Beware of some English slides!

Betriebssysteme

Vorlesung im Herbstsemester 2008
Universität Mannheim

Kapitel 3a: Grobarchitektur von Laufzeitsystemen und Überblick über ULIX

Prof. Dr. Felix C. Freiling

Lehrstuhl für Praktische Informatik 1

Universität Mannheim

(Diese Folien sind nicht Teil des Basiskurses)

Motivation

- Viel Theorie im Basiskurs Betriebssysteme (und in den bekannten Lehrbüchern)
- Untersuchung eines lebenden Beispiels, um Konzepte in der Praxis kennen zu lernen
- Beispiel ULIX, ein Betriebssystem für die akademische Lehre
- Heute: Überblick über den Entwurf von ULIX

Überblick

- Warum ULIX?
- Literarisches Programmieren
- Grobarchitektur von Laufzeitsystemen
- Entwurf von ULIX
- ULIX-Hardware Überblick

Warum ULIX?

Personal Motivation

- Operating systems (OS) are not my main research topic but one which is interesting and one which I teach
- (Almost) nobody teaching operating systems has written one him/herself
  - (Almost) nobody writes operating systems today anyway
- Convenient sabbatical in the spring 2008:
  - I enjoy programming, so ...
  - ... let's write an operating system!

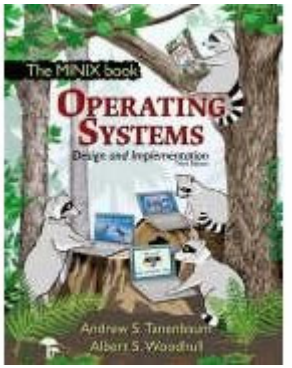
Detail vs. Abstraction in OS Courses

- Implementation details matter
  - Example questions:
    - How is the context switch really programmed?
    - How can user level threads be implemented on multiple kernel level threads?
    - How does the page fault handler ensure that the violating instruction is re-executed?
  - Details are very architecture dependant
  - But real code is very motivating in a course
- Implementation details are unpleasant
  - Abstracted away in (almost) all the popular textbooks
  - Lack of real code

A New OS

- Goal: Find a new balance between detail and abstraction for the OS course in Mannheim
  - Show real code to the students without boring them with HW details
- Principles:
  - Explain the basics of OS implementation in a book showing **real code**
  - Don't target a real architecture: Use a **hypothetical architecture** designed for the OS
  - Implement basics **as simple as possible**: Don't design for speed or efficiency

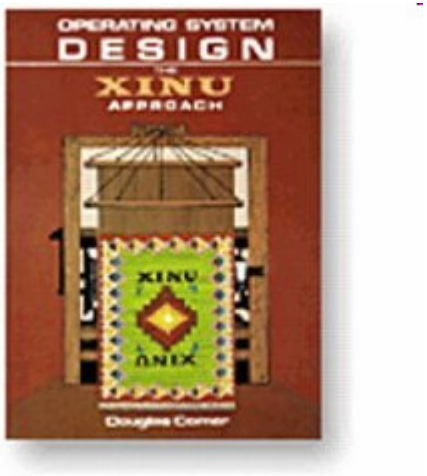
Related Work: The MINIX Book



- Around 1000 pages
- First 500 pages: Textbook on operating systems with references to source code
- Last 500 pages: Source code of MINIX

- For me it is yet another textbook on operating systems (just with a long appendix)

XINU



- System by Douglas Comer
- XINU is not Unix
- Layered operating system for a PDP 11 variant
- No virtual memory
- Not currently supported
- Surprisingly simple...

ULIX

- First idea: re-implement MINIX as a literate program
- Second idea: why not go for a new OS
  - Not tied to particular (real) HW
  - Not aimed at practical fitness
- Result:
  - ULIX: A Literate Unix

Überblick

- Warum ULIX?
- Literarisches Programmieren
- Grobarchitektur von Laufzeitsystemen
- Entwurf von ULIX
- ULIX-Hardware Überblick

Literarisches Programmieren

Programs for Humans not Machines

Let us change our traditional attitude to the construction of programs: Instead of imagining that our main task is to instruct a computer what to do, let us concentrate rather on explaining to human beings what we want a computer to do.

