

Technische Fakultät der Christian-Albrechts-Universität Kiel
Institut für Informatik

Studienarbeit im Diplomstudiengang Informatik

Programming Asynchronous Execution

Gabriel Wicke
wicke@wikidev.net

Themensteller: Prof. Dr. Willem-Paul de Roever
Betreuer: Prof. Dr. Marcel Kyas, Freie Universität Berlin
Prof. Dr. Willem-Paul de Roever, CAU Kiel
Dr. Friedemann Simon, CAU Kiel

Abstract

The execution of computers, physical artifacts built to process information, is governed by physical laws. One of the most fundamental limits is the maximum speed of information travelling through a system- the speed of light. This limit imposes itself both on the design of computer hardware and on communication over long distances, e.g. in computer networks. The independent- or asynchronous- execution of physically separated systems is thus the natural state for spatially separated systems.

Computers have long used closely coordinated- synchronous- execution and decreased the size of systems to make this possible even at very high clock rates. This scheme has now reached electrical limits- new computers are produced with many relatively independent processors. The programming of asynchronous execution is thus not only important for the interaction with remote systems, but increasingly also for the effective use of mainstream computers.

Both the execution of systems and the interaction between systems can be described as communication. I will trace this idea through the layers of computational systems, with a special emphasis on the interfaces between different layers and their primitives to represent asynchronous execution and communication.

Finally, a practical implementation of a library for the efficient interaction with many asynchronously operating systems- typically using a network- is discussed.

Acknowledgements

I would like to thank my supervisors Marcel Kyas, Willem de Roever and Friedemann Simon for their advice and patience.

Contents

1	Latency and Communication	4
1.1	Latency in Modern Digital Computers	5
2	Digital Information Processing	5
3	Synchronization in Digital Communication	7
3.1	Self-clocking Protocols	9
3.1.1	Dual Rail Encoding	9
3.1.2	Start-stop Encoding	9
3.1.3	Single Wire Synchronous Encoding	10
3.2	Data/Strobe Synchronous and Single-rail Bundled Data Path .	10
3.3	Globally Synchronous Communication: Central Clock Signal .	11
3.4	Error Detecting/Correcting Codes	11
3.5	Bidirectional Communication	12
3.5.1	Rate Control using ACKs	12
3.5.2	Referencing Remote Resources using Handles	13
3.5.3	Recovering Message Loss using Buffers, NACKs and Timeouts	14
4	Degrees of Synchrony and Blocking at the Interface between Layers	14
4.1	Event Multiplexing: Select and Barrier	15
4.2	Interrupts: Starting New Activity	16
4.3	Summary	16
5	Interference, Shared Resources and Mutual Exclusion	17
6	Algorithms, Computation and Programs	18
6.1	Models of Computation	19
6.2	Recovering Dependency Information from Sequential Specifi- cations	21
7	Electronic Digital Computers	23
7.1	Instruction-Level Parallelism (ILP): Pipelining and Out-Of- Order Execution	24
7.2	Memory Hierarchy	25
7.3	Asynchronous Processors	26

7.4	Multi-Core CPUs	26
7.5	Mutual Exclusion	28
8	I/O Concurrency	30
9	Task or Execution Concurrency	32
9.1	Cooperative Multitasking	32
9.2	Preemptive Multitasking and Parallel Execution	33
10	Operating Systems	34
10.1	Processes and Threads	35
10.1.1	Threading Models	36
10.2	I/O Concurrency and Event Loops	37
11	Programming Languages and Runtimes	39
11.1	Objects as State or Processes	40
11.1.1	Object State Synchronization: The Monitor	40
11.2	Asynchronous Function Calls	41
11.3	Data/Loop Parallelism with Annotations	41
11.4	Lightweight User-Space Threads and Non-Blocking I/O Con- currency	42
11.5	Implicit Parallelism	43
12	Socketmachine: A User-Space Lightweight Thread Library with Non-Blocking I/O	44
12.1	Implementation	44
12.2	Operation Principle	46
12.3	I/O Operations	46
12.4	Scheduling, Timeouts and Error Handling	47
12.5	Related Work	47
12.6	Summary and Outlook	48
	References	48

1 Latency and Communication

In Newtonian mechanics, communication using light is assumed to be instantaneous—the speed of light infinite. This allows the simple definition of a global time, but has the disadvantage of eliminating causality.

The assumption of infinite speed of light was not confirmed by empirical data. For most applications far removed from these limits this did and still does not matter, as this assumption allows a very simple formulation of natural laws with sufficient precision.

While astronomers had earlier determined approximate limited speeds for light, the practical need for a new theory was increasing at the end of the 19th century as technological progress allowed high-speed communication far beyond the horizon. The problem of synchronization between imprecise and far separated clocks, important for astronomical determination of longitudes and the coordination of fast train systems, occupied several researchers including Poincaré and Einstein. Synchronization methods based on the communication of timestamps and the estimation of the propagation delay were proposed. Experiments had found the speed of light to be independent of movement of the sender/receiver pair through an imaginary 'aether' [Michelson and Morley, 1887], and quantitative measurements had established relatively precise values for c , the speed of light in vacuum.

In 1905, Einstein published *Zur Elektrodynamik bewegter Körper*, the foundation of Special Relativity. He assumed c to be the upper limit of any propagation and constant in any inertial system and thus broke completely with the earlier aether theories. In Special Relativity, c is the central constant relating space to time and mass to energy ($e = mc^2$).

While Einstein's theory was very controversial for a long time, it has proven consistent with empirical data and forms one of the fundamentals of modern physics. More precise measurements have since then established c at 299792458 m/s, or about 30 cm per ns. So far, no propagation of information faster than c was observed.

In modern communication networks, e.g. the internet, propagation delays close to the speed of light are easily observable, e.g. by using the 'traceroute' command:

```
traceroute to wikipedia.org (208.80.152.2), 30 hops max, 40 byte packets
 1  lan-gw.wikidev.net (81.3.6.17)  0.251 ms
(..)
 5  ge-3-0.ir1.frankfurt-he.de.xo.net (80.81.192.182)  12.881 ms
```

```
6 207.88.15.74.ptr.us.xo.net (207.88.15.74) 19.653 ms
7 te0-3-4-0.rar3.washington-dc.us.xo.net (207.88.13.198) 105.141 ms
8 te-3-0-0.rar3.atlanta-ga.us.xo.net (207.88.12.9) 132.595 ms
9 te-4-0-0.rar3.miami-fl.us.xo.net (207.88.12.6) 132.121 ms
(..)
12 rr.pmtpa.wikimedia.org (208.80.152.2) 138.759 ms
```

1.1 Latency in Modern Digital Computers

For systems used to execute a high number of elementary operations in a short time, e.g. common electronic computers, the speed of light is also limiting at a spatially smaller scale. At an execution rate of 4×10^9 operations per second, the maximum distance information can travel in the processing of this instruction is limited to about 7.5 cm if propagated by light in vacuum, and much less for electricity through a network of capacitive logic gates. As a result, the propagation of a single clock signal commonly used to globally coordinate communication through modern processors now takes multiple clock cycles. Further miniaturization of semiconductor electronics is complicated by disappearing isolation properties at feature sizes approaching a small number of atoms, limiting the complexity of electrical circuits which can be coordinated by a single clock signal. As a consequence, the focus for further performance improvement has recently shifted to increasing numbers of synchronous processors on a single chip- each capable of independent computations- which makes the programming of asynchronous execution necessary to use the computational resources even of commodity computers.

This has stimulated the search for improved abstractions to express potentially asynchronous executions and methods to optimize executions for particular physical systems (especially relative to the system's latency) based on the information expressed using these abstractions.

2 Digital Information Processing

Human sensory input is limited in resolution- different physical states are observed as equivalent. The choice of resolution and equivalence class boundaries is influenced by the utility of discrimination, allowing a human to make optimal use of limited physical resources to process them. This form of lossy compression- a kind of abstraction- can be applied recursively and with dif-

ferent utility functions at multiple levels of information processing. Simple equivalence classes give way to far more complex and ambiguous concept structures at these higher processing levels, giving humans the ability to gradually develop semantical structures to 'make sense' of raw sensory input.

In this human-centered view, information- the colourful concept informatics tries to understand and manipulate- could thus be defined as physical interaction- e.g. observation- with associated semantics. This differs from the definition used for digital information, which concentrates on lossless compression and communication of already digital states.

Humans soon constructed measurement devices to amplify their abilities for discrimination. The shadow of a sundial allows better discrimination of the earth's position relative to the sun and thus of time. A continuous quantity- the angle between sun and earth- is translated into another continuous quantity, which can be observed more precisely- the position of the shadow on the dial. As the sundial processes physical interaction (sunlight) into a physical state which can convey more (semantic) information to humans, it can be seen as a simple analog information processing machine- an analog computer.

The number of weights equalizing an item on a scale can be counted (e.g. using digits as an aid). The result is a discrete or digital number- an example of a very simple digital computer. Here, the size of individual weights and thus the resolution of the resulting weight information is chosen by humans to reflect semantically meaningful units, which can vary with the application of the scale.

Claude Shannon, who was mainly interested in communication issues and was thus able to disregard issues of semantics, used this simpler notion of digital information in mathematical information theory [Shannon, 1948]. In his noiseless channel theorem, he defined the storage required for communication of information using the bit as the primitive unit of digital information, laying the foundation for digital computation.

Physical systems cannot be completely isolated from unwanted interaction with state outside the system and will deviate from deterministic behaviour by a degree summarized as noise. Shannon's second fundamental theorem, the noisy channel theorem, quantifies how much information can be reliably transmitted through a noisy communication channel, given that signal amplitudes smaller than the noise amplitude cannot be reliably distinguished from noise. The resulting noisy channel capacity represents an upper

bound achievable using perfect error correcting codes, encodings that trade some redundant transmission against the capability to correct (and usually additionally detect) a limited number of errors introduced by a random noise process.

In addition to noise, the realization of physical systems precisely according to an idealized specification is rarely practical, leading to additional systematic non-ideal behaviour of the system.

These two problems together limit the power of analog computers for the processing of digital information, as noise and imperfections of the system directly change the result. The superior ability to eliminate the effects of noise and other non-ideal behaviour made digital computation the dominating method for the processing of digital information.

The advantages of digital information combined with the rapid development of semiconductors have enabled an impressive development of digital computers in the last 60 years, solving most problems associated with the processing of digital information and allowing modifications of information processing using physical state with digital semantics: programs. Their computational power is equivalent to Turing machines, although modern machines deviate from the purely sequential model by embracing parallelism to improve performance.

Physical systems are distributed in space; the movement of information through such a system- an execution- can be described in terms of different forms of communication. Depending on the relative rate of communication between parts of a system the execution is classified as synchronous (closely matched) or asynchronous (mostly independent, with occasional synchronization points).

In the following, I will concentrate on mainstream electronic digital information processing systems- in short, currently the most common form of computers.

3 Synchronization in Digital Communication

The serial transmission of discrete symbols, e.g. bits, words or morse code characters over a physical medium implies *continuous* physical state changes spread out in time and space. A receiver needs to recover individual symbols from a continuously changing physical signal, so requires some knowledge about the extent of a single symbol in time and space (if moving relative to

the sender). Some degree of synchronization between the receiver and the signal's state changes is needed.

This synchronization can either be recovered from the transmission, especially if state transitions follow the periods of a steady clock signal and occur at regular intervals (*self-clocking protocols*), or using a dedicated clock signal (*externally clocked*). In any case, synchronization is also digital information converted to an analog signal, which needs to be converted back to a digital value at the receiver.

