Otherwise, the flag is set, and the caller of `lock()` may proceed with its critical section. When done, it calls `unlock()`, which checks if there are threads waiting for this mutex. If so, one of them is returned to execution with a call of `sys::resume()`. We have used `sys::atomic` to make the bodies of `lock()` and `unlock()` atomic operations. This is crucial, because shared data - `lock` and `waiting` - is inspected and modified. The shared data also implicitly includes a data structure akin to `sys::thread_set` representing the set of running threads; this is modified by the underlying implementation. Consequently we must include `sys::suspend()` and `sys::resume()` in the atomic operations, otherwise the logically

absurd situation where a thread is simultaneously waiting and running might arise.

We have thus derived a general purpose mutual exclusion facility from the underlying concept of an atomic operation. The essential ingredients for the above construction can be found in (Loopstra 1959b, 343), (Codd 1962) and (Dijkstra 1965a).

It is, however, impractical to construct a multi-CPU system with a global facility for disabling interrupts. Also, the other CPUs would continue to run regardless, thereby invalidating mutual exclusion. Again, however, we can expect the hardware to come to the rescue, this time with a *test-and-set* machine instruction. As a high level interface to this facility and an illustration of the concept, consider the function

```
bool sys::test_and_set(bool& flag)
{
    if (flag == true)
        return true;
    else
    {
        flag = true;
        return false;
    }
}
```

where the function body is one atomic operation. With `sys::test_and_set()` we can replace `sys::atomic` as follows:

```
namespace sys
{
    class atomic
    {
    public:
        static void begin(void)
        {
            while (sys::test_and_set(spin) == true)
                while(spin == true);
        }

        static void end(void)
        {
            spin = false;
        }

    private:
        static bool spin;
    };
};
```

With this our previous implementation of a mutex has been ported from a single-CPU system to a multi-CPU one. This illustrates the important point that it is essentially immaterial to the logical considerations thereof whether a population of cooperating threads executes on a single or a multi-CPU system (Dijkstra 1965a, 82): although we had to cater for the difference in the fundamental hardware method of obtaining mutual exclusion as an implementation detail, the interface and the concept of the mutex remained unchanged. (Bovet & Cesati 2006; Comer 2005; Paterson & Hennessy 2005; Tanenbaum 2001).

The alert reader may now wonder whether we have actually implemented efficient waiting: the nested loops making up the body of `sys::atomic::begin()` appear to result in unmitigated busy waiting. However, for modern multi-CPU systems with a per-CPU, coherent cache this implementation of a *spinlock* is actually very efficient. *Cache coherency* means keeping the view of the main memory of each CPU consistent despite caching. This can be achieved by *snooping*: essentially, hardware on each CPU monitors the bus traffic between the other CPUs and the main memory and updates the cache as needed. When a CPU with a local, coherent cache finds `spin` to be `true` upon entry to `sys::atomic::begin()`, the resulting repeated reads of `spin` in the inner `while` loop are satisfied from the cache, until the CPU that set `spin` to `true` sets it back to `false` in `sys::atomic::end()`. As the machine instructions making up `sys::atomic::begin()` also get cached, the waiting CPU doesn't use the bus unnecessarily, and so the performance of other CPUs is not degraded. The computational power of the waiting CPU is still wasted while looping and might be more profitably used by switching to some other thread. However, the overhead of a context switch is likely to be far greater than the average overhead of a spinlock. Provided that the critical sections protected by spinlocks are kept short and few, using them is an attractive solution in that it requires no special hardware; a coherent local cache is desirable in any case. (Bovet & Cesati 2006; Comer 2005; Mellor-Crummey & Scott 1991; Paterson & Hennessy 2005; Tanenbaum 2001).

3.4 History

In this section we return to our historical theme. In section 3.4.1 we make a further remark on hardware development. This has been postponed from section 2 with the rationale that it may be better dealt with after the fundamental concepts of synchronization have been established. In sections 3.4.2 and 3.4.3 we explore the history of multi-threading and synchronization, in particular that of the terms and concepts thereof.

3.4.1 A further remark on hardware

In the original EDVAC design it was necessary modify the address fields of the instructions in memory to achieve conditional jumps and indirect memory access (Neumann 1945). In a multi-thread environment this arrangement results in the instructions thus modified becoming shared data, which means that they should either be protected by critical sections, or that such code cannot be shared.

As a part of its most rudimentary computing facilities the SSEM included an instruction to skip the next instruction, if the accumulator was less than zero (Williams & Kilburn 1948 , 415). Modifying code words at run-time was thus unnecessary to achieve a conditional jump. Indirect memory access without self-modifying code became available with the Manchester Mark I: the first version of this machine had two *index registers* for this very purpose (Lavington 1978, 6; Napper 1998, 371). These were known as "B-lines"; they were physical lines on the surface of the CRT tubes used to implement the memory. (Lavington 1978; 1993; Napper 1998).

3.4.2 Multi-threading and synchronization

According to (Codd 1962, 81) the earliest use of the term "multiprogramming" was in (Rochester 1955); we haven't been able to locate an earlier reference either. This original usage referred to a technique where instead of solely polling for the completion of an i/o-operation as a part of one job the CPU would interlace the polling with running another job, thereby making use of the computational capability of the CPU while waiting (Rochester 1955, 66). It is mentioned in the discussion section of (Rochester 1955) that the making use of an interrupt instead of polling had been

considered during the development of the IBM equipment discussed (Rochester 1955, 69).

(Gill 1958) introduces the term "parallel programming" in essentially the same sense that we have been using multi-threading. "Time-sharing" using interrupts and multiple CPUs are recognized as its hardware basis (Gill 1958, 2, 5). The potential of "mutual interference" of threads executing on different CPUs sharing the same memory is noted (Gill 1958, 3), as is the fact that "very similar logical problems can occur" (Gill 1958, 3) in the context of time sharing a single CPU using interrupts (Gill 1958, 5).

However, the nature of these problems is not elaborated, nor is their solution discussed, except for introducing the concept of FORK/JOIN (Conway 1963), a suggestion for a memory protection system (Gill 1958, 7) and a note of the necessity of preserving the CPU state as a part of a context switch (Gill 1958, 7). The possibility of combining interrupt based time sharing with multiple CPUs is mentioned (Gill 1958, 6) as is the idea of *swapping* programs and their data between the main memory and a backing store (Gill 1958, 7). In addition of making use of the CPU(s) while i/o- operations are in progress, parallel programming is envisioned to be useful in real-time applications such as the control of a chemical plant (Gill 1958, 7-8). Arguably, (Gill 1958) can be considered to have been the first to expose what in our view is the core problem of multi-threading, that is, the emergence of indeterminism from variables shared by two or more threads; however, the key expression, "... there is a need for control links to prevent one computer *anticipating* another" (Gill 1958, 3), is rather ambiguous.