Donald Knuth, Literate Programming, 1984

Programs as Works of Literature

I believe that the time is ripe for significantly better documentation of programs, and that we can best achieve this by considering programs to be works of literature. Hence, my title: "Literate Programming."

Donald Knuth, Literate Programming, 1984

Programming without Programming Language Restrictions

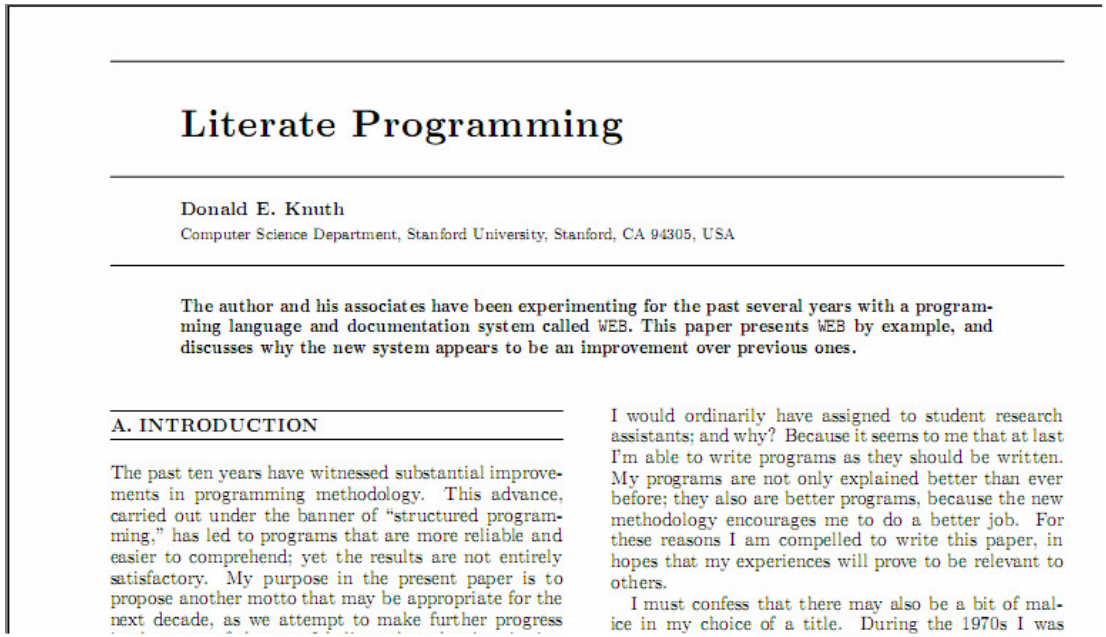
The practitioner of literate programming can be regarded as an essayist, whose main concern is with exposition and excellence of style. Such an author, with thesaurus in hand, chooses the names of variables carefully and explains what each variable means. He or she strives for a program that is comprehensible because its concepts have been introduced in an order that is best for human understanding, using a mixture of formal and informal methods that reinforce each other.

Donald Knuth, Literate Programming, 1984

Donald E. Knuth

http://www-cs-faculty.stanford.edu/~knuth/

- Author of "The Art of Computer Programming" and the TeX typesetter
- Invented literate programming while writing TeX in 1980s
- Main reference is article "Literate Programming", appeared in The Computer Journal, 27(2):97-111, 1984



Literate Programming

Donald E. Knuth
Computer Science Department, Stanford University, Stanford, CA 94305, USA

The author and his associates have been experimenting for the past several years with a programming language and documentation system called WEB. This paper presents WEB by example, and discusses why the new system appears to be an improvement over previous ones.

A. INTRODUCTION

The past ten years have witnessed substantial improvements in programming methodology. This advance, carried out under the banner of "structured programming," has led to programs that are more reliable and easier to comprehend; yet the results are not entirely satisfactory. My purpose in the present paper is to propose another motto that may be appropriate for the next decade, as we attempt to make further progress

I would ordinarily have assigned to student research assistants; and why? Because it seems to me that at last I'm able to write programs as they should be written. My programs are not only explained better than ever before; they also are better programs, because the new methodology encourages me to do a better job. For these reasons I am compelled to write this paper, in hopes that my experiences will prove to be relevant to others.