The conversion from a digital to a continuous domain- D/A conversion- is relatively straightforward and can result in a successful transmission if the dynamic range of the chosen physical variable vs. the expected combined noise and non-ideal behaviour amplitude at the receiver is sufficient to allow the receiver to distinguish the full range of possible discrete values.

Implementations of the opposite direction- analog to digital or A/D conversion- runs into difficulties when the continuous physical state is close to the boundary between two discrete output values. This occurs necessarily at least once on each state change, and any number of times shortly after a state change in the presence of oscillations and noise. So far, no solution for any realizable communication medium is known which can guarantee a decision in limited time- a metastable state can be entered without deciding on one of the discrete values [Lamport, 1986]. This property of asynchronous concurrent systems is called *unbounded nondeterminism*- the timing and thus potentially the result of the computation cannot generally be deterministic in the presence of metastability.

A nondeterministic delay can result in a plain failure whenever a timely decision is important. As an example, globally clocked systems are based on static timing analysis and return undefined results if the timing assumptions are violated. Even if downstream processing could tolerate a delayed decision, further sampling will also be delayed and will thus potentially miss some state changes, missing or corrupting symbols in a digital transmission or distorting the dynamic representation of a digitized analog variable (e.g. sound, light etc). Another important application for atomic decisions is mutual exclusion in the access to shared resources. An unstable state could allow multiple non-exclusive accesses, resulting in arbitrary interference and likely failure. Mutual exclusion is discussed in more detail in section 5.

Solutions with a probability of metastability low enough to rank behind other more common failure modes were developed for electronic circuits, making it a solved problem in digital computers for most practical purposes

(e.g. Kinniment and Woods [1976] and Ginosar [2003]).

Once some form of synchronization is established, sampling can be timed to occur when intermediate states and oscillations associated with state changes have subsided. This only leaves the typically smaller-amplitude noise and allows either smaller overall signal amplitudes (and thus less power, higher distance or higher frequency) or more distinguishable states (higher amplitude).

A common technique to recover a relatively clean and regular local clock signal even if clock edges are only received irregularly as part of communication is the phase-locked loop, which is widely used in serial self-clocking protocols.

3.1 Self-clocking Protocols

All unidirectional self-clocking protocols described here operate asynchronously to any global clock- the sender rather uses a local clock source in the encoding of messages. Additionally, the unidirectional nature of this communication allows the sender to send messages at any time, asynchronously from the reception or processing of these messages at the receiver, but synchronously to the local clock signal used for the encoding of the message.

3.1.1 Dual Rail Encoding

Dual rail encoding uses two transmission lines to encode binary information as a high signal on either of the two lines and a separation period between bits with both lines low. Each clock transition is explicitly transmitted, which allows the use of non-periodic or quickly varying clock signals. It is possible to build combinational logic which directly operates on dual-rail encoding, which helps to maintain the necessary close delay match between the two 'rails' over longer distances, a very useful property for combinational logic in circuits without global clock synchronization ('asynchronous circuits').

3.1.2 Start-stop Encoding

An example for the start-stop communication of individual ascii chars is the RS-232 serial protocol. Starting from an idle state without any state changes, a state change corresponding to a start bit signals the begin of a character transmission to the receiver. After this, typically seven data bits followed

by one parity bit and one stop bit are sent synchronously to a local clock approximating a known frequency. The receiver decodes the message using an approximately matching local clock signal and then again listens for the next start bit without making any assumptions about timing, potentially for a very long time.

The short bursts of fixed-frequency transmission place only very modest requirements on the clock accuracy at sender and receiver and allow the start of a transmission at any time. This protocol was first used for the transmission of key input from teletypewriters, and due to its simplicity continues to be popular for low-level communication in modern computers and embedded microprocessors.

3.1.3 Single Wire Synchronous Encoding

Single-wire synchronous communication encodes data in way that allows the receiver to recover an implicit clock signal and tune a local clock reference to the signal transmission (typically a phase-locked loop, PLL).

To ensure robust clock synchronization, specialized run length limited codes (RLL) with a minimum number of state changes in a given period are used to help the receiver in keeping local timing information in synch. A classical example with high overhead is the Manchester Code, which inserts one extra state transition between each data bit transmitted. More efficient protocols only insert extra transitions after a given number of data transmissions (e.g. the 8b/10b used in SATA, PCI-Express, USB 3 etc links) or even only when needed depending on the data transmitted (bit stuffing, e.g. in USB 1 and 2).

3.2 Data/Strobe Synchronous and Single-rail Bundled Data Path

In these protocols, the communication of timing information is split out to a separate transmission path. The receiver samples one or more parallel pure data channels at times coordinated by the clock or strobe channel. The explicit transmission of each clock transition has the disadvantage of higher overheads in the form of a second line, but allows rapid changes in the clock rate.

The delay on the paths for strobe and data need to be conservatively matched to the worst-case stabilization time on the data paths. All propa-

gation characteristics on both strobe and data paths need to be known to make static timing analysis possible.

3.3 Globally Synchronous Communication: Central Clock Signal

Global synchronization uses a central clock signal to coordinate all communication in a system- not only a single communication path. This makes the transmission of additional timing information between sender and receiver unnecessary, but typically limits the communication distance to how far electricity can travel in a fraction of a clock cycle (or, in other words, the clock frequency to the maximum delay in all coordinated communication paths in the system). All propagation delays in the system need to be known for global static timing analysis, restricting this method to tightly controlled systems.

The distribution of the clock signal itself becomes more difficult with high speeds and consumes significant resources- both space and energy. The disadvantages of global synchrony are less pronounced at a spatially limited scale, where the missing need for synchronization of individual data paths often results in higher throughput than comparable asynchronous circuits.

A popular strategy to leverage the respective strengths of both synchrony and asynchrony is termed 'globally asynchronous, locally synchronous' (GALS). This architecture connects locally synchronous systems with asynchronous communication, exploiting simplicity and efficiency of local clock synchronization without being limited by a slow global clock speed and expensive clock distribution.

In a globally synchronous system, messages are sent synchronous to the global clock, but asynchronous to the actual reception or processing of the message at the receiver. However, the receiver shares the same global clock signal and is normally statically designed with sufficient resources to handle any incoming messages.

3.4 Error Detecting/Correcting Codes

The modification or corruption of messages during transmission, propagation or reception needs to be considered if interference cannot be ruled out. Without any special measures, a receiver cannot distinguish a modified message from a correct one, let alone correct minor errors.

Error detecting/correcting codes solve this issue by adding some redundant information derived from the message's contents to each message. Most of these codes- especially those with low overheads in size and processing complexity- only achieve a limited probability of error detection and cannot detect deliberate message modifications as an attacker can also recompute the error correcting code. Very high levels of protection against undetected modification can be achieved using signatures based on public-key cryptography. This method requires knowledge of the sender's public key at the receiver, which needs to be established using a safe means of communication before this protection can be used. Message modification can be detected even with relatively low overheads, and can be treated as a message loss in further processing or recovery.

Error correction capabilities are typically very limited, as most modifications are minor and the overheads for more correction capabilities quickly grow. Codes with higher overheads and correction capabilities are mainly used in latency-sensitive applications, where other methods of error recovery would introduce unacceptable delays in the communication.

3.5 Bidirectional Communication

Synchronization is implemented by delaying an action while waiting for a signal- a clock tick, the arrival of a (reply-)message, or the finished transmission of a message. In the case of global synchronization, both state change at the sender and the sampling at the receiver are delayed until a transition in a global clock signal occurs. In self-clocking protocols, the same delays are used with local clocks or transitions in the signal.

All of these examples are unidirectional and independent of propagation delays, capacities at the receiver etc- messages are sent asynchronously from any operations at the receiver.

3.5.1 Rate Control using ACKs

A feedback path- *bidirectional communication*- allows the synchronization on acknowledgement messages, which can control the rate of further message transmissions to match variable progress at the receiver. In the simplest version, any further message transmissions are *blocked* until the preceding message is acknowledged. In digital electronics, this local form of synchronization is used in circuits without a global clock- so-called *asynchronous*

circuits.

The round-trip latency of the communication channel limits the rate of message transmissions in this simple protocol. The amount of blocking can be reduced by allowing multiple outstanding ACK messages. With multiple messages and ACKs in flight, reordering of messages can be an issue.

With this scheme, the sender's ability to send bursts of messages in very quick succession until the limit on outstanding ACKs blocks it can cause the message arrival rate at the receiver to exceed the receiver's capabilities, and messages can be lost. A dynamic equilibrium can be reached if the sender initially only allows a small number of outstanding ACKs and scales back the transmission rate on noticing message loss. If no further messages are lost, the rate can then be slowly increased until another message loss triggers a repeat of this cycle. This method of congestion control is widely employed on the internet in the TCP protocol [Postel, 1981].

3.5.2 Referencing Remote Resources using Handles

If in-order transmission is desired, but the transmission channel cannot guarantee first-in-first-out (FIFO) behaviour, *handles*, opaque references to a remote resource, can be used to establish an order dependency. Sequence numbers are one simple possible handle, another would be an arbitrary number as a message handle and a reference to the respectively preceding messages's handle to establish order. Similarly, ACKs can reference the message they acknowledge using that message's handle.

The handle concept can be applied to many other remote (not locally controlled) resources- e.g. to an open file, a transaction or a credit card account.

Cancellation of an ongoing operation can be implemented by sending a special message referencing the original request's handle to the processing entity.

Similarly, the destination of the reply could be redirected by passing a handle associated with an ongoing request to a third party, which can then request a redirection of the result to itself by sending a special message including a reference to the handle to the original receiver. This method is also known as 'promise pipelining' [Liskov and Shrira, 1988].

3.5.3 Recovering Message Loss using Buffers, NACKs and Timeouts

In the rate control scheme outlined above, message loss eventually results in a *deadlock*, as lost messages are indistinguishable from a very long delay [Fischer et al., 1985]. If the parties in the communication have access to an approximate time source and the intervals between messages (in both directions) can be bounded, *timeouts* can be employed to initiate retransmissions of messages or acknowledgements to recover this situation.

For a retransmission, the message still needs to be available- either by recreating it on demand or by keeping a copy in a buffer until the successful transmission is eventually (given limited message loss) achieved. The number of buffers needs to equal the maximum number of outstanding ACKs; in practice, it is often the number of available buffer resources which determine the maximum number of outstanding ACKs.

With retransmissions, a message might be received twice, which can be detected using message handles.

Messages missing in an order and uncorrectable message modifications are quickly detected at the receiver without any need for timeouts. To speed up the recovery, the receiver can initiate an immediate retransmission by sending a NACK message for the corresponding handle.

4 Degrees of Synchrony and Blocking at the Interface between Layers

All previous protocols are synchronous with the transmission of individual messages- all further transmissions are blocked during an ongoing transmission.

The unidirectional transmission protocols are asynchronous to propagation latency, congestion or error recovery, while the reliable bidirectional protocols synchronize on ACK messages- they are synchronous to the successfully acknowledged transmission of messages.

Bidirectional communication so far assumed that ACKs would be sent immediately after receiving the message, independent of further execution influenced by the message. If the ACK or reply is instead delayed until this execution has finished, the sender's execution is synchronized to the *execution* of the receiver. This synchronization at the same level- execution synchro-

nizes to the speed of execution elsewhere- is also called *fully synchronous* or *rendezvous*.