The first documented instance of multi-threading in practice we have been able to locate is a demonstration run with the Dutch X-1 computer (Loopstra 1959a; 1959b), with two applications time-sharing the CPU making use of automatic switching based on interrupts (Loopstra 1959a, 43). (Loopstra 1959b) discusses the interrupt system of the X-1 as the basis of organizing the use of the i/o facilities of the machine. As a part of this discussion we find the crucial observation that at certain times interrupts must be disabled (or, rather, postponed). In particular, an example is given where a counter variable is shared between the interrupt handler and the main program; a facility for disabling interrupts under program control is deemed necessary in this context.

(Loopstra 1959b, 343). Again, this can be interpreted as an exposition of our core problem, complete with a concrete solution applying to interrupt handlers in a single-CPU system.

(Strachey 1959) points out that a special purpose buffering unit, such as the IBM Tape Record Coordinator in (Rochester 1955) can be replaced with interrupt driven i/o; such units tend to approach the computer itself in complexity and, therefore, price.

Furthermore, the extensive use of time sharing is suggested for a paramount advantage in economy and utility: a fast and expensive computer with a large memory can be used not only for the most computationally demanding problems, but also for a number less demanding, even trivial ones sharing the machine - at the same time it is possible to share the machine among multiple users. The essential hardware requirements are the inclusion of an interrupt mechanism and a memory protection facility. The rest can be arranged with some permanently resident software, the "Director". A pair of limit registers under the sole control of the Director is suggested as the chief means of memory protection. This constitutes a fundamental plan for a multi-tasking, multi-user operating system, where, in modern terms, the Director is the operating system kernel. Assigning execution priorities is discussed; whether these are to apply to application programs as well as interrupts is somewhat unclear, though. The concept of a virtual machine is implied in: "To each of the operators the machine appears to behave as a separate machine (smaller, of course, and slower than the real machine)." (Strachey 1959, 340). (Strachey 1959).

A major goal of memory protection in combination of an operating system is to provide a private address space to each program running on the system, and it may thus seem like an antithesis of our premise of shared address space. However, the operating system as laid out in (Strachey 1959) must have access to all the resources of the machine including the all the memory to carry out its function; interrupt handling is also in the domain of the operating system. (Codd 1962). Therefore multi-threading considerations are central in the design and implementation of such an operating system.

(Codd & al. 1959) makes further important points on multi-tasking operating systems. It should be possible to design and implement the individual programs sharing the facilities of a system in isolation from each other (Codd & al. 1959, 14). Using a hardware interval timer generating interrupts is discussed as a means of imposing time limits on individual programs (Codd & al. 1959, 15). This is a direct precursor to the concept of the virtual processor (Dijkstra 1967; Salzer 1966).

The vision of a multi-tasking operating system was soon put in practice. Two early implementations were the Manchester University Atlas operating system (Kilburn & al. 1961; Kilburn & Payne, 1961; Lavington S. 1978; 1993; Morris & al. 1967) and the CTTS from the MIT "Real-Time, Time-Shared Computer Project" (Corbato & al. 1962; Corbato & al. 1963; Teager 1962).

3.4.3 Fiat lux

(Codd 1962) offers a comprehensive recap and elaboration of the issues discussed in the papers in the previous section, and, in particular, provides a considerable new insight on the issues pertaining to multiple CPUs. The concept of multi-threading essentially in the sense that it exists in modern operating systems is introduced: an operating system process may include more than one concurrently executing thread sharing the same address space. A concrete, detailed plan for the implementation for such an operating system is given. This time it is unambiguously stated that a priority scheme applies to the user programs (Codd 1962, 125). (Codd 1962, 122-150).

In a multi-CPU system "a ... requirement is a means of preventing more than one " [C]"PU from concurrently modifying an item in memory." (Codd 1962, 124). An example of two CPUs concurrently decrementing a counter follows, with the familiar result of the counter becoming effectively decremented only once. "To overcome this difficulty it is necessary to provide in the system for temporarily blocking ... entry by more than one " [C]"PU into selected sections of code." (Codd 1962, 124). This is exactly what we have deemed the core problem of multi-threading, complete with the solution by critical sections, that is, mutual exclusion. Strictly speaking the problem and the solution have been given in the context of a multi-CPU system, though. Then

again it is recognized that: "For the most part, however, the problems encountered are similar" [in both single and multi-CPU systems] (Codd 1962, 78).

A set of special operations is made available to the user programs by the operating system. **START** is used to create an additional thread. **STOP** may be invoked by a thread to terminate itself. **BLOCK** and **UNBLOCK** are used to enter and leave a critical section, respectively. **GET** and **PUT** are used to invoke i/o-operations and include an integrated wait-signal facility used by the operating system to suspend and resume threads, so that a thread can wait for the completion of i/o operation removed from execution. Underlying **BLOCK** and **UNBLOCK**, in fact, there is the concept of the mutex, or a binary semaphore. Therefore Codd's plan for an operating system includes a universal synchronization primitive. Ultimately, the implementation of mutual exclusion in the system rests on a hardware arrangement whereby only one CPU at a time can execute *supervisor*, that is, operating system kernel code. (Codd 1962, 130-134).

(Dijkstra 1965a) begins with a thorough exposition of the concept of the "sequential process"; the crucial observations are that a program execution on a computer is such a process and that "its only essential feature is that its elementary steps are performed in sequence" (Dijkstra 1965a, 10). It is then stated that "the subject matter of this monograph is the cooperation between loosely connected sequential processes" (Dijkstra 1965a, 10). By loosely connected it is meant "that apart from the (rare) moments of explicit intercommunication, the individual processes themselves are to be regarded as completely independent of each other. In particular we disallow any assumption about the relative speeds of the different processes." (Dijkstra 1965a, 10). A problem is then posed in terms of such cyclic processes communicating through shared memory. It is stipulated that the accesses to this shared memory are "indivisible", that is, excluding each other in time. The problem is to construct a cyclic program, where "mutual exclusion" exists; only one of the processes may be executing the "critical section" of the program at any given time. The problem is first discussed in terms of two processes, and the solution, credited to Th. J. Dekker, presented. A general solution for an arbitrary number of processes follows. (Dijkstra 1965a, 11-22). The general

problem and its solution has been separately published as (Dijkstra 1965b) with a proof of the solution, which in (Dijkstra 1965a) was left to the reader.

