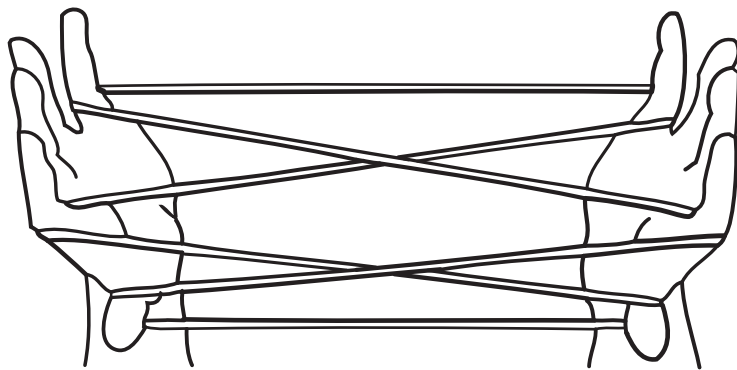


Institutionen för datavetenskap

Department of Computer and Information Science

Master's Thesis

Multithreaded concurrency for faster web browsers



Anders Karlsson

Reg Nr: LITH-IDA-EX--07/069--SE
Linköping 2007



Linköpings universitet
INSTITUTE OF TECHNOLOGY

Department of Computer and Information Science
Linköpings universitet
SE-581 93 Linköping, Sweden

Institutionen för datavetenskap

Department of Computer and Information Science

Master's Thesis

Multithreaded concurrency for faster web browsers


Anders Karlsson

Reg Nr: LITH-IDA-EX--07/069--SE
Linköping 2007

Supervisor: **Niklas Larsson**
Opera Software

Examiner: **Christoph Kessler**
IDA, Linköpings universitet

Department of Computer and Information Science
Linköpings universitet
SE-581 93 Linköping, Sweden

	Avdelning, Institution Division, Department Division of Software and Systems Department of Computer and Information Science Linköpings universitet SE-581 83 Linköping, Sweden	Datum Date 2007-12-20										
Språk Language <input type="checkbox"/> Svenska/Swedish <input checked="" type="checkbox"/> Engelska/English <input type="checkbox"/> _____	Rapporttyp Report category <input type="checkbox"/> Licentiatavhandling <input checked="" type="checkbox"/> Examensarbete <input type="checkbox"/> C-uppsats <input type="checkbox"/> D-uppsats <input type="checkbox"/> Övrig rapport <input type="checkbox"/> _____	ISBN _____ ISRN LITH-IDA-EX--07/069--SE <table border="0"> <tr> <td>Serietitel och serienummer</td> <td>ISSN</td> </tr> <tr> <td>Title of series, numbering</td> <td>_____</td> </tr> </table>	Serietitel och serienummer	ISSN	Title of series, numbering	_____						
Serietitel och serienummer	ISSN											
Title of series, numbering	_____											
URL för elektronisk version http://urn.kb.se/resolve?urn=urn:nbn:se:liu:diva-10689												
<table border="0"> <tr> <td>Titel</td> <td></td> </tr> <tr> <td>Title</td> <td>Multithreaded concurrency for faster web browsers</td> </tr> <tr> <td> </td> <td></td> </tr> <tr> <td>Författare</td> <td>Anders Karlsson</td> </tr> <tr> <td>Author</td> <td></td> </tr> </table>			Titel		Title	Multithreaded concurrency for faster web browsers	 		Författare	Anders Karlsson	Author	
Titel												
Title	Multithreaded concurrency for faster web browsers											
Författare	Anders Karlsson											
Author												
Sammanfattning Abstract <p>Web browsers do lots of things concurrently. This applies both to presentation of new web pages as well as background tasks such as script execution and image animation. The way this concurrency is implemented differs.</p> <p>One common way to implement concurrency is multithreading. This is also an efficient way to exploit the power of multiple processor cores. Unfortunately there are many pitfalls and a bad design can give even worse performance and stability than for a single-threaded application. Still the trend of an increasing number of processor cores in normal PCs has made it a hot topic. This has made the developers of applications that previously used other concurrency mechanisms, such as cooperative multitasking, interested in multithreading.</p> <p>This thesis examines possible ways that multithreading can be added to a previously single-threaded web browser. Possible solutions are examined and detailed simulations and performance measurements are performed. Design patterns and design principles that can help implementation of multithreading are identified and described. Finally design suggestions and a step by step list for a redesign of a single-threaded browser is presented.</p>												
Nyckelord Keywords web browsers, SMP, multithreading, Pthreads, design patterns, worker threads, Perfmon2, cache												

Abstract

Web browsers do lots of things concurrently. This applies both to presentation of new web pages as well as background tasks such as script execution and image animation. The way this concurrency is implemented differs.

One common way to implement concurrency is multithreading. This is also an efficient way to exploit the power of multiple processor cores. Unfortunately there are many pitfalls and a bad design can give even worse performance and stability than for a single-threaded application. Still the trend of an increasing number of processor cores in normal PC:s has made it a hot topic. This has made the developers of applications that previously used other concurrency mechanisms, such as cooperative multitasking, interested in multithreading.

This thesis examines possible ways that multithreading can be added to a previously single-threaded web browser. Possible solutions are examined and detailed simulations and performance measurements are performed. Design patterns and design principles that can help implementation of multithreading are identified and described. Finally design suggestions and a step by step list for a redesign of a single-threaded browser is presented.

Sammanfattning

Webbläsare utför många saker samtidigt. Detta gäller både visning av webbsidor så väl som bakgrundoperationer som skriptexekvering och bildanimation. Sättet som denna samtidighet implementeras på varierar.

Ett vanligt sätt att implementera samtidighet är multitrådning. Detta är också ett effektivt sätt att exploatera kraften i flera processorkärnor. Tyvärr finns det många fallgropar och en dålig design kan till och med ge sämre prestanda och stabilitet än för ett enkeltrådat program. Dock har trenden med ett ökande antal processorkärnor i vanliga PC-datorer gjort detta till ett hett ämne. Detta har medfört att utvecklare av applikationer som tidigare använde andra former av samtidighet, så som cooperative multitasking, har blivit intresserade av multitrådning.

Denna rapport undersöker möjliga sätt som multitrådning kan införas i en tidigare enkeltrådad webbläsare. Möjliga lösningar undersöks och detaljerade simuleringar och prestandamätningar genomförs. Designmönster och designprinciper som kan underlätta implementationen av multitrådning identifieras och beskrivs. Slutligen presenteras designförslag och en steg-för-steg-lista för omdesign av en enkeltrådad webbläsare.

Acknowledgments

I would like to thank all that supported me when I returned back to the university, to finish my studies.

Most importantly, my family and my friends, who kept on supporting and helping me, you have a larger part of my degree than I can ever express or enough thank for!

I wish I could give a copy of this report to my grandfather Göte, who throughout my entire childhood encouraged me to seek knowledge and explore whatever I found interesting. My grandfather is now long gone, but it is much because of him I took up my postponed studies. He really wanted to see me graduate and would have loved to see this thesis.

Contents

1	Introduction	1
1.1	Processor evolution	1
1.2	Web browser evolution	2
1.3	Concurrency	3
1.4	Opera Software	4
1.5	Document outline	4
2	Web browser functionality	5
2.1	Retrieving data	5
2.2	Parsing HTML	5
2.3	Decoding of images	6
2.4	Recursion	6
2.5	Cascading stylesheets	7
2.6	Document layout	7
2.7	Document painting	8
2.8	Form response	8
2.9	Cookies	9
2.10	Client side scripting	9
2.11	Plugins	9
3	Multiprocessing Hardware	11
3.1	The processor	11
3.2	RAM memory	12
3.3	Cache memory	12
3.4	Multiprocessing	14
4	Concurrency mechanisms	17
4.1	The process	17
4.2	Concurrency levels	19
4.3	Cooperative multitasking	19
4.4	Multithreading	21
4.5	Race conditions	23
4.6	Synchronization	24

5	Threadlab evaluation system	27
5.1	System design	27
5.2	Supported concurrency variants	28
5.3	Evaluation scripts	30
5.4	Web browser simulation	31
6	Patterns and methods	39
6.1	Scoped locking	39
6.2	Strategized locking	41
6.3	Thread specific storage	43
6.4	Thread pool	44
6.5	Locking strengths	46
6.6	Lock free algorithms	48
7	Evaluation	49
7.1	Detecting the number of processors	49
7.2	Performance counting	51
7.3	Pattern evaluation	58
7.4	Threading scenarios	60
8	Results	69
8.1	Threading on uniprocessor machines	69
8.2	Threaded browser design	70
8.3	Redesign of non-threaded browser	75
A	Threadlab details	81
A.1	Threadlab dependencies	81
B	Test scripts and measured results	83
B.1	Load and paint all images - cooperative	83
B.2	Load and paint all images - threaded serial	84
B.3	Load and paint all images - thread pool	84
B.4	Download multiple large files - cooperative	85
B.5	Download multiple large files - threaded	85
	Index	86

Chapter 1

Introduction

Web browsers do lots of things concurrently. This applies both to presentation of new web pages as well as background tasks such as script execution and image animation. The way this concurrency is implemented differs.

One common way to implement concurrency is multithreading. This is also an efficient way to exploit the power of multiple processor cores. Unfortunately there are many pitfalls and a bad design can give even worse performance and stability than for a single-threaded application. Still the trend of an increasing number of processor cores in normal PC:s has made it a hot topic. This has made the developers of applications that previously used other concurrency mechanisms, such as cooperative multitasking, interested in multithreading.

This thesis examines possible ways that multithreading can be added to a previously single-threaded web browser. Possible solutions are examined and detailed simulations and performance measurements are performed. Design patterns and design principles that can help implementation of multithreading is identified and described. Finally design suggestions and a step by step list for a redesign of a single-threaded browser is presented.

1.1 Processor evolution

After many years of faster and faster single processor machines the industry seems to stand in the middle of a big change. The big race for higher and higher clock speeds ended around 2003 and it is no longer easy to make any quantum leaps upwards. Instead the leading processor manufacturers have started to build processors with multiple processor cores on one single chip. Earlier multi processor machines were, by economical reasons, dedicated mostly for server halls. But nowadays we see desktop machines with two or four processor cores and that is a trend that probably will continue.

1.2 Web browser evolution

Web browsers have been around for many years. With the fast paced computer industry it feels like we have had them if not forever but for a long time. The first ones that appeared, around 1991, were all very simple programs. The most important features were text, hyper links and pictures. There were simple web applications, but all relied on form based transactions based on simple CGI functionality. The content was mostly static, far away from the advanced web applications of today. Still it was a revolution!

The first years were very turbulent. The demands for more interactivity and better ways to design web content lead to more advanced HTML standards, more advanced form functionality, Cascading Style Sheets (CSS), ECMAScript (commonly called JavaScript, from the initial implementation), new image formats, Scalable Vector Graphics (SVG) and layers. Also the server side was improved. The simple CGI scripts evolved to advanced application environments.

On top of the web browsing functionality the browser companies added other features like mail clients, news readers and HTML editors. Features like tabbed browsing, where the user could open a multitude of pages in the browser window, helped making the earlier pretty simple browsers more advanced to develop and maintain. The web browsers had evolved from being simple programs that loaded and painted static web pages, to some of the most complex and advanced programs we have today.

The described feature race heavily increased the demands of concurrency in the browser. The earliest browsers could just parse and paint the document and then forget the data. The load and paint operations could be done as atomic operations and the user did not expect much interactivity or feedback during the process. One big change came when ECMAScript was added. That meant that a script language could affect the document also after it was loaded and displayed. This was first used mostly to pop up dialogs and showing irritating text scrolls, but the technique has grown more and more important and nowadays it is a fundamental part of the more advanced web applications. A steady trend has been that more and more computation is moved from the web server to the web browser. The operations that earlier were made by submit and response, to and from the server, can now be made mainly in the client. The web server role has changed somewhat, from *being* the application, to more and more handling bootstrapping, data communication and data storage.

Under the surface of the web browser a lot has changed during the years. At an early stage it became necessary to implement better internal representations of the displayed document. The resulting document tree has become a central part of every modern browser. Other functions, like CSS, ECMAScript and SVG require internal representations, language parsers and runtime environments.

1.3 Concurrency

It is natural to us humans that things happen simultaneously. It is natural to expect the same from a computer system and computer application.

Definition

In computer science “concurrency” normally means that two or more computations appear to be executed at the same time. Edsger Dijkstra summarized this as “Concurrency occurs when two or more execution flows are able to run simultaneously.”. If the computer system has multiple processors the computations can be executed independent and simultaneously. If the computer has only one processor the operating system can switch back and forth between the computations making it appear as if they execute simultaneously. This work will mainly focus on the first case, with multiple processors.

The word parallelism is also commonly used in relation to the subject. This can easily be mixed up with parallel programming, where a problem often is divided and sent for parallel calculation on multiple computational resources. This is often used for heavy computation on super computers.

Variants

This work will focus on two variants of concurrency; cooperative multitasking and multithreading.

Cooperative multitasking in a program means that a program tries to achieve concurrency by itself, by organizing the program execution. Operations need to be designed not to take too long time to execute and a mechanism that switches between concurrent tasks has to be implemented. If implemented well the result can be good and since it is guaranteed that no two functions run at the exactly same time the execution order can be deterministic and the data can not be corrupted by the concurrency. If one function gets stuck in a loop or on a blocking function call the whole program will freeze until that is solved. The increasing market of multicore processor computers also makes cooperative multitasking looking a bit obsolete, since cooperative programs can run on only one processor and therefore not make use of the full machine power.

A more true parallel way is to use different forms of multithreading. This means that the program execution process is divided to threads that run in parallel. Each thread can run its own part of the program, with the data shared among threads. When running on a single processor core machine it will result in an execution scheme not very far from the cooperative case. The main difference is that the execution is cut up by the operating system that keeps running shares of each thread. When running on a computer with multiple processor cores the threads can run truly parallel.

1.4 Opera Software

Opera Software has since over ten years developed an Internet web browser, known for being fast, portable and economical with computer resources. This application mainly uses a design based on single threaded cooperative multitasking. This design has proved to work well and will probably be kept also in the future, since the browser needs to be portable to many different devices and environments, also those without multithreading support.

To make good use of multiple processor cores within a program you need to use multithreading or some other technique that makes it possible for different parts of the program to run on different processor cores. An application built with cooperative multitasking will only be a single process and only be running on one core. An interesting question is: what could a good combination of those two techniques look like, and it is possible to get the best of the two worlds?

Opera would like to know whether it is reasonable to add various levels of multithreading to their web browser. They want to know more about parts that can benefit from being parallelized and techniques that could be used. It should be possible to use this information for future decision making.

1.5 Document outline

This thesis report will first describe basic web browser functionality (chapter 2). It will then describe the basics of multiprocessing hardware (chapter 3) and the concurrency mechanisms used on this hardware (chapter 4).

To simulate aspects of web browser concurrency a simulation environment called Threadlab was developed. This is described in chapter 5. This chapter also describes how modules of a generalized web browser can be simulated by Threadlab.

This is followed by an overview of design patterns and design methods that could be used to help concurrent programming of web browsers (chapter 6).

Based on chapter 2, 4 and 5, the evaluation of web browser concurrency is described in chapter 7. This chapter also describes various scenarios of concurrent execution.

Finally the experimental results from chapter 7 are discussed in chapter 8.

Chapter 2

Web browser functionality

To discuss web browser functionality there is a need to define the basic operations of a web browser.

This chapter can be jumped over if the reader already has a good view of web browser functionality.

2.1 Retrieving data

A key function of a web browser is to retrieve data to be displayed. The data to be retrieved can be HTML documents, images, CSS style sheet definitions, and other kind of data needed to present a web application.

The data can be retrieved using several different kinds of protocols from different kinds of sources. Most common is the *Hypertext Transfer Protocol* (HTTP), but data could also be retrieved from local file storage, through the File Transfer Protocol (FTP) or other customized protocols.

The data to be loaded is uniquely identified by the use of a *Uniform Resource Locator* (URL). URLs are a subset of the more general Uniform Resource Identifiers (URIs), with information about the retrieval mechanism.

The retrieved data is normally stored in RAM memory. Files that can not be displayed by the browser are often downloaded to a secondary storage. This can also be manually triggered by the user.

2.2 Parsing HTML

The very most common way to write web pages is to use Hypertext Markup Language (HTML). Initially HTML was a rather loose standard that gave lots of freedom to browser developers and web page creators. Later on it showed that the loose definitions caused problems and since much HTML is automatically created from different design tools more strict versions were preferred. Current versions of the HTML standard define a much stricter markup language. To be

able to show also old pages the web browsers need to be backward compatible with the older standard versions.

The earliest web browsers did not allow any or little local interaction with the page contents. Link clicks loaded whole pages and web form handling relied on data being sent to the server which returned the results in form of new generated web pages. This means that early web browser implementations did not have any special needs for a local representation of the parsed HTML document. The necessary operations after parsing were to calculate the document layout and to paint the result. Later on came script languages that could change parts of the document, style sheet definitions that could affect the style of document elements and other mechanisms that made it more important to have a local representation of the parsed document.

Since the HTML and XML standards often have hierarchical properties the internal representation normally is a document tree following the document contents. Script languages and other interactive functionality can then access the document tree to change nodes.

2.3 Decoding of images

When images are referenced from a HTML document they are first downloaded to local memory. The images are normally encoded in image formats such as JPEG, PNG or GIF. Those formats normally need to be decoded by the browser, to get a matrix with color values.

Some image formats allow several image frames to be encapsulated in one image file, allowing various forms of animation. This requires all image frames to be decoded.

Vector based images, such as images defined with the Scalable Vector Graphics (SVG) standard, are often treated as embedded data which is parsed and painted as individual documents.

2.4 Recursion

Most modern web browsers allow documents to define sub documents that will be displayed as part of the whole. The sub documents need to be treated individually. An early example of this is the *frameset* definition that makes it possible to divide the document into frames that all contain different sub documents. Those frames can also contain frameset definitions recursively.

A newer way to include sub documents as part of a document is the use of *IFrames*. This enables documents to specify that an area should be defined by another document. This document will be loaded recursively.

How this recursion is represented internally may differ practically between different web browsers. They could either gather all sub documents as sub trees of the larger document tree, or have references to individual trees.

2.5 Cascading stylesheets

An early HTML concept is that contents and style should be separated. That is the reason why HTML entries relate to logical document entities like headers and lists, not to how they actually should be displayed. When first browsers evolved the content designers wanted more flexibility and tag attributes that changed the look of things evolved. This resulted in hard coded looks of pages that often were hard to manage.

A solution to the lack of flexibility from hard coded visuals was *Cascading Style Sheets* (CSS). This means that documents, in their header part, can reference a file containing style definitions for different parts of the document. This can be things as font selection, font sizes and weight, colors, borders, margins and positioning. The CSS standard also added the possibility to define areas in the document that could be given properties. This is intended to be more flexible than using HTML tables for document layout definitions.

Figure 2.1 illustrates how CSS definitions can control the browser output.