All asynchronous modes of communication either omit a synchronization on a reply or separate the transmission of the request from waiting for the reply. The send operation normally returns a handle, which can be passed to the receive operation to wait for the specific, matching ACK or reply.

The send or request operation can synchronize on ACKs sent on arrival at different points in a transmission path. A transmission path often traverses multiple layers, and the arrival at each of these layers at both the sender and receiver can send an ACK.

Bounded buffers can prevent a request from being executed immediately- the send operation can be blocked for a long time. A sender wishing to avoid this can alternatively request *non-blocking* operation with a quick reply even if the request was not or only partially executed. The sender can then retry the (remaining) operation later.

The receive operation can similarly be made non-blocking, potentially returning a part of the incoming data and an indication if more can be expected.

Apart from the non-blocking interface, the degree of synchrony of the blocking receive interface is much less varied. The remaining choice is mostly about buffering and partial data. One possible blocking solution can return some data as soon as it becomes available, while another could return only after the entire message is locally buffered.

4.1 Event Multiplexing: Select and Barrier

Non-blocking variants of both the send and receive operations provide the requesting system with more flexibility in the handling of many concurrent interactions. At the same time, non-blocking operations could lead to inefficient polling of each member of a set of ongoing operations. A more efficient alternative in the form of select or barrier primitives can be provided by lower layers.

The *select*¹ primitive takes a list of handles with the intention to send, receive, or both specified for each, and blocks until at least one handle can accept or provide data, returning the list of 'ready' handles.

The *barrier* primitive takes the same inputs and returns once *all* handles

¹Molded on the `select()` system call, which first appeared in 4.2BSD in August 1983.

are ready to perform the requested action. This primitive can easily be emulated if a select-like multiplexing operation is available.

4.2 Interrupts: Starting New Activity

Both blocking and polling are passive methods in which the requesting layer checks if a reply is yet available. If the requesting system supports the creation of new activities or tasks, the arrival of messages can actively trigger the creation of a new activity to handle the message. In case the original activity which started the request terminated, the new activity can continue, potentially using left-behind state.

Alternatively, the new activity can transfer the incoming message to a buffer and unblock any blocked activity waiting for the arrival of this message.

4.3 Summary

Reliable, but not necessarily completely asynchronous communication in the face of a bounded number of transmission errors and bounded buffers is achievable if the participating entities have access to an approximate time source to implement timeouts. The correctness of this type of system including an initial handshake procedure to establish initial handles or sequence numbers for both directions was proven correct by Lamson, Lynch and Sogaard-Andersen and is widely used in computer networks- the TCP protocol [Lynch, 1996, ch. 23].

Asynchronous unidirectional communication cannot be reliable in the presence of arbitrary transmission errors.

Communication- as most of computation in general- can be structured in layers (Dijkstra [1968a], Dijkstra [1976], Bush and Meyer [2002]), with a wide array of choices in the degree of synchrony in the communication primitives at the interface between two layers.

Asynchronous execution can be implemented using any communication pattern that does not enforce fully synchronous execution (rendezvous), i.e. divorces the amount of blocking during the send or request operation from its corresponding execution time.

Communication, synchronization, rate coordination and error recovery are very general problems, so solutions were developed in many areas. With some squinting, it should be possible to notice parallels between the solu-

tions discussed for electronic digital systems above and those used in other contexts- e.g. human speech, commercial or institutional organizations.

5 Interference, Shared Resources and Mutual Exclusion

The communication protocols so far have carefully avoided any unwanted interaction between logically separate signals by assuming physically separated communication paths for each signal. This is not always possible- either because the point of a system is precisely the integration of information and thus communication from multiple sources or because limited physical resources need to be multiplexed between multiple parallel communication paths.

Mutual exclusion avoids interference or the access to inconsistent state by organizing exclusive access to a shared resource in a sequence. A degree of fairness is often implemented by organizing waiting accesses in a FIFO queue.

Decisions on the order of access need to be atomic, even if two requests arrive virtually at the same time. In a globally synchronized circuit with a global clock signal, arrival in the same clock cycle can be handled with a fixed decision rule. However, in the absence of a clock cycle and the associated stable state guarantees, a decision between virtually simultaneous requests is more difficult. An earlier example of a kind of mutual exclusion between discrete values based on continuous inputs- A/D conversion- suffered from the possibility of metastability. The same issue and the same solutions apply to implementations of mutual exclusion in an asynchronous setting, so-called arbiters [Kinniment and Woods, 1976].

The literal implementation- termed *pessimistic*- of free mutually exclusive access to a live resource in so-called *critical sections* [Dijkstra, 1968b] can suffer from long delays when an access to a resource is not kept brief. A more robust method- termed *optimistic*- avoids potential waiting for slow processes at the cost of duplicate computations in the case of contention. Potential writers prepare transactions using isolated copies of information and optimistically submit the results for atomic commit along with the versions or state of the information used. If the used information was not changed by other mutually exclusive operations in the meantime, the results are atomically committed. If the information used in preparation was modified in the

meantime, the transaction is refused and can be retried using updated information. The results are required to be identical to a mutually exclusive, serial execution of all transactions- a fundamental property termed *linearizability* [Herlihy and Wing, 1990].

The need for isolation between concurrently prepared transactions only allows the direct execution of operations without observable side effects (e.g. no observable I/O). Observable side effects can only be executed during a successful commit. These properties need to be manually enforced if the specification of the transaction does not support a clear identification of side-effectful operations. If this is not possible, all operations can be treated as potentially side-effectful- which yields the pessimistic solution. Some programming languages cleanly separate side-effectful operations from pure ones, which allows them to experiment with highly automated transactions, so-called software transactional memory (STM) [Shavit and Touitou, 1995].

With low contention, the non-blocking property of optimistic mutual exclusion algorithms potentially allows a higher degree of parallelism than a naïve pessimistic solutions by performing the actual computations asynchronously- although an optimized pessimistic solution should achieve the same performance at the cost of implementation complexity.

In the case of high contention (and ignoring overheads), the throughput can drop to a similar rate as a pessimistic solution. However, considering only throughput ignores the resource utilization from recalculations (which could be used for unrelated calculations) and potential overheads introduced by dependency tracking.

Any need for mutual exclusion severely impacts the degree of parallelism attainable, so minimization and localization of mutual exclusion are more important than the choice of mutual exclusion method.

The mutual exclusion problem is again very general, and endless examples from everyday life can be found- including both optimistic and pessimistic solutions.

6 Algorithms, Computation and Programs

After discussing the concurrent nature of physical computation and communication, let us now look at possible specifications for computational processes.

The succinct instruction 'cook a delicious meal' requires significant back-

ground knowledge for successful execution. What exactly constitutes 'delicious' depends on the requester, the cultural background, the occasion, possibly the available ingredients and tools, time or cost constraints and any number of other things.

A *recipe* tries to reduce the required context knowledge significantly by breaking down 'cook a delicious meal' to a sequence of very specific instructions. The execution of each of these instructions only requires basic knowledge of how to execute individual steps (how to use basic kitchen tools), allowing beginners to execute an *algorithm* which solves the 'cook a nice meal' problem.

The abstraction to a sequence of precisely defined and simple instructions potentially enables execution on a computer (which typically does not have any context knowledge). Indeed, there are machines believed to be able to effectively simulate any 'effectively computable' algorithm.

6.1 Models of Computation

The most influential model of computation is the Turing machine, invented before real computers became widely available and proven equivalent to a large class of machines including modern computers. This abstract machine *sequentially* applies rules which manipulate cell content and position on an infinite storage tape depending on an integer state variable and the current cell content. According to the Church-Turing thesis, the class of machines equivalent to the Turing machine can effectively compute any 'effectively computable' algorithm- an informal notion inspired by what human calculators *provided with paper, pencil, and rubber, and subject to strict discipline* could effectively calculate [Turing, 1948]. Central to the effectiveness of an algorithm is its complexity, the degree to which computation time grows with input size. The original Church-Turing thesis conjectured that a Turing-like machine could solve any algorithm at least as efficient- with the same or lower complexity- as a 'mechanical' or human solution could.

However, a newer class of computers based on quantum mechanical effects- which were not yet built in competitive sizes- would have a better complexity for some important algorithms (including integer factorization, used in cryptography), but can still be (less efficiently) simulated by a Turing-like machine. Quantum computers are believed to be able to effectively simulate any finite physical system- a remarkable property with some interesting philosophical repercussions (Zuse [1967], Deutsch [1985], Nielsen and Chuang

[2000]).

Conditional iteration is a characteristic capability of Turing-equivalent machines- allowing subalgorithms like 'fry the onions until golden brown' and infinite loops to be expressed. Where this capability is not needed, a simpler model of computation is sufficient, which simply passes the data sequentially through all specified operations- a pipeline (e.g. McIlroy [1964] and Sutherland [1989]).

Turing-equivalent *concurrent* models of computation lift the restriction on sequential communication and execution patterns. One obvious source of concurrency is the interaction with peripherals and through them potentially with the 'outside world'- short I/O. Independent local computations are the other source of concurrency, which can be exploited to increase the utilization of computational resources while other computations are blocked on I/O or to utilize independent computational resources in parallel.

Distributed computation is a subfield of concurrent computation, often defined as *computation distributed across computers connected through a network*, emphasizing the higher probability of communication failures and absence of hierarchy and globally atomic operations. The most important distributed algorithms try to solve the problems arising from limited local information, symmetry and missing atomic decisions- variants of the agreement problem like mutual exclusion, commit protocols, leader election etc. The presence of failures makes the reliable solution to the agreement problem impossible in some restricted models of computation, while it can be solved if transmission errors are limited, some approximate measure of time is available and arbitrary computations on messages are permitted.

A very good introduction to distributed systems, including the partially asynchronous network model, can be found in Lynch [1996]. A survey covering many impossibility results is available in Fich and Ruppert [2003].

To allow more formal reasoning about properties of concurrent systems, models in the spirit of the Turing machine were developed for non-sequential computations. Petri-Nets, invented by 1939 by Carl Adam Petri for the description of chemical processes, were adapted to concurrent computation in 1962. The Actor model, inspired by physical laws and gradually gaining formalization, followed in 1973.

Process calculi, with emphasis on the support for equational reasoning, followed with Robin Milner's Calculus of Communication Systems (CCS) between 1973 and 1980 (and later the π -calculus) and C.A.R. Hoare's Communicating Sequential Processes (CSP) in 1978, which was formalized to a

full process calculus beginning with Francez et al. [1978] and continuing until the early 1980s. The Algebra of Communication Systems (ACS) introduced in 1982 emphasizes algebraic structures for processes. Numerous other calculi and extensions followed, collectively improving support for the formal reasoning and the automatic verification of concurrent systems.

6.2 Recovering Dependency Information from Sequential Specifications

The naïve execution of each instruction in sequential algorithms is fully blocking and modifies global state as a side effect. Treating global state as an opaque object enforces full synchronization between instructions, as parts of the state accessed by the next instruction could be modified. A finer analysis following the actual part of the state accessed and manipulated by each instruction could however recover a dependency graph between instructions, which could well include independent arcs- potential concurrency. Unfortunately, this is only guaranteed to yield the same results (more about *the same* later) if the state changes have no additional side effects which- if executed in a different order or in parallel- could change the semantics of the observable result of the computation.

An expert cook 'executing' a recipe has sufficient knowledge about the semantics of relevant side effects to allow him or her to perform this analysis effortlessly and exploit the discovered concurrency to increase his or her throughput (by using multiple plates at once and multiplexing attention between them). Automatic analysis is missing all this background knowledge. To allow a safe concurrent execution of sequentially-specified algorithms, the semantics of state changes and potential side effects would need to be formally specified.