I must confess that there may also be a bit of malice in my choice of a title. During the 1970s I was

Excerpts

- Article "is" a program to print the first 1000 prime numbers

1. Printing primes: An example of WEB. The following program is essentially the same as Edsger Dijkstra's "first example of step-wise program composition," found on pages 26–39 of his *Notes on Structured Programming*,[2] but it has been translated into the WEB language.

of the main characteristics of WEB is that different parts of the program are usually abbreviated, by giving them such an informal top-level description.]

⟨ Program to print the first thousand prime numbers 2 ⟩

2. This program has no input, because we want to keep it rather simple. The result of the program will be to produce a list of the first thousand prime numbers, and this list will appear on the *output* file.

Since there is no input, we declare the value $m = 1000$ as a compile-time constant. The program itself is capable of generating the first m prime numbers for any positive m, as long as the computer's finite limitations are not exceeded.

11. Generating the primes. The remaining task is to fill table p with the correct numbers. Let us do this by generating its entries one at a time: Assuming that we have computed all primes that are j or less, we will advance j to the next suitable value, and continue doing this until the table is completely full.

The program includes a provision to initialize the variables in certain data structures that will be introduced later.

⟨ Fill table p with the first m prime numbers 11 ⟩ ≡

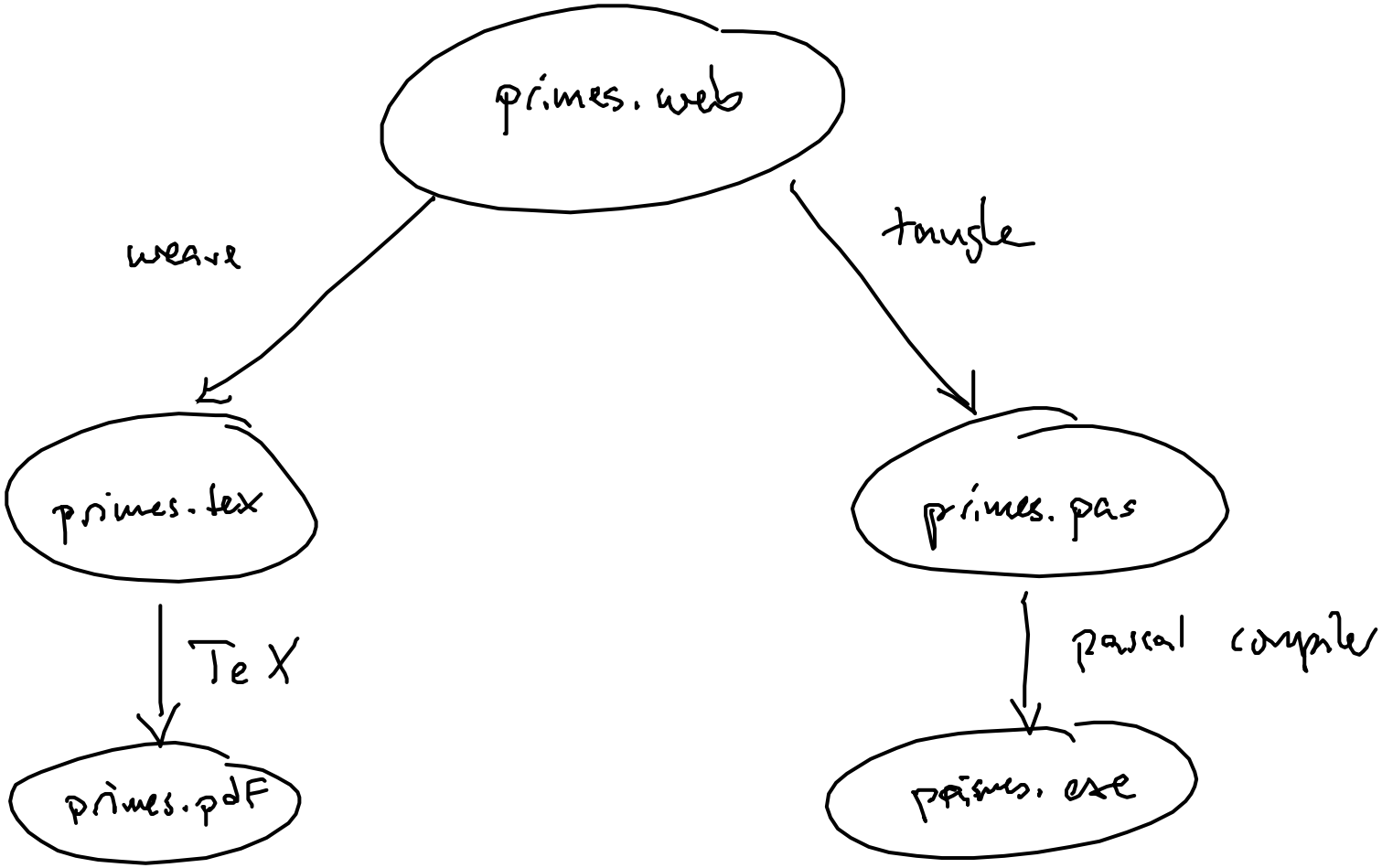
```
⟨ Initialize the data structures 16 ⟩;
while k < m do
  begin ⟨ Increase j until it is the next prime
        number 14 ⟩;
  k ← k + 1; p[k] ← j;
  end
```