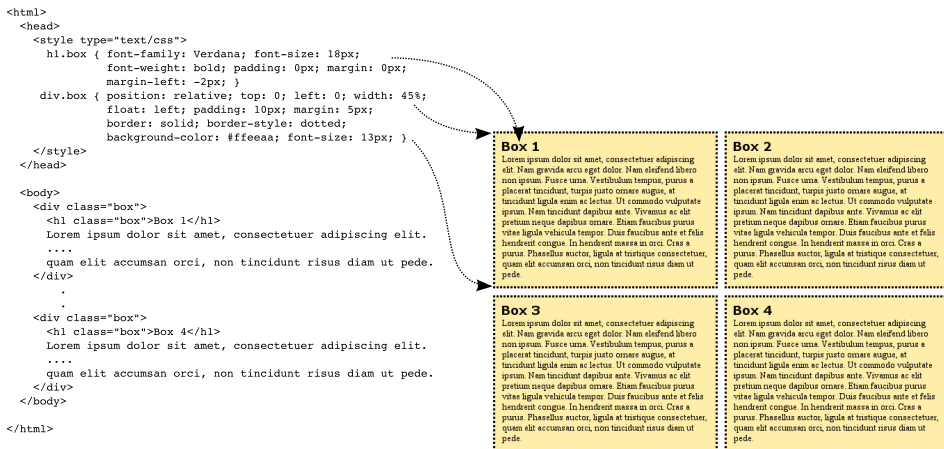


Figure 2.1. CSS style definitions controlling result

2.6 Document layout

When all data is loaded and decoded the resulting document tree has to be painted. Before the document elements can be painted the document layout has to be calculated. Size and position has to be determined for each element. This process is very complex. There are many different layout aspects to take into account and all has to be calculated together. There are also elements that need special care, like text blocks where the text has to be wrapped to fit the available size and for example flow around other elements.

Before the CSS standard arrived document layout was mainly done with HTML tables. Tables were initially supposed for data tabulation and not directly for layout, but the table element tags have size and layout properties that makes it possible to achieve advanced layouts. Table cells can also contain tables, recursively. The more and more popular use of visual WYSIWYG editors caused designers, often without their knowledge, to produce totally unreadable HTML documents with many levels of nested tables. This became a bit better when the CSS layers were introduced, but still many documents use a mixture of several layout techniques. This makes layout very complex.

2.7 Document painting

The actual painting is commonly done by use of the painting primitives given by the graphical environment the client is running under. The client now has to traverse the document tree and paint all the elements at their calculated positions. Text data is then written graphically on the paint canvas and image data is copied from memory.

Just before the actual paint is done the look of elements must be decided. The look of elements can be decided by old special tags such as the font-tag, but more common today is to use CSS definitions. Since it is possible to change the CSS definitions from client side scripts it may be necessary to read from the CSS representation in memory during the paint operations. Another possibility is to have properties for all elements also in the internal document representation. That is a trade off between memory usage and paint speed.

If the painted web page contains animated images those have to be repainted repeatedly.

2.8 Form response

When the user fills in a web form and triggers a submit operation the browser has to encode all the form data and perform a new request. The data can either be encoded into the HTTP request protocol or as part of the URL used to retrieve the form result. Special care has to be taken to handle character encodings correctly.

The form submit normally results in a new web document to be returned as response. This triggers the whole process, as described above, to be started all over.

The entry of more advanced client side scripting techniques has lowered the use of the big submit and response forms somewhat. Scripts can interact directly with the web browser and more fine grained user interaction can be achieved. The changes still require the browser to recalculate the layout and repaint the documents.

2.9 Cookies

As a method to add states to the otherwise stateless nature of HTTP requests the idea of HTTP cookies was realized. The name comes from the *magic cookie* concept in the Unix world. The idea is that the server, in the response header, can add a small portion of data that is stored in the client browser. This data is then sent as part of the HTTP request each time the client makes accesses to that server in the future.

2.10 Client side scripting

The idea of a client side scripting language is not new. For example the early web browser ViolaWWW contained an advanced embedded scripting language that could be used to obtain advanced user interaction [Cailliau and Gillies, 2000]. Later on Sun Microsystems and Netscape Communications Corporation began the work on JavaScript. Later Netscape submitted the JavaScript specification to Ecma International, a standardization organization, for standardization. The work resulted in the standard called ECMAScript, with the standard name ECMA-262 [ECMA, 1999]. This standard is implemented in all modern web browsers.

The ECMAScripts are normally running interpreted in the web browser. This means that the browser at least has to parse the script to a more efficient internal form, set up symbol tables and reserve memory for variables.

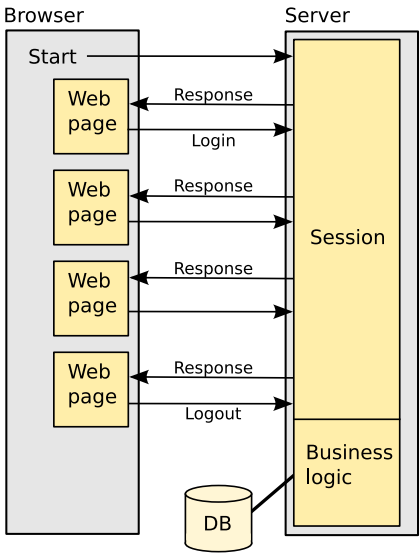
The ECMAScripts would be rather dull if they did not interact with the browser in some ways. This is done by using programming interfaces that give access to the document DOM tree and other features in the browser. Document elements can also reference script functions to be run when the element is accessed in some way.

The use of client side scripting has evolved to a level where the web browser, for advanced applications, no longer statically shows documents, but acts more like a run time environment. Techniques such as AJAX (Asynchronous JavaScript and XML) have evolved, where the client side can exchange information with the server without reloading whole pages. Figure 2.2 shows the difference between a traditional submit and response application and a modern AJAX application.

2.11 Plugins

Plugins were more important in the early days, where they were the only way to extend the web browser functionality. Plugins are packaged programs following a predefined API specification that makes it possible to register them to and run them inside a web browser. The plugin is normally activated by a HTML tag that embeds data that the plugin handles into a document. The plugin will be activated and normally occupy a frame on the document page where it can present the result. This was often used to execute Java code written especially for web documents and for different forms of media players. Nowadays it is more

Traditional web application



AJAX client side application

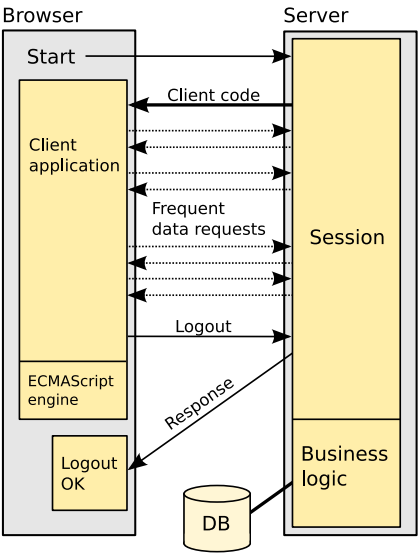


Figure 2.2. Traditional server based application and modern client side application

commonly used for running clients for Adobe Flash media and embedded media players.

Chapter 3

Multiprocessing Hardware

This chapter gives an overview of multiprocessing with a focus on shared-memory multiprocessing (SMP). The purpose of this chapter is to give a basic understanding of hardware mechanisms that may affect choice of different concurrency mechanisms and what effect they may have on performance.

3.1 The processor

Modern processors are complex and different processors may differ significantly. Nonetheless there are some important basic features that all processors have in common and that are important in respect to concurrency.

3.1.1 Execution

The processor executes program instructions that reside in the computer memory hierarchy. The instruction to be executed is pointed out by a register, the *program counter* (PC). After each instruction execution this counter is updated so that it points to the instruction to follow.

3.1.2 Registers

The program counter is one of many values kept track of in the processor. All processors contain fast *registers* that can hold values and addresses during execution. Compilers are constructed to optimize execution by keeping as much of the current calculations as possible in registers instead of the much slower RAM memory.

3.1.3 Flags

Certain aspects of execution result can be interesting to further instructions. For example it can be interesting if a calculation result was negative or zero. Therefore

the processor updates internal values called *flags* accordingly.

3.2 RAM memory

When being executed programs reside in the primary RAM memory. Even if it is much faster than secondary storage, such as hard disk drives, it is relatively slow compared to modern processors. To speed up execution computers have faster cache memories that stores recently addressed data. The relation between primary memory and caching is important when deciding on concurrency.

3.2.1 Memory connection

The processor is normally connected to the RAM memory by a fast speed bus. This bus, together with the speed of the RAM circuits, is often a bottleneck in modern computers. Much research is done on ways to speed up the connection between the CPU and the RAM memory, but it will likely continue be considerably slower than the execution inside the processor. When multiple processors work against a single RAM memory area, as with an SMP machine, they will also compete about access to the memory bus.

Modern computers often use *Direct Memory Access* (DMA) techniques that allow storage units such as hard disk and CD-ROM drives as well as peripheral units such as network and sound hardware to communicate directly with the memory. This frees the processor from the cumbersome work of moving large blocks of data. The con is that all those units will have to compete for access to the memory.

The individual memory circuits can normally only be addressed by one client at a time, reading one data word at a time. To speed up memory access memory multiple circuits are often installed and organized to allow *interleaving* techniques. Data is then stored non-contiguously, spread symmetrically over multiple circuits, exploiting the data access limitations for single circuits.

3.2.2 Process memory

Modern computer systems implements *memory protection*, which means that the system kernel ensures that a process can only affect the real memory space allocated to that process and nothing else. The system normally maps the physical memory allocated by a process into a much bigger virtual memory space that the process can allocate logical memory in. Together with concurrency mechanisms this means that each process will act as if they had exclusive access to the computer.

3.3 Cache memory

An important way to speed up execution is to minimize the accesses to the slow RAM memory. This is done by adding one or more levels of cache memory in

between.

3.3.1 Cache levels

Cache memories are today normally located inside the processor circuits. For speed reasons it is good to have caches as close to the CPU as possible. Often multiple levels of cache memory are used, with the fastest, the *level 1* (L1) cache, closest to the processor, followed by the *level 2* (L2) cache that normally is larger but slower. This can be followed by further levels. The L1 cache is often divided into one part that caches instructions (L1I) and one part that caches data (L1D). This may apply also to the L2 cache, but it is often general, storing both instructions and data.

Initially caches were placed outside the processor. Later it became common with a small CPU internal L1 cache and in high-end machines an external L2 cache. Last years many manufacturers have moved to have both L1 and L2 caches inside the processor circuit, to enhance performance. Even if they reside in the same circuit, the L1 cache is normally faster and closer to the executing parts of the CPU. Some processors have the L1 cache on the same circuit die as the processor core and the L2 cache on another die that also resides inside the processor casing. To speed up memory further the L1 cache can be placed inside the processor core and the L2 cache can be placed on the same die.

Figure 3.1 illustrates how a single processor is connected to the RAM memory through a cache hierarchy.

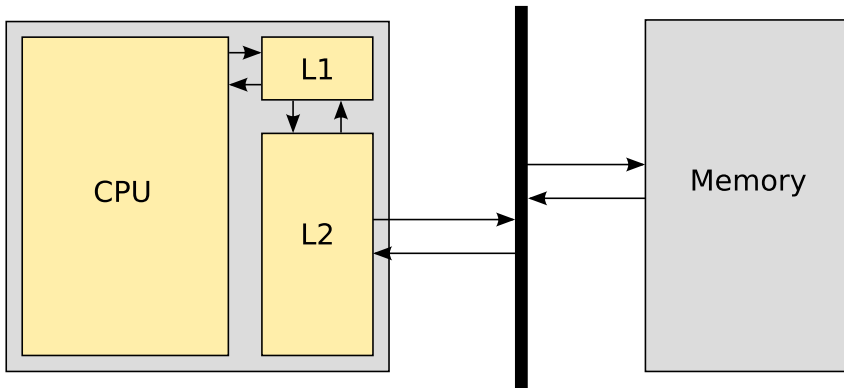


Figure 3.1. Cache level architecture

When the processor needs data from the RAM memory it first checks the L1 cache. The data in the cache is stored together with a *tag*, referencing the memory address and the processor looks for a line with correct tag. Figure 3.2 shows how two entries in the cache points to corresponding elements in real memory. If the data exists in that cache it is immediately returned, otherwise the processor checks the L2 cache. If the data exists there it is returned and written to the L1

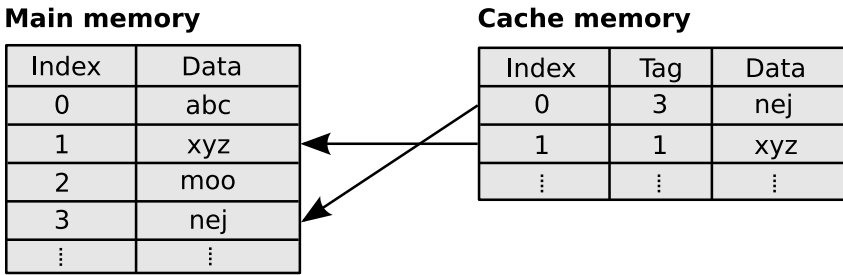


Figure 3.2. Cache memory references

cache. If it did not exist in any cache it has to be read from the slow RAM memory. When data is read from RAM memory it will be stored in cache memories for later retrieval.

3.3.2 Cache misses

A failed attempt to read or write data from or to the cache is called a cache miss. The hardware then needs to access the next cache level or eventually the slower main memory to continue. Cache misses can occur in three cases, when reading an instruction, reading data and writing data. The effect of those are slightly different and listed below.

- Cache misses when *reading an instruction* normally cause the largest delays. That thread of execution will then have to wait while the instruction is read from main memory.
- Cache misses when *reading data* normally cause less delay than when reading instructions. Modern processors can, within some limit, continue to execute instructions not dependent of the missing data while it is being read from main memory.
- Cache misses when *writing data* normally cause least delay. Processors often handles a queue for data to be written and can continue the execution until that queue becomes full.

3.4 Multiprocessing

The previous sections described a single processor environment. This section takes a step further and describes machines built with multiple processors.

3.4.1 Symmetric multiprocessing

There are several types of multi processor environments available today and several other types have existed historically. The main differences are how the pro-

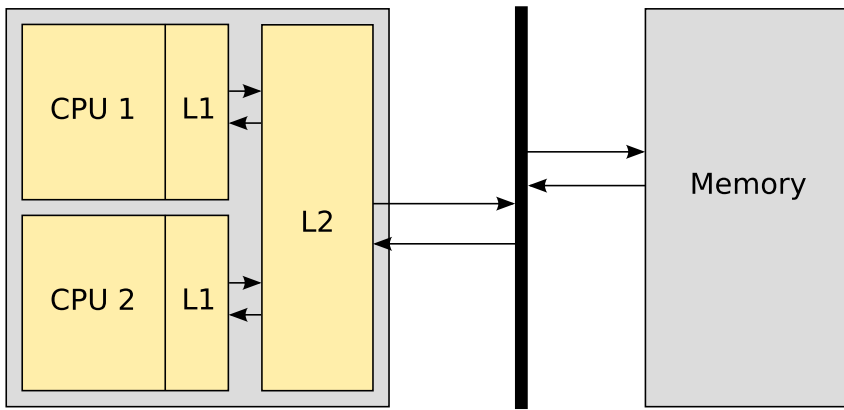


Figure 3.3. SMP cache level architecture

processors are interconnected, how they are connected to the memory and what type of communication is used.

The currently most common multiprocessing model for desktop computers is *Symmetric multiprocessing* (SMP). SMP machines are built with one RAM memory that is symmetrically accessed by multiple processors. Symmetric means that the mechanism and time for access to the memory is generally the same for all processors and for all parts of the memory. Caches can of course speed up the memory accesses.

The individual processors may have processor internal caches as described in section 3.3.1. Since execution of a program can be migrated between the processors by the operating system and since several threads of one program may need access to the same data it is also common with a shared cache. Figure 3.3 shows how an SMP cache hierarchy may look like.

3.4.2 Cache coherence

When multiple processors read or write the same memory line special care has to be taken to guarantee data consistency. If all processor cores have individual cache memories it has to be guaranteed that a memory line that exists in multiple caches are exactly the same.

Problem appears when one processor core writes to a memory line. Then copies of that memory line has to be disqualified in all other caches where it resides. There are several ways to handle this disqualification and lot of research is performed to find better schemes to guarantee cache coherence.

It is important to know that disqualification must be performed if one processor writes to a memory area that other processors have in their caches. This needs to be taken in mind when a concurrency model is decided. Since disqualification results in cache misses it slows down execution and should be minimized.

Chapter 4

Concurrency mechanisms

Running multiple operations concurrently can solve design problems as well as enhance the user experience. But certain measures need to be taken to ensure correct results. It is very important with a basic understanding of problems that can occur. That knowledge can be used to find suiting design, with design principles and patterns that can help to ensure the function in a concurrent environment.

This chapter aims to describe the major topics and pitfalls of this area and how they relate to the area of web browsers.

4.1 The process

A process is a running instance of a program. A program is a list of instructions together with other necessary information about how to execute it. A process is the actual execution of such a program. This section will describe the different mechanisms involved when running a process.

4.1.1 Process components

A process consists not only of its program instructions. The operating system also needs to keep track of all the process registers. When the process is switched out the operating system has to copy the contents of all registers to the internal data structure that represents the process. This data structure is often called *Process Control Block* (PCB). When the process is switched in again those values have to be written back to the processor registers.

Also the memory address space layout, priority and accounting information has to be handled by the OS.

4.1.2 Process execution

When the running process is executed the program counter (PC) always points at the actual position in the program and the registers contain currently actual data.

For program blocks without any conditional branches everything flows on and the processor pipeline will be well filled. If the processor reaches a subroutine call it has to store the current processor state on the stack and change the PC to point to the first subroutine instruction. If also the subroutine makes a subroutine call the same procedure happens again. When the processor has reached the end of a subroutine it has to restore the earlier saved execution state.

For temporary storage the processor uses the stack. The stack is a part of the RAM memory ordered in a LIFO (Last In First Out) order. A *stack pointer* (SP) points at the current stack top.

4.1.3 Process costs

The cost of creating and maintaining a process is relatively high. For every process the operating system needs to initiate the memory layout, the stack, registers, program counter and other necessities. After initialization it needs to keep track of those things as records in memory to be able to switch between the processes.

Traditionally it has been common to use multiple processes to achieve concurrency. Many Unix server daemons have traditionally been built to spawn off a number of worker processes that handle incoming requests. The pros are that it is easy to design and make robust, but it comes with the cost of a big process overhead.

4.1.4 Process scheduling

As described in section 3.1 the processor continuously executes process instructions. The operating system will switch between processes, trying to keep the instruction flow up, even if some process may be temporarily stopped while waiting for some resource.

There are many important aspects of process scheduling that have to be considered when designing the operating system. It is important that the processor is kept busy even with many processes that keep jumping between active and waiting for resources. The system has to feel responsive and it has to be impossible to lock up the operating system from a process.