This can quickly lead to very complex specifications. For example, in modern computers I/O is mostly programmed by modifying state in registers and memory. This can produce arbitrary effects in the 'world outside the computer', all of which would need to be precisely defined to allow a decision if a change in execution order (or even the timing) of state-manipulating instructions produces a semantically equivalent result. Following this path would reverse the big advantage of algorithms- a high degree of abstraction.

Instead, simpler models of observably-equivalent executions are used for most purposes. A still relatively demanding model of observational equiva-

lence is used for so-called real-time systems, in which not only the relative order of some or all externally observable actions is relevant for the semantics, but also their timing- *too early* can be just as wrong as *too late*. Probably the most common (and also a very simple) definition of equivalence considers the relative order of all or even only between some externally observable events in a *trace*.

The identification of instructions resulting in externally observable events in the recovery of implicit concurrency can be difficult if the same mechanism- typically global state manipulation- is used for both internal communication and I/O. Programming languages can and some already do support more aggressive automatic concurrency analysis by separating 'pure' (side-effect free) parts from effectful and thus not reorderable parts of a computation.

Declarative functional languages take this even further by departing from sequential program structures in favour of a dependency graph expressing the logic of the computation while keeping the evaluation order deliberately unspecified. In these languages, the specification of sequential computations is confined to special constructs separate from 'pure' computations, with order actually enforced by constructing a data dependency graph using a global 'world state' passed through the sequential operations [Wadler, 1993].

A communication and dependency graph structure can also be manually defined in more traditional sequential languages using explicit communication primitives- e.g. process creation and inter-process communication/ message passing, arbitrary I/O. The manual nature of these specifications can expose programmers to communication issues otherwise handled by hardware or compilers and runtimes. Newer programming languages try to provide higher-level communication primitives, which are discussed in later chapters.

The machinery involved in typical computer systems is using a heavily hierarchical architecture with lower hierarchy layers closely mirroring the underlying physical processes and upper layers gradually hiding (abstracting/compressing) details not important in problem domains- a structure advocated by Dijkstra and others since the end of the 1960s (Dijkstra [1968a], Dijkstra [1976]). Each layer defines an *interface* or contract for the layer above it, allowing parts of the stack to be swapped for replacements providing the same interface- for example allowing the same program to be executed on different hardware. Additionally, and more importantly, this structure makes reasoning about complex systems tractable- for both humans and machines.

As discussed in section 4, the communication primitives provided at the interface between two layers can support asynchronous execution to varying

degrees, with all interfaces supporting asynchronous execution sharing a split between sending a request and synchronization when trying to access a result.

The use of asynchronous interfaces provides opportunities for concurrent execution, which can be used to increase performance by using parallel hardware or hiding the impact of latency on throughput.

7 Electronic Digital Computers

While the term 'computer' initially referred to a person carrying out calculations and later to programmable mechanical machines, the modern term refers to programmable electronic machines whose still rapid development started around 1940. Early programs for these electronic machines took the form of punched cards or electronic switches, while current computers use various forms of semiconductor-based random-access digital memory for the storage of programs and data- shared in the von Neumann architecture and separated in the Harvard architecture. Random access storage distinguishes modern computers from Turing machines and enables more efficient implementations of some algorithms. Despite these minor differences, modern computers are regarded as equally powerful as Turing machines.

Internally, the central processing unit (CPU) of modern computers consists of highly integrated functional units fundamentally based on transistors, usually including a multi-purpose arithmetic logic unit (ALU), memory cells called registers, I/O controllers for communication with peripherals or memory and a control unit. Communication in current computers is nearly universally synchronized using a global clock, although asynchronous communication is typically used for communication with peripherals, following the GALS architecture.

Programs are sequences of binary-encoded instructions specific to a CPU which- by determining a sequence of hardware operations in the specific CPU- define algorithms.

The control unit inside a CPU decodes the linear program bytestream from memory and manages the corresponding hardware operations. This includes the maintenance of the current location inside the program's bytestream using an instruction pointer, loading data from internal or external memory and initiating corresponding operations of the ALU or other hardware units. After this operation is completed and the result stored to a register, the location of the next instruction might be changed again (in the case of conditional

instructions) and the cycle starts anew.

The cyclic execution of sequences of general instructions splits a potentially long data flow graph in short subgraphs which implicitly receive the state produced by preceding cycles as input. The availability of many different operations in each cycle of an execution and the ability to control the data flow using programs makes computers general-purpose. For popular yet resource-intensive applications (e.g. media and graphics processing), larger subgraphs can be implemented in hardware to increase efficiency for these operations- but this can also slow down simpler operations if not used frequently enough as space and thus latency is consumed [Hennessy and Patterson, 2007, chapter 1].

7.1 Instruction-Level Parallelism (ILP): Pipelining and Out-Of-Order Execution

The clock rate for globally synchronous processors is determined by the longest path and thus the longest subgraph traversed in a cycle. In the naïve cyclic model which executes one instruction in each cycle, this includes at least fetching the instruction and data from memory, decoding it, executing it and writing back the results to some form of memory. To increase the throughput of a processor with a higher clock rate, each of these phases can be executed in a single clock cycle, splitting up the processing of a single instruction across several cycles. Most high-performance processors split the execution into even smaller parts, resulting in pipelines with up to 31 stages. The performance of such a pipeline depends on the overlapping- asynchronous- execution of sequentially independent instructions.

The pipelined execution of branch instructions forces the pipeline to make a speculative choice or stall, as further instructions are not independent of the branch taken. Elaborate statistical branch prediction schemes are used to pick a branch for speculative execution, reducing the probability of getting delayed by a pipeline stall on misprediction as far as possible.

An even higher degree of potential concurrency can be exploited by issuing multiple instructions in a single cycle if sufficient functional units are available for a parallel execution. More elaborate dependency analysis at runtime is needed to discover sufficient parallelism to make this worthwhile. Tomasulo's algorithm allowed the dependency analysis to move past the simpler register-use algorithms employed earlier- a significant step as registers are scarce

in many architectures and thus heavily reused. Complex synchronization logic is needed to manage the resulting parallel execution using buffering and stalling. Exceptions (e.g. division by zero, invalid memory access) need to be handled in way that keeps the observable behaviour identical to a naïve implementation. This capability, called out-of-order execution (OOE), is present in all high-performance microprocessors since the Intel Pentium Pro in 1995, with early implementations as early as 1964 [Hennessy and Patterson, 2007, chapter 2].

Any mainstream microprocessor thus recovers a limited amount of parallelism from a sequential program specification and employs this to asynchronously execute independent subsequences.

The length of independent instruction sequences discoverable using runtime analysis is very limited, and often fails to fully utilize the high degree of available internal parallelism of current processors. To improve utilization, many processors now expose a single CPU core as a dual-core processor, which allows two independent instruction streams to saturate the functional parallelism available in the processor [Tullsen et al., 1995].

7.2 Memory Hierarchy

Large memory consumes significant space, which limits the amount of memory close to a processor core. Additionally, the lowest-latency memory technologies are more expensive and require more space than higher-capacity technologies. On current hardware, the latency of an access to high-capacity memory is more than two magnitudes higher than basic arithmetic operations, which makes the memory subsystem very important for the overall performance of current computer systems.

Current designs use a memory hierarchy with 2-3 cache levels of increasing capacity and latency on each processor chip and large memory chips as separate components connected with a memory bus to the processor. From the perspective of a processor executing a stream of instructions, the memory load process is a slow and unpredictable operation, during which many instructions could be executed.

Memory management units in processors try to minimize this by identifying memory access patterns to speculatively pre-fetch memory. This works relatively well for sequential memory accesses, e.g. in loops, but not so well in more random access patterns- e.g. when traversing graph structures or accessing hash tables.

Special prefetch instructions are widely available to start a memory load to specific cache layers before the memory is actually used. Issued early enough, the data is in the fastest cache when it is needed, which allows the execution to proceed without or with much reduced blocking- probably the lowest-level example of the programming of asynchronous execution.

A very good and practical survey about current memory architectures is available in Drepper [2007]. Another excellent treatment can be found in [Hennessy and Patterson, 2007, chapter 5].

7.3 Asynchronous Processors

Fully asynchronous processors have been studied since the 1950s², but have not yet been able to break the dominance of synchronous designs. Asynchronous processors can potentially achieve very high energy efficiencies due to their ability to only switch transistors and thus dissipate significant energy in the data paths actually used. Another advantage is the missing need for large safety margins in timing to account for manufacturing tolerances, electrical instabilities, temperature etc, which allows the exploitation of the full speed the actual logic circuits are capable of in the given conditions (Sutherland [1989], Sutherland and Ebergen [2002]).

On the other hand, the synchronization logic required for each 'asynchronous' data path (dual rail coding etc) adds bulk and latency, which can probably offset the advantages of asynchronous circuits in small structures.

Hardware design is a heavily automated process, with the leading proprietary systems all assuming synchronous designs. CAD systems for the design of high-performance asynchronous systems did not yet receive the same attention as their synchronous counterparts, which makes it difficult to draw meaningful conclusions from comparing the performance of existing designs.

The need to save energy has pushed some techniques from asynchronous processors into otherwise synchronous processors, e.g. clock gating and asynchronous memory interfaces [Konstadinidis, 2002].

7.4 Multi-Core CPUs

The turning away from the conventional organization came in the middle 1960s, when the law of diminishing returns began to take effect in the effort

²One of the first asynchronous processors was the ILLIAC II, finished in 1962.

to increase the operational speed of a computer. . . . Electronic circuits are ultimately limited in their speed of operation by the speed of light . . . and many of the circuits were already operating in the nanosecond range.

W. Jack Bouknight et al. The Illiac IV System (1972)

We are dedicating all of our future product development to multicore designs. We believe this is a key inflection point for the industry.

Intel President Paul Otellini, describing Intel's future direction at the Intel Developers Forum in 2005 (both citations from [Hennessy and Patterson, 2007, chapter 4])

Successful continuous miniturization and the exploitation of instruction-level parallelism have long delayed the change from single- to multi-core processors in mainstream computers, providing a 'free lunch' of ever-faster execution of traditional sequential programs. The relative difficulty of programming distributed and even parallel computers combined with inertia (existing codebases, education) has long delayed a change away from the traditional single sequential instruction stream interface.

Miniturization eventually ran into hard problems when a significant drop in the isolation properties of structures close to a few atoms wide made further miniaturization energetically problematic. Around 2005, some chips dissipated 135W in a very small volume, of which at least 25% was statically lost to imperfect isolation [Hennessy and Patterson, 2007, p. 19].

Leveraging the existing investment in synchronous cores and shared-memory based multiprocessor communication, the leading chip manufacturers eventually decided to increase performance of individual processors further by integrating many cores on a single die, reaching 6 cores in 2010. Cores are exposed to the programmer as interconnected, independent processors with shared memory, following the mold of traditional multiprocessors with multiple chips in a single computer. Current mainstream processors start by running code on a single core, which can set up instruction pointers for other processors and start their execution by triggering a specific inter-processor interrupt ³.

Communication using shared memory combined with local caches places a question mark behind cache consistency. The latency of larger distances again challenges the creativity of hardware designers, this time to balance global memory consistency with performance. Local cache layers allow low-latency memory access if the corresponding *cache line*- typically between 16

³See e.g. <http://web.archive.org/web/20080613190758/http://www.cheesecake.org/sac/smp.html>

and 256 bytes- is not concurrently modified by other processors.