This code is used in section 3.

Basic Principles

- Literate programming combines ...
  - a **typesetting language** for writing comments (informal code) like TeX or LaTeX
  - a **programming language** for writing (formal) code like Pascal, Java, C
  - plus some "meta" markup to separate both
- Programs are broken down into small named parts called **chunks**
  - Each chunk starts first with informal text, then the formal text
  - Each chunk should be understandable by itself
  - Each chunk has a name and can be used (by that name) in other chunks

Workflow of a WEB System



TANGLE

- Generates code for compiler
  - Mainly performs named chunk replacement (like macro replacement)

```
\font\ninerm=cmr9
\let\mc=\ninerm % medium caps
\def\WEB{{\tt WEB}}
\def\PASCAL{{\mc PASCAL}}
\def\[{\ifhmode\ \fi$[\mkern-2mu[$}
\def\]{$]\mkern-2mu]$\ }
    :
\hyphenation{Dijk-stra}

@* Printing primes: An example of \WEB.
The following program is essentially the same
as Edsger Dijkstra's @^Dijkstra, Edsger@>
``first example of step-wise program
composition,'' found on pages 26--39
of his {\sl Notes on Structured
Programming},$^\Dijk$  but it has been
translated into the \WEB\ language. @.WEB@>

\[Double brackets will be used in what
follows to enclose comments relating to \WEB\
    :
an informal top-level description.\]

@p @<Program to print the first thousand
prime numbers@>
```

```
@ This program has no input, because we want
to keep it rather simple.  The result of the
program will be to produce a list of the
first thousand prime numbers, and this list
will appear on the |output| file.

Since there is no input, we declare the value
|m=1000| as a compile-time constant. The
program itself is capable of generating the
first |m| prime numbers for any positive |m|,
as long as the computer's finite limitations
are not exceeded.


@<Program to print...@>=
program print_primes(output);
const @!m=1000;
@<Other constants of the program@>@;
var @<Variables of the program@>@;
begin @<Print the first |m| prime numbers@>;
end.
```

WEAVE

- Generates typeset version of program
  - Does prettyprinting and indexing

⟨ Fill table p with the first m prime numbers 11 ⟩
Used in 3.
⟨ Give to *j_prime* the meaning: j is a prime number 22 ⟩
Used in 14.
⟨ If $p[n]$ is a factor of j, set *j_prime* ← *false* 26 ⟩
Used in 22.
⟨ Increase j until it is the next prime number 14 ⟩
Used in 11.
⟨ Initialize the data structures 16, 18 ⟩ Used in 11.
⟨ Other constants of the program 5, 19 ⟩ Used in 2.
⟨ Output a line of answers 10 ⟩ Used in 9.
⟨ Output a page of answers 9 ⟩ Used in 8.
⟨ Print table p 8 ⟩ Used in 3.
⟨ Print the first m prime numbers 3 ⟩ Used in 2.
⟨ Program to print the first thousand prime numbers 2 ⟩
Used in 1.
⟨ Update variables that depend on j 20 ⟩ Used in 14.
⟨ Update variables that depend on *ord* 21, 25 ⟩
Used in 20.