If the system contains multiple processors the operating system maps the running processes on those. It is then preferable that the same process will be scheduled to the same processor as much as possible to prevent cache misses that occur when it is scheduled to another processor. But it may still occur that processes migrate between processors if one becomes very loaded. Some operating systems has shown tendencies of bouncing, where processes, due to high load or I/O wait from other processes, can bounce back and forth between processors every task switch. This gives very bad performance.

4.2 Concurrency levels

Concurrency means that several computations are executed quasi simultaneously. This applies both to the operating system level, where different processes are executed simultaneously and on a process level where a program may want several functions to be executed simultaneously.

4.2.1 System level concurrency

On the operating system level several program processes are executed concurrently. This is handled by the system kernel. The kernel can either execute as a single thread, cooperative, or multithreaded.

The operating system handles concurrent execution of all programs. Earlier there were operating systems performing this with cooperative multitasking but today all serious systems use preemptive multitasking. With cooperative multitasking the system relied on each process voluntarily giving up the processor when its time slice was used. With preemptive multitasking the system kernel switches between the processes to be executed, lets them execute as long as they can be allowed and then switches to the next process.

4.2.2 Program level concurrency

On the program level it is also often necessary to have concurrency. It is natural for a program to do several things at the same time. When an advanced computation or other kind of operation is performed it may be necessary to interact with outside resources as well as giving response to the user about the progression.

Traditionally this was achieved by the programmers, using either several processes that communicated with each other or different kinds of cooperative multitasking. Nowadays this has in many cases been replaced by multithreading.

For more information on cooperative multitasking, see section 4.3 below. For more information on multithreading, see section 4.4 below.

4.3 Cooperative multitasking

Cooperative multitasking is a method nowadays mainly used within applications. The main idea is that different tasks within a process voluntarily leaves processing time to other tasks.

Earlier there were operating systems completely relying on cooperative multitasking, but since time sharing between processes depended heavily on running application behavior they have all moved away from cooperative multitasking.

On the application level cooperative multitasking still keeps on living. There are several reasons for this. First it is a simple technique. You achieve concurrent behavior within one process, which is simple to manage. You execute only one function at a time, which means that operations on data are more secure than they are in a multithreaded program. When altering a variable you can be sure that

there are not any other functions running, at the very same time, also altering the data.

For programs that need to be highly portable there can be problems with more advanced concurrency techniques, like multithreading, since they can depend on platform specific libraries and technology. Cooperative multitasking is normally more independent from other software.

4.3.1 Message handlers

One common way to achieve cooperative multitasking is to establish a central message handler to which tasks can send messages. The message handler then schedules the corresponding functions for execution. The functions must not execute too long, or the program would appear to be locked. If a task needs to do more calculations than it can possibly do within reasonable time it will have to register itself for rerun to the message handler and return. Then, when other tasks have been executed, the message handler will restart the postponed task which now can continue the calculations.

With a message handler different tasks can continuously register themselves to be run later and in this way appear as independent tasks.

For object oriented programs it is common that classes that should be able to receive messages from the message handler implement a `message()` method that the message handler can call. When an object registers a message it calls the method handler with a pointer to the receiving object and a reference to the message. The message reference can for example be a constant number or, for more advanced systems, objects of a `Message` class, that for example can carry states.

The evaluations of different concurrency techniques that are presented in chapter 7 include tests of cooperative multitasking using a central message handler.

4.3.2 Select loops

Another common way to implement cooperative multitasking is to use a *select loop*. The idea is to use a blocking function call, often called `select`, which takes as parameters a collection of file descriptors. The `select` call monitors the file descriptors until one of them becomes ready for some kind of I/O. In classic terminal based program interfaces interaction was commonly built around key commands. The key inputs could then be monitored by the `select` loop, which returns when a key is pressed making it possible to read and perform the corresponding action. The same can be used for graphical environments where the paint protocol as well as the different input devices such as keyboard and mouse can be monitored. The `select` function call can also be given a timeout value that is called if no I/O operation happens for a certain interval. That makes it possible to perform timed background tasks.

Cooperative multitasking based on a `select` loop is often used in relation to the *Reactor* pattern [Schmidt, 1995]. Objects do then register themselves as responsible for different resources and the `select` loop dispatches events to the registered

objects when they occur.

4.4 Multithreading

The most popular and common concurrency technique today is multithreading. The execution is then split among several running *threads*, which are executed concurrently by the operating system. If the computer executing the program has only one processor core the OS will task switch back and forth between the different threads making it appear as if they were executed in parallel. With an SMP machine, with multiple processor cores, the threads can be executed truly parallel.

4.4.1 System implementation

Implementation of multithreading can be done in various ways and have differed and do differ between different hardware architectures and operating systems. A necessary feature is that the threading mechanisms should contain synchronization mechanisms. Those mechanisms are normally implemented using memory operations that are guaranteed to be atomic. Modern processors often contain special functionality to assure this. Special embedded systems and other specialized devices may contain processors without this support. Then the multithreading must be built on top of operating system functionality guaranteeing the same synchronization mechanisms.

4.4.2 Differences between processes and threads

Every process has its own process control block with information about the process. It has its own memory area and data is not shared between processes without special request from the programmer. Creating a process means setting up all those things. Unix server programs traditionally used a number of cooperating processes and often spawned a new one to handle each incoming service request. The process overhead could then become big.

Threads are different execution flows within a program. They all use the same shared environment. The global program data and many parts of the process control block is shared between the threads. The only things that have to be unique for each thread is things as the program counter, the registers and the stack for that thread.

The thread concurrency means that you never can be sure which thread currently accesses a variable or other kind of shared resource that can be accessed by several threads. This can cause random unwanted behavior unless it is explicitly handled. The solution is different forms of synchronization mechanisms that make sure that access to a resource is only done by one thread at a time.

In general you gain more true concurrency with multithreading, but at the cost of more complex programming and overhead from synchronization mechanisms.

4.4.3 Thread overhead

A common mistake is to believe that the power of multithreading comes without any cost. And even if many understand that the management of threads does cost something it is common to believe that the multiple processors of an SMP system compensate for this cost.

It is true that multiple processors gives more power, but there are pitfalls that novice programmers can fall into that makes the programs slower, regardless of the number of processors.

When running on only one processor core the operating system needs to switch between threads, just as with multiple processes. With multiple processors the system will still most likely switch in and out threads, but can do things more truly parallel. The *base overhead* for the thread management, at the system layer, is fairly small. For a program with a couple or a dozen of threads the overhead will most likely be just a fraction. But for programs with more threads, maybe a hundred, all competing to be switched in and out, it will be a factor to consider.

A more important aspect than the system thread overhead is the cache hit rate for a certain program. As described in section 3.3, cache memories and locality are extremely important for the execution speed of a program. This also applies for a multithreaded program and especially when running on multiple processors it is important that the programmer knows how to minimize cache misses.

4.4.4 Affinity

If multiple threads mainly work on same shared data they could be locked to run on the same processor or processors. This is often called *processor affinity*. This normally lowers the benefits from multithreading on multiple processors, but in some cases the threads are synchronized to be running serially at a degree where it could still be positive. Pinpointing a thread to a processor this way can keep up the cache hit rate if the thread otherwise tends to be bouncing between processors.

Since few of the main operations in web browsers benefit from massively threaded operations *on the same data* this is not very applicable when running on well working systems. One problem that affinity solves is that poorly designed process schedulers may bounce threads between processors, often when they interchangeably wait for I/O, causing high cache miss rates. The Linux O(1) scheduler, implemented from the Linux 2.5 branch and on, contains mechanisms trying to prevent bouncing.

4.4.5 Pthreads

Initially different hardware and software vendors implemented their own versions of threading mechanisms. They normally differed much from each other, which made it hard to develop portable threaded programs. To address this problem the work on a standardized thread API was initiated. The work was finally

standardized 1995, as IEEE POSIX 1003.1c [IEEE, 1995]. The standardized library are commonly called POSIX threads or Pthreads.

The Pthreads API consists of a number of C language functions and types. Pthreads consists of methods that handle the lifespan of threads, different kinds of synchronization functionality, such as mutex locks and condition variables.

4.5 Race conditions

When two threads, running in parallel, try to read and write a shared resource it can not be predetermined which thread will first begin to operate on the resource. If the access to the shared resource is not handled explicitly by the programmer the result can be very problematic. This is called a *race condition*, a term that originates from the idea of two software signals racing each other to first influence the output. The solution to this problem is some sort of synchronization making it impossible for the two threads to manipulate the shared data at the same time.

A classic example is two threads both working on a number value variable. If both threads change the value by first reading the variable and then doing the change which is written back, then it is possible that the operations happen in an order that causes the result value to end up wrong. Consider the following mathematical operations, running in two threads.

Thread 1

$i = i + 1$

Thread 2

$i = i + 1$

This could result in the following internal sequence of instructions.

Thread 1

Load i in register

Increment register by 1

Store register to i

Thread 2

Load i in register

Increment register by 1

Store register to i

The registers used in the above instruction example are thread local. The result is that the operation done by the first thread is overwritten by the second thread. The second thread works on the value read before the first thread has finished and written back the result.

Another example is two threads working on the same linked list structure. If one thread wants to add an element at the beginning of the list, just after the head element, and the other thread wants to delete the first element of the list, then they may end up totally damaging the list. The major reason of the problem is that the operation of linking in a new element requires several pointer operations. First the old tail must be linked to the new element and then the new element must be linked to, from the head element.

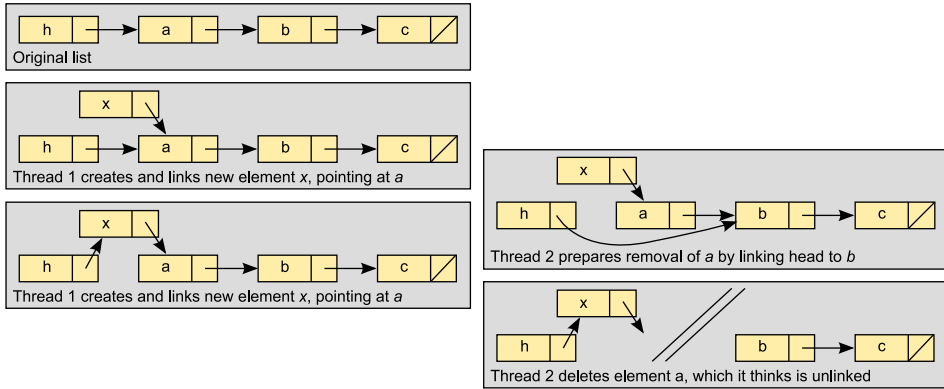


Figure 4.1. Race condition on list structure

This could result in the following internal sequence of instructions, where thread 1 wants to add a new element (here called *x*) at the beginning of the list and thread 2 wants to delete the current first element (here called *a*).

Thread 1

Create new element *x*

Point *x*->*next* to *a*
(*head*->*next*)

Point *head*->*next* to *x*

Thread 2

Get address of *a*
(*head*->*next*)

Point *head*->*next* to *b*
(*a*->*next*)

Delete element *a*

At the end of the scenario the head points to the new element *x*, created by thread 1, which will erroneously point to the deleted element *a*. The rest of the list is lost.

Figure 4.1 illustrates this scenario.

Since element linking is also used as a foundation of tree formed data structures the example with damaged list structures also applies to tree structures. This means that for example the document tree in the web browser would be very vulnerable without any solution to the problem above.

4.6 Synchronization

The solution to the race condition problem described above is different types of synchronization mechanisms. When multiple threads may access a shared resource it is important to ensure that access is synchronized so that only one thread

have access to the resource at a time. Methods such as semaphores, mutex locks and condition variables are commonly used and will be described here.

4.6.1 Critical sections

When a program performs read or write operations on shared resources it is often possible to identify regions that make out the core of those operations. Those sections of the code are called *critical sections*. Those sections have to be protected so that other threads do not access the same shared resources at the same time. It is considered a good practice to try to keep the critical sections as short and few as possible.

Critical sections are heavily related to different synchronization mechanisms, since those will be used to ensure that only one thread has access to the shared resource at one time.

During program development it is necessary to identify the critical sections and apply suitable synchronization. When shared data is used it should be locked before entering the critical section and unlocked after leaving it. When the data is locked in other threads, working on the same data, critical sections will have to wait until the data is unlocked by the first thread. This means that the critical sections should be kept as short as possible. In many cases a long critical section can be rearranged to a shorter one by moving code not directly related to the shared resource.

On single processor machines it may be possible to prevent race conditions by disabling interrupts before entering the critical section and enabling them after. That will block any other thread from being activated before the interrupts are restored. This method is simple, but not interesting for a modern web browser, since it does not work on multi processor machines.

4.6.2 Semaphores

A semaphore is a classic way of restricting access to a shared resource by the use of a protected variable. The method was invented by Edsger Dijkstra and uses a variable that is initialized to the number of identical resources to be protected. Whenever a thread needs to use one of the limited resources the semaphore is checked to assure that there are free ones. If there is a free resource the semaphore variable will be atomic tested and decreased by one. When the thread is finished the semaphore variable is increased again. A special case is semaphores where the number of resources is one. The semaphore will then act like a mutex lock (see section 4.6.3).

To implement semaphores the system needs some way to assure that the test and decrease operation is not interrupted when the semaphore variable is found to be not null. In the same way the increase operation may not be interrupted. Many processor architectures have atomic test-and-set instructions to help with this.

4.6.3 Mutex locks

The mutex lock is a popular special case of semaphores where a single resource is locked. The name comes from *mutual exclusion* and implies that one access to a resource excludes access from others.

Mutex locking has become one of the fundamental synchronization mechanisms and is a central part of the Pthreads library, which is described in section 4.4.5, below.

4.6.4 Condition variables

In a multithreaded program it is often necessary to have threads wait for certain conditions to happen. If some threads are responsible for decoding incoming compressed image data they will have to wait until such data appears. Instead of spinning like mad in a while loop consuming lots of processor power just while waiting it can voluntarily release the CPU and enter a sleep mode waiting for data. The event of having new data is signaled by a condition variable that the thread will enter as argument to the wait operation.

Condition variables often work together with a related mutex variable. In those cases the conditional wait function will ensure locking of the mutex before it returns back to the now woken up thread. In the example above it can for example be the mutex guarding the list structure storing image objects.

4.6.5 Deadlock

A very dangerous condition for a multithreaded program is if two threads end up waiting for each other to release a resource. This can for example happen if both threads have locked one unique resource and then both try to lock the resource locked by the other thread. It can also happen if a thread has locked a resource and by program error failed to release the lock when finished. Then the same thread or other threads, wanting to enter the same critical section, may wait forever. The problem with a lock not being unlocked can also be caused by such things as C++ exceptions.

There are design patterns that can help programmers to ensure that locks are properly unlocked. The Scoped locking pattern, described in section 6.1, defines a way to ensure release of locks by the use of C++ scopes and release of local variables.

Chapter 5

Threadlab evaluation system

To simulate and evaluate the ideas, methods and patterns described in earlier chapters an evaluation system called *Threadlab* was developed. The system can be executed on various hardware platforms and executes evaluation scripts that describe a certain test. The evaluation scripts are executed to *bootstrap* simulations, not executed during the actual simulation.

During execution the system measures some basic performance aspects, such as execution times, lock waiting and number of processor cycles. After execution Threadlab produces a report of collected values.

Threadlab only collects basic information that is easy and portable to collect. To measure more detailed information it is recommended to use Perfmon2 and Pfmmon (see section 7.2.3).

The results from this system are later used as base for the design and choices presented in chapter 8.

5.1 System design

This section describes aspects of the Threadlab system design.

5.1.1 Software platform

The Threadlab evaluation system is designed and programmed in C++ under Linux. The main focus has been to test different aspects of thread functionality and patterns on various hardware platforms. Since it has not been a target to test various operating systems, portability outside the Unix world has not been a main issue.

Threadlab relies on several open source libraries for command operations. Those libraries are all widely used. A complete list of library dependencies can be found in section A.1.

The low level multithreading system used is Pthreads (see section 4.4.5).

5.1.2 Simulations

The Threadlab simulations operate on known data structures such as lists, trees and memory areas. The system defines a number of operation commands that are common for those structures and common for web browser operation. The initialization of threads, data structures to be used and definitions of what operations to perform is defined by Python based initialization scripts. This is described in section 5.3 and more detailed in appendix A. An initialization script is executed to *setup* a certain simulation, not for the actual simulation execution. Python is in this case a powerful way to describe simulations. It is used to initialize the threads, data structures and commands.

When the python script is executed C++ objects corresponding to the created Python objects will be created and registered to Threadlab. When setup properly the simulation is executed. Execution means that the threads created earlier begin to execute commands. When all threads are done the simulation is over and measured data is presented.

During execution the system spawns threads that then perform wanted operations on the described data structures. After execution the collected result is presented.

The performance data monitored during execution is collected by the responsible threads and commands. Local storage is preferred in this case since a central storage of data needs some sort of synchronization and could become a bottleneck that damages the results.

After execution the collected data is analyzed and presented. No analyzing is done during the test execution, to ensure that it does not affect the result.

Figure 5.1 below shows how a simulation is initialized. The Threadlab application creates a Script object that first reads and parses a Python script. That Python script creates a number of threads that are automatically registered to the script object that manages a list of threads. After that the Python script creates data structures such as memory areas, DOM trees and images. In the next step the Python script creates commands that are told to work on the previously created data structures. The commands are added to the previously created threads.

5.2 Supported concurrency variants

By the use of threads and commands, as described above, many combinations of possible designs can be tested. This section describes the main concurrency concepts that can be used in simulations.

5.2.1 Multithreading

Variants of any number of threads executing commands can be evaluated by creation of TLThread objects. Each such object will correspond to a real Pthreads thread. Normally an initialization script will create a number of command objects that will be added to the TLThread object. Each TLThread object manages

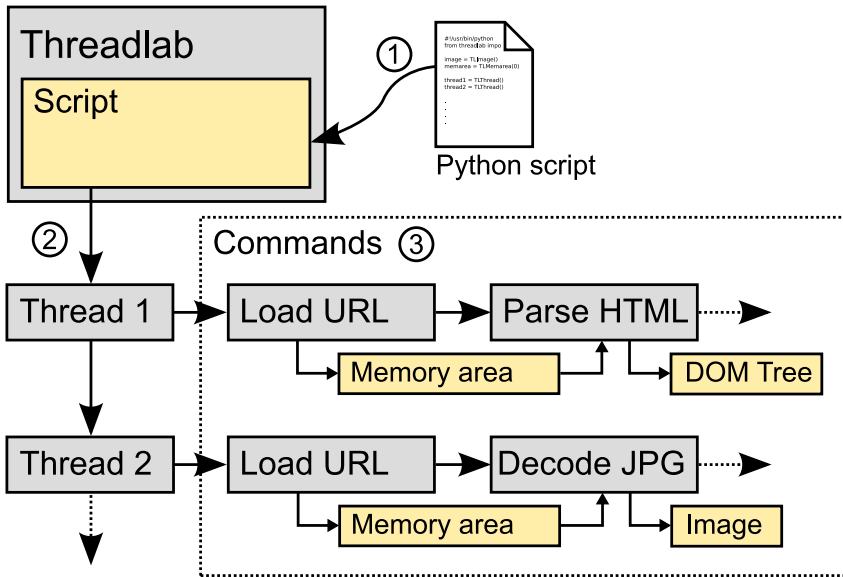


Figure 5.1. Threadlab initialization

a list of added commands, as shown in figure 5.1. During execution the thread will sequentially execute all its commands.