In (Dijkstra 1965a) it is admitted that the solution to the mutual exclusion problem explained above is of limited practical value. In particular, the "busy way of waiting" employed is deemed impractical as it is wasteful in consuming processor time and bus bandwidth, which might be profitably used by other processes. (Dijkstra 1965a, 26-28). The semaphore is presented as a practical solution to the mutual exclusion problem. A distinction is made between "general semaphores" and "binary semaphores", the latter being a special case of the former. C. S. Scholten is credited "to have demonstrated a considerable field of applicability for" (Dijkstra 1965a, 28) general semaphores. The term "synchronizing primitive" is introduced with the semaphore. (Dijkstra 1965a, 28-30). A thorough discussion of the application of the semaphore with numerous programming examples follows. The term "mutex" is effectively introduced as the name of a binary semaphore variable used to obtain mutual exclusion (Dijkstra 1965a, 51). "The purpose of the critical sections is to resolve any ambiguity in the inspection and modification of the" [shared] "variables" (Dijkstra 1965a, 64). The concept of "deadly embrace", or deadlock is introduced (Dijkstra 1965a, 73-75), and in the concluding remarks the point is made that it is immaterial to the logical considerations thereof whether a population of cooperating sequential processes, or threads executes on a single or a multi-CPU system. (Dijkstra 1965a, 82).

It is surprising to note that none of the well-known, excellent textbooks on operating systems and concurrent programming of (Ben-Ari 2006), (Brinch Hansen 1973), (Madnick & Donovan 1974), (Tanenbaum 2001) and (Silberschatz & al. 2005) don't include (Codd 1962) in their references / bibliography, whereas all except (Ben-Ari 2006) duly include (Dijkstra 1965a). The point we wish to make with this observation is that the early history of multi-threading doesn't appear to be quite as well known as one might hope, and that further research of this subject would therefore seem to be well warranted.

4 Evaluating concrete synchronization primitives

This section discusses the concrete, well-known synchronization primitives the semaphore, the monitor and the POSIX mutex / condition variable. A concise description of each is given followed by an evaluation of their universality based on the criteria established in the previous section.

4.1 Semaphore

Below the semaphore is presented as a partial C++ class declaration:

```
class semaphore
{
public:
    semaphore(int init) : count(init) { ... }
    void P(void);
    void V(void);

private:
    int count; // invariant: count >= 0
    ...
};
```

The operations $P()$ and $V()$ are defined as follows:

$P()$: If $count > 0$ decrement $count$ by one, else remove the calling process from execution and add it to the set of processes waiting for this semaphore.

$V()$: Increment $count$ by one. If the set of processes waiting for this semaphore is not empty, remove one process from the set and cause that process to resume execution.

It is stipulated that the operations $P()$ and $V()$ must be indivisible in time and that $count$ must be non-negative at all times. (Dijkstra 1965a).

(Dijkstra 1965a) presents a binary semaphore as a special case of the semaphore (general semaphore) described above where $count$ is restricted to the values of 0 and 1. It is demonstrated in (Dijkstra 1965a) that semaphores can be constructed from binary semaphores. A binary semaphore is equivalent to a mutex as presented in section 3.3. The concept of a mutex and therefore that of the binary semaphore is due to (Codd

1962).

The fact that the semaphore includes the counter result in a desirable feature of its waiting semantics: if no thread is currently waiting at the time of $v()$, the first thread subsequently entering $p()$ will return immediately. This is an important point as otherwise a *lost wakeup* could occur. That is, if waiting and waking up are implemented in terms of modifying the sets of running and waiting threads only, it could happen that a thread that is about register itself as waiting is preempted by another, which finds that the set of threads waiting for this instance of the wait-signal facility is currently empty. Consequently, when the first thread is eventually resumed it completes its registration as waiting and removes itself from the set of running threads. The result is that it may never be resumed although the operation to enter the wait was started before the other thread's attempt for a wakeup. (Dijkstra 1971).

Above, we have used the original convention of the operations P and V of (Dijkstra 1965a). It has since become common to use wait for P and signal for V (Ben-Ari M. 2006; Brinch Hansen 1973; Silberschatz & al. 2005).

4.2 Monitor

A monitor is a combination of the programming language concept class, mutual exclusion and the wait-signal facility: it is a special kind of a class, where the executions of the (public) method bodies are critical sections protecting the data of a monitor instance, which can be accessed in class methods only, as can the related instances (if any) of the intimately related *condition variables*, an implementation of the wait-signal facility. (Hoare 1974).

In the context of a monitor mutual exclusion could be implemented in terms of a mutex from section 3.3 as a hidden instance variable of the monitor, with `mutex::lock()` invoked just before entering a (public) method and `mutex::unlock()` invoked just before the control returns to the caller. Assuming that the underlying concept of a class has private methods these don't need similar protection as they are accessible from the public methods only.

To illustrate the concept of a monitor, let us pretend that we can use the keyword `monitor` that is equivalent to `class` with the distinction that there is a hidden mutex among the instance variables, which is used to obtain mutual exclusion as described above, and that the class `condition_variable` is available with this interface:

```
class condition_variable
{
public:
    void wait(void);
    void signal(void);
    ...
};
```

A call to `wait()` causes the calling thread to be removed from execution and added in the set of threads waiting on this particular instance of the condition variable. A call to `signal()` causes one of the threads in this set to be removed from the set with the resumption of its execution. The `condition_variable` class also has implicit knowledge of the hidden mutex of the `monitor`. This is used to release the critical section as an atomic part of `wait()` and to reclaim the critical section before returning from `wait()` as a result of a call to `signal()` by some other thread. We also postulate that all member variables must be `private`, and that a `monitor` may not be used as a base class, and therefore `protected` members are forbidden in a `monitor`. The former restriction is a part of the `monitor` concept, while the latter is merely a simplification to facilitate focusing on the most relevant. (Hoare 1974).

With the above we can construct a semaphore in terms of a `monitor` as follows:

```

monitor semaphore
{
public:
    semaphore(int init) : count(init)
    {
        ...
    }

    void P(void)
    {
        if (count == 0)
            cond.wait();

        --count;
    }

    void V(void)
    {
        ++count;
        cond.signal();
    }

private:
    int count;
    condition_variable cond;
    ...
};

```

Above we make the assumption that the underlying condition variable has the exact semantics of (Hoare 1974), that is, upon a call to `signal()`, the thread thus waken up is directly passed the critical section from the thread that called `signal()`; there is no possibility of any other thread intervening. Therefore, it is assured that `count` is in fact positive upon return from `wait()` in `P()`. (Hoare 1974).