However, modification of the same cache line by multiple processors necessarily exposes the latency of memory interconnects and high-capacity memory as the cache coherency protocol sends messages between caches and memory. This determines the cost of globally atomic operations in the presence of contention- a very important characteristic for the global communication and coordination between cores.

The default memory coherency guarantees employed by different processors vary significantly, with some leaving nearly all coherency requirements to be manually enforced by the programmer using special atomic and order-restricting (barrier) instructions, and others providing relatively 'comfortable' default consistency guarantees [McKenney, 2009].

7.5 Mutual Exclusion

Mutual exclusion between concurrent entities is based on atomic decisions about the order of access. The fundamental and difficult nature of atomic operations discussed earlier suggests a need for their implementation in the lowest layer of abstraction, exposed to higher abstraction layers with suitable interfaces. The interface for atomic decisions presented to programmer by digital computer hardware mainly consists of atomic instructions of varying complexity.

All of these atomic operations can be used to implement pessimistic mutual exclusion, with the simplest atomic read or store instructions requiring more elaborate code than more complex read-modify-write instructions. The earliest known-correct mutual exclusion algorithms- Dekker's algorithm (and Semaphores) were published by [Dijkstra, 1968b]. The bakery algorithm by Lamport [1974] is another popular algorithm which only requires atomic store and load operations. Good surveys of shared-memory synchronization algorithms can be found in Andrews [1999] and Lynch [1996].

Spinlocks are a primitive based on the repeated atomic modification of a binary variable, typically using a swap instruction. The atomic instruction replaces the original value with one indicating 'locked' and returns the original value. If the original value indicates 'unlocked', the process can proceed to execute its critical section and finally setting the binary lock variable back to 'unlocked' using a normal atomic store. If the value was already 'locked', the spinlock re-spins.

Dijkstra essentially extended and abstracted the spinlock primitive with

counting semaphores [Dijkstra, 1968b]. He defined an abstract integer-like datatype semaphore which can only be manipulated with two operations- $P()$ and $V()$. The semaphore is initialized to positive integer, corresponding to the number of permissible parallel holders of the lock. The $P()$ operation tries to atomically decrement the semaphore variable, which succeeds if the previous value was greater than zero and blocks if the value was zero. The completion of the $P()$ operation equals the entry to the critical section, which is again left by executing the $V()$ operation- an atomic increment of the semaphore.

The abstract interface of semaphores allows multiple implementations of the $P()$ and $V()$ operations. Spinlocks can be employed to execute the $P()$ and $V()$ operations with mutual exclusion. Conditional read-modify-write instructions (e.g. *CAS*) can be used in an adapted form of spinning. More efficient techniques which avoid busy waiting are possible in the presence of multitasking, scheduling and typically operating system.

Herlihy [1991] showed that some of these instructions are not sufficiently powerful for the implementation of wait-free optimistic mutual exclusion. In particular, atomic read or write instructions cannot be used to solve wait-free agreement between multiple processes.

These atomic primitives are however still very useful to ensure consistent and wait-free modification of state with a single writer. The restriction to word-sized data can be lifted by using an atomic word as an indirection, e.g. as an offset or pointer, in the access of consistent state. Readers access the state by atomically reading the indirection register, followed by access to the consistent state pointed to. The single writer prepares a new, isolated version of the state in a spare buffer and- once consistent- atomically modifies the indirection register to point to the new state. Essentially, this method implements the transaction pattern for a single writer, which removes the possibility of conflicts.

More complex atomic read-modify-write instructions can be used to solve wait-free agreement for at least two processes. Those read-modify-write instructions with the capability for conditional modification (*test&set*, *CAS* etc) allow the implementation of wait-free agreement for arbitrary numbers of processes, and are widely used in the implementation of higher-level mutual agreement and mutual exclusion primitives- both optimistic and pessimistic.

8 I/O Concurrency

Computers are especially useful if they accept inputs and are able to present or otherwise communicate their results. Peripherals and CPU are relatively independent systems; clocks and processing rates are not necessarily matched (e.g. consider key presses on a keyboard), and parallel processing is the norm.

Communication in both directions is needed; let us first consider the direction from CPU to peripheral, as the CPU is directly controlled by the programmer, which simplifies the process.

The most common method for sending messages to peripherals is the manipulation of specific registers or device memory mapped into a global memory space. Pure store instructions to registers and memory are close to non-blocking, allowing the asynchronous sending of messages to peripherals. The memory or register represents a shared resource between peripheral and CPU, so potential inconsistency and- if both are able to modify the state- interference need to be considered.

For the manipulation of single word-sized memory cells or registers, atomic stores and (depending on the processor) barrier instructions guarantee consistency. Depending on the peripheral, following specific manipulation protocols/orders for registers or memory locations can yield a consistent result- e.g. by first configuring options and finally enabling a feature the options are used in. Variants of this scheme with support for larger state include the atomic pointer change mentioned earlier.

Reliable communication of bulk data with bounded buffers requires synchronization on communication from peripherals.

A very simple unidirectional ACK-based method is polling on a register or memory cell, which is modified by the device to acknowledge reception of a message- so-called *programmed I/O* (PIO). The inefficiency of polling makes this method only appropriate for short polling periods (especially if the setup latency of other mechanisms is too high) or where no other mechanisms are available. For early batch processors without multitasking and multi-user capabilities and limited urge to save power, this was not an issue.

The practical problem of interruption of running programs- which could for example have entered an infinite (polling) loop- was the motivation for the invention of the real-time interrupt in the mid-1950s. An arriving message (typically the assertion of an interrupt line) triggers the immediate execution of an *interrupt service routine* (ISR), interrupting any currently running computation in the process.

If the execution of the interrupted code should continue after the ISR has finished, the corresponding execution state at the time of the interruption needs to be captured at the start of the ISR and restored on its exit. The introduction of stacks in 1960 by Dijkstra and others (for the implementation of recursion) provided a simple solution to this problem. Similar to a subroutine call, the processor saves the instruction pointer and some processor state to the stack and creates a new stack frame before calling the ISR. The saved state might not be complete, so some additional state (typically registers) might need to be saved and restored manually.

Multiple interrupt sources are typically mapped to their respective handlers using a vector of (ISR-)addresses at a fixed location in memory.

Soon, interrupts were used to interact with other concurrent computations by modifying shared state. This had unexpected consequences; in the words of Edsger W. Dijkstra in [Dijkstra, 2001]: *It was a great invention, but also a Box of Pandora. Because the exact moments of the interrupts were unpredictable and outside our control, the interrupt mechanism turned the computer into a nondeterministic machine with a nonreproducible behavior, and could we control such a beast?*

Dijkstra went on to earn his PhD with a solution to this problem: *It comprised slightly over 900 instructions, it was the work with which I earned my Ph.D. and no one would call it an operating system: the machine was envisaged as a uniprogramming system with concurrent activity of the peripherals, which can request the attention of the central processor.*

Soon, it was clear that mutual exclusion was needed for the manipulation of shared data structures between ISRs and other code.

Masking or disabling interrupts while manipulating shared code from the only program running on the machine was the first method employed. Disabling all interrupts can have serious consequences for the latency of other interrupts, so both interrupt handler processing and the manipulation of data structures from non-interrupt context should be kept as short as possible. While running an ISR on a uniprocessor no other code (apart from other ISRs, if not disabled) can run, so any infinite loops or other infinite blocking in ISRs can lead to unrecoverable deadlock.

Interrupt priorities can further help to manage latency of critical interrupt tasks. In some processors, these are assigned statically (e.g. by the interrupt handler's position in the interrupt vector), while high-performance processors include more flexible *programmable interrupt controllers* (PIC).

Interrupts are often used as the main organization structure for the pro-

gramming of small microcontrollers, with the main code typically exiting after the setup of hardware and interrupts. This program structure minimizes latency- very important for microcontrollers used to control physical processes- and power consumption. A disadvantage of this program structure is a highly non-linear and non-local control flow which can make large programs hard to understand by looking at the code alone. A large part of the control flow depends on characteristics of the hardware which cannot always be guessed by looking at the code.

In computers with operating systems, interrupts are the dominant mechanism for synchronization in the interaction between the kernel/ hardware drivers and peripherals. Bulk data transfers between peripherals and memory typically use asynchronous *Direct Memory Access* (DMA) delegated to a separate DMA controller, which signals the completion of transfers to the CPU by triggering an interrupt. Similarly, peripherals often trigger interrupts to signal state changes in shared memory or registers. The triggered ISRs then typically unblock waiting computations by manipulating shared data structures in the operating system, which are then scheduled to continue execution by the operating system's scheduler. The same methods of PIO, DMA and interrupts are used to communicate with controllers for further communication protocols (PCI, SATA etc).

The high latencies inherent in the interaction with peripherals have resulted in the predominant use of asynchronous communication, with separation of asynchronous memory writes from synchronization on incoming communication using polling or interrupts.

9 Task or Execution Concurrency

9.1 Cooperative Multitasking

With growing performance computing hardware, human operators and slow peripherals limited the utilization of the expensive hardware. Methods to overlap the processing of multiple instruction streams- threads or tasks- were soon devised to rectify the situation.

Cooperative multitasking enables the overlapping of tasks by switching control between tasks at defined yield points, which are either manually defined or implicit in blocking operations which start a slow interaction with a peripheral. The *context switch* to another task entails the saving of pro-

cessor state needed for the resumption of the task to memory (often to a stack), figuring out which task to execute next, loading the next task's state from memory to the processor and jumping to the next instruction address. This task can either be executed directly in the code (e.g. using coroutines [Conway, 1963]), by calling a scheduling function or by returning to a calling scheduler (trampolining).

Further integration of interrupt processing with scheduling and hardware abstraction naturally leads to operating systems, but cooperative multitasking can also be implemented on bare hardware.

Initially, this capability was only used for the overlapping of unrelated computations, which were not expected to communicate or interfere with each other. This changed soon when programmers started to use the shared memory space to communicate between processes. Fortunately, mutual exclusion was relatively easy to achieve on uniprocessors as cooperative tasks could only be interrupted by enabled interrupts between yield points, but not by other tasks. As long as relevant interrupts are disabled and each interaction with a shared resource is finished between yield points, no interference can occur.

The cooperative division of processor time depends on correct behaviour of individual tasks and sufficient switch opportunities in each task. A misbehaving task entering an endless loop without yield points can stop the machine from making any further progress, leaving only interrupts to force a termination of the task. For interactive applications in later personal computers, the necessity to continue running the current task to the next yield point limits the achievable responsiveness of the system.

9.2 Preemptive Multitasking and Parallel Execution

The execution of interrupts preempts the execution of a running program without any need for explicit yield points. Using a cyclic timer interrupt, the corresponding interrupt handler can initiate a context switch at regular intervals. This allows fine-grained interleaving between tasks, but leaves the programmer exposed to arbitrary context switches and thus interference in the use of shared resources at any time. Disabling all interrupts to protect critical sections would run counter to the intention behind preemptive multitasking- safety and interactivity- and would restrict context switching more than necessary.

The mutual exclusion primitives described earlier can be extended to sup-

port preemptive multitasking by associating a queue with each entry primitive. A task trying to enter an occupied critical section registers itself in the associated queue and yields to a scheduler instead of spinning. The scheduler marks the task as blocked and does not try to run it. A process leaving the desired critical section performs the exit protocol (e.g. the V() operation on semaphores), which implicitly causes the scheduler to remove the first waiting thread from the associated queue and mark it as runnable again.