When commands are executed from a thread they will perform proper mutex locking of data structures they work on. This will prevent race conditions to occur if multiple threads work on the same data structures and will also make it possible to measure locking overhead.

The commands that are executed this way will do *all* their work before they finish. This means that the commands in a thread will all be executed and finished sequentially.

5.2.2 Cooperative multitasking

It is possible to evaluate cooperative multitasking with Threadlab. This is allowed by the special `TLCooperative` command, which gives the possibility to test cooperative execution of a number of commands. The script can add other commands to a `TLCooperative` object, just as to `TLThread` objects. The `TLCooperative` object is then added to a thread, just as normal commands. When the thread, during execution, reaches the `TLCooperative` object this object will manage a message handler and execute all the registered commands concurrently, until the message handler is empty.

When commands are executed inside a `TLCooperative` command they will *not* perform any mutex locking of data structures they operate on. The reason is that cooperative multitasking execution normally does not suffer from race

conditions. This gives more true values when measuring the simulation.

Commands that are executed cooperatively will try to behave nicely. This means that commands that have time consuming calculations to do will divide it into parts. The command will execute one part each time it is called and then it will register itself to the message handler which later will call the command again to allow it to calculate another part. Commands that wait for I/O will also register itself and return in case it can not proceed at the moment. Since the commands execute concurrently and can be in a non finished state other commands, that depend on a non finished one, have to check if they can proceed or not. If possible they will start working on what is currently available. This is for example used when painting images with `PyCmdPaintImage`. That command checks what data is currently decoded by the JPEG decoder command `PyCmdDecodeJPG`, which in turn will decode what data is currently loaded by for example the `TLCmdLoadURL` command.

5.2.3 Thread pool

It is also possible to test the thread pool pattern (see section 6.4). Instead of creating threads the script can create a thread pool with a predetermined number of worker threads. Commands that are added to this thread pool will be concurrently executed in a round robin manner.

The thread pool manages a list of commands to execute and the worker threads will grab commands and execute them. Mutex locking is of course used on the command list.

Commands executed by a worker thread will perform mutex locking of data structures they work on, just as for the normal multithreaded concurrency variant. This will prevent race conditions to occur if multiple threads work on the same data structures and will also make it possible to measure locking overhead.

The commands will, just as with the cooperative case, perform parts of their full operation if the operation is large or for example waits for I/O. The command will then be readded to the end of the command list.

If the command list is empty when a worker thread is free, then that thread will enter a sleep state where it sleeps in wait of new commands. When new commands are added to the command list sleeping worker threads will be woken up. This is implemented using condition variables (see section 4.6.4).

5.3 Evaluation scripts

To initialize Threadlab Python based script files are used. Those script files tell the system which data structure to use, what threads to use and what operations different threads should perform and on what data structures.

The Python based script format is easy but powerful. The Threadlab program provides a Python module, `threadlab`, that is made available to scripts. This module is used to read detected information about the hardware and system environment and to set up the tests.

```

import threadlab

print 'Threadlab script: %s' % threadlab.script_file
print 'Number of processors: %d' % threadlab.processor_count

IMG1 = 'images/test_picture_1.jpg'
IMG2 = 'images/test_picture_2.jpg'

image1 = threadlab.PyImage()
image2 = threadlab.PyImage()
memarea = threadlab.PyMemarea( 0 )

window = threadlab.PyWindow( 10, 10, 800, 600 );

thread1 = threadlab.PyThread()
thread2 = threadlab.PyThread()

thread1.addCommand( threadlab.PyCmdDecodeJPG( IMG1, image1 ) )
thread1.addCommand( threadlab.PyCmdPaintImage( window, image1, 10, 10 ) )

thread2.addCommand( threadlab.PyCmdMemLoadFile( IMG2, memarea ) )
thread2.addCommand( threadlab.PyCmdDecodeJPG( memarea, image2 ) )
thread2.addCommand( threadlab.PyCmdPaintImage( image2, window, 100, 100 ) )

thread1.addCommand( threadlab.PyCmdSleep( 5 ) )

```

Figure 5.2. Threadlab script that loads and paints two images

As example the script in figure 5.2 first prints the name of the running script and the number of processors on the current machine. It then creates two image objects and a memory area. Those will be used by commands during execution. It also creates a graphical window to paint on. It creates two threads that both will decode JPEG images. The first thread decodes the image directly from the file and the second by first loading it to a memory area and then decode the image from there. The operations are setup by creation of command objects that through their parameters are told how to work. The commands are added to the threads. During execution the data will be loaded, decoded and painted.

Detailed information about Threadlab and the script environment is available in appendix A.

5.4 Web browser simulation

This section describes browser modules for a generalized web browser and how they can be simulated within Threadlab.

The generalized browser model will consist of a number of modules and structures responsible for well identified and isolated tasks. The following sections will describe those modules and structures.

Each section will contain a description of the module, a description of data

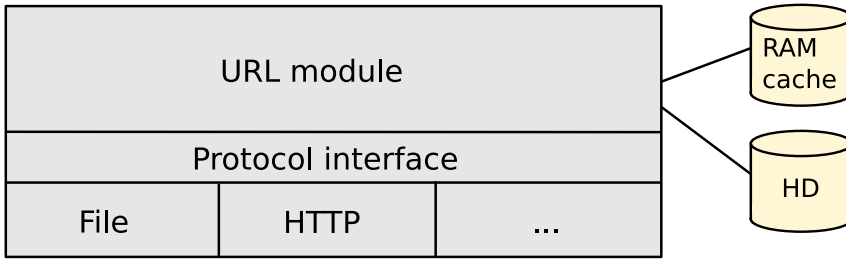


Figure 5.3. Network module design

structures managed by the module and, if applicable, notes about data integrity in concurrent systems.

5.4.1 Network module

Many central operations involve retrieval of remote resources. The resources are normally referenced by a URL string containing information about protocol and location. Our generalized browser model will have a central network module that retrieves and returns data referenced by a URL.

The network module can implement an arbitrary number of protocols, but should at least implement local file system access (by the protocol string “file”) and the HTTP protocol (by the protocol string “http”).

The network module will normally download resources to RAM memory. It may also be told to download resources to secondary disk storage. It can also be told to default to RAM but fall back to disk storage if the resource is larger than a predetermined limit.

The network module will also implement a local file cache that stores retrieved files in a round robin manner to lower the number of remote retrievals of previously retrieved data. The local file cache can be configured to have a default time out value that causes elements to be reloaded if accessed again after a certain time. It is also possible that certain elements have individual time out values.

The network module can be probed by other modules about download progression. Other modules may in this way start to work on the data before it is completely downloaded. This can for example be used by image decoders that can start decompressing image data before it is completely downloaded. The image decoder can in turn be asked about progression to make it possible to start paint operations before it is totally finished.

Figure 5.3 pictures the network module organization.

- ☞ The network module will be simulated in Threadlab by the use of the `TLCmdLoadURL` command. This command can load a URL to a memory area or to a file.

5.4.2 Parsers

Naturally the most common document format that needs to be parsed in a web browser is HTML. This is slightly harder than it may sound. There are several versions of the standard that have to be supported seamlessly. Worse is that many documents do not follow the standards correctly or follow an intermix of standards. Still the browser must be forgiving on errors and present good enough efforts also for badly formed documents.

There are several other data formats that have to be parsed. An important format, related to HTML in some aspects, is XML. This can and should generally be made more strict than HTML parsing. Another parser has to be able to parse CSS style sheet data.

One of the more complex parsers is the ECMAScript parser. This parser has to be able to parse scripts and together with other functions create the internal representation that will be used in the script runtime environment. This can for example be a syntax tree representing the input script or in more advanced cases a byte code representation. In the first case the script execution could be done directly in the syntax tree. That is generally simpler to prepare and simpler to execute, but it will generally be slower than running byte code in a virtual machine. The second case is more complex to produce, but the result may be faster and have smaller memory footprint.

Generally the parsers in our generalized browser model read input data and construct a representation in memory for use by other modules. The parsing can for example be done by a recursive descent parser or a table driven parser.

In the generalized browser model the parsers will take data as parameter and generate the corresponding parsed data representation as output.

- ☞ Parsing is simulated in Threadlab by the use of the `TLCmdParseHTML` and `TLCmdParseXML` commands. Those commands can parse HTML and XML data available in a memory area or pointed to by a file path. The result will be stored in a `TLXMLTree`.

5.4.3 Document module

A web page that is loaded is done so by being downloaded by the network module, parsed by a parser and displayed by the layout module. The module responsible for coordinating all this is the document module. The document module will get requests to load a document by, for example, the user interface module. A document object will be created and the document module will request the page from the network module. When the page is loaded a HTML parser will be used to create a document tree. The document tree can then be referenced from the document object.

When the document is parsed the document module will traverse the document tree to identify other data to be loaded. This can be images, cascading style sheets, ECMAScripts, other web pages and other data that can be referenced from a web page. Requests are then sent to the network module that will download the data. When the data is downloaded the document module will have to look

at the data to determine if it has to be prepared in any ways. Images will for example have to be decoded by the image module and HTML or CSS data will have to be parsed.

- ☞ The Document module will be simulated by multiple commands and data structures that together define the document module behavior. Most notable is the `TLCmdLoadURL`, `TLCmdParseHTML` and `TLCmdPaintAllImages` commands.

5.4.4 Document tree

A central part of the document module and central to the browser is the hierarchical document tree that represent a loaded and parsed web page. The document tree represents the hierarchical structure of the source file as well as the structure of the resulting rendered web page. The tree also contains node attributes and necessary meta information.

The document tree is the result output from an HTML parser (see section 5.4.2 above). It will be stored in memory as a tree of node objects.

Figure 5.4 shows a simple HTML document example and figure 5.5 shows the corresponding document tree.

- ☞ The Document tree is represented in Threadlab by the `TLXMLTree` data structure.

5.4.5 Layout module

Before the document can be painted the document tree has to be traversed and the layout has to be calculated. The layout calculation is generally a complex operation. The relative sizes of all elements have to be taken in count and correct positioning should be calculated. This is hard because of the almost infinite number of layout combinations and the fact that web page designers often use different web browsers during their design process. The designers may use strangely selected combinations of layout methods and the users expect the layout results to be similar regardless of the client.

The result from the layout calculations could be stored in some sort of parallel data structure, but in our general browser model it will be stored together with other meta data in the document tree nodes.

The browser model tree nodes will have references to a dictionary construct where nodes can have name- and value-pairs assigned.

As with other information in the document tree the layout data has to be protected if written and read from different threads.

5.4.6 Graphic canvas

Each web page retrieved and displayed by the generalized browser will be painted on a graphical canvas. The painting will be done by use of graphical primitives.

```
<html>
  <head>
    <style type="text/css">
      h1.box { font-family: Verdana; font-size: 18px; ... }
    </style>
  </head>

  <body>
    <h1 class="box">Bill the pirate</h1>
      <p>
        <ul>
          <li>Alfa</li>
          <li>Bravo</li>
          <ul>
            <li>Charlie</li>
            <li>Delta</li>
          </ul>
          <li>Echo</li>
        </ul>
      </p>
    
    <p>
      Lorem ipsum dolor...
    </p>
    <p>
      Nam tincidunt dapibus ante...
    </p>
  </body>
</html>
```

Figure 5.4. HTML document for simple web page

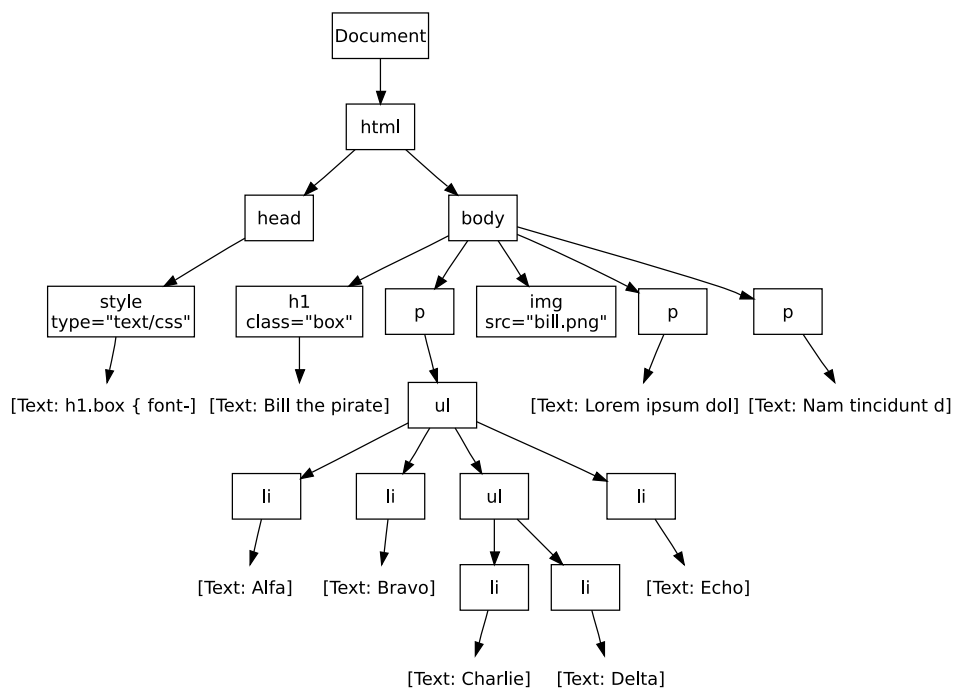


Figure 5.5. Document tree for a simple web page

The actual implementation may vary. Some systems provide native paint operations and a native canvas concept. On other systems it may be necessary to paint in a memory area using own paint primitives and then copy that memory area to the display memory. For our generalized browser model we just focus on the existence of a graphical canvas and not on underlying implementation.

- ☞ The graphic canvas is represented in Threadlab by the `TLWindow` data structure. When such an object is created a corresponding graphical window will be created and prepared for paint operations.

5.4.7 Image decoding

Before images can be painted on a graphical canvas they have to be translated from the image format used by the web page to a raw array of color values. This is done by image decoders.

The image decoder module will present an interface containing mainly a `decode()` function. This function takes a `File` object as parameter and returns an `Image` object. The `decode` function will prepare the decoding by looking at the given `File` object and from that trying to decide necessary information about the image type. It will then return. Actual decoding will be started concurrently and can be monitored through the `Image` object.

It is important to realize that the `File` object can be so newly created that it does not yet contain enough data to do the decode initialization or decoding. The decoder module will then have to wait for data concurrently and continue when it can.

Actual image decoding is done by decoder modules for different supported image formats. The correct module is selected during initialization of the decoding.

When the decoding is prepared a memory area for the resulting image will be allocated. This memory area requires special care if the decoding is done in another thread than the use of the decoded data.

It is assumed that the decoding will proceed concurrently. The loading of a web page often results in multiple image decodings occurring in parallel. The image decoders work as much as they can with the data they have available and the layout module (see section 5.4.5) will paint as much image data that is available.

- ☞ Image decoding is in Threadlab represented by the `TLCmdDecodeJPG` command. This command reads compressed picture data from a file or a memory area and stores the decoded result in a destination memory area.

5.4.8 ECMAScript engine

The ECMAScript engine is responsible for running the ECMAScripts used on web pages. The engine will use data result from an ECMAScript parser (see section 5.4.2 above). ECMAScript engines can be written at various levels of complexity. Some existing engines execute by traversing directly into the parse tree output

from the ECMAScript parser. This is rather simple and requires less work for the parser step, but the memory footprint is larger and the execution speed is normally lower. More advanced engines use a byte code representation of the scripts. The byte code contains instruction codes and arguments to be executed in the script engine. The byte code is normally produced during or after the parsing step. This is normally more complex to implement, but can give smaller footprint and faster execution speeds than parse tree execution. The exact execution is not important for the generalized browser model. The model engine will take input from the parser step and execute it with corresponding side effects.

During the execution the script engine will need to allocate memory for variables, objects and other things that require memory space. This is normally done dynamically at run time and unused memory is freed by a garbage collector mechanism. The script engine needs to keep track of literals such as named variables by handling a string pool. The engine also needs to store and retrieve environment scopes during execution. The scopes contain all the data necessary for execution of a function and have to be stored before a new function call and retrieved after that function returns, to be able to continue execution.

- ☞ Execution of ECMAScripts can in some cases be simulated by execution of multiple corresponding Threadlab commands. This will not simulate the script engine overhead, but may simulate interaction between ECMAScripts and other operations. As an example it is easy to simulate a web browser loading and showing a web page with the script engine running in the same thread or in a thread of its own.

5.4.9 Browser user interface

The browser user interface will contain the usual interface setup used for modern web browsers. The web browser will be able to display multiple web pages simultaneously, for example by use of multiple windows or windows with tabs that allow changing of loaded web pages inside a single window.

The user interface is the main connection between the user and the underlying modules. The user interface manages windows where the graphical canvas is displayed. When the user, through the interface, enters a URL or selects a bookmark, then the interface will send messages to the underlying modules to make a page load to happen.

- ☞ The web browser UI can in some ways be simulated by loading images representing the interface from file, which then can be decoded and painted to simulate the looks of the interface. The user interaction can not be simulated.

Chapter 6

Patterns and methods

Developing with concurrency mechanisms, such as Pthreads, can be complex. Data is shared among the threads and it is not possible to be sure about data integrity and the order of data access without using special mechanisms. Mechanisms such as mutex locks, monitors and condition variables require special care to not cause more trouble than they solve. This can be especially hard if the development is done by multiple programmers. For a big programming project, such as a web browser, different programmers often have different responsibilities and work on different parts of the project. This can cause one programmer to miss synchronization due to lack of knowledge about parts his/her areas interact with. It would be of good help if there were design principles and patterns that could aid the programmers to handle concurrency.

To make concurrency programming easier to maintain and overview, several techniques and design patterns have been defined. The general goal is to add abstraction layers that hide and encapsulate the fine grained synchronization mechanisms. Some patterns help by making locking and unlocking more automatic and different locking techniques seamlessly interchangeable. Other patterns address the need for local storage for threads and the way functions are called from threads.

The following sections will describe some important concurrency patterns and discuss the applicability to web browsers. When applicable the patterns will be illustrated by C++ examples.

6.1 Scoped locking

Scoped locking is a method to implement locking for critical sections by the use of language scopes. This addresses the need of releasing locks after a critical section. Forgetting to release a lock can result in undefined behavior and deadlocks. The release of a lock can be extra hard to make correct if made in a section with error control and where exceptions can be thrown.

The *scoped locking* pattern addresses this. The pattern was initially defined

by Douglas C. Schmidt [Schmidt, 1999].