We have used the monitor in the sense that is has been presented in (Hoare 1974). However, the essence of the above example can be found in (Hoare & Perrot 1972, 110) as an example presented in a seminar discussion by Brinch Hansen. From this discussion (Hoare & Perrot 1972, 109-113) it is apparent that the concept of the monitor was in the process of being gradually refined in collaboration between Brinch Hansen, Hoare, and, to some extent, Dijkstra; "the development of the monitor concept is due to frequent discussions and communications with E.W. Dijkstra and P. Brinch-Hansen" (Hoare 1974, 557). A detailed account of the development process can be found in (Brinch Hansen 1993). Because of the gradual development it is not easy to say what exactly constitutes a monitor. In fact, the refining of the concept has continued after (Hoare 1974) as evidenced by (Howard 1976).

As the condition variable in (Hoare 1974) is implemented in terms of semaphores it has similar waiting semantics: if no thread is currently waiting at the time of `signal()`, the first thread subsequently entering `wait()` will return immediately. Consequently, a lost wakeup is not an intrinsic problem. (Hoare 1974).

4.3 The POSIX mutex and condition variable

The POSIX mutex and condition variable are defined as a part of *pthread* ('Posix threads') -specification (IEEE 2004). Because this definition in its entirety is quite extensive, only the fundamentals of the POSIX mutex and condition variable are presented.

4.3.1 The POSIX mutex

Below the POSIX mutex is presented as a partial C++ class declaration.

```
class posix_mutex
{
public:
    void lock();
    void unlock();
    ...
};
```

The POSIX mutex is a synchronization primitive that provides for mutual exclusion where the critical section to be protected by the POSIX mutex is enclosed between invocations of the `lock()` and `unlock()` operations:

```
posix_mutex m;
...
m.lock();

// critical section
...
m.unlock();
```

While a thread is within the critical section (owns the mutex) other threads calling `lock()` on the same mutex will be removed from execution until the thread within the critical section calls `unlock()` on the mutex (releases the mutex). That is, the POSIX mutex is equivalent to the mutex introduced in section 3.3. (IEEE 2004).

4.3.2 The POSIX condition variable

Below the POSIX condition variable is presented as a partial C++ class declaration.

```
class posix_condition_variable
{
public:
    void wait(posix_mutex& m);
    void signal();
    ...
};
```

The POSIX condition variable is a synchronization primitive that a thread can wait on, removed from execution until another thread signals it to continue. Multiple threads can wait on a single condition variable. The argument to `wait()` specifies a POSIX mutex that must be owned by the calling thread. Invoking `wait()` causes the mutex to be released and the calling thread to be removed from execution in an atomic operation. Before returning `wait()` invokes `lock()` on the mutex; after the call the mutex is again owned by the thread that made the call. If there are threads waiting on a condition variable, calling `signal()` causes one of the waiting threads to resume execution. If no threads are waiting at the time `signal()` is called the call has no effect. The condition of the condition variable is expressed in code, for example:

```
posix_mutex mtx;
posix_condition_variable cnd;
...
mtx.lock();
...
while (...) // <- condition
    cnd.wait(mtx);

// condition is present: handle it
...
mtx.unlock();
...
```

The loop above is mandated by the pthreads-specification (IEEE 2004) as a *spurious wake-up*, that is, a return from a call to `wait()` like the above without a corresponding call to `signal()`, is allowed in an implementation. In general, the above construct is necessary even without spurious wake-ups as a thread other than those involved in the signaling could make the condition false before the thread that called `wait()` can obtain the related mutex and return, even if the data on which the condition is based is

protected by the mutex (as it should be). (IEEE 2004).

The occurrence of a lost wakeup is prevented by the requirement that a POSIX condition variable must be associated with a POSIX mutex, and that the mutex must be locked upon entry to `wait()`. With this arrangement the underlying data structure representing the set of threads waiting on a POSIX condition variable is implicitly protected by a related POSIX mutex, and therefore another thread cannot make an attempt to wake a thread up while it is in the process of being added to the set of threads waiting for a POSIX condition variable.

In the following example we implement the semaphore in terms of the POSIX mutex and condition variable.

```
class semaphore
{
public:
    semaphore(int init) : count(init)
    {
        ...
    }

    void P()
    {
        mtx.lock();

        while(count == 0)
            cnd.wait(mtx);

        --count;

        mtx.unlock();
    }

    void V()
    {
        mtx.lock();
        ++count;
        cnd.signal();
        mtx.unlock();
    }

private:
    int count;
    posix_mutex mtx;
    posix_condition_variable cnd;
};
```

4.4 Discussion

It is demonstrated in (Dijkstra 1965a) that a general semaphore can be used to effect mutual exclusion. This is formally proved in (Dijkstra 1971, 125-126). The semaphore also directly implements a wait-signal facility. It can therefore be considered a universal synchronization primitive.

The monitor provides mutual exclusion. The condition variable used in conjunction with the monitor is a wait-signal facility. So the two together can be considered to be a set of universal synchronization primitives.

A POSIX mutex provides for mutual exclusion. Used in conjunction with a POSIX condition variable the two provide the wait-signal facility, so the POSIX mutex and condition variable can be considered to be a set of universal synchronization primitives.

On the premise that the semaphore is a universal synchronization primitive, a set of synchronization primitives that can be used to construct a semaphore can obviously replace the semaphore and is therefore also a universal set of synchronization primitives. Likewise, the scope of the formal proof from (Dijkstra 1971, 125-126) that semaphores can effect mutual exclusion can be extended to cover such a set of synchronization primitives. The 'THE'-multiprogramming system, an early, yet advanced multi-tasking operating system, uses semaphores as its only means of synchronization (Dijkstra 1967). This lends further credence to the the statement that "... a semaphore can be used to solve arbitrary synchronization problems" (Brinch Hansen 1973, p. 96), that is, a semaphore is a universal synchronization primitive.

Generalization: Let S_0 and S_1 be sets of synchronization primitives. If the synchronization primitives in set S_0 can be constructed from the synchronization primitives in set S_1 , the synchronization primitives in set S_0 can be replaced by the said constructs. Therefore, if set S_0 has a property P , set S_1 also has the property P .

The *solution space* of a set of synchronization primitives S is defined as the set solutions to synchronization problems that can be expressed in terms of S . Let P_0 be the

solution space of S_0 and let P_1 to be the solution space of S_1 . In the context of the above generalization $P_0 \subset P_1$, that is, the solution space of S_1 is at least equal to that of S_0 . If the synchronization primitives in S_1 can also be constructed from the synchronization primitives in S_0 , it follows that $P_1 \subset P_0$. Therefore, $P_0 \subset P_1 \wedge P_1 \subset P_0 \Rightarrow P_0 = P_1$, that is, the solution spaces of S_0 and S_1 are equal.