A problem with this scheme is the ability to override 'safe' scheduling with mutual exclusion methods. This becomes especially acute if the scheduler provides multiple priority levels and resources are shared between processes of different priorities. A low-priority task in a critical section can block higher-priority tasks in need of the same resource. At the same time, other medium-priority tasks could prevent the low-priority thread from running and thereby prevent progress of the blocked high-priority task, a situation known as *priority inversion*. Solutions to this problem include various forms of priority inheritance, and are especially important in real-time systems [Sha et al., 1990]. An anecdote of priority inversion on mars can be found in Jones [1997].

10 Operating Systems

Operating systems provide the programmer with a uniform and more problem-oriented interface than bare hardware and implement many global services on which programs can build. The definition of an operating system used here is minimal and mainly follows the interfaces for concurrent execution provided to tasks consisting of instructions of the underlying processor ('native code').

In a shared memory space, a bug in any task can crash and corrupt unrelated tasks or the scheduler. *Virtual memory*- running each user task in its separate virtual memory space associated with a *process*- solves this issue and separates the *kernel* (the core operating system including the scheduler) from *user space*. Context switches now occur between processes and kernel, and-after a scheduling decision- back to (possibly another) process and virtual memory space.

Many interfaces are provided for communication between processes and the kernel. The different memory access capabilities result in extensive data copying especially between kernel- and user space to preserve isolation, which can be a performance problem for processes with high communication band-

width requirements. Some specialized so-called *zero-copy* operations avoid these overheads, but at least some variants which support shared buffers between user-space and kernel need to be used carefully to avoid inconsistencies.

The most common interface is system calls, function calls which initiate a context switch to the kernel which then executes an associated operation, e.g. I/O. System calls are typically exposed to the programmer with a wrapper library, which manages calling conventions, initiates the context switch to the kernel and makes system calls easy to use from programming languages.

Some system calls return handles, opaque references to kernel-managed resources, which can then be used to interact with the resource using further system calls. Examples of handles include file descriptors for the manipulation of files and similar state, sockets for network connections and PIDs for the identification of processes.

System calls can be both blocking (fully synchronous) and non-blocking (asynchronous to the execution of the associated operation), often switchable using arguments or an operation on the handle.

Blocking calls only switch the context back to the calling process when the associated operation is finished, returning any results in the process.

Non-blocking calls on the other hand return control quickly back to the calling process, but can indicate an incomplete operation which needs to be resumed with another system call.

An additional interface mirrors the interrupt pattern in processes. Each process can register *signal handlers* corresponding to various events, which are called in the processes' memory space, sharing the stack with an interrupted thread. Similar to interrupts, signals can be masked. In the case of multi-threaded processes, the dispatching of signals to threads might need careful management by using per-thread masking. This limits the use of signals especially in libraries.

A good introduction to operating systems can be found in Silberschatz et al. [2008].

10.1 Processes and Threads

Some system calls allow the creation of new processes (*fork*) which start as a clone of the parent process and share only very limited resources with the parent (file descriptors). While processes operate in separate memory spaces, various degrees of lesser separation can be achieved with the *clone*

system call. The posix *thread* library, *pthread*s, provides additional tools for the creation of threads- additional independent instruction streams sharing a memory space with other threads in the same process, which are preemptively scheduled by the operating system.

Processes provide superior isolation and can use explicit shared memory segments or copying inter-process communication (IPC) operations for communication. Context switches between processes (common with preemptive scheduling with more processes than processors) take longer than between threads, as a hardware cache used in the translation of virtual to physical memory location needs to be flushed when the memory space switches.

Threads on the other hand need to carefully structure their communication via shared memory using mutual exclusion and atomic operations. Bugs in one thread can corrupt memory in other threads, which can be hard to track down.

Both processes and the threads described here are relatively resource-intensive. The operating system allocates significant resources with each thread or process. In Linux, at least 8KB physical memory are allocated in the kernel for a stack and control information, plus a userspace stack which consumes at least 4KB physical memory and often several MB of virtual memory- which severely limits the number of stacks on 32-bit systems. Context switches and process/thread creation consume at least several hundred, but typically more than 1000 CPU cycles.

These costs have led to the development of *threading models* with lower overheads in user space.

10.1.1 Threading Models

- 1:1** threading is the association of each user space thread with one separately scheduled kernel thread- in other words, the plain processes or pthreads as described above.
- m:1** threading is the mapping of multiple user space threads to a single kernel thread. The meaning of *thread* in this context is less tightly bound to a contiguous sequence of instructions- *thread of control* fits probably best. Many other names have been used- e.g. fibers or green threads. Userspace threads are typically scheduled cooperatively (very similar to cooperative multitasking) and can be as 'light' as a few machine words or nearly as heavy as an operating system process. Preemptive

scheduling can be implemented using signals and the POSIX library procedures `getcontext` and `setcontext`, but requires a stack per thread. I/O is typically managed using non-blocking I/O and *event loops* (see section 10.2). The single underlying operating system thread prevents the utilization of multiple physical processors.

m:n threading is an extension of userspace threading to multiple operating system threads, often matching the number of processors in the system. This threading model is very similar to *thread pools*, in which userspace threads or a description of work to be done is submitted to a work queue and successively executed by a pool of threads. This paradigm works well for truly independent (no mutual exclusion needed) and non-blocking computations- e.g. data parallelism⁴ or task parallelism where the restrictions are enforced by a programming language or guaranteed by the programmer.

Variants of thread pools are used as backends to many libraries and language runtimes. A similar principle can also be extended to many machines over networks.

10.2 I/O Concurrency and Event Loops

Concurrent I/O operations using blocking system calls require the association of a dedicated kernel thread with each operation. Especially for computationally 'lightweight', but I/O-heavy tasks, this causes significant overheads and limits the degree of concurrency achievable with limited resources. An alternative is the use of non-blocking I/O, in which system calls return quickly even if the attempted operation is not yet complete, returning information about progress and errors. The state associated with each concurrent task can be minimized, which enables very high levels of I/O concurrency, e.g. in network servers.

As mentioned in section 4, non-blocking send operations typically return the amount of data actually sent to allow another call with the remaining data, or an error code. Receive operations generally return any buffered data and an indication if more data can be expected or an error occurred.

Incomplete calls can be resumed by repeated polling, but are mostly used in conjunction with *select*-like system calls (see also section 4.1). These calls

⁴The parallel application of similar and independent operations to parts of a large data set.

block on a set of (handle, event mask) pairs, in which the event mask normally indicates interest in a combination of readable, writeable and sometimes handle-specific events. An additional timeout parameter can be specified, which causes *select* to return an empty list of events if no other events occurred before the timeout elapsed. Once at least one of the registered events for a handle occurs (e.g. the handle becomes writeable), the call returns the list of handles with new events.

The returned handles can then be processed sequentially or concurrently—either in the same operating system thread, or by e.g. pushing the operation to a work queue associated with a thread pool.

For each returned handle, the associated task state is looked up in a local data structure and the associated handler called with this state. The handler calls its non-blocking I/O operation and normally makes progress, resulting in a call to the next handler in a callback chain if it is complete, or another direct registration with the event loop to continue the operation once the handle is ready again.

After all returned handles are processed, the *event loop* repeats with another call to *select* or similar.

Bare event loops are a relatively low-level method to control highly concurrent execution. The control flow corresponding to the handling of a complete request is split into many handlers, each corresponding to a single, potentially long-running I/O operation. This structure resembles a *state machine*, with transitions both back into the same state if incomplete, or to the next state—the next handler. The overall control flow contains many non-local jumps, which makes the program hard to comprehend and write.

The use of libraries can be problematic if blocking system calls are used. This significantly reduces the library choices for many applications and often forces a reimplementaion based on the event loop and data structures used.

Long-running computations need to be manually split to avoid blocking the processing of other tasks for extended periods— and issue inherent in all cooperative multitasking systems.

The *select* call is normally implemented as a form of polling based on the inspection of a bitmap. Select-capable handles correspond to small integers in Unix-like systems, so the size of the bitmap is determined by the highest handle number in use. An alternative, but still inefficient method for the monitoring of very sparse handle sets is the *poll* call, which traverses the list of monitored handles. Both are obsoleted by non-polling primitives (e.g. *epoll*

in Linux ⁵, *kqueue* [Lemon, 2000] in FreeBSD and derivatives) based on events triggered directly by interrupts, which keeps their runtime independent of the number of monitored handles.

In Unix-like systems, the 'everything is a file' philosophy allows the use of select-like primitives for the monitoring of events connected to a wide range of resources. Additionally, signal handlers or other concurrent code can be used to trigger a blocked select call using the so-called 'self-pipe' trick- writing a byte to a non-blocking pipe (a handle-based IPC primitive) monitored for reading by a select call.

Abstraction libraries which select the best available mechanism and provide additional support for this programming style are available (e.g. [Provos and Mathewson, 2010] and [Lehmann, 2010]). A survey of options for the implementation of highly concurrent network servers can be found in Kegel [2010].

User interface frameworks use event loops internally to collect UI events (mouse clicks, keyboard presses) and dispatch them to their respective handlers. The I/O multiplexing calls used are often specific to the windowing system.

Runtimes for languages with a focus on high degrees of I/O concurrency are also using event loops internally (e.g. Haskell [Marlow et al., 2004] and Erlang). The disadvantages of event loops mentioned above can be mostly addressed by languages.

11 Programming Languages and Runtimes

Asynchronous execution can theoretically be programmed in any language (even binary instructions) by performing I/O or starting concurrent computations in the form of processes, threads or userspace threads. 'Higher-level' programming languages and library abstractions typically restrict what can be programmed, but expand what can be practically programmed and reasoned about by humans. Programs written in higher-level languages conform to specific structures, e.g. avoid *goto* in favour of structured loops, conditionals and method calls. These *control abstractions* were soon complemented with *data abstractions*, the encapsulation of data structures with defined interfaces for their manipulation- Abstract Data Types (ADT) [Liskov and Zilles, 1974].

⁵See `man epoll`, introduced in Linux 2.5.44.

11.1 Objects as State or Processes

Object-oriented programming extends this principle to structure entire programs as interaction between independent *objects* encapsulating state and behaviour. Several additional features (including classes, inheritance, dynamic dispatch) are not very important from a concurrency perspective, but both encapsulation and interaction between independent objects *are* interesting properties.

The concurrent execution of each object as a passive 'server' interacting with other objects using *message passing* expresses a high degree of concurrency and implies mutual exclusion by running each object as a single thread. In addition to local computation, each object (or process) has the ability to asynchronously create new objects/processes. Communication between objects can be organized using synchronous or asynchronous message passing, one convertible in the other by creating processes to asynchronously send or synchronize on a reply.

Another, more popular execution scheme treats interaction between objects as blocking function calls executed on the caller's thread, resulting in a structured, but essentially synchronous, shared-memory execution model. In this execution model, the object abstraction can support concurrency by providing language support for the enforcement of defined interfaces to shared data and mutual exclusion in the manipulation of an object's data from multiple threads. Apart from this, methods applicable to general imperative programming can be employed to implement asynchronous execution- manual thread creation (see previous sections) and asynchronous function calls (see 11.2).)

11.1.1 Object State Synchronization: The Monitor

The monitor primitive was invented by C.A.R. Hoare and Per Brinch Hansen to organize mutually exclusive access to methods of an object. In addition to a global mutex per object with associated FIFO queue, queues associated with condition variables can be used to improve control over scheduling.