6.1.1 Implementation

As a base for scoped locking a *Guard* class is defined. The guard object takes a real lock, such as a Pthreads mutex, as a constructor parameter. The guard object locks the lock object in the guard constructor and releases the lock in the guard destructor. At the beginning of each critical section a stack based local *Guard* object is created. When the program leaves the scope all local objects will be destructed. The guard destructor will automatically be executed and the lock released. That means that the unlocking is guaranteed. The interface is also somewhat cleaner, since the user does not have to bother about thread specific function calls.

The following code shows a possible *Guard* class definition.

```
class Guard
{
public:
    Guard ( pthread_mutex_t * lock )
        : lock_ (lock)
        {
            result = pthread_mutex_lock( lock_ );
        }

    ~Guard (void)
        {
            if ( result == 0 )
                pthread_mutex_unlock( lock_ );
        }

private:
    pthread_mutex_t * lock_;
    int result;
};
```

The following example shows the possible use of the *Guard* class.

```
class User
{
public:
    // ...

    void resetTime()
    {
        Guard guard( lock_ );
        long result = time( NULL );
        if ( result < 0 )
            return;
    }
};
```

```

        time = result;
    }

    // ...

private:
    pthread_mutex_t * lock_;
    long time;
    // ...
};

```

6.1.2 Applicability for web browsers

The scoped locking pattern is highly interesting for a web browser. It will help the programmers to keep track of lock use and help with lock release. It also encourages small critical sections, not larger than the current scope.

The scoped locking pattern is also useful in connection to the strategized locking pattern (see section 6.2 below).

The Guard class can also be extended for debug purposes, for example with log capabilities that can be compiled in by setting an `#ifdef` variable in the source code. If this is done it is important that the logging interface does not require synchronization. If the log interface requires locking the behavior may be hard to predict and it may cause deadlocks.

- ☞ The runtime overhead of the scoped locking pattern is evaluated and described in section 7.3.1.

6.2 Strategized locking

The Scoped locking pattern, described above, uses locally defined lock objects as parameters to the Guard class. The Guard class is hard coded to take the low level lock type as argument. This may be undesirable, especially in programs developed for a wide range of hardware and software platforms where synchronization mechanisms may differ. Some sort of abstraction of the actual lock type may be desirable. The *Strategized locking* pattern does help with this.

Strategized locking is a way to implement different types of locks transparently. All the lock types implement a common interface. This can be implemented either by polymorphism or parameterized during creation.

Use of polymorphism can be useful when the wanted types are not known at compile time. Parameterization during creation is suitable when the wanted types are known at compile time.

An interesting possible concrete lock class is a *Null lock*, that can be used when the program is compiled for systems with no concurrency. The Null lock returns 0 for all operations and if it is implemented as inline code, then good compilers may optimize it away completely.

The strategized locking pattern was initially defined by Douglas C. Schmidt [Schmidt, 1999] (same publication as scoped locking).

6.2.1 Implementation

An abstract class `Lockable` is first defined, with virtual `acquire` and `release` methods.

```
class Lockable
{
public:
    // Acquire the lock.
    virtual int acquire (void) = 0;
    // Release the lock.
    virtual int release (void) = 0;
    // ...
};
```

Subclasses can then override the virtual methods in `Lockable` to implement concrete locking, for example by use of *threads*.

```
class Thread_Mutex_Lockable : public Lockable
{
public:
    // Acquire the lock.
    virtual int acquire(void) {
        result = pthread_mutex_lock( lock_ );
        return result;
    }

    // Release the lock.
    virtual int release(void) {
        if ( result == 0 )
            return pthread_mutex_unlock( lock_ );
        else
            return 1;
    }

private:
    // Concrete lock type.
    pthread_mutex_t * lock_;
    int result;
};
```

The Bridge pattern [Gamma et al., 2000] is used to define a non polymorphic interface that holds a reference to the polymorphic `Lockable`.

```
class Lock
{
public:
    // Constructor stores a reference to the base class.
    Lock(Lockable &l) : lock_(l) {};

    // Acquire the lock by forwarding to the
```

```

// polymorphic acquire() method.
int acquire(void) { lock_.acquire(); }

// Release the lock by forwarding to the
// polymorphic release() method.
int release(void) { lock_.release(); }

private:
    // Maintain a reference to the polymorphic lock.
    Lockable &lock_;
};

```

6.2.2 Applicability for web browsers

A browser that should be possible to compile on many platforms could benefit much from having the actual lock types abstracted. Support for a new platform can be as easy as implementing a new concrete subclass to the `Lockable` class.

Strategized locking can also help the programmers that do not have to bother about low level locking techniques anymore.

By the use of Null lock class the locking can be turned off in a better way than using `#ifdef`-constructions around all places where locking is used.

- ☞ The runtime overhead of the strategized locking pattern is evaluated and described in section 7.3.2.

6.3 Thread specific storage

Single threaded programs can by several reasons rely on global data. For example many Unix functions set a global `errno` variable when something results in an error. This variable can be globally accessed by all threads. This means that it can cause bad race conditions. By use of macros or other language tools it is possible to transparently change this in a way that makes the global data thread specific. Access to the variable is changed to a function call that looks up and returns a specific copy of the variable for the running thread.

The thread specific storage may also increase performance since thread specific data may be used without the need of mutex locking. If a thread executes a longer sequence of function calls that operate on globally accessible data it often has to lock and unlock the data mutex multiple times. This will also be avoided by thread specific storage.

The active object pattern was initially defined by Douglas C. Schmidt [Schmidt, 1997].

6.3.1 Implementation

Thread specific storage is normally implemented using a `Collection` class whose objects keep track of all thread specific variables for a certain thread. Those variables are accessed by a certain key. To hide the keys a proxy class is used. This

proxy class knows the key and gets and sets the thread specific variables by calling the Collection class.

The use of thread specific storage can be hidden by use of macros. The following example, taken from the original pattern definition, shows how the `errno` variable can be replaced by a macro if running multithreaded. The macro runs the `_errno()` function that returns a thread-specific version of `errno`. The result is that no user code has to be changed to implement thread specific storage.

```
// A thread-specific errno definition (typically defined in <sys/errno.h>).
#ifdef defined (_REENTRANT)
// The _errno() function returns the thread-specific value of errno.
#define errno (*_errno())
#else
// Non-MT behavior is unchanged.
extern int errno;
#endif /* REENTRANT */
```

More detailed implementation details are available in [Schmidt, 1997].

6.3.2 Applicability for web browsers

The thread specific storage pattern may be interesting when multithreading is added to a non threaded browser. If the non threaded browser uses globally defined variables those can be secured by thread specific versions.

The pattern should only be used when the different threads work on individually different data. It should not be used when multiple threads work on a larger global data set.

6.4 Thread pool

The thread pool pattern describes how a number of worker threads are created to perform tasks. A queue of pending tasks is implemented and the worker threads pick tasks from this queue. Access to the queue, from the worker threads, has to be serialized to prevent disruption. When a worker thread is finished with a task it picks a new one from the queue. If the queue is currently empty the thread will enter a sleep state. When new tasks are added to the queue the sleeping threads are woken up.

6.4.1 Implementation

The following `Threadpool` class spawns a number of worker threads that are kept in a vector of `TPThread` objects.

```
class Threadpool
{
public:
    Threadpool( int num_threads );
```



```

~ThreadPool();

void addCommand( Command * command );

void run();
void join();

// Function to be called from pthreads spawn
static void * thread_function( void * obj );

std::vector<Command *> command_list;

pthread_mutex_t command_list_mutex;
pthread_cond_t command_available;

private:
    void spawnThreads();

    int num_threads;
    std::vector<TPThread *> thread_list;
};

```

The `addCommand()` function adds a new command to be executed to be handled by the worker threads. It also initiates a mutex for the command list and a condition variable that is used by sleeping threads.

The `thread_function()` is called during thread creation and it contains a while-loop that repeatedly takes `Command` objects from the queue and executes them. If no commands are available the loop will perform a `pthread_cond_wait()` on the `command_available` variable with the `command_list_mutex`. In this example the command list, the mutex and the condition variable is public, since the thread function has to operate by pointer references. This can of course be hidden.

Each thread is represented by a `TPThread` object that keeps a reference to the running thread and the threadpool.

```

class TPThread
{
public:
    TPThread( ThreadPool * threadpool );
    ~TPThread();

    pthread_t * thread;
    bool running;
    ThreadPool * threadpool;
};

```

6.4.2 Applicability for web browsers

The thread pool pattern is highly interesting for web browsers. If the browser already has concurrency based on a message handler it may be possible to reuse the current code with little modification.

This is an easy way to exploit the power of multiple processor cores.

- ☞ The thread pool pattern can be simulated by Threadlab and is used in several of the scenarios evaluated in section 7.

6.5 Locking strengths

Normally locking is made every time a thread wants to enter a critical section. This is often called *pessimistic locking*. The advent of database and web systems where multiple clients asynchronously read and write to the common data has given birth to other ideas. Often different concurrent operations against a database or a web server do not affect each other and no errors occur. This leads to ideas called *optimistic locking*.

6.5.1 Pessimistic locking

Pessimistic locking is the most common synchronization policy and what most people think about when they think about locking in relation to multithreading. It means that all related shared data is locked before entering a critical section. The main idea is the pessimistic assumption that the operation can always go wrong when performed without synchronization and should therefore always be locked.

The execution cost from pessimistic locking is as high as locking overhead can be. Synchronization is used for every access to shared resources. On the other hand the development is relatively simple and straightforward. With use of the concurrency patterns described above a safe locking mechanism can be implemented without any special functionality above the synchronization.

6.5.2 Optimistic locking

When pessimistic locking is used on very frequently used central data structures it can result in a high overhead. For some structures this can be relatively unnecessary since different threads may often access different parts of the structure. One solution could be micro locks, on every data element in the structure, but that would result in a big memory overhead. The idea with optimistic locking is to not do any locking at all and then detect and take care of conflicts as they appear. Optimistic locking is also known as transaction-based programming, non-blocking synchronization and speculative parallelization.

6.5.3 Common uses of optimistic locking

Optimistic locking is commonly used in database implementations. In a database application data queries can be made from multiple client applications concurrently. Locking can traditionally be made on whole tables or on single cells. Locks on whole tables can cause time consuming delays, when other clients need to wait for the lock to be released. Locks on single cells can heavily increase the lock overhead in form of memory and CPU. Even worse than the lock overhead is the sequentialization effect of pessimistic locking. This effect leads to long idle times on multiprocessor systems. To solve this it has become common with optimistic locking. In the database case it means that operations are made without any locking. Instead data integrity is controlled when needed, for example when data is about to be stored. If some operations have been made on related data, when the integrity problem was detected, then those operations will be reversed. With database terminology this is called a *rollback*.

Also for web applications optimistic locking has become popular. The stateless nature of web applications makes normal locking hard. What would happen if one user started to use a system that locks a certain resource and then just closes the web browser? Other users would most likely have to wait forever or at least for a timeout. To solve this the data integrity is checked and solved when the store operations are made.

A third example of optimistic locking is many version management systems, like Subversion and CVS. It would not work to lock a source code tree for others when one developer works on the code. Instead code access is allowed to all interested. They can get the code and work on it. When they are done they send it back to the server which controls that the changes made by the user do not conflict with changes made by others.

6.5.4 Optimistic locking strategies

To assure data integrity after each operation that involves shared data it is necessary to check that the shared resource has not been changed since it was copied and modified by a client. The overhead from this check should be as small as possible.

One commonly used way to check integrity of data is time stamps.

6.5.5 Applicability for web browsers

At first glance optimistic locking may seem suiting for a web browser. The applicability depends heavily on how the browser is threaded.

The structure that most of all could benefit from optimistic locking is the document tree (see section 5.4.4). If multithreading is introduced in a way so that the document tree is accessed by multiple threads then optimistic locking could be used to avoid lots of mutex waiting. But as specified in chapter 8, such a threading should be prevented.

Another reason why optimistic locking is unsuitable for web browsers is that conflicts that occur can be very complex to handle. A transaction in a database can be unrolled and an error message can be sent back to the user who then can redo the operation. A web browser does not have the possibility to rollback and abort. For example if one thread is executing the ECMAScript engine and another is layouting and painting the web page, then both may need to change values in the document tree. To correctly handle a conflict when the script engine changes the tree would require the script engine to keep track of multiple execution steps backwards. This would increase the memory load of the engine and slow down the execution. A conflict during update from the layout and painting thread would require the layout and paint operation to be cancelled and redone, since the layout was changed by the script thread. This is easier to handle than conflicts in the script engine, but the user experience can be hurt.

Use of optimistic locking for web browsers has been investigated by David Vest, in his thesis work "*A version aware data structure designed for incremental rendering of structured documents*" [Vest, 2007].

6.6 Lock free algorithms

One way to solve concurrency problems that has been a research topic last decades is lock free algorithms. The idea is to construct algorithms, for common operations on commonly used data structures, that do not need any locking. Normally this require more processor cycles per operation, to ensure data integrity, but can give better performance due to no waiting for locks.

6.6.1 Applicability for web browsers

Lock free algorithms may be interesting for data structures such as the document tree. Unfortunately the algorithms are often complex and the time and memory overhead from lock free algorithms may be hard to overview.

Chapter 7

Evaluation

The previous chapters present various concurrency techniques, design principles and patterns that could be used. This chapter presents evaluations made on different principles and patterns, as well as evaluation theory needed to understand the evaluations.

The evaluation theory background is not only good help understanding the tests presented here, but also further on during future development. Methods to measure process cycles and cache misses can be used during future threading decisions.

7.1 Detecting the number of processors

To be able to dynamically adjust concurrency for a program, at runtime, it might be interesting to detect the number of processors available. Most operating systems have function calls that can be used to acquire this information. This section shows a number of ways for various systems.

7.1.1 Portable under Unix

On many Unix systems the `/proc/cpuinfo` can be read to see information about available processor cores. The file contents are outputted by the system kernel and will list numerous information about each available processor. For each processor there will normally be a block beginning with the keyword `processor` and a number.

7.1.2 Posix `sysconf`

POSIX.1 defines the function `sysconf ()` that can be used to get configuration information at runtime. The function call is portable, but the set of information that can be retrieved and the keys differ. The following C++ code retrieves the number of processors for some operating systems, among them Linux.

```

#include <unistd.h>
#include <iostream>

using namespace std;

int main()
{
    long procs = 1;

#ifdef _SC_CRAY_NCPU
    procs = sysconf( _SC_CRAY_NCPU );
#elif defined( _SC_NPROC_CONF )
    procs = sysconf( _SC_NPROC_CONF );
#elif defined( _SC_NPROCESSORS_CONF )
    procs = sysconf( _SC_NPROCESSORS_CONF );
#endif

    cout << "Number of processors: " << procs << endl;
    return 0;
}

```

7.1.3 Linux affinity

On Linux systems the affinity settings, which can be used to explicitly lock threads to processors (see section 4.4.4) can be used to also read information about processors. The C++ code below counts and prints the number of processors.

```

#include <stdio.h>
#include <unistd.h>
#include <iostream>

using namespace std;

int main()
{
    cpu_set_t mask;
    CPU_ZERO( &mask );

    unsigned int len = sizeof( mask );
    sched_getaffinity( 0, len, &mask );

    int procs = 0;
    for ( int i = 0; i < 16; i++ )
        if ( CPU_ISSET( i, &mask ) )
            ++procs;

    cout << "Number of processors: " << procs << endl;
    return 0;
}

```

7.1.4 Windows processors

Windows systems contain several methods to retrieve the number of processors. One way is to read the `NUMBER_OF_PROCESSORS` environment variable by calling `System.Environment.GetEnvironmentVariable()`.

A possibly more reliable way is to call `GetSystemInfo()`. That function call fills in a `SYSTEM_INFO` struct that has the member variable `dwNumberOfProcessors`. That value does not distinguish between hyperthreading and real processor cores. To be more sure it is possible to use the `GetLogicalProcessorInformation()` function.

Just as Linux Windows has an affinity API that can be used to read affinity information and determine number of processors a process runs on.

7.2 Performance counting

All modern processors keep, in various amounts, count of events and values that can be used to evaluate the performance of a program. The information is often stored in processor registers that can be accessed by user programs.

The number of values counted and the number of available registers has increased during the years. The first Intel Pentium processors as well as early AMD, Alpha and Ultra Sparc processors had 2 registers. This increased in later processors and the Pentium 4 processor had 18. Last years the trend of increasing number of registers have changed to an increasing number of measurable values that are mapped to a fix number of registers. A program can select what values the processor should count.

The number of features and the interfaces used to access different kinds of performance values differs significantly between processors. Not only do different manufacturers implement different sets of information and different APIs, they can also differ between different processors from the same manufacturer and they are often badly documented. To make the life easier for programmers there are several libraries that help abstracting access to performance information. One such library is `Perfmon2`, which is described in section 7.2.3 below. Another solution is `PAPI` (Performance Application Programming Interface) [PAPI] which abstracts performance counting giving a unified interface and unified event names.

It is important to understand that reading a performance counter in many cases can affect the values being read. The programming interfaces commonly require instructions being executed to start and stop counting of an event. Those instructions may as well being counted in some implementations.

To prevent a large task switch overhead when retrieving the values, they can be read by an *inline* function call. The value is normally read by using inline assembler in the source code, that calls an assembler instruction reading the register.

7.2.1 Measuring process cycles

One of the earliest kind of performance counting that was widely implemented is count of process cycles. This is often used to measure the number of cycles used for certain operations. Some special care and understanding is needed to get a correct picture of the results.

The actual usage value of process cycle counting has become slightly harder to estimate last years. The Hyper-Threading technology, introduced by Intel in their Xeon and Pentium 4 processors, means that the operating system believes the processor has two cores, even if there is only one. When the system schedules threads to both virtual processors the real one pipelines the instructions so that cycle count can be misleading. At the moment of writing the Hyper-Threading technology seems to be abandoned by Intel, so cycle counting may again be useful.

The example below reads the cycle count for an Intel Pentium 3 or Pentium 4 processor as well as modern compatible AMD processors.

```
static inline u_int rdtsc32()
{
    u_int lo;
    asm volatile("rdtsc" : "=a" (lo));
    return lo;
}
```

7.2.2 Measuring cache misses

Most modern processors are able to measure cache misses. Measurement of cache misses is an important part of evaluation of different concurrency designs. Every cache miss means that the system has to access much slower RAM memory or even very much slower disk systems. The cache hit rate should be regarded as a main input to future decisions.

The abstraction layers presented below can be used to easily measure fine grained details about cache misses. It is possible to measure many aspects of hit rates for different cache levels.

7.2.3 Perfmon2

The many differences in number of performance counters between processors and the API differences has resulted in the *Perfmon2* project [Perfmon2]. The goal is a Linux kernel module that abstracts the API differences upwards, presenting a generic interface.

The interface is designed to be generic, flexible and extensible. A goal is to move away from the earlier fragmentation, where many analyzing tools used their own low level interfaces.

The results from performance counting is mapped to the `/proc` file system by the kernel module. If the running processor can not provide a certain counter a best effort can in some cases be estimated by use of other counter values.

To help programming, the library *libpfm* is provided. It communicates with the lower layers and makes it possible to control and monitor performance counting. The *libpfm* interface and the lower levels of Perfmon2 are described in [Eranian, 2005].