Pascal WEB

- Knuth wrote the first WEB tool as a combination of Pascal and TeX to write TeX
- Resulted in a special volume of "Computers and Typesetting" (TeX - The Program)
  - "The first program which you could read during a pleasant evening on the sofa"
- Excessive indexing, even automatically generated "mini indexes" at the bottom of odd pages
  - Showing all definitions and uses of identifiers on the open double page
- System soon extended to C (Knuth and Levy)
  - CWEB combines C and TeX
  - More tools followed ...

Norman Ramsey

http://www.eecs.harvard.edu/nr/

- Found restriction to C/Pascal and TeX is too severe
- Wanted to program literately in any programming language - with (almost) any formatting language
- Invented noweb
  - First programming language independent literate programming tool
  - Focus on simplicity and extensibility

noweb

- You can use any programming language with noweb
  - Sacrifice automatic indexing and prettyprinting
  - You can add heuristic filters for different languages for indexing
- You can use (almost) any formatting language with noweb
  - Write "plain" LaTeX, generate LaTeX, Troff, HTML

Printing Primes Example

```
\title{Printing Primes: An example of \nw}

\section{Printing Primes: An example of \nw}

The following program is essentially the program that appears in
Reference~\cite{knuth:literate}, the first article on literate programming,
but rendered using \nw. An important differents is the {\tt WEB} has macros, but \nw\ does not.
Knuth's program is itself essentially the same as Edsger Dijkstra's
``first example of step-wise program composition.''~\cite[pages 26--39]{dahl:structured}.

Dijkstra's program prints a table of the first thousand prime numbers.
We shall begin as he did, by reducing the entire program to its
top-level description.

<<*>>=
<<program to print the first thousand prime numbers>>

@ This program has no input, because we want to keep it simple.
The result of the program will be to produce a list of the first
thousand prime numbers, and this list will appear on the [[output]]
file.

Since there is no input, we declare the value [[m = 1000]] as a
compile-time constant.
The program itself is capable of generating the first [[m]] prime
numbers for any positive [[m]], as long as the computer's finite
limitations are not exceeded.

<<program to print the first thousand prime numbers>>=
program `print_primes(output);
  const m = 1000;
        <<other constants of the program>>
  var <<variables of the program>>
    begin <<print the first [[m]] prime numbers>>
    end.
@
...
```

Typeset noweb Example

1 Printing Primes: An example of noweb

The following program is essentially the program that appears in Reference [1], the first article on literate programming, but rendered using noweb. An important differents is the WEB has macros, but noweb does not. Knuth's program is itself essentially the same as Edsger Dijkstra's "first example of step-wise program composition." [2, pages 26–39].

Dijkstra's program prints a table of the first thousand prime numbers. We shall begin as he did, by reducing the entire program to its top-level description.

1a ⟨* 1a⟩≡
⟨*program to print the first thousand prime numbers* 1b⟩

⟦Double brackets will be used in what follows to enclose comments relating to noweb itself. This definition of the root module could have been eliminated by choosing to use

```
notangle -R'program to print the first thousand prime numbers'
```

to extract the program.⟧ This program has no input, because we want to keep it simple. The result of the program will be to produce a list of the first thousand prime numbers, and this list will appear on the output file.

Since there is no input, we declare the value m = 1000 as a compile-time constant. The program itself is capable of generating the first m prime numbers for any positive m, as long as the computer's finite limitations are not exceeded.

1b ⟨*program to print the first thousand prime numbers* 1b⟩≡

```
program 'print_primes(output);
  const m = 1000;
        ⟨other constants of the program 2b⟩
  var ⟨variables of the program 2a⟩
    begin ⟨print the first m prime numbers 1c⟩
    end.
```

Überblick

- Warum ULIX?
- Literarisches Programmieren
- Grobarchitektur von Laufzeitsystemen
- Entwurf von ULIX
- ULIX-Hardware