A general semaphore can be constructed using binary semaphores (Dijkstra 1965a, 35-39). The converse is also true: since the binary semaphore is a special case of the general semaphore, the former can obviously be implemented in terms of the latter. Therefore, the binary semaphore can be considered an universal synchronization primitive in its own right, and the solutions spaces of the two synchronization primitives are equal.

Hoare demonstrates that a semaphore can be implemented in terms of monitors and condition variables (Hoare 1974, 550) and that monitors and condition variables can be implemented using semaphores (Hoare 1974, 551). A construction demonstrating the first point has also presented under section 4.2. Therefore, on the premise that the semaphore is an universal synchronization primitive the monitor and the condition variable are a set of universal synchronization primitives. Also, the solution spaces of this set and the semaphore are equal.

In section 4.3 we have presented an implementation of the semaphore in terms of the POSIX mutex and condition variable. Therefore, on the premise that the semaphore is an universal synchronization primitive the POSIX mutex and condition variable are a set of universal synchronization primitives. Also, the solution space of this set is at least equal to that of the semaphore.

5 Conclusion

We have presented a historical overview of the development of the computer in the modern sense. While the role of Babbage appears to be well-known, we feel that the pioneering work roughly coinciding the second world war is less so. A case in point: as late as 2006 we can find the the suggestion of the ENIAC being the first electronic digital computer in such a venerable forum as Nature (Ruttiman 2006, 399). While this is arguably correct in a strict sense, it is prone to leave the reader with the incorrect impression that the ENIAC was the first computer. In fact, the concrete realization of Konrad Zuse's work as the Z3 predates the ENIAC.

The EDVAC-architecture is an important milestone: it embodies the concept of the computer in the modern sense. In connection to this we have discussed von Neumann's role in its development: while he certainly was one the key contributors, his role was not quite as central as the still widely used term "von Neumann architecture" implies.

In our discussion the EDVAC architecture is the baseline of later developments, in particular, the introduction of the interrupt and multiple CPUs. These constitute the necessary hardware basis of multi-threading, but at the same time give rise to race conditions that we have deemed the core problem of multi-threading: race conditions result in losing the determinism of operation that makes a computer useful in the first place. Race conditions can be eliminated with the introduction of mutual exclusion. A practical implementation of mutual exclusion requires a wait-signal facility, the means for efficient waiting, also useful in its own right. This solution comes at the price of additional problems: deadlock, livelock, starvation, priority inversion and lost wakeup. These can be dealt with by the correct application and implementation of synchronization.

Mutual exclusion is sufficient to eliminate race conditions, the problem that ultimately makes multi-thread programming different from single-thread programming. In this we have found a pragmatic criteria for the universalness of synchronization primitives: a set of synchronization primitives providing mutual exclusion and a wait-signal facility can be considered universal in the sense that it can be used to deal with all practical

synchronization problems.

We have discussed the implementation issues of synchronization through examples gradually refining an implementation of the synchronization primitive mutex. Here, elementary mutual exclusion provided by the hardware was combined with effective waiting implemented with facilities provided by the underlying thread implementation. In a single-CPU system the hardware method of obtaining mutual exclusion is temporarily disabling interrupts, while an atomic test-and-set machine instruction in combination of a spinlock is used for multi-CPU implementation. Although the ultimate method of obtaining mutual exclusion differs, the upper levels, and crucially, the external interface of the implementation are the same in both cases. This illustrates the point that it is effectively irrelevant whether multi-threading is implemented in terms of a single or multiple CPUs.

We then return to our historical theme by providing an exposition of the early history of multi-threading and synchronization. In particular, the history of the terms and concepts previously introduced is discussed. In this context we note that (Codd 1962) appears to have been the first to expose the problem of race conditions and their solution by mutual exclusion. This, in particular, appears to be poorly known. We suspect that the same applies to the early history of multi-threading and synchronization in general; we feel that further study thereof would be well warranted.

Finally, we have discussed the well-known, concrete synchronization primitives the semaphore, the monitor and the POSIX mutex / condition variable. As each provides mutual exclusion and the wait-signal facility they are all deemed universal by the above universality criteria. We have also observed that if a set of synchronization primitives can be implemented in terms of another such set, then the latter can be used to replace the former and therefore has a solution space at least equal to that of the one replaced. As semaphores and monitors can be implemented in terms of each other their solution spaces are equal. The POSIX mutex / condition variable can be used to implement the semaphore, and therefore has a solution space at least equal to that of the semaphore and the monitor. Implementing the POSIX mutex / condition variable in terms of the

semaphore or the mutex would complete the loop. However, this is not straightforward as the wait semantics differ, and so this is suggested as an issue for further research: it might be possible to do this by making use of the equivalence of different wait semantics of monitors as presented in (Howard 1976).

As the motivation of our pragmatic approach we have stated that no formal exposition on the issue of universality of synchronization primitives seems to exist. If this is indeed the case, then providing such a formal treatment in terms of which this problem and its solution could be posed is an obvious suggestion for further research. This could, perhaps, be obtained on the basis of (Lamport 1986a; 1986b). Ultimately, the problem with such an approach rooted in the fundamentals of physics would seem to be that for the moment general relativity and quantum mechanics cannot be fully reconciled; either one or both of these fundamental theories will probably need to be refined to make this possible. (Greene 1999; Weinberg 1986).

References

Aiken, H. 1937, Proposed automatic calculating machine. Reprinted in (Randell 1982), p. 195-201.

Aiken, H. & Hopper, G. 1946a, The Automatic Sequence Controlled Calculator - I, IEEE: Electrical Engineering 65(8-9), p. 384-391. Reprinted in (Randell 1982), p. 203-210.

Aiken, H. & Hopper, G. 1946b, The Automatic Sequence Controlled Calculator - II, IEEE: Electrical Engineering 65(10), p. 449-454. Reprinted in (Randell 1982), p. 210-216.

Aiken, H. & Hopper, G. 1946c, The Automatic Sequence Controlled Calculator - III, IEEE: Electrical Engineering 65(11), p. 522-528. Reprinted in (Randell 1982), p. 216-222.

Alt, F. 1948a, A Bell Telephone Laboratories Computing Machine - I, Mathematical Tables and Other Aids to Computation, 3(21), p. 1-13. Reprinted in (Randell 1982), p. 263-276.

Alt, F. 1948b, A Bell Telephone Laboratories Computing Machine - II, Mathematical Tables and Other Aids to Computation, 3(22), p. 69-84. Reprinted in (Randell 1982), p. 277-292.

Anderson, J., Hoffman, S., Shifman, J. & Williams, R. 1962, D825 -- A multiple computer system for command and control, Proceedings of the FJCC 1962, p. 86-96.

Ashcroft E. 1975, Proving assertions about Parallel Programs, Journal of computer and system sciences 10(1), p. 110-135.