After entering the mutex, a thread might leave the mutex again by *waiting* on a condition variable. Other threads might *signal* condition queues to release either one or all waiting threads to the mutex entry queue. Depending on the *monitor discipline*, a signaling thread executing in mutual exclusion either continues in the mutual exclusion section or is relegated to the front

or back of the entry queue.

This concept was implemented in many object-oriented languages including Concurrent Pascal, Modula-3, Java (synchronized keyword) and C#.

11.2 Asynchronous Function Calls

Asynchronous function calls can be used both for I/O concurrency and parallel computations. Following the familiar pattern, they typically return a handle, which can later be used to synchronize on the result of the computation. Additionally, some types of handles allow non-blocking polling and multiplexed synchronization. Handles in this application are typically called *promises* or *futures* [Liskov and Shriram, 1988].

Most of the machinery can be implemented in libraries- e.g. on top of thread pools. The synchronization step in a pure library implementation involves an explicit call to a synchronization function using the handle. In most languages, implicit synchronization- the use of a handle as a regular value which synchronizes on access- requires the introduction of a new native data type with support from compiler and runtime system.

Blocking synchronization or system calls are problematic in connection with thread pools, as the number of worker threads is intentionally limited. In I/O-heavy applications, the degree of concurrency is either limited to a static number of threads in the pool, or the pool reverts to 1:1 threading for I/O by creating additional threads.

11.3 Data/Loop Parallelism with Annotations

OpenMP is an interface for the specification of parallel tasks, e.g. of parallel loops, using inline pragma statements in newer C, C++ and Fortran compilers. Specific directives are available for mutual exclusion, data decomposition and synchronization (especially barriers).

An advantage of the annotation system is the ability to incrementally parallelise existing single-threaded code by adding a few annotations. As is expected in the relatively low-level languages targeted, control of side effects and mutual exclusion is left to the programmer.

Most current runtimes focus on shared-memory architectures and thread pools (e.g. in GCC), but extensions to clusters are in development. In the cluster area, the Message Passing Interface (MPI) library is widely used for scientific applications, but is not directly integrated in a language.

11.4 Lightweight User-Space Threads and Non-Blocking I/O Concurrency

Lightweight threads- normally implemented in user-space with non-blocking I/O- can be both used to map relatively fine-grained task parallelism to hardware parallelism and for the implementation of high or very high I/O concurrency.

The needed control over scheduling and blocking favours the implementation of lightweight threads in the runtime of higher-level languages which can restrict access to blocking system calls. Similarly, the ability of static isolation enforcement between threads or 'processes' actually sharing an underlying memory space incurs less overheads than virtual memory space protection as implemented by operating system processes.

Building on a non-blocking I/O subsystem, these languages are free to expose I/O operations with a blocking or non-blocking interface. Fine-grained yield points for cooperative scheduling can be inserted by the compiler rather than the programmer.

Erlang, a functional language centered around message passing between lightweight threads (called processes to emphasize their large degree of isolation), is a relatively popular example of this type. A blocking receive operation supports pattern matching on messages and triggers an execution which can asynchronously send messages to other threads in response. A supervision hierarchy between processes is used to cleanly handle errors. Local state is only allowed in the form of arguments to functions, which are often recursive. This clear definition of arguments allows the dynamic replacement of looping code at runtime, an important feature for Erlang's original use in telecommunication switches. Global shared state is implemented as a transactional database which is again accessed using message passing.

Haskell provides similar features as Erlang for lightweight concurrency, but differs in its strict separation between purely functional from effectful code and in its lazy evaluation model, which only evaluates an expression when it is actually needed. Implicit synchronization on handles (futures) is particularly easy to achieve in Haskell without any specific compiler support as a byproduct of its general by-need evaluation semantics.

Pure computations in Haskell can be more aggressively transformed than similar effectful code, which makes Haskell a good candidate for the automatic parallelisation of computationally expensive code. The current Haskell implementation supports semi-automatic parallelization of purely functional

code using annotations to trigger *sparks*, the asynchronous evaluation of subexpressions before they would be evaluated by-need [Trinder et al., 1998]. With lazy evaluation, no change in the code is needed to synchronize on the result.

11.5 Implicit Parallelism

Could programming languages perform similar automatic parallelisation as processors do for instruction-level parallelism?

To some degree, this is already done.

For imperative languages, the Static Single Assignment Algorithm (SSA) [Cytron et al., 1991] helps to build a data and control dependency graphs, which can expose cycles (for loop parallelism or Single Instruction Multiple Data / SIMD instruction) or parallel branches. This analysis depends on purity- the absence of side effects not captured by the direct dependency graph. In principle, any memory access can launch missiles, so compilers simply assume code to be relatively pure and rely on special annotations-memory barriers- to enforce stricter ordering. Still, compilers for imperative languages are limited in their reordering by the e.g. library calls and other code they cannot directly analyze. The first auto-parallelizing compiler features focus on loop parallelism, e.g. the Gnu Compiler Collection (GCC) autopar feature [GCC, 2010a].

More aggressive transformations are possible in languages with a strong separation between purely functional and effectful code, but experiments e.g. in Haskell [Harris and Singh, 2007] have so far resulted in limited success.

Even in languages which expose a very high degree of parallelism, a precise cost model of the execution environment is needed to make good decisions on granularity, scheduling and resource use. A minimum granularity is imposed by communication latency and bandwidth inherent in a parallel execution- a single integer addition will always be slower when executed in parallel. Too coarse granularity on the other hand exposes less (or no) parallelism. An accurate cost model needs to take into account a very high number of details-memory resources, cache behaviour, communication latency and bandwidth, data locality, expected inputs- all not specified in programs.

Some of this information could be automatically collected- a technique already practiced by Just In Time (JIT) compilers at runtime- which introduces costs of its own, but could in principle dynamically choose from a set of equivalent algorithms and distribution strategies depending on inputs.

Another possible approach could be the separate development of cost models for particular execution environments- a method already successfully employed in computer aided design.

12 Socketmachine: A User-Space Lightweight Thread Library with Non-Blocking I/O

Socketmachine [Wicke, 2009] is a proof-of-concept implementation of an extremely lightweight userspace threading library for I/O-bound applications, especially highly concurrent network servers. It is based on the *Libevent* [Provos and Mathewson, 2010] event loop, scheduler and utilities.

A disadvantage of bare event loops (see section 10.2) is the difficulty of programming state machines in which the control flow is chopped up into many individual handlers connected by non-local jumps.

Socketmachine provides a set of I/O operation and subroutine call macros which allow the programmer to use an efficient event loop without splitting the control flow in explicit handlers, resulting in a programming style virtually identical to a sequential program with blocking I/O operations and subroutine calls.

12.1 Implementation

The implementation is entirely based on macros, which results in several restrictions for the procedures in a potentially 'blocking' call graph.

All procedures in the call graph need to take a continuation⁶ pointer argument at a fixed position in their signature. Additionally, the current prototype requires two additional standard arguments- a file descriptor and a status argument- for compatibility with Libevent callbacks, but this requirement could be lifted.

An example procedure illustrating the signature, local state and simple, potentially blocking operations:

```
int echo_handler(int fd, short status, struct conti *co){
    // Declare local state
    SM_SETUP_this (co, &echo_handler,
                  int total;
```

⁶Here: Local call state linked to further call state in a call graph.

```

        int i;
        unsigned char buf[64];
        int read;
    );
    // initialize total
    this->total = 0;
    for(this->i = 0; this->i < 10; this->i++) {
        // Read at most 64 bytes from fd into buf. This could block.
        this->read = SM_READ(fd, this->buf, 64);

        // Handle error status..
        if(!this->read) {
            // print the error
            perror(__func__);
            SM_RETURN(-1, NULL);
        }

        // ..or do something with the buffer contents..
        this->total += this->read;
        if(!SM_WRITE(this->buf, this->read, fd)) {
            perror(__func__);
        }
    }
    // return to or actually call the caller
    SM_RETURN(this->total, NULL);
}

```

Additional arguments are passed in using a 'state' member of struct `conti`, normally transparently set up using a wrapper macro similar to `SM_READ` or `SM_WRITE`.

Local data in a method is initially declared on the stack to avoid memory allocation overheads for non-blocking calls. The `SM_SETUP_this` macro serves to mark local state members and implicitly defines a local struct pointer, *this*, so an access to local state *foo* has the form `this->foo`. Calls to potentially blocking subroutines are wrapped in another macro, `SM_CALL`, or convenience wrappers like `SM_READ` and `SM_WRITE`.

12.2 Operation Principle

Once a non-blocking system call returns `EWOULDBLOCK` (instead of blocking), the corresponding I/O method registers with the Libevent event loop and returns a status code to the caller which signals it to unwind the stack.

The caller saves its own function pointer and the instruction pointer of the call site to the local continuation frame (using portable computed labels, a GCC feature) and checks if the local continuation is still on the stack, in which case it is transferred to the heap and the callee's continuation pointer is updated to point to the continuation on the heap. Finally, the process repeats by returning the same status code until the top-level continuation is reached, which allows Libevent to process the next request. The result is a callback/continuation chain on the heap, starting from the blocked I/O operation and corresponding to the call graph.

Once the I/O operation is finished- successful or not-, the `SM_RETURN` macro returns or calls (if previously blocked) the continuation's function pointer while passing in the continuation and the result of the I/O operation. A conditional jump in the local state macro at the start of the called procedure checks the instruction pointer field in struct `conti` and- if set- jumps to the corresponding code position just after the original call and returns the passed-in result from the original callee. The 'blocking' call macro (using another GCC feature, multi-statement macro expressions) now evaluates to this result, and the local execution in the procedure proceeds, e.g. by handling a returned error code or performing further operations. Finally, a call to `SM_RETURN` passes the result to the original caller- using a plain return if nothing blocked, or a call if the stack was unwound.

All access to local state is using the *this* pointer indirection, which makes local state independent of its location on the stack or the heap.

12.3 I/O Operations

Handlers for the transfer of data between memory buffers and file descriptors (sockets, actual files, pipes) and between two file descriptors are implemented. For the transfer between two file descriptors, both a naïve iterative read/write procedure using user-space buffers and a more efficient variant based on the relatively new `splice` system call introduced in Linux 2.6.17 are available. `Splice` avoids the double copying from kernel- to user-space and back by mapping kernel memory in a pipe buffer, and from the pipe buffer to another

file descriptor. In the Linux kernel, this so-called 'zero-copy' approach can result in a direct DMA transfer from the page cache to the network card- a very efficient method for bulk transfers, e.g. while serving large files.

Additional operations allow the creation of new threads, blocking for a predefined time and on arbitrary conditions associated with file descriptors.

12.4 Scheduling, Timeouts and Error Handling

All blocking operations support a timeout argument, which is passed to the Libevent scheduler. Libevent sorts timeouts using a minheap and sets the input multiplexing operation (normally epoll on Linux) timeout to the time remaining until the next timeout event. Priorities are available in Libevent, but are not currently exposed by the Socketmachine operations.

Errors are handled similar to normal system calls, the global `errno` variable is not touched. All implicit local resources are freed automatically on return- both successful and with error. Explicitly allocated resources need to be deallocated before returning. The sequential and local code structure should make it easier to cover all error code paths, especially when compared with state machine callbacks.

12.5 Related Work

Protothreads [Dunkel, 2009] follow a very similar overall concept and implementation, but miss support for the preservation of local state across blocking operations. I/O is supported using simple polling, no event loop integration is available. The main focus of protothreads is the programming of microcontrollers, where an event loop library and operating system would not make sense.