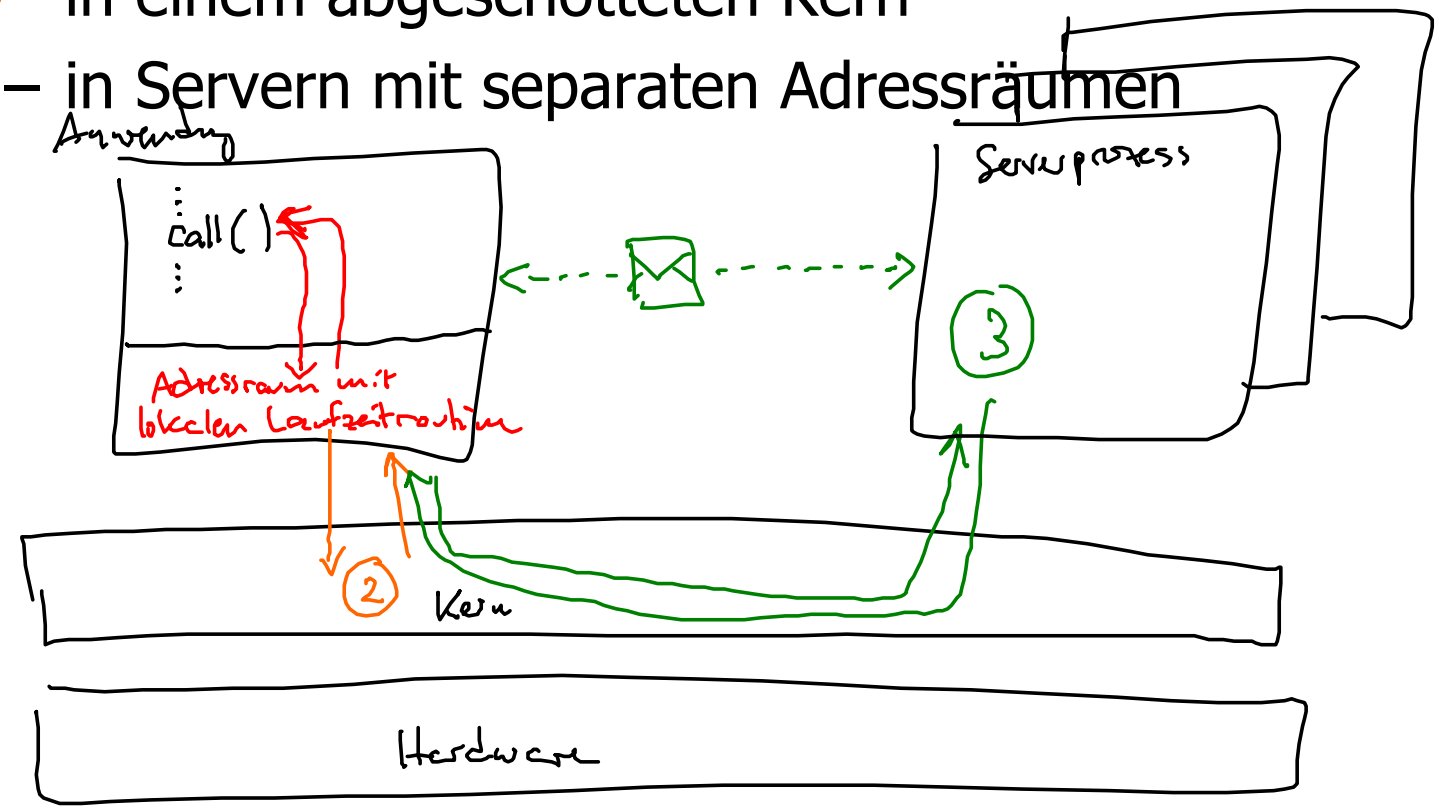
Grobarchitektur von Laufzeitsystemen

Notwendige Dienste

- Prozessmanagement
  - Virtueller Speicher
  - Virtuelle Prozessoren (Threads)
- Prozessinteraktion
  - Über Speicher oder Nachrichten
- Ggf. noch persistenter Speicher und Gerätemanagement

Bereitstellung der Dienste

- Drei mögliche Realisierungsorte:
  - 1 – im Adressraum der Anwendung
  - 2 – in einem abgeschotteten Kern
  - 3 – in Servern mit separaten Adressräumen



Realisierung im Adressraum der Anwendung

- Zugriff auf den Dienst erfolgt immer durch einen lokalen Prozeduraufruf (im Adressraum der Anwendung)
  - Dienst kann dann ggf. vollständig im Adressraum der Anwendung erbracht werden
  - Ansonsten Anforderung von Unterstützung durch den Kern oder andere Server
- Adressraum-lokale Realisierung eines Dienstes bietet sich an, wenn
  - der Dienst eine anwendungsbezogene Spezialisierung eines allgemeineren, durch den Kern oder externe Server angebotenen Dienstes darstellt, und
  - durch den Dienst keine Schutzprinzipien verletzt werden (Aktionen geschehen mit den Rechten der Anwendung)
- Beispiele: Berechnungsroutinen