Aspray W. 1990, MIT Press, Cambridge, Massachusetts: John von Neumann and the

Origins of Modern Computing.

Babbage, C. 1864, Of the Analytical Engine. Reprinted in (Pylyshyn 1970), p. 16-28.

Blaauw G. 1964, The Structure of System/360 Part V - Multisystem Organization
IBM Systems Journal 3(2), p. 181-195.

Blaauw G. & Brooks F. 1964, The structure of SYSTEM/360, Part I: Outline of the
logical structure, IBM Systems Journal 3(2), p.119-135.

Bashe C., Buchholz W., Hawkins G., Ingram J. & Rochester N. 1981, The Architecture
of IBM's Early Computers, IBM Journal of Research and Development 25(5), p. 363-
376.

Bauer, F. 1980, Between Zuse and Rutishauser - The Early development of Digital
Computing in Central Europe, Academic Press, New York: Metropolis, N., Howlett J.
& Rota C. 1980, A History of Computing in the Twentieth Century, A collection of
essays., p. 505-523.

Ben-Ari M. 2006, Addison-Wesley, Harlow, England: Principles of Concurrent and
Distributed Programming, second edition.

Bovet D. & Cesati M. 2006, O'Reilly, Sebastopol, CA: Understanding the Linux
Kernel, 3rd edition.

Brinch Hansen P. 1973, Prentice-Hall, Englewood Cliffs, N.J.: Operating System
Principles.

Brinch Hansen P. 1993, Monitors and Concurrent Pascal: A personal history. ACM
SIGPLAN Notices 28(3), p. 1-35.

Burks A., Goldstine H. & Neumann J. 1946, Preliminary discussion of the logical

design of an electronic computing instrument. Reprinted in (Pylyshyn 1970), p. 37-46.

Cesareo, O. 1946, The relay interpolator, Bell laboratories record, volume 23, p. 457-460. Reprinted in (Randell 1982), p. 253-256.

Codd E. 1962, Multiprogramming, Academic Press, New York: Alt F.L. & M. Rubinoff M. (eds.), Advances in Computers, Vol. 3, p. 77-153.

Codd E., Lowry E., McDonough E. & Scalzi C. 1959, Multiprogramming STRETCH: Feasibility Considerations , Communications of the ACM, 2(11), p. 13-17.

Comer, D. 2005, Pearson, Prentice Hall, Upper Saddle River, New Jersey: Essentials of Computer Architecture.

Comrie L. 1946, Babbage's Dream Comes True, Nature, volume 158, p. 567-568.

Conway M. 1963, A multiprocessor system design, Proceedings of the Fall Joint Computer Conference, 1963, p. 139-146.

Corbato F., Dagget M. & Daley R. 1962, An experimental time-sharing system, Proceedings of the AFIPS 1962 Fall Joint Computer Conference, p. 335-344.

Corbato F., Dagget M., Daley R., Creasy R., Hellwig J., Orenstein R. & Korn L. 1963, MIT Press: The Compatible Time-Sharing System, A Programmer's Guide.

Desmonde, W. & Berkling K. 1966, The Zuse Z3, Datamation 12(9) p. 30-31.

Dijkstra, E. 1965a, Cooperating Sequential Processes, Technological University, Eindhoven, The Netherlands.

Dijkstra, E. 1965b, Solution of a problem in concurrent programming control, Communications of the ACM, 8(9) p. 569.

Dijkstra, E. 1967, The structure of the 'THE'-multiprogramming system, Proceedings of the first ACM symposium on Operating System Principles.

Dijkstra, E. 1971, Hierarchical ordering of sequential processes, Springer-Verlag, New York: Acta Informatica 1(2), p. 115-138.

Eckert, J. 1956, Univac-Larc, the next Step in Computer Design , Proceedings of the EJCC 1959, p. 16-20.

Eckert, J., Chu, J., Tonik, A. & Schmitt, W. 1959, Design of Univac LARC-System, Part I , Proceedings of the EJCC 1959, p. 59-65.

Einstein, A. 1905, Zur Elektrodynamik bewegter Körper, Annalen der Physik, 322(10), p. 891-921.

Einstein, A. 1954, Routledge, London: Relativity, The Special and the General Theory, 15th edition (1996).

Elbourn, R. & Witt, R. 1953 , Dynamic Circuit Techniques Used in SEAC and DYSEAC , Proceedings of the IRE 41(10) p. 1380-1387.

Enslow P. 1974 (Ed.), John Wiley and Sons, New York: Multiprocessors and parallel processing.

Feynman, R. 1992, Random House, London: Surely you are joking Mr. Feynman.

Flynn, M. 1972, Some Computer Organizations and Their Effectiveness, IEEE Transactions on Computers 21(9) p. 948-960.

Gill S. 1958, Parallel Programming, The British Computer Society: The Computer Journal, 1(1), p. 2-10.

Goldstine H. & Goldstine A. 1946, The Electronic Numerical Integrator and Computer (ENIAC), M.T.A.C. 2(15), p. 97-110. Reprinted in (Randell 1981), p. 359-373.

Gray, G. 1999a, Some Burroughs Transistor Computers, Unisys History Newsletter 3(1), [Web-document]. Available:

<http://www.cc.gatech.edu/~randy/folklore/v3n1.html>. [Referenced 06. 11. 2007].

Gray, G. 1999b, The UNIVAC 1108, Unisys History Newsletter 3(2), [Web-document]. Available: <http://www.cc.gatech.edu/~randy/folklore/v3n2.html>.

[Referenced 26. 03. 2008].

Greene, B. 1999, Vintage, London: The Elegant Universe, Superstrings, Hidden Dimensions and the Quest for the Ultimate Theory.

Greenstadt J. 1957, The IBM 709 computer, Proceedings of the Symposium: New Computers, a Report from the Manufacturers (ACM), Los Angeles, March 1957, p. 92-96.

Gustafson, J. 1998, Reconstruction of the Atanasoff-Berry Computer in (Rojas & Hashagen 2000), p 91-106.

Halliday D. & Resnick R. 1986, John Wiley & Sons, New York: Fundamentals of physics, 2nd edition.

Hoare C. 1971, Towards a theory of parallel programming, in (Hoare & Perrot 1972), p. 61-71.

Hoare, C. 1974, Monitors: an operating system structuring concept, Communications of the ACM 17(10), p. 549-557.

Hoare C. & Perrot R. (eds.) 1972, Academic Press, London: Operating Systems Techniques.

Howard J. 1976, Signaling in monitors , Proceedings of the 2nd international conference on Software engineering (IEEE-CS, ACM/SIGSOFT, NBS), p. 47-52.