Tatham [2000] describes implementation techniques for coroutines in C, which was the initial inspiration for the Socketmachines implementation.

Capriccio [von Behren et al., 2003] intercepts system calls from regular blocking code and transforms them into non-blocking operations including epoll I/O multiplexing. A regular threading interface (pthreads-compatible) is emulated and regular stack usage is supported. The size of stacks is minimized using static analysis and linked stack segments where variable-size allocations and recursion are possible.

GCC split stacks [GCC, 2010b] is a compiler feature to minimize virtual memory use for stacks by introducing non-contiguous stacks- very similar to

Capriccio. This is currently in development.

The Go programming language supports similar split stacks combined with synchronous message passing between 'goroutines' and event-loop based non-blocking I/O including *epoll* support.

Similarly, runtimes of at least the *Haskell* [Marlow et al., 2004], *Erlang* and *E* [Miller and Bornstein, 1997] programming languages use I/O multiplexing and lightweight threading internally.

Epoll support is in development and expected for Haskell's upcoming GHC 6.14 runtime [Haskell Bug Tracker Ticket, 2010].

12.6 Summary and Outlook

The implementation of the Socketmachine library was an interesting learning exercise about low-level thread implementations and non-blocking I/O. The result performs similar to regular event-driven state machines, but is only a minor improvement in terms of composability and usability. Syntactic improvements using a preprocessor or *semantic patches* [Coccinelle] could improve the usability, but not composability with external libraries potentially containing blocking I/O operations.

The use of existing high-level languages with good support for high degrees of I/O concurrency and parallelism- particularly Haskell and Erlang- appears more promising for server applications from my current point of view. Some of the high-performance features implemented in the Socketmachine library can potentially be ported, which makes them useful to a wider range of applications.

References

- G. R. Andrews. *Foundations of Parallel and Distributed Programming*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999. ISBN 0201357526.
- R. Bush and D. Meyer. Some Internet Architectural Guidelines and Philosophy. RFC 3439 (Informational), Dec. 2002. URL <http://www.ietf.org/rfc/rfc3439.txt>.
- Coccinelle. Program matching and transformation engine, supports semantic patches. URL <http://coccinelle.lip6.fr/>.

- M. E. Conway. Design of a separable transition-diagram compiler. *Commun. ACM*, 6(7):396–408, 1963. ISSN 0001-0782. doi: <http://doi.acm.org/10.1145/366663.366704>.
- R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13(4):451–490, 1991. ISSN 0164-0925. doi: <http://doi.acm.org/10.1145/115372.115320>.
- D. Deutsch. Quantum theory, the church-turing principle and the universal quantum computer. 400:97–117, 1985.
- E. W. Dijkstra. The structure of the “the”-multiprogramming system. *Commun. ACM*, 11(5):341–346, 1968a. ISSN 0001-0782. doi: <http://doi.acm.org/10.1145/363095.363143>.
- E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.
- E. W. Dijkstra. Cooperating sequential processes. published as EWD123, 1968b. URL <http://www.cs.utexas.edu/users/EWD/ewd01xx/EWD123.PDF>.
- E. W. Dijkstra. My recollections of operating system design. circulated privately as EWD1303, Apr. 2001. URL <http://www.cs.utexas.edu/users/EWD/ewd13xx/EWD1303.PDF>.
- U. Drepper. What every programmer should know about memory, 2007. URL <http://people.redhat.com/drepper/cpumemory.pdf>.
- A. Dunkel. Protothreads, 2009. URL <http://www.sics.se/~adam/pt/>.
- A. Einstein. Zur Elektrodynamik bewegter Körper. In *Annalen der Physik*, volume 17, 1905.
- Erlang. URL <http://www.erlang.org>.
- F. Fich and E. Ruppert. Hundreds of impossibility results for distributed computing. *Distrib. Comput.*, 16(2-3):121–163, 2003. ISSN 0178-2770. doi: <http://dx.doi.org/10.1007/s00446-003-0091-y>.
- M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process, 1985.

- N. Francez, C. Hoare, and W. de Roever. Semantics of nondeterminism, concurrency and communication. *Mathematical Foundations of Computer Science*, pages 191–200, 1978.
- GCC. Gnu compiler collection. URL <http://gcc.gnu.org/>.
- GCC. Gnu Compiler Collection - automatic parallelisation, 2010a. URL <http://gcc.gnu.org/wiki/Graphite/Parallelization>.
- GCC. Gnu Compiler Collection - split stacks, 2010b. URL <http://gcc.gnu.org/wiki/SplitStacks>.
- R. Ginosar. Fourteen ways to fool your synchronizer. In *ASYNC '03: Proceedings of the 9th International Symposium on Asynchronous Circuits and Systems*, page 89, Washington, DC, USA, 2003. IEEE Computer Society. ISBN 0-7695-1898-2.
- Go programming language. URL <http://golang.org/>.
- T. Harris and S. Singh. Feedback directed implicit parallelism. In *ICFP '07: Proceedings of the 12th ACM SIGPLAN international conference on Functional programming*, pages 251–264, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-815-2. doi: <http://doi.acm.org/10.1145/1291151.1291192>.
- Haskell. URL <http://www.haskell.org>.
- Haskell Bug Tracker Ticket, may 2010. URL <http://hackage.haskell.org/trac/ghc/ticket/635>.
- J. L. Hennessy and D. A. Patterson. *Computer Architecture - A Quantitative Approach*. Morgan Kaufmann, fourth edition, 2007.
- M. Herlihy. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1):124–149, 1991. ISSN 0164-0925. doi: <http://doi.acm.org/10.1145/114005.102808>.
- M. P. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12:463–492, 1990.

- M. Jones. What really happened on mars?, 1997. URL https://research.microsoft.com/en-us/um/people/mbj/Mars_Pathfinder/Mars_Pathfinder.html.
- D. Kegel. The c10k problem, 2010. URL <http://www.kegel.com/c10k.html>.
- D. Kinniment and J. Woods. Synchronisation and arbitration circuits in digital systems. *ieep*, 123(10):961–966, October 1976.
- G. K. e. a. Konstadinidis. Implementation of a third-generation 1.1-ghz 64-bit microprocessor. *IEEE Journal of Solid-State Circuits*, 37(11), 2002. URL <http://research.sun.com/vlsi/pubs/01046088.pdf>.
- L. Lamport. A new solution of dijkstra’s concurrent programming problem. *Commun. ACM*, 17(8):453–455, 1974. ISSN 0001-0782. doi: <http://doi.acm.org/10.1145/361082.361093>. URL <http://research.microsoft.com/en-us/um/people/lamport/pubs/bakery.pdf>.
- L. Lamport. Buridan’s principle, 1986. URL <http://research.microsoft.com/en-us/um/people/lamport/pubs/buridan.pdf>.
- M. Lehmann. Libev- an event loop library, 2010. URL <http://software.schmorp.de/pkg/libev.html>.
- J. Lemon. Kqueue: A generic and scalable event notification facility, 2000. URL <http://people.freebsd.org/~jlemon/papers/kqueue.pdf>.
- B. Liskov and L. Shrira. Promises: linguistic support for efficient asynchronous procedure calls in distributed systems. In *PLDI ’88: Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*, pages 260–267, New York, NY, USA, 1988. ACM. ISBN 0-89791-269-1. doi: <http://doi.acm.org/10.1145/53990.54016>.
- B. Liskov and S. Zilles. Programming with abstract data types. *SIGPLAN Not.*, 9(4):50–59, 1974. ISSN 0362-1340. doi: <http://doi.acm.org/10.1145/942572.807045>.
- N. A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1996. ISBN 1558603484.

- S. Marlow, S. P. Jones, and W. Thaller. Extending the haskell foreign function interface with concurrency. In *Haskell '04: Proceedings of the 2004 ACM SIGPLAN workshop on Haskell*, pages 22–32, New York, NY, USA, 2004. ACM. ISBN 1-58113-850-4. doi: <http://doi.acm.org/10.1145/1017472.1017479>.
- M. D. McIlroy. Summary- what's most important, 1964. URL <http://cm.bell-labs.com/cm/cs/who/dmr/mdmpipe.pdf>. Summary of design brief for Unix outlining the pipeline.
- P. E. McKenney. Memory barriers: a hardware view for software hackers, 2009. URL <http://www.rdrop.com/users/paulmck/scalability/paper/whymb.2009.04.05a.pdf>.
- A. A. Michelson and E. W. Morley. On the relative motion of the earth and the luminiferous ether. *American Journal of Science*, 34:333–345, 1887.
- M. S. Miller and D. Bornstein. E programming language, 1997. URL <http://www.erights.org/>.
- MPI. Message passing interface. URL <http://www.mpi-forum.org/>.
- M. A. Nielsen and I. L. Chuang. *Quantum Computation and Quantum Information*. Cambridge University Press, 1 edition, October 2000. ISBN 0521635039.
- OpenMP. The OpenMP API specification for parallel programming. URL <http://openmp.org/wp/>.
- J. Postel. Transmission Control Protocol. RFC 793 (Standard), Sept. 1981. URL <http://www.ietf.org/rfc/rfc793.txt>. Updated by RFCs 1122, 3168.
- N. Provos and N. Mathewson. Libevent- an event loop library, 2010. URL <http://monkey.org/~provos/libevent/>.
- L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Trans. Comput.*, 39(9):1175–1185, 1990. ISSN 0018-9340. doi: <http://dx.doi.org/10.1109/12.57058>.
- C. E. Shannon. A Mathematical Theory of Communication. *The Bell System Technical Journal*, 27:379–423, july, october 1948.

- N. Shavit and D. Touitou. Software transactional memory. In *Proc. of the 12th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 204–213, 1995.
- A. Silberschatz, P. B. Galvin, and G. Gagne. *Operating System Concepts*. Wiley Publishing, 2008. ISBN 0470128720.
- I. E. Sutherland. Micropipelines. *Commun. ACM*, 32(6):720–738, 1989.
- I. E. Sutherland and J. Ebergen. Computers without clocks. *Scientific American*, 287(2), Aug. 2002.
- S. Tatham. Coroutines in C, 2000. URL <http://www.chiark.greenend.org.uk/~sgtatham/coroutines.html>.
- P. W. Trinder, K. Hammond, H.-W. Loidl, and S. L. Peyton Jones. Algorithm + strategy = parallelism. *J. Funct. Program.*, 8(1):23–60, 1998. ISSN 0956-7968. doi: <http://dx.doi.org/10.1017/S0956796897002967>.
- D. M. Tullsen, S. J. Eggers, and H. M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *ISCA*, pages 392–403, 1995.
- A. M. Turing. Intelligent machinery. submitted to the National Physical Laboratory, 1948.
- R. von Behren, J. Condit, F. Zhou, G. C. Necula, and E. Brewer. Capriccio: scalable threads for internet services. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 268–281, New York, NY, USA, 2003. ACM. ISBN 1-58113-757-5. doi: <http://doi.acm.org/10.1145/945445.945471>.
- P. Wadler. Monads for functional programming. In *Program Design Calculi: Proceedings of the 1992 Marktoberdorf International Summer School*. Springer-Verlag, 1993. URL <http://homepages.inf.ed.ac.uk/wadler/papers/marktoberdorf/baastad.pdf>.
- G. Wicke. Socketmachine: A user-space thread library with non-blocking I/O integration, 2009. URL <http://dl.wikidev.net/socketmachine/>. Source code.
- K. Zuse. Rechnender Raum. In *Elektronische Datenverarbeitung*, volume 8, pages 336–344, 1967.