Realisierung im Kern

- Kerne stellen die Basisabstraktionen Adressraum, Thread und Prozeßinteraktion bereit
  - Benötigen dafür vollständigen Zugriff auf die Hardware
  - Müssen von den Anwendungen vollkommen abgeschottet sein
    - Kein direkter Zugriff aus der Anwendung in den Speicher des Kerns
  - Aufruf von Kernfunktionen nur über den TRAP-Mechanismus mit Umschaltung in den Supervisor-Modus
- Minimale Kerne sind am universellsten einsetzbar
- Abgeleitete Dienste auf höheren Ebenen implementiert
- Realisierung eines Dienstes in einem separaten Server-Prozeß sinnvoll, wenn der Dienst mit dem geschützten Zugriff auf Geräte verbunden bist
  - Beispiel: Dateidienst

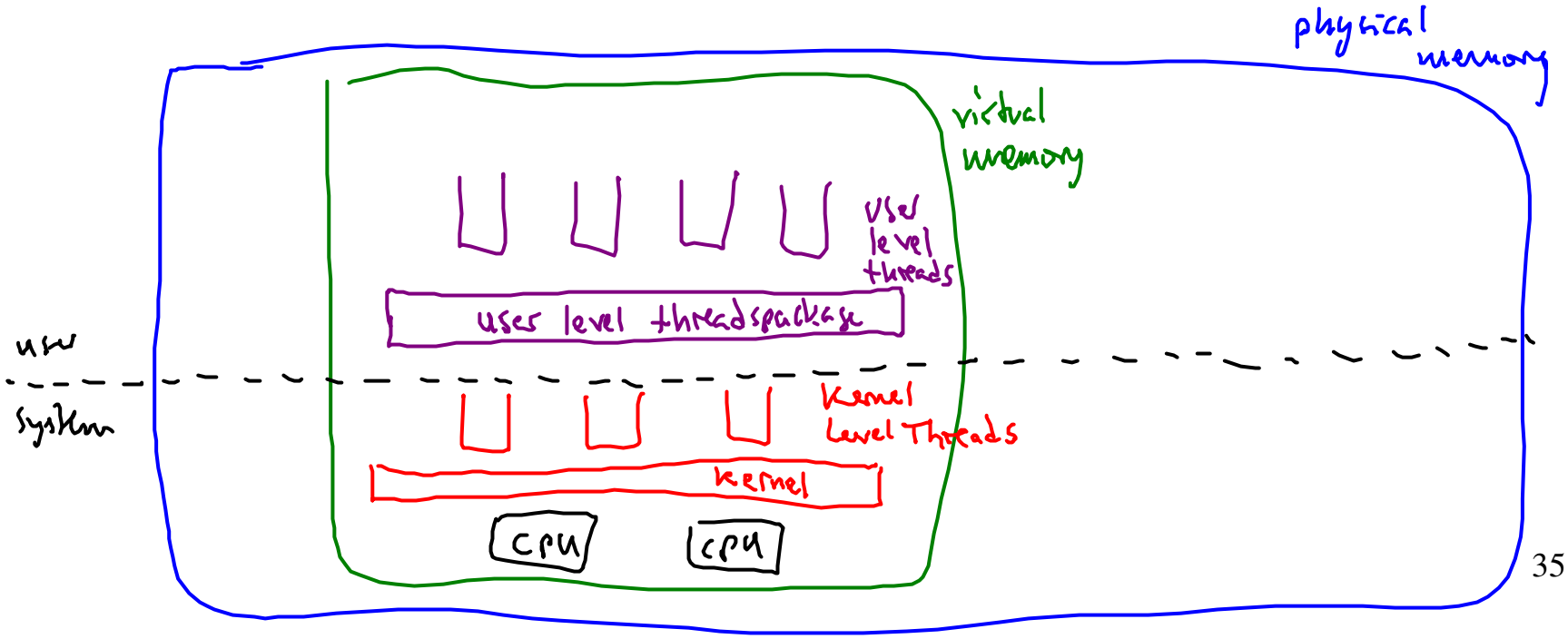
Überblick

- Warum ULIX?
- Literarisches Programmieren
- Grobarchitektur von Laufzeitsystemen
- Entwurf von ULIX
- ULIX-Hardware Überblick

Entwurf von ULIX

Basic Abstractions in ULIX

- Basic abstractions of operating systems:
  - Virtual memory as abstraction of physical memory
  - Virtual processors (threads) as abstraction of physical processors



What ULIX should have...