IEEE 2004, The Open Group Base Specifications Issue 6 / System Interfaces / Threads, IEEE: Std 1003.1.

ISO/IEC, 1998, International Standard 14882:1998, Programming Languages — C++.

Kilburn T., Howarth D., Payne R. & Sumner F. 1961, The Manchester University Atlas Operating System Part I: Internal Organization , The Computer Journal, 4(3), 222-225.

Kilburn T. & Payne R. 1961, The Atlas supervisor, Proceedings of the AFIPS 1961 Eastern Joint Computer Conference, p. 279-294.

Knuth, D. 1997, Addison-Wesley, Boston: The Art of Computer Programming, volume 1, 3rd ed.

Lampert, L. 1986a, The Mutual Exclusion Problem: Part I - A Theory of Interprocess Communication, Journal of the ACM 33(2), p. 313-326.

Lampert, L. 1986b, The Mutual Exclusion Problem: Part II - Statement and Solutions, Journal of the ACM 33(2), p. 327-348.

Lampson B. & Redell D. 1980, Experience with Processes and Monitors in Mesa, Communications of the ACM, 23(2), p. 105-117.

Lavington S. 1978, The Manchester Mark I and Atlas: A Historical Perspective, Communications of the ACM 21(1) p. 4-12.

Lavington S. 1993, Manchester Computer Architectures 1948-1975, IEEE Annals of the History of Computing 15(3), p. 44-54.

Leiner, A. & Alexander, S. 1954, System organization of DYSEAC, IRE Transactions on Electronic Computers, EC-3, p. 1-10.

Leiner, A. 1954, System specifications for the DYSEAC, Journal of the ACM 1(2) p. 57-81.

Loopstra B. 1959a, The X-1 Computer, The British Computer Society: The Computer Journal, 2(1), p. 39-43.

Loopstra B. 1959b, Input and output in the X-1 system, Unesco, Paris: Proceedings of the international conference on information processing, Unesco, Paris 15-20 June 1959, p. 342-344.

Lukoff, H., Spandorfer, L. & Lee, F. 1959, Design of Univac LARC-System Part II, Proceedings of the EJCC 1959, p. 66-74.

Madnick, S. & Donovan J. 1974, McGraw-Hill Kogakusha Ltd., Tokyo: Operating systems, International Student Edition.

Mellor-Crummey J. & Scott M. 1991, Algorithms for scalable synchronization on shared-memory multiprocessors, ACM Transactions on Computer Systems, 9(1), p. 21-65.

Mersel, J. 1956, Program interrupt on the UNIVAC scientific computer, Proceedings of the Western Joint Computer Conference 1956, p. 52-53.

Metropolis, N. & Worlton, J. 1972, A Trilogy on Errors in the History of Computing, Proceedings, First USA-Japan Computer Conference Tokyo, October 3-5. Reprinted in IEEE Annals of the History of Computing, 2(1), p. 49-59 (1980).

Metropolis, N. 1987, The Los Alamos Experience, 1943-1954 in (Nash 2000), p. 237-250.

Morris D., Sumner F. & Wyld M. 1967, An appraisal of the Atlas supervisor, ACM/CSC-ER Proceedings of the 1967 22nd national conference, p. 67-75.

Nash, S. (ed.) 2000, ACM: A History of Scientific Computing, proceedings of the Conference on the History of Scientific and Numeric computation, Princeton, 1987.

Napper R. 1998, The Manchester Mark 1 computers in (Rojas & Hashagen 2000), p 365-377.

Neumann, J. 1945, First Draft of a Report on the EDVAC. Reprinted in (Stern 1981), p. 177-246.

Newton I. 1676, A letter to Robert Hooke dated Feb. 5, 1675[6]. Reprinted in Turnbull H. (ed.), 1959, The Royal Society, Cambridge: The correspondence of Isaac Newton, volume I, 1661-1675, p. 416-417.

Newton, I. 1687, Royal Society, London: Philosophiæ Naturalis Principia Mathematica.

Padegs A. 1981, System/360 and Beyond , IBM Journal of Research & Development 25(5), p. 377-390.

Paterson, D. & Hennessy, J. 2005, Morgan Kaufmann Publishers, San Fransisco, CA: Computer organization and design, 3rd ed.

Pylyshyn, Z. (ed.) 1970, Prentice-Hall Inc., Englewood Cliffs, N.J.: Perspectives on the Computer Revolution.

Randell, B. 1977, Colossus: Godfather of the Computer, New Scientist, volume 73, issue 1038, p. 346-348. Reprinted in (Randell 1982), p. 349-354

Randell, B. (ed.) 1982, Springer-Verlag, Berlin: The Origins of Digital Computers, Selected Papers, 3rd ed.

- Renwick W. 1949, The EDSAC demonstration, Cambridge University Mathematical Laboratory: Report of a Conference on High Speed Automatic Calculating Machines 22-25 June 1949, p. 9-11. Reprinted in (Randell 1982), p. 423-429.
- Rochester, N. 1955, Computer and its peripheral equipment, Proceedings of the Eastern Joint Computer Conference 1955, p. 64-69.
- Rojas R. 1997, Konrad Zuse's Legacy: The Architecture of the Z1 and Z3, IEEE Annals of the History of Computing 19(2), p. 5-16.
- Rojas, R. & Hashagen U. (eds.) 2000, The MIT Press, Cambridge, Massachusetts: The First Computers, History and Architectures, proceedings of the Conference on the History of Computing, Paderborn, Germany, 1998.
- Rosen, S. 1969, Electronic Computers: A Historical Survey, ACM Computing Surveys, 1(1), p. 7-36.
- Rosen S. 1972, Programming systems and languages 1965-1975¹, Communications of the ACM 15(7), p. 591-600.
- Ruttimann J. 2006, Milestones in scientific computing, Nature, volume 440, p. 399-405.
- Sale, A. 1998, The Colossus of Bletchley Park - The German Cipher System in (Rojas & Hashagen 2000), p. 351-364.
- Salzer, J. 1966, Traffic control in a multiplexed computer system, MIT doctoral thesis.
- Schmidhuber J. 2006, Colossus was the first electronic digital computer, Nature, volume 441, p. 25.
- Sha L., Rajkumar R. & Lehoczky J. 1990, Priority Inheritance Protocols: An Approach

to Real-Time Synchronization, IEEE Transactions on Computers, 39(9), p. 1175-1185.

Silberschatz A., Galvin P. & Gagne G. 2005, John Wiley & Sons, New York:
Operating System Concepts, 7th edition.

Smith, T. 1970, Some perspectives in the early history of computers, in (Pylyshyn 1970), p. 7-15.