Highest in the Perfmon2 suite is a performance analyzing tool, *Pfmon*. This tool can be used as a wrapper around programs that should be monitored. *Pfmon* is executed with flags telling what counters to measure, the program to be measured and arguments to that program. *Pfmon* sets up the wanted counters, executes the given program and presents the collected results. This is an easy way to use performance counting, but many times a more fine grained measuring is needed. This can be done by using *libpfm* directly in the program to be evaluated.

Perfmon2 supports all the commonly used modern processors and the developers continuously add support for new models.

- ☞ Perfmon2 is used within the Threadlab simulation tool to gather runtime data. Performance counting is started and stopped just before and after the simulation execution.

Perfmon2 C++ API

The Perfmon2 suite makes it easy to make fine grained performance counting on Linux systems. Threadlab contains an object oriented abstraction to the Perfmon2 API that makes it very easy to add performance counting. The counting can act on the whole application or on the individual application threads. It is possible to measure multiple threads in parallel. Perfmon2 will store counters for each thread during task switches. The counting can be applied just around the operations that should be monitored.

It is encouraged that this is used during the future development work, to make sure that the application execution really behaves like supposed.

Figure 7.1 shows a simple example of use of the `PFMonitor` class defined in Threadlab.

7.2.4 Cachegrind

The Valgrind project contains many highly useful tools. One such tool is Cachegrind. This tool is a cache profiler and it simulates the different caches in the CPU. It keeps track of memory accesses, cache hits and cache misses. The simulation of the memory hierarchy makes the program run about 20 to 100 times slower than normal.

Cachegrind can measure cache performance for functions, modules or whole programs. The simulation may not be as accurate as CPU performance counters, but may be easier to use.

Picture 7.2 shows a typical Cachegrind report for execution of a whole program.

```

#include "pfmonitor.h"

int main(int argc, char *argv[])
{
    // Initiate the Perfmon2 library
    PFMonitor::setOptions( NO_DEBUG, NO_VERBOSE );
    PFMonitor::initLib();

    // Each counting uses a PFMonitor object
    PFMonitor * pfmonitor = new PFMonitor();

    // Set which event to count
    pfmonitor->setEvent( "L1D_REPL" );
    // Bind the monitor to this process
    pfmonitor->bind( getpid() );
    // Start counting
    pfmonitor->start();

    // ... something to measure

    // Stop counting
    pfmonitor->stop();
    // Print counting results
    pfmonitor->report();

    delete pfmonitor;
    return 0;
}

```

Figure 7.1. Perfmon2 C++ API example

```

==12630== I   refs:           8,728,783,528
==12630== I1  misses:           1,297,490
==12630== L2i misses:           19,435
==12630== I1  miss rate:           0.01%
==12630== L2i miss rate:          0.00%
==12630==
==12630== D   refs:           6,414,843,399 (4,730,566,224 rd + 1,684,277,175 wr)
==12630== D1  misses:           11,712,120 (   5,856,020 rd +   5,856,100 wr)
==12630== L2d misses:           4,988,998 (   1,786,247 rd +   3,202,751 wr)
==12630== D1  miss rate:           0.1% (   0.1% +   0.3% )
==12630== L2d miss rate:           0.0% (   0.0% +   0.1% )
==12630==
==12630== L2  refs:           13,009,610 (   7,153,510 rd +   5,856,100 wr)
==12630== L2  misses:           5,008,433 (   1,805,682 rd +   3,202,751 wr)
==12630== L2  miss rate:           0.0% (   0.0% +   0.1% )

```

Figure 7.2. Typical Cachegrind report

7.2.5 Performance ratios

Performance counting, for example by the use of Perfmon2, can measure many different events. Many of those events are only interesting in relation to others. For example the number of retired (finished) L1 cache misses is only interested in relation to the number of executed operations overall.

Here follows a description of especially interesting ratios between events. The events described here follows the event naming used for Intel Core 2 Duo processors, but the ratio descriptions are valid also for other processors with different naming schemes.

The first parts of the names, before the colon sign, names the event. The second parts, after the colon sign, describe a subgroup for the event. That can for example be counting of an event for the current core or any core. Descriptions of the events used in the performance ratios below are shown in table 7.3.

Clocks per instruction (CPI)

CPU_CLK_UNHALTED : CORE_P/INST_RETIRED : ANY_P

CPI shows the average number of clock cycles needed per retired (finished) instruction. A high value indicates that instructions need more cycles to execute. On this processor the value can get as low as 0.25 cycles per instruction. A high value may for example appear when instructions and micro-ops can not be provided fast enough to fill the Reservation Station within the CPU.

Branch misprediction per retired micro-operation

BR_INST_RETIRED_MISPRED/UOPS_RETIRED : ANY

A high value indicates bad branch predictability.

Branch misprediction performance impact

RESOURCE_STALLS : BR_MISS_CLEAR/CPU_CLK_UNHALTED : CORE_P

This value indicates the performance impact coming from branch mispredictions. In many cases an improved predictability can greatly improve application performance.

If the code before the mispredicted branch has a high CPI value, for example due to cache misses, then a large part of the execution can not be parallelized. Reduction of the CPI of that code may reduce the impact from the misprediction.

Bus not ready ratio

2 · BUS_BNR_DRV : ALL_AGENTS/CPU_CLK_UNHALTED : BUS

This ratio indicates a high number of Bus Not Ready signals in relation to number of clock cycles for the memory bus. A high value shows that the bus is very loaded. The latency of the memory sub-system will probably impact on performance.

Event	Description
INST_RETIRED:ANY_P	The number of instructions that retire (finish) execution, any core.
UOPS_RETIRED:ANY	The number of micro-ops retired, any core.
CPU_CLK_UNHALTED:CORE_P	The number of core cycles while the core is not in a halt state.
BR_INST_RETIRED_MISPRED	The number of branch instructions retired that were mispredicted.
RESOURCE_STALLS:BR_MISS_CLEAR	The number of cycles after a branch misprediction is detected at execution until the branch and all older micro-ops retire.
BUS_BNR_DRV:ALL_AGENTS	The number of Bus Not Ready (BNR) signals that the processor asserts on the bus to suspend additional bus requests by other bus agents.
BUS_TRANS_ANY:ALL_AGENTS	Counts all bus transactions.
ITLB_MISS_RETIRED	The number of retired instructions that missed the ITLB when they were fetched.
DTLB_MISSES:ANY	The number of Data Table Lookaside Buffer (DTLB) misses.
LOAD_BLOCK:L1D	Indicates that loads are blocked due to some reason.
L1I_MISSES	Counts all instruction fetches that miss the Instruction Fetch Unit (IFU) or produce memory requests.
L1D_CACHE_LD:MESI	Counts the number of data reads from cacheable memory.
L1D_REPL	Counts the number of lines brought into the L1 data cache.
L2_LINES_IN:SELF	Counts the number of cache lines allocated in the L2 cache.
LAST_LEVEL_CACHE_REFERENCES	Count each request originating from the core to reference a cache line in the last level cache.

Figure 7.3. Important performance events

Bus utilization

$2 \cdot BUS_TRANS_ANY : ALL_AGENTS / CPU_CLK_UNHALTED : BUS$

This ratio shows number of bus transactions in relation to number of clock cycles. A highly loaded bus indicates heavy traffic between processor and memory. The latency of the memory sub-system will probably impact on performance. Computation intensive applications can be improved by better data and code locality.

ITLB miss rate (spread out execution)

$ITLB_MISS_RETIRED / INST_RETIRED : ANY_P$

A high value shows that the executed program is spread out over too many pages and therefore causes a high number of instruction table misses. This often affects the pipelining badly.

DTLB miss rate (too many data page accesses)

$DTLB_MISSES : ANY / INST_RETIRED : ANY_P$

A high value shows that the application accesses too many different data pages too fast, which causes a high number of data table misses.

Loads blocked by L1 data cache rate

$LOAD_BLOCK : L1D / CPU_CLK_UNHALTED : CORE_P$

A high value shows that load operations are blocked by L1D cache. This normally happens with too many concurrent L1D cache misses.

L1 instruction cache miss rate

$L1I_MISSES / INST_RETIRED : ANY_P$

A high value shows a high number of L1 instruction cache misses in relation to number of retired instructions. An improved code locality may reduce the number of L1I cache misses and improve performance.

L1 data load rate

$L1D_CACHE_LD : MESI / CPU_CLK_UNHALTED : CORE_P$

A high value shows that the execution may be limited by memory read operations. One read operation can be handled by a core each clock cycle.

L1 data cache miss rate

$L1D_REPL / INST_RETIRED : ANY_P$

A high value shows that the code often misses the L1D cache and that the data has to be read from the slower L2 cache.

L1 data miss performance impact

$8 \cdot L1D_REPL / CPU_CLK_UNHALTED : CORE_P$

This value shows the performance impact coming from L1D cache misses given that the value has to read from L2 cache instead.

L2 cache miss rate

$L2_LINES_IN : SELF / INST_RETIRED : ANY_P$

A high value shows that many L2 cache misses occur. This may happen because the data set used by the current workload is larger than the L2 cache size. Reading data from RAM memory has a high impact on performance.

Last level cache load rate

$LAST_LEVEL_CACHE_REFERENCES / INST_RETIRED : ANY_P$

A high value shows that many load operations read from the last level cache, missing the first level.

Last level cache miss rate (per instruction)

$LAST_LEVEL_CACHE_MISSES / INST_RETIRED : ANY_P$

A high value shows that many load operations miss the last level cache. This may have a huge impact on performance, since the value then has to be read from the much slower RAM memory.

7.3 Pattern evaluation

This section describes evaluation of some of the more important design patterns described in section 6.

The CPU used for evaluation of scoped and strategized locking is an Intel Core 2 Duo processor with 2 cores running at 3001 MHz.

7.3.1 Scoped locking

The scoped locking pattern has been evaluated with focus on speed overhead. The size overhead is rarely a problem, since the number of locks is moderate and the overhead for a small Guard object is small.

The speed overhead is measured using a program that spawns a number of threads that concurrently lock, change and unlock a data structure. A fixed number of operations is used, in this case 100 000 000. The program is executed multiple times and an average time is calculated. The evaluated execution time is the *real* time, not the processor time. The measurements are executed for 1 to 10 concurrent threads.

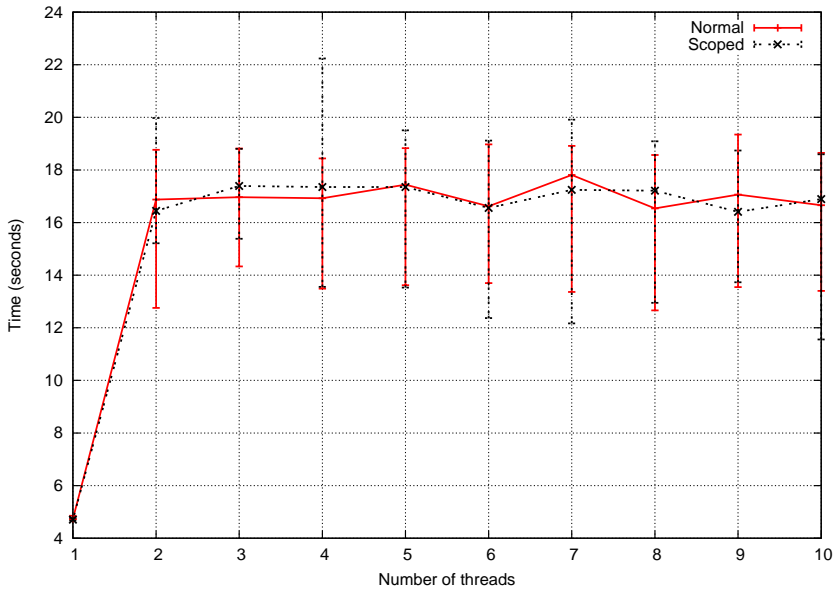


Figure 7.4. Time for 100 000 000 operations on data structure with scoped locking

The resulting speed comparison between low level Pthreads locking and scoped locking can be seen in figure 7.4. As seen in this figure there is no distinguishable time overhead between low level Pthreads locking and use of scoped locking.

7.3.2 Strategized locking

The strategized locking pattern has, as the scoped locking pattern, been evaluated with focus on speed overhead.

The speed overhead is measured using the same program that was used for testing scoped locking. The measurement procedure was identical. The version of strategized locking used in this case is the one that is parameterized during runtime.

The resulting speed comparison between low level Pthreads locking and strategized locking can be seen in figure 7.5. In contrast to the scoped locking evaluation there is a distinguishable time overhead when using strategized locking. The overhead is likely coming from runtime parameterization and polymorphism. The overhead of about 3-5 seconds is still small when taken in count that the number of lock operations is large.

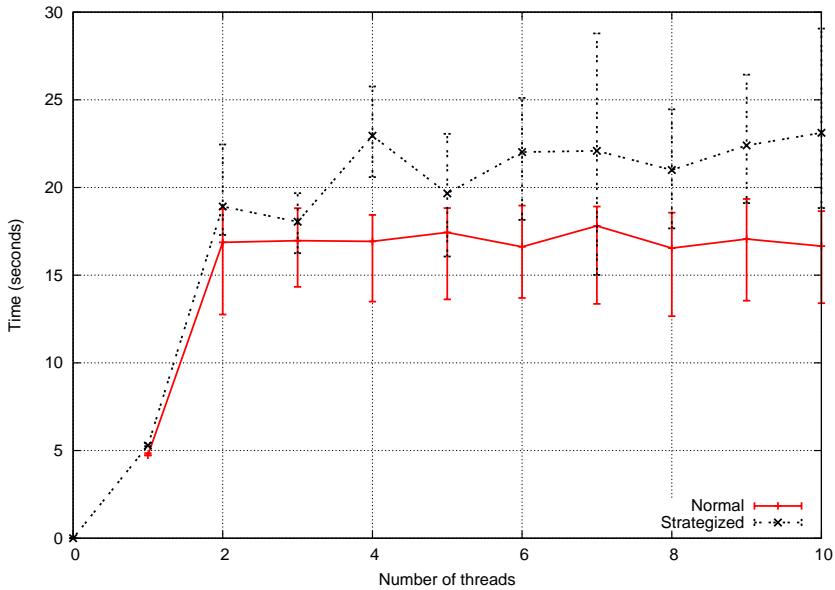


Figure 7.5. Time for 100 000 000 operations on data structure with strategized locking

7.4 Threading scenarios

Depending on how multithreading is applied to the web browser different behavior will arise. This section lists the scenarios that have been simulated to gather knowledge about this.

7.4.1 Scenario: Load web pages and paint all images

This scenario will load a number of web pages concurrently, parse them and handle each JPEG image URL on the page. Each image URL will be handled by adding three new commands, that loads the image data, that decodes it and that paints the decoded result. The painting will be done in a graphical window, spread out over the area. The layout functionality is very simple and aims only to not clutter all images over each other.

Each scenario will load 5 different web pages. Those web pages are copies of real web pages and they are served from another computer on the local network. This will reduce the influence of remote server performance on the results.

All measuring is done on a computer with an Intel Core 2 Duo processor (two cores). Each core has 32 KB L1I and 32 KB L1D cache and 4096 KB shared L2 cache. The running OS was GNU/Linux with kernel 2.6.23.1. The scenario will be simulated using the following concurrency models.

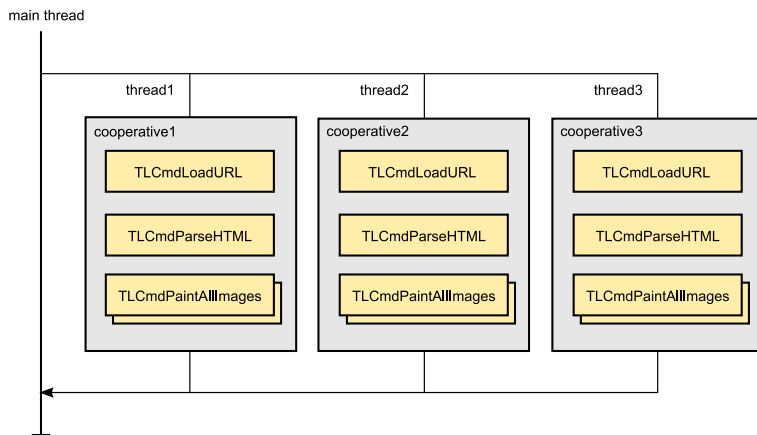


Figure 7.6. Parsing documents and paint images cooperative

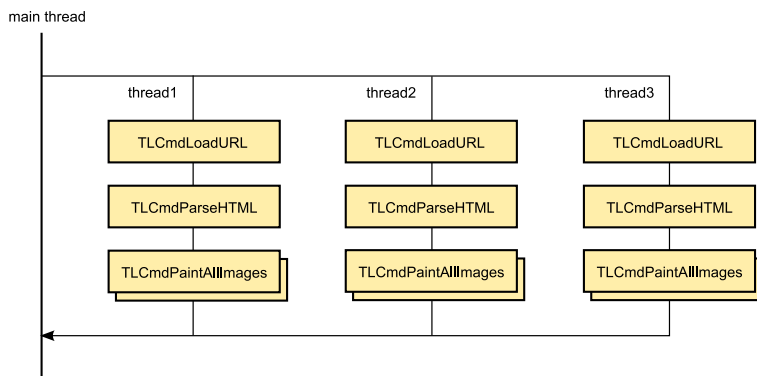


Figure 7.7. Parsing documents and paint images threaded serial

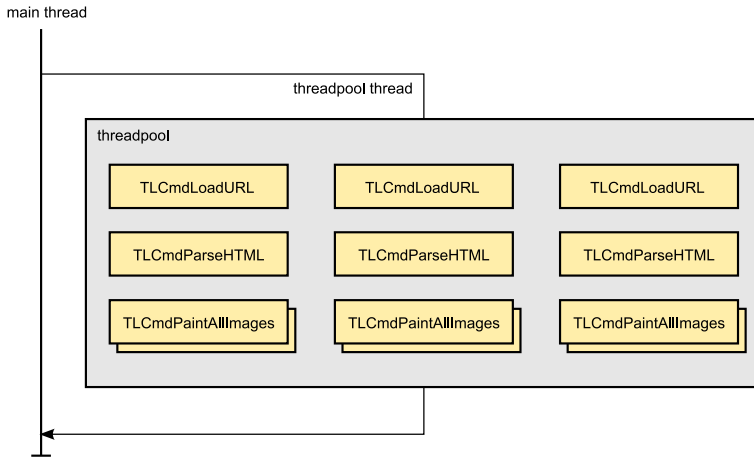


Figure 7.8. Parsing documents and paint images with thread pool

Concurrency model: cooperative

This version will perform the scenario cooperative. One thread per web site is created and a `TLCooperative` command is created for each of them. Each thread will execute all its commands with cooperative multitasking. This is illustrated in figure 7.6. The corresponding Threadlab script can be found in section B.1.

Concurrency model: threaded serial

This version consists of one thread per web page that will perform the load, decode and paint operations. The commands will, for each thread, be executed one after another. This is illustrated in figure 7.7. The corresponding Threadlab script can be found in section B.2.