Smotherman, M. 1989, A sequencing-based taxonomy of I/O systems and review of historical machines, ACM SIGARCH Computer Architecture News 17(5) p. 5-15.

Smotherman, M. 2006, A Survey and Taxonomy of I/O Systems [Web-document].
Available: http://www.cs.clemson.edu/~mark/io_hist.html [Referenced 30. 10. 2007].

Smotherman, M. 2007, Interrupts [Web-document]. Available:
<http://www.cs.clemson.edu/~mark/interrupts.html> [Referenced 29. 10. 2007].

Sperry Rand Corporation, 1965, Univac 1108-II, The big system with the big reputation.

Sperry Rand Corporation, 1970, Univac 1108 multi-processor System, Programmer's reference, processor and storage.

Stanga, D. 1967, UNIVAC 1108 multiprocessor system, AFIPS Press, Montvale, N.J.:
Proceedings of AFIPS 1967 Spring Joint Computer Conference, p. 67-74.

Stern, N. 1980, John von Neumann's Influence on Electronic Digital Computing, 1944-1946 , IEEE Annals of the History of Computing, 2(4), p. 349-362.

Stern, N. 1981, Digital Press, Bedford, Massachusetts: From ENIAC to UNIVAC, An appraisal of the Eckert-Mauchly Computers.

Stibitz, G. 1966, The First Computers, Science, volume 153, p. 814-816.

Strachey C. 1959, Time sharing in large fast computers, Unesco, Paris: Proceedings of the international conference on information processing, Unesco, Paris 15-20 June 1959, p. 336-341.

Stroustrup B. 1997, The C++ Programming Language, 3rd ed, Addison-Wesley: MA, USA.

Tanenbaum, A. 2001, Prentice-Hall Inc. Upper Saddle River, New Jersey: Modern operating systems, 2nd ed.

Teager H. 1962, Real-time, time-shared computer project , Communications of the ACM, 5(1), p. 62.

Thompson, R. & Wilkinson, J. 1963, The D825 automatic operating and scheduling program, Proceedings of the SJCC 1963, p. 41-49.

Webster, 1913, G & C. Merriam Co., Springfield, Mass. U.S.A: Webster's Revised Unabridged Dictionary, edited by Noah Porter.

Weik, M. 1961, A Third Survey of Domestic Electronic Digital Computing Systems, Ballistic Research Laboratories, Aberdeen Proving Ground, Maryland: Report No. 1115.

Weinberg, S. 1986, Towards the final laws of physics, Cambridge University Press: Elementary particles and the laws of physics (1987).

Wilkes, M. 1975, How Babbage's dream came true, Nature, volume 257, p. 541-544.

Wilkes, M. 1985, The MIT Press, Cambridge, Massachusetts: Memoirs of a Computer Pioneer.

Wilkes M. 1990, Computers before silicon, design decisions on Edsac , IEE Review, 36(11), p. 429-431.

Wilkes M., Wheeler D. & Gill S. 1951, Addison-Wesley, Cambridge, Mass.: The preparation of programs for an electronic digital computer.

Wilkes M. & Renwick W. 1949, The EDSAC, Cambridge University Mathematical Laboratory: Report of a Conference on High Speed Automatic Calculating Machines 22-25 June 1949, p. 9-11. Reprinted in (Randell 1982), p. 417-421.

Williams M. 1998, A Preview of Things to Come: Some Remarks on the First Generation of Computers in (Rojas & Hashgen 2000), p. 1-13.

Williams F. & Kilburn T. 1948, Electronic digital computers, Nature, volume 162, p. 487. Reprinted in (Randell 1982), p. 415-416.

Wirth N. 1977 Toward a discipline of real-time programming, Communications of the ACM 20(8), p.577-583.

Zuse, K. 1936, Verfahren zur selbsttätigen Durchführung von Rechnungen mit Hilfe von Rechenmaschinen, German patent application Z23624. Reprinted in (Randell 1982), p. 163-170 as Method for Automatic Execution of Calculations with the Aid of Computers, translated to English by Mr. R. Basu.

Zuse, K. 1993, Springer-Verlag, Berlin: The Computer - My Life. Translated from Zuse, K. 1984, Springer-Verlag, Berlin: Der Computer - Mein Lebenswerk.

¹ Given the time of publication it appears that there must be a typo in the title of this article.

Appendix A: reprints and web sources for the references

(Aiken 1937), also reprinted in (Pylyshyn 1970), p. 29-46.

(Bashe & al. 1981), available as <http://www.research.ibm.com/journal/rd/255/ibmrd2505C.pdf>.

(Blaauw 1964) , available as <http://www.research.ibm.com/journal/sj/032/blaauw.pdf>.

(Blaauw & Brooks 1964), available as <http://www.research.ibm.com/journal/sj/032/brooks.pdf>

(Anderson & al. 1962), reprinted in Bell C. & Newell, A. 1971, McGraw-Hill, New York: Computer Structures - Readings and Examples., p 447-455.

(Corbato & al. 1962), available as <http://larch-www.lcs.mit.edu:8001/~corbato/sjcc62/>

(Corbato & al. 1963), available as http://www.bitsavers.org/pdf/mit/ctss/CTSS_ProgrammersGuide.pdf

(Dijkstra 1965b), reprinted in 1968, Genuys, F. (ed.) Programming languages, New York: Academic Press, p. 43-112, available as: <http://www.cs.utexas.edu/users/EWD/ewd01xx/EWD123.PDF>.

(Dijkstra 1967), reprinted in 1968, ACM: Communications of the ACM 11(5) p. 341–346.

(Dijkstra 1971), reprinted in (Hoare & Perrot 1972), p. 72-93.

(Einstein 1905), available as http://www.physik.uni-augsburg.de/annalen/history/papers/1905_17_891-921.pdf.

(Neumann 1945), available as <http://www.virtualtravelog.net/entries/2003-08-TheFirstDraft.pdf>.

(Newton 1687), available as <http://books.google.com/books?id=XJwx0lnKvOgC>, a translation to English can be found at <http://rack1.ul.cs.cmu.edu/is/newton>.

(Padegs 1981), available as <http://www.research.ibm.com/journal/rd/255/ibmrd2505D.pdf>

(Salzer 1966), available as <http://www.bitsavers.org/pdf/mit/lcs/tr/MIT-LCS-TR-030.pdf>.

(Sperry Rand Corporation 1965), available as http://archive.computerhistory.org/resources/text/Remington_Rand/SperryRand.UNIVAC1108II.1965.102646105.pdf

(Webster 1913), available as <http://machaut.uchicago.edu/websters> and <http://1913.mshaffer.com>.

(Weik 1961), available as <http://ed-thelen.org/comp-hist/BRL61.html>.