Concurrency model: thread pool

This version will load, decode and paint all the images concurrently, by the use of a thread pool. This is illustrated in figure 7.8. The threadpool thread are spawned before any measuring is made. The corresponding Threadlab script can be found in section B.3.

Execution time results

The execution times for the different scenarios were measured. The threadpool version was tested with different threadpool sizes. Table 7.9 shows the average execution times for the three concurrency variants. The threadpool version uses the value from 8 threadpool threads. A high number of execution runs was made in each case and averages were calculated.

Version	Execution time
Cooperative	4.98
Threaded serial	5.14
Threadpool	3.84

Figure 7.9. Load web pages and paint all images - execution times

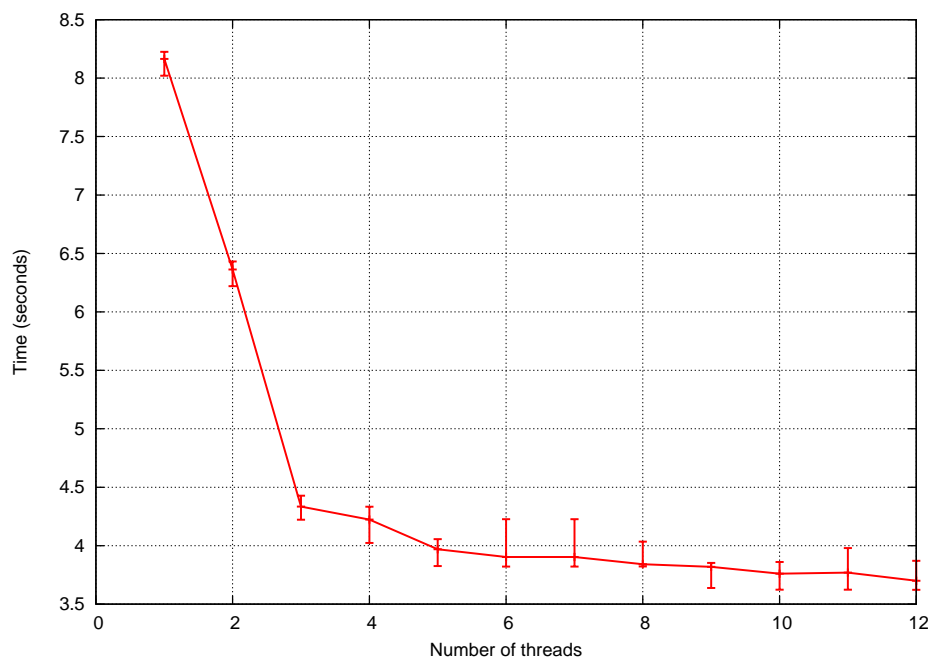


Figure 7.10. Parsing documents and paint images - threadpool execution time

The threadpool version is significantly faster than the two other versions. The speed increase relative to the number of worker threads can be seen in figure 7.10. As stated earlier the values come from a computer with two processor cores. As seen in the figure the execution speed decreases dramatically with three threads and then more slowly. The speed continues to decrease slowly, also with a high number of threads in relation to the number of processor cores. This likely comes from a better flow of scenario instructions and better use of the threadpool.

A good value for the number of threadpool threads can be hard to determine. It is important to remember that each thread also means a possible overhead and that the memory area is divided between threads. Figure 7.10 shows that 6 or 8 threads could be a reasonable value for a two core machine.

Performance counting results

Table 7.11 shows important event values gathered from simulations of this scenario. A total of five simulation runs per event was performed and an average was calculated. The threadpool variant used 4 worker threads.

The table shows that the number of finished instructions, `INST_RETIRED:ANY_P`, is greatly higher in the cooperative case. It is around 35.3 times higher than the threaded case and 29.4 times higher than the threadpool case. This comes from the fact that the download operations need many iterations by the message handler and that the operations dependent on download will have to spin as well. A better message handler would check file descriptors for the transfers before handling a message. The higher number of retired operations in several cases falsely affects the calculated ratios in favor of the cooperative case. This means that the ratio for the cooperative case is of no use in relation to the other two cases.

The number of branch mispredictions is about 7% higher for the cooperative case. The number of data and instruction table misses is significantly lower in the threaded and threadpool cases. The number of L1I misses is significantly higher for the cooperative case (about 2.8 times the value for the others). This comes from the message handler that continuously handles different operation code that is too large to fit in the small L1I cache. The number of L1D misses is larger as well for the cooperative version. In the same way the number of L2 cache misses is larger when running cooperative.

Table 7.12 shows important performance ratios based on the events in table 7.11. The ratios for the cooperative case are misleading due to the heavy load of the message handler when operations wait for data. This means that the only really possible comparison is that between the threaded and the threadpool version. But also here a difference in the number of used operations makes it hard to do good comparison. The interesting result is that the values for the most important events are pretty much the same for the threaded and the threadpool cases. The threadpool case has just slightly larger L1I and L1D miss rates, but the L2 miss rate is significantly smaller.

It is interesting to see that it differs so little between the threaded and the threadpool case. The threaded case is pretty stupid and has no concurrency of the operations within the handling of each loaded web site. This means that the execution has a well predicted flow and that it does not switch between different operations by itself before the current one is finished. It will of course be preempted by the kernel. The threadpool case is more advanced and gives a concurrency within the handling of each web page. It is also compatible with cooperative message API making it a suiting strategy when adding multithreading to a non threaded web browser.

According to table 7.12 the L2 cache miss rate (per instruction) is 2.89% for the threadpool case. This corresponds to a cache hit rate of 97.10%. Regarding how many and how different the operations are in the simulation this is a good result.

Event	Coop	Threaded	Threadpool
INST_RETIRED:ANY_P	373048966	10562365	12695175
UOPS_RETIRED:ANY	1681894832	971706089	972200967
CPU_CLK_UNHALTED:CORE_P	3941429140	3015484608	3065160667
BR_INST_RETIRED_MISPRED	22086904	20653444	20607168
ITLB_MISS_RETIRED	1353985	240200	210600
DTLB_MISSES:ANY	3589067	1338993	1384619
LOAD_BLOCK:L1D	9355896	6082993	7093538
L1I_MISSES	6150413	2203008	2253693
L1D_CACHE_LD:MESI	1863024912	1599746391	1600187970
L1D_REPL	17408185	9165387	9753766
L2_LINES_IN:SELF	1143061	498352	368049

Figure 7.11. Load web pages and paint all images - important events

Ratio	Coop	Threaded	Threadpool
Clocks / instr (CPI)	10.56545	285.49333	241.44296
Branch Misprediction Per Micro-Op Retired	0.01313	0.02125	0.02120
Branch Misprediction Performance Impact	0.01004	0.00745	0.00630
Bus Not Ready Ratio	0.00129	0.00047	0.00123
Bus Utilization	0.00665	0.00399	0.00948
ITLB Miss Rate (spread out execution)	0.00363	0.02274	0.01659
DTLB Miss Rate (too many data page accesses)	0.00962	0.12677	0.10907
Loads Blocked by L1 Data Cache Rate	0.00237	0.00202	0.00231
L1 Instruction Cache Miss Rate (per instruction)	0.01649	0.20857	0.17752
L1 Instruction Cache Miss Rate (per cpu cycle)	0.00156	0.00073	0.00074
L1 load rate	0.47268	0.53051	0.52206
L1 Data Cache Miss Rate (per instruction)	0.04666	0.86774	0.76830
L1 Data Miss Perf. Impact	0.03533	0.02432	0.02546
L2 Cache Miss Rate (per instruction)	0.00306	0.04718	0.02899
L2 Cache Miss Rate (per cpu cycle)	0.00029	0.00017	0.00012

Figure 7.12. Load web pages and paint all images - performance ratios

7.4.2 Scenario: Download multiple large files

Concurrency model: threaded

This scenario will download a number of large files with one thread per file download. The corresponding Threadlab script can be found in section B.5.

Concurrency model: cooperative

This scenario will download a number of large files with cooperative multitasking. The corresponding Threadlab script can be found in section B.4.

Results

Execution of the scenario showed no difference in transfer speeds between the two variants. The transfer speed was limited by network bandwidth and the simulation managed to use the whole 100 Mbit/s bandwidth in both cases. One obvious difference was the much larger fraction of user and system time used for the cooperative case. This was due to the overhead of millions of messages that had to be handled by the message handler. Many times the message handler called the `TLCmdLoadURL` call-back function just to find that no new data was available. A serious cooperative application solves this in a better way, by checking file descriptors for activity before calling a message call-back.

Even if the lack of file descriptor monitoring in Threadlab the simulation shows that the threaded case offloads the application and the system considerably.

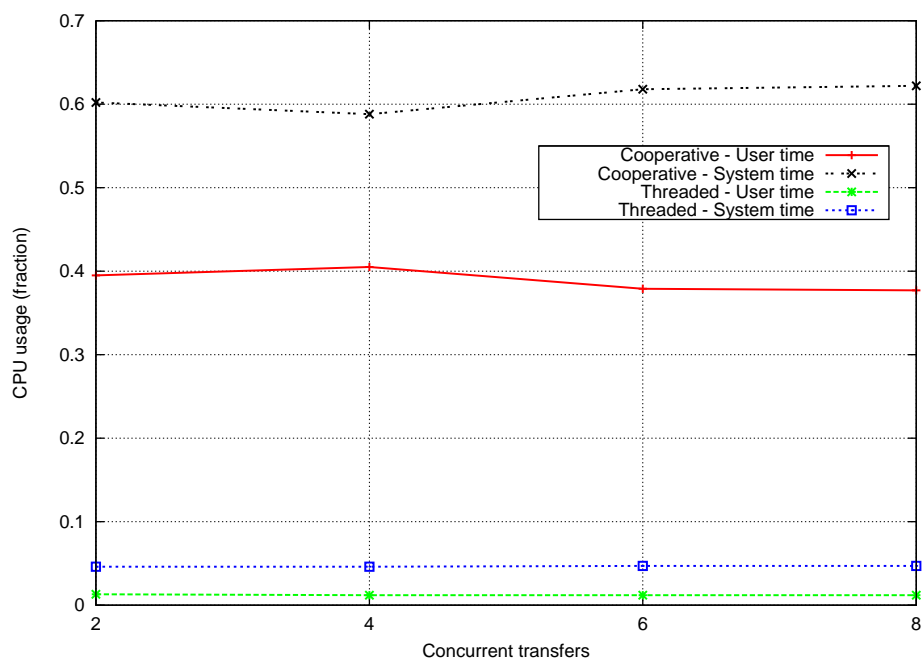


Figure 7.13. CPU usage in percent for concurrent file transfers

Chapter 8

Results

This chapter presents design decisions and recommendations based on all previous chapters. It will first present general threading conclusions. That will be followed by motivated structural designs of threaded web browsers and finally a section with redesign guidelines for a non threaded browser.

8.1 Threading on uniprocessor machines

When using multithreading on a system with only one processor there is no performance gain from having parallel execution. This means that there is less or no reason to stress implementations of threading for those systems. Also there exist platforms where multithreading is not supported.

A lesson from chapter 4 is that a program running on a good operating system in some occasions can benefit from multithreading also when running with only one processor. The reason is that the system can detect that one thread is waiting for I/O and schedule another thread of the same program for execution. When running without threads the program would instead be preempted in benefit of another program. The result is a higher throughput, not actual computation time. It is important to know though, that fine grain threading on uniprocessor machines most likely would result in more thread overhead than performance gain.

The type of concurrency used can be either decided during the compile phase or, more dynamically, during execution. Selection during compilation is suitable for well defined hardware platforms that are known to have a specific definition. This can for example be PDA:s, media devices, or telephones. Dynamic selection of concurrency is suitable for diverse platforms such as desktop computers and laptops.

Below are some variants of the degree of threading possible to use on a uniprocessor system.

No multithreading at all On a uniprocessor system there is normally no big performance gain using multithreading compared to cooperative concurrency.

For hardware that is known in advance to have no more than one processor it is safe to, at compile time, select to use no multithreading at all. For less known hardware the choice should be made dynamically in run time. If the running program detects that it has only one processor it can select to use no multithreading at all. It then executes everything concurrently, using for example a message handler (see section 4.3.1).

Some multithreading When the web browser is compiled for systems with diverse variants of hardware, but where it is well known that multithreading is supported by the system, then some parts could be selected to run multithreaded regardless of number of processors. A reason for this can be to keep the binary size down by not having functionality for both cooperative and multithreaded execution of some part.

It may also be desirable to keep apart the execution of parts that are identified to always execute parallel of each other. One example of this is the idea to use one thread per handled web page, described in section 8.2.1.

Things that are suitable to not run in separate threads, if using some multithreading also for uniprocessor machines, is things like image decoding or file transfer. Those are threaded in a multi processor environment mainly to exploit the extra performance gained from having true parallel execution. When running on a uniprocessor machine they will instead cause lots of thread switching and cause delays.

Same degree as for multi processor machines A multithreaded program that is designed with multithreading from the start probably uses the same, or near the same, multithreading regardless of whether it executes on a machine with one or multiple processors. It can simplify the source code significantly to have only one concurrency mechanism. On the other hand some operations that on a multi processor machine give good performance increase due to parallel execution can give bad levels of thread overhead when executed on one processor machines.

If it is chosen to use full threading throughout a program, then a detected uniprocessor machine should preferably still result in a lower number of threads. If the program uses thread pools (see section 6.4) to handle common operations the number of pool threads should be lowered to one.

8.2 Threaded browser design

This section describes and motivates design possibilities for threaded browsers.

8.2.1 One thread per page

In a web browser the data of each web page is highly related to that page. There is a one-to-one relation between the document and things such as the DOM tree representation, parsed CSS definitions, running ECMAScripts, layout calculations

and other important elements. This high level of data integrity for web pages can be exploited in threaded design by using one thread per web page. This would mean that the contents of each tab within a browser window or each individual browser window would be handled by a thread.

Internally each page thread can handle concurrency cooperatively. The overhead of a message handler or `select` loop (see section 4.3.2) is little.

A problem with this design is that some data resources should be shared between pages. This can for example be loaded and decoded images and CSS files. In browsers with tabs it is common that users use multiple tabs for a single web site. A user can for example, from a news agency site, spawn tabs for each of the top news headlines. To prevent the data to be downloaded and parsed each time and then use unnecessarily amounts of memory it might be shared among page threads. This is not problematic as long as the threads are not supposed to do any changes to the data. But for example CSS definitions can be changed by ECMAScripts. This would mean that the CSS file data can be shared, but the parsed result of the data should be thread local.

The following operations and modules should be running in the page thread.

- Downloading HTML, CSS, etc.
- Parsing of HTML, CSS, ECMAScripts, etc.
- Initialization of image downloads
- Layout calculations
- ECMAScript execution
- Layout calculations
- Page paint operations
- Continuous plug-in execution

8.2.2 Thread pool for images

Modern web pages are very graphical and it is not uncommon with hundreds of images on a single web page. Even if many images are small the loading and decoding of every one may hurt the user experience. Cooperative multitasking may give the impression that all the images are loaded concurrent, but it will not gain from an increasing number of processors. To exploit multiple processors an easy way would be to use a thread pool for images. Jobs for those threads would then be to load and decode the images. By using one job request for the download operation and one for the decode operation it is possible to make those operations work sequentially. Operations that take long time can re-register themselves to the thread pool and return prematurely to handle more concurrent operations than the number of worker threads.

Since images, as mentioned in section 8.2.1 above, could be shared among many pages, data protection for images would be needed anyways.

One reason to use a threadpool especially for images is the possibility of an increased cache hit rate. If the threads execute a limited set of instructions they will easier fit in the cache.

8.2.3 General thread pool

Section 7.4 showed that it was possible to reach a cache hit rate of 97.10% in the simulation of concurrent loading, parsing and painting of web pages using a thread pool. This shows that a thread pool could be used instead of a message handler for a wider area of execution.

The classes that currently contain functions that are called by a message handler could be modified to be called by worker threads. The most important difference is that a single threaded message handler may guarantee that the messages are executed in order and that a succeeding message will not be executed before the preceding message has finished. This may not be true if it is instead called by the worker threads of a thread pool. This means that the operations that are moved to be handled this way have to handle the possibility that the data they require for their operation may still be pending. For example an image decoder must handle the case where the data retrieval is currently not finished.

The number of worker threads should be determined dynamically. If the thread operations are computation intensive one thread more than the number of cores should be enough. All cores would then be busy with the computation. If the operations are data intensive threads will likely be preempted when it waits for I/O. Then a higher number of threads would give better performance.

8.2.4 Threaded file transfers

The most common file transfer in a web browser is the transfer of page elements, like pictures. But it has also become popular to use web browsers to download data that is stored locally on the users hard disk and not used in any web page. This is normally done by use of the HTTP or FTP protocols. It is not uncommon that those methods are used to transfer large amounts of data, like CD images. The possibility to send data from the client browser to the server has normally been used sparsely, but the use may increase with the growing popularity of web sites that encourages user participation. This section apply to those file transfers, to and from the user hard disk drive.

When a non threaded and cooperative multi tasked web browser transfers larger amounts of data it may be heavily slowed down and there will be a hard balance between user browsing experience and file transfer performance. Regardless of cooperative design it will have to repeatedly handle the file transfers in the same single thread as other browser functionality. Since I/O operations are involved the time for handling the file transfers can be hard to determine. This area could benefit a lot from multithreading, especially on multi processor systems. Section 7.4.2 showed a big overhead for concurrency mechanisms when using cooperative multitasking and a much less system load using a multithreaded approach.

Below follows some possible granularity of threading for file transfers.

One file transfer thread The simplest design would be to create one extra thread for handling file transfers to disk. This would free the cooperative concurrency mechanism from handling file transfers. Multiple concurrent downloads could for be handled, within the thread, by a `select` loop waiting for data on file descriptors. Using a single thread for file transfers is simple, but does not scale well with increasing number of processors.

Thread pool When many concurrent file transfers occur within a single file transfer thread that thread may be scheduled out if one of the transfers locks when waiting for I/O. It does also scale badly with an increasing number of processors. This can be solved by using a pool of threads responsible for file transfers. A suiting number of threads can for example be the same as the currently available processor cores.

When this design is used it is recommended that the same thread is responsible for the whole transfer operation for a file. This will reduce the need of data serialization to a minimum.

One thread per transfer Using one thread for each file transfer is a simple way to move all the file transfer concurrency logics to the operating system. It will scale well and if designed right there will be little need for data serialization since the individual file transfers will not affect each others. Status information about each file transfer can be made available for reading to other parts of the browser without race condition problems.

One negative thing with this solution is that a large number of file transfers may cause the application to reach the cap on number of threads.

Simulated results

Simulations of file transfers, according to the scenarios mentioned in section 7.4.2 show that there is no gain on transfer speeds when one thread is used per transfer, than if it is handled cooperatively. But an important result is that the cooperative transfers used lots of processor power handling the concurrency. This processor power is used on behalf of other browser functionality, most importantly it may affect the response and user experience.

BitTorrent transfers

If the browser contains support for the BitTorrent peer-to-peer file sharing protocol it might be even more useful with multiple threads. A file that is downloaded using the BitTorrent protocol is divided in a normally large amount of pieces. The pieces may be downloaded in any order and pieces that are finished downloading may be concurrently uploaded to other users that want those pieces. This often results in large amounts of I/O and processing. Each file transfer may result in hundreds of small file transfers. Using a thread for each handled torrent download will offload other parts of the web browser. This will also ensure that

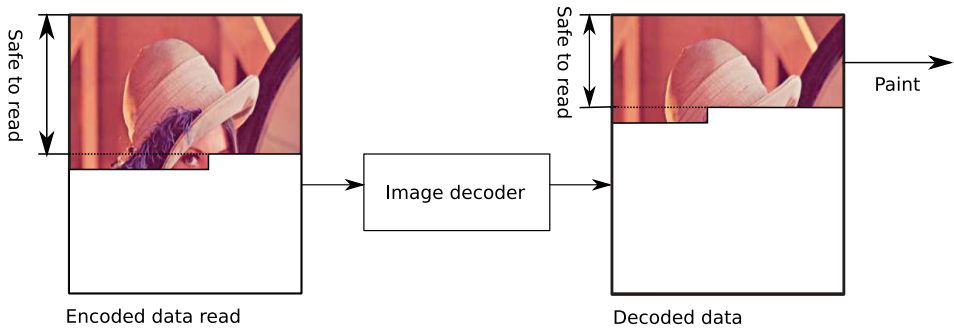


Figure 8.1. Decoding partially read sequential data

transfers that wait for I/O operations do not slow down the system unnecessarily. The data integrity within each torrent download is high and there is no special needs for data serialization. More fine grained threading could be used, to get more benefit from multiple processors. To minimize the need of serialization a data decomposition should be made, where the data pieces are divided among the threads.

8.2.5 Sequentialism

When for example an image is loaded from an Internet web server, to be displayed in the local web browser it is normally loaded sequentially. The network layers below the application assures that the data is read in the same order as it is initially transmitted.

It is common that resources received sequentially are received once and then never changed. This can in many cases be used to make read access to the received data secure also for concurrent environments. By adding a counter that is increased during data retrieval, it is possible to tell how much of the data currently is safe to use.

In the same manner operations like for example image decoding of compressed image formats to raw bitmap data can be done sequentially, working on byte after byte or line after line of loaded data. This means that an image decoder could continuously decode the data that is received, up to what currently is flagged to be safe. As mentioned above this could be achieved by an integer number telling how much of the data block can be safely read.

In a further step the painting of the decoded image data could be done in a similar fashion. The image decoding process in many cases produces sequential output. This could be used in a multithreaded system to concurrently paint the decoded image data at the same time as the decoder concurrently decodes the currently loaded compressed data.

This pipelining principle is shown in figure 8.1.

8.2.6 Use scoped locking

Section 7.3.1 showed that scoped locking had no measurable time overhead. The memory overhead is small since it only requires a small object per lock. Use of scoped locking is therefore encouraged when possible. Scoped locking will encourage small critical sections, not larger than the current scope. It will also ensure that locks are unlocked.

8.3 Redesign of non-threaded browser

This section contains guidelines for redesign of a currently single threaded web browser.

8.3.1 Using thread pools

As mentioned in section 8.2.2 above the use of thread pools is a simple way to exploit multiple processors. This is mostly interesting for operations that occur numerous times when generating a web page and where the operations work on a tightly coupled set of data.

Implementation of a thread pool could be a good first step towards better performance in multi processor environments. Instead of adding the suiting operations to the cooperative message handler they are added to the thread pool work list. If the APIs for both alternatives are made similar it is possible to change behavior both statically and dynamically.

8.3.2 Split up pages

A suiting second step, after implementation of a thread pool, would be to split up the application to use one thread per web page, as mentioned in section 8.2.1 above.

8.3.3 Move away file downloads

Download operations that are not part of browsing could be moved to separate threads, as mentioned above. This could for example be files being downloaded to local disk as well as BitTorrent transfers.

8.3.4 Use of global data

If a single threaded web browser is redesigned to use threading only for heavy and isolated parts, such as file transfers and image decoding, then global data related to page rendering and other things running in the main thread will not be a problem. But in a design where every page is executed in its own thread global data may be target for race conditions. This includes not only global variables, but global objects that may contain local variables.

Before it will become possible to run each page in a single thread it is necessary to make sure that global data is localized and handled correctly. This can either be done by making the global data locally known only to single threads or by adding serialization mechanisms to prevent race conditions.

Global objects that contain system wide information, such as program settings, are normally rarely changed and the critical sections during read operations are normally small. In this case readers-writer locks could be used. It is a synchronization primitive that allows multiple threads to read from but only one at a time to write to a resource.

8.3.5 Other protocols

If the browser contains for example mail functionality, polling and reading of RSS sources, IRC-client (Internet Relay Chat) and other protocols not related to the main browser functionality then the code for those protocols could benefit from being run in their own threads.

8.3.6 Background work as threads

Background work such as memory management, cookie expiration and other tasks could be moved to a thread for this. This would make it possible to do browsing and background operations concurrently and would release some load from the message handler. One suitable operation could be expiration of cached images, web pages and other elements. Expiration of cookies is another.

The redesign work for this may be pretty large. It may be necessary to add mutex locks around some central parts to make sure that for example a web page that currently is being loaded does not try to use cached data that has been freed or use the old cookie data.

8.3.7 Watchdog thread

When cooperative multitasking still is used within threads, as described in several of the possible threaded designs, it is possible that a thread will lock if a function by some reason does not return. This problem is the same as with browsers written totally cooperative.

One way to handle this is to use one thread that on regular intervals monitors the other threads. The monitored threads could update a watchdog counter and if that counter stops updating the watchdog thread could initiate disposal of that thread and make sure that a new one is created.

If the thread that locked up handled the contents in a browser tab, as described in section 8.2.1, then a restart of that thread could simply appear as a page reload to the user.

8.3.8 ECMAScript engine

The increasing use of ECMAScripts for more and more advanced tasks results in an increased performance need. It may initially be tempting to put the ECMAScript engine for each web page in a separate thread. Unfortunately that can give worse performance. Since ECMAScripts may, and often do, alter the DOM tree for a web page the web page may have to be recalculated and repainted. That means that the calculation and presentation of the web page and the execution of the ECMAScript both access the DOM tree. Running them in separate threads means that there will be a need for a mutex lock on the DOM tree. A possible way to avoid those locks could be the use of multi-version trees [Vest, 2007].

Bibliography

- Robert Cailliau and James Gillies. *How the Web Was Born: The Story of the World Wide Web*. Oxford University Press, 2000. ISBN 0192862073.
- ECMA. *ECMA-262: ECMAScript Language Specification*. ECMA (European Association for Standardizing Information and Communication Systems), pub-ECMA:adr, third edition, December 1999. URL <http://www.ecma.ch/ecma1/STAND/ECMA-262.HTM>.
- Stephane Eranian. The perfmon2 interface specification. Technical Report HPL-2004-200, HP, Internet Systems and Storage Laboratory, 2005. URL <http://www.hpl.hp.com/techreports/2004/HPL-2004-200R1.html>.
- Gamma, Helm, Johnson, and Vlissides. *Design Patterns Elements of Reusable Object-Oriented Software*. Addison-Wesley, Massachusetts, 2000. ISBN 0-201-63361-2.
- IEEE. *IEEE 1003.1c-1995: Information Technology — Portable Operating System Interface (POSIX) - System Application Program Interface (API) Amendment 2: Threads Extension (C Language)*. IEEE, 1995. URL <http://www.ieee.org/>.
- PAPI. Papi website, 2007. URL <http://icl.cs.utk.edu/papi/>. Visited: 2007-05-22.
- Perfmon2. Perfmon2 website, 2007. URL <http://perfmon2.sourceforge.net/>. Visited: 2007-05-22.
- Douglas C. Schmidt. *Reactor: an Object Behavioral Pattern for Event Demultiplexing and Event Handler Dispatching*. Addison-Wesley, 1995. ISBN 0-201-6073-4. URL <http://www.cs.wustl.edu/schmidt/PDF/reactor-siemens.pdf>.
- Douglas C. Schmidt. Strategized locking, thread-safe decorator, and scoped locking: patterns and idioms for simplifying multithreaded c++ components. *C++ Report, SIGS*, 11(9), 1999. URL <http://www.cs.wustl.edu/schmidt/PDF/locking-patterns.pdf>.
- Douglas C. Schmidt. Thread-specific storage: an object behavioral pattern for accessing per-thread state efficiently. *C++ Report, SIGS*, 9(10), 1997. URL <http://www.cs.wustl.edu/schmidt/PDF/TSS-pattern.pdf>.

David Vest. A version aware data structure designed for incremental rendering of structured documents. Master's thesis, Linköpings universitet, 2007. LITH-IDA-EX-07/007-SE.

Appendix A

Threadlab details

This appendix describes more detailed information on the concurrency evaluation tool Threadlab. Threadlab is a toolkit of data structures and commands that can be combined to perform simulations.

A.1 Threadlab dependencies

Threadlab uses the following open source libraries and programs for various parts of its functionality.

Library	Related modules	Description
libpthread	Thread	The Pthreads library are used as base for threaded execution.
libjpeg	CommandDecodeJPEG, DataStructureImage	CommandDecodeJPEG decodes a memory area or file containing a JPEG image and saves the data to a DataStructureImage.
libxml	CommandParseXML, CommandParseHTML, DataStructureTree	The Gnome xml library is used to parse XML and HTML. The result is stores in a DataStructureTree object.
libcurl	CommandLoadURL	The libcurl library is used to load data from a general URL. The result is stores in a DataStructureMemory object.
libX11	DataStructureWindow	DataStructureImage objects can be painted on graphical windows represented by DataStructureWindow objects.
SWIG	(Threadlab Python API)	SWIG is used to automatically generate Python bindings for the Threadlab C++ interface.
libpython	(Script engine)	The Python bindings created by SWIG are compiled using libpython.
libpfm	(Performance counting)	The Perfmon2 library libpfm is used to communicate with the Perfmon2 kernel module to setup and run performance counting.
Perfmon2	(Performance counting)	To be able to do performance counting with Perfmon2 a kernel patch has to be applied. This patch programs the performance counters in the CPU and stores values during task switches to enable fine grained counting.

Appendix B

Test scripts and measured results

This chapter describes the most important test scripts that were executed as part of the evaluations. The scripts are presented together with the resulting execution results.

B.1 Load and paint all images - cooperative

The following Threadlab script evaluates the scenario described in section 7.4.1. The concurrency model will handle painting of all images on a web page with cooperative multitasking.

```
import threadlab

urls = [
    "http://ilsa.pugo.org/tmp/exjobb/websites/foto_wgt/",
    "http://ilsa.pugo.org/tmp/exjobb/websites/foto_berlin/",
    "http://ilsa.pugo.org/tmp/exjobb/websites/www.dn.se/",
    "http://ilsa.pugo.org/tmp/exjobb/websites/humor/",
    "http://ilsa.pugo.org/tmp/exjobb/websites/www.idg.se/",
]

window = threadlab.TLWindow( 10, 10, 1024, 800 );

for url in urls:
    memarea = threadlab.TLMemarea( 0 )
    tree = threadlab.TLXMLTree()
    thread = threadlab.TLThread()

    cooperative = threadlab.TLCmdCooperative()
    cooperative.addCommand( threadlab.TLCmdLoadURL( url, memarea ) )
    cooperative.addCommand( threadlab.TLCmdParseHTML( memarea, tree ) )
    cooperative.addCommand( threadlab.TLCmdPaintAllImages( tree, window, url ) )
    thread.addCommand( cooperative )
```

B.2 Load and paint all images - threaded serial

The following Threadlab script evaluates the scenario described in section 7.4.1. The concurrency model will handle painting of all images on a web page with a number of threads.

```
import threadlab

urls = [
    "http://ilsa.pugo.org/tmp/exjobb/websites/foto_wgt/",
    "http://ilsa.pugo.org/tmp/exjobb/websites/foto_berlin/",
    "http://ilsa.pugo.org/tmp/exjobb/websites/www.dn.se/",
    "http://ilsa.pugo.org/tmp/exjobb/websites/humor/",
    "http://ilsa.pugo.org/tmp/exjobb/websites/www.idg.se/",
]

window = threadlab.TLWindow( 10, 10, 1024, 800 );

for url in urls:
    thread = threadlab.TLThread()
    memarea = threadlab.TLMemarea( 0 )
    tree = threadlab.TLXMLTree()

    thread.addCommand( threadlab.TLCmdLoadURL( url, memarea ) )
    thread.addCommand( threadlab.TLCmdParseHTML( memarea, tree ) )
    thread.addCommand( threadlab.TLCmdPaintAllImages( tree, window, url ) )
```

B.3 Load and paint all images - thread pool

The following Threadlab script evaluates the scenario described in section 7.4.1. The concurrency model will handle painting of all images on a web page with a thread pool.

```
import threadlab

urls = [
    "http://ilsa.pugo.org/tmp/exjobb/websites/foto_wgt/",
    "http://ilsa.pugo.org/tmp/exjobb/websites/foto_berlin/",
    "http://ilsa.pugo.org/tmp/exjobb/websites/www.dn.se/",
    "http://ilsa.pugo.org/tmp/exjobb/websites/humor/",
    "http://ilsa.pugo.org/tmp/exjobb/websites/www.idg.se/",
]

window = threadlab.TLWindow( 10, 10, 1024, 800 );
threadpool = threadlab.TLThreadpool( 4, threadlab.SPAWN_AT_BOOTSTRAP )

for url in urls:
    memarea = threadlab.TLMemarea( 0 )
    tree = threadlab.TLXMLTree()

    threadpool.addCommand( threadlab.TLCmdLoadURL( url, memarea ) )
    threadpool.addCommand( threadlab.TLCmdParseHTML( memarea, tree ) )
    threadpool.addCommand( threadlab.TLCmdPaintAllImages( tree, window, url ) )
```


B.4 Download multiple large files - cooperative

The following Threadlab script evaluates the scenario described in section B.5, downloading a number of large files with cooperative multitasking.

```
import threadlab

NUM_THREADS = 8

thread = threadlab.TLThread()
cooperative = threadlab.TLCmdCooperative()

for i in range( NUM_THREADS ):
    cooperative.addCommand( threadlab.TLCmdLoadURL( \
        'http://ilsa.pugo.org/tmp/exjobb/download_test_%d' % (i%4+1),
        '/tmp/download_test_%d' % (i+1) ))

thread.addCommand( cooperative )
```

B.5 Download multiple large files - threaded

The following Threadlab script evaluates the scenario described in section B.5, downloading a number of large files with one thread per file download.

```
import threadlab

NUM_THREADS = 8

for i in range( NUM_THREADS ):
    thread = threadlab.TLThread()
    thread.addCommand( threadlab.TLCmdLoadURL( \
        'http://ilsa.pugo.org/tmp/exjobb/download_test_%d' % (i%4+1),
        '/tmp/download_test_%d' % (i+1) ))
```

Index

- Affinity, 22
- BitTorrent, 73
- Browser user interface, 38
- Cache
 - levels, 13
 - memory, 12
 - miss, 14
- Cachegrind, 53
- Cascading stylesheets, 7
- Client side scripting, 9
- Concurrency, 3
 - levels, 19
 - patterns, 39
- Condition variables, 26
- Cookies, 9
- CooperativeMultitasking, 19
- Critical sections, 25
- Deadlock, 26
- Decoding of images, 6
- Document
 - layout, 7
 - module, 33
 - painting, 8
 - tree, 34
- ECMAScript engine, 37
- Evaluation, 49
 - cache misses, 52
 - patterns, 58
 - process cycles, 52
 - scoped locking, 58
 - strategized locking, 59
- flags, 12
- Form response, 8
- frameset, 6
- Graphic canvas, 34
- Hypertext Markup Language, 5
- IFrames, 6
- Image decoding, 37
- Layout module, 34
- Lock free algorithms, 48
- Locking
 - optimistic, 46
 - pessimistic, 46
 - strengths, 46
- Memory
 - connection, 12
 - memory protection, 12
- Message handlers, 20
- Multiprocessing, 14
 - cache coherence, 15
 - symmetric, 14
- Multiprocessing Hardware, 11
- Multithreading, 21
- Network module, 32
- Opera Software, 4
- Optimistic locking
 - strategies, 47
- Parsers, 33
- Parsing HTML, 5
- Perfmon2, 52
- Performance counting, 51
- Performance ratios, 55
- Plugins, 9
- Process

- components, 17
- costs, 18
- execution, 17
- memory, 12
- scheduling, 18

Processor

- execution, 11
- flags, 11
- registers, 11

program counter, 11

Program level concurrency, 19

Pthreads, 22

Race conditions, 23

RAM memory, 12

Recursion, 6

registers, 11

rollback, 47

Scoped locking, 39

select loop, 20

Semaphores, 25

Sequentialism, 74

Strategized locking, 41

Synchronization, 24

System level concurrency, 19

Thread overhead, 22

Thread pool, 44

Thread specific storage, 43

Threaded file transfers, 72

Threading scenarios, 60

Threadlab, 27

- concurrency variants, 28
- scripts, 30
- simulations, 28
- software platform, 27
- system design, 27

Watchdog thread, 76

Web browser evolution, 2

Web browser functionality, 5

Web browser simulation, 31

Upphovsrätt

Detta dokument hålls tillgängligt på Internet — eller dess framtida ersättare — under 25 år från publiceringsdatum under förutsättning att inga extraordinära omständigheter uppstår.

Tillgång till dokumentet innebär tillstånd för var och en att läsa, ladda ner, skriva ut enstaka kopior för enskilt bruk och att använda det oförändrat för icke-kommersiell forskning och för undervisning. Överföring av upphovsrätten vid en senare tidpunkt kan inte upphäva detta tillstånd. All annan användning av dokumentet kräver upphovsmannens medgivande. För att garantera äktheten, säkerheten och tillgängligheten finns det lösningar av teknisk och administrativ art.

Upphovsmannens ideella rätt innefattar rätt att bli nämnd som upphovsman i den omfattning som god sed kräver vid användning av dokumentet på ovan beskrivna sätt samt skydd mot att dokumentet ändras eller presenteras i sådan form eller i sådant sammanhang som är kränkande för upphovsmannens litterära eller konstnärliga anseende eller egenart.

För ytterligare information om Linköping University Electronic Press se förlagets hemsida <http://www.ep.liu.se/>

Copyright

The publishers will keep this document online on the Internet — or its possible replacement — for a period of 25 years from the date of publication barring exceptional circumstances.

The online availability of the document implies a permanent permission for anyone to read, to download, to print out single copies for his/her own use and to use it unchanged for any non-commercial research and educational purpose. Subsequent transfers of copyright cannot revoke this permission. All other uses of the document are conditional on the consent of the copyright owner. The publisher has taken technical and administrative measures to assure authenticity, security and accessibility.

According to intellectual property law the author has the right to be mentioned when his/her work is accessed as described above and to be protected against infringement.

For additional information about the Linköping University Electronic Press and its procedures for publication and for assurance of document integrity, please refer to its www home page: <http://www.ep.liu.se/>