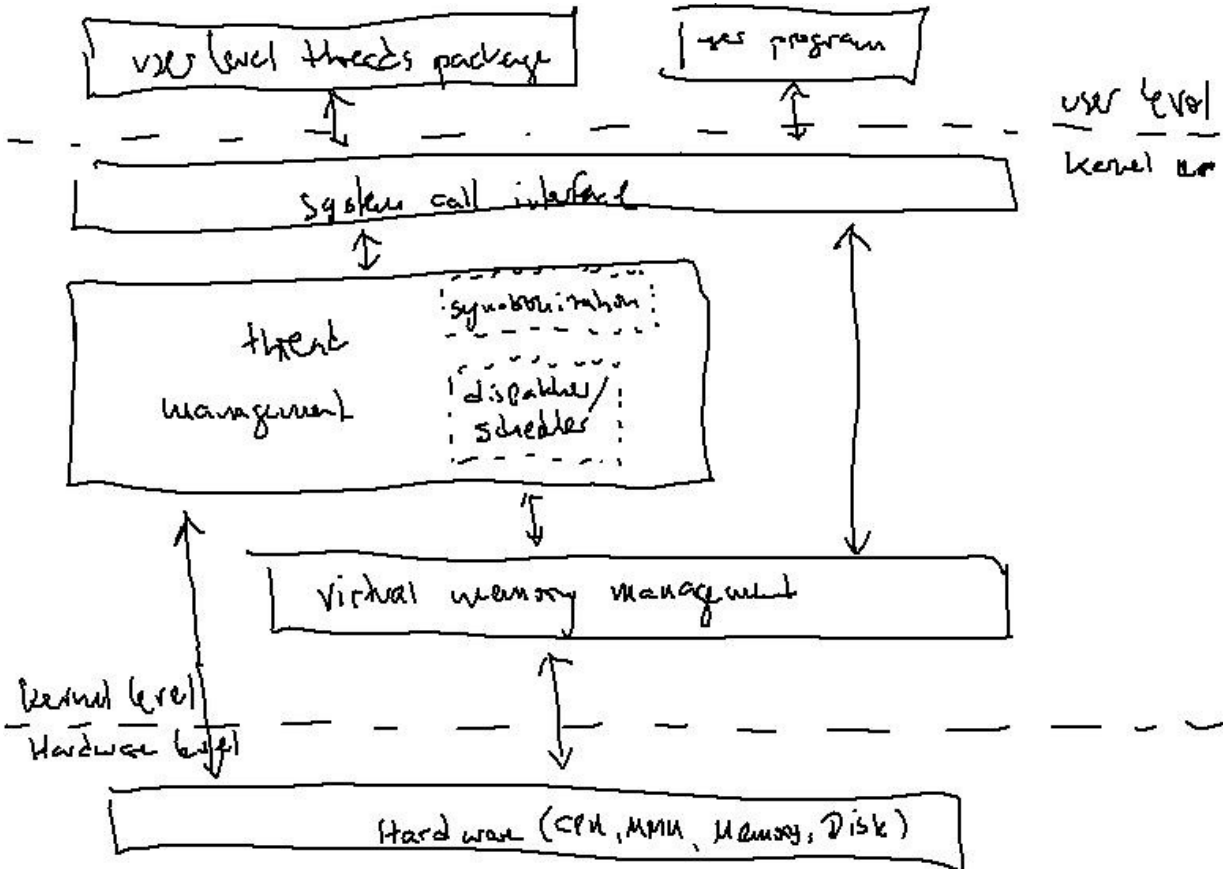
- Paged virtual memory based on a MMU
- Kernel level threads (multiple in one address space possible, simple scheduler)
- Kernel level synchronization (HW-based, semaphores, signals)
- User level threads
- User level synchronization (kernel level based, user level semaphores)
- Multiprocessing (multiple CPUs)

What ULIX should not have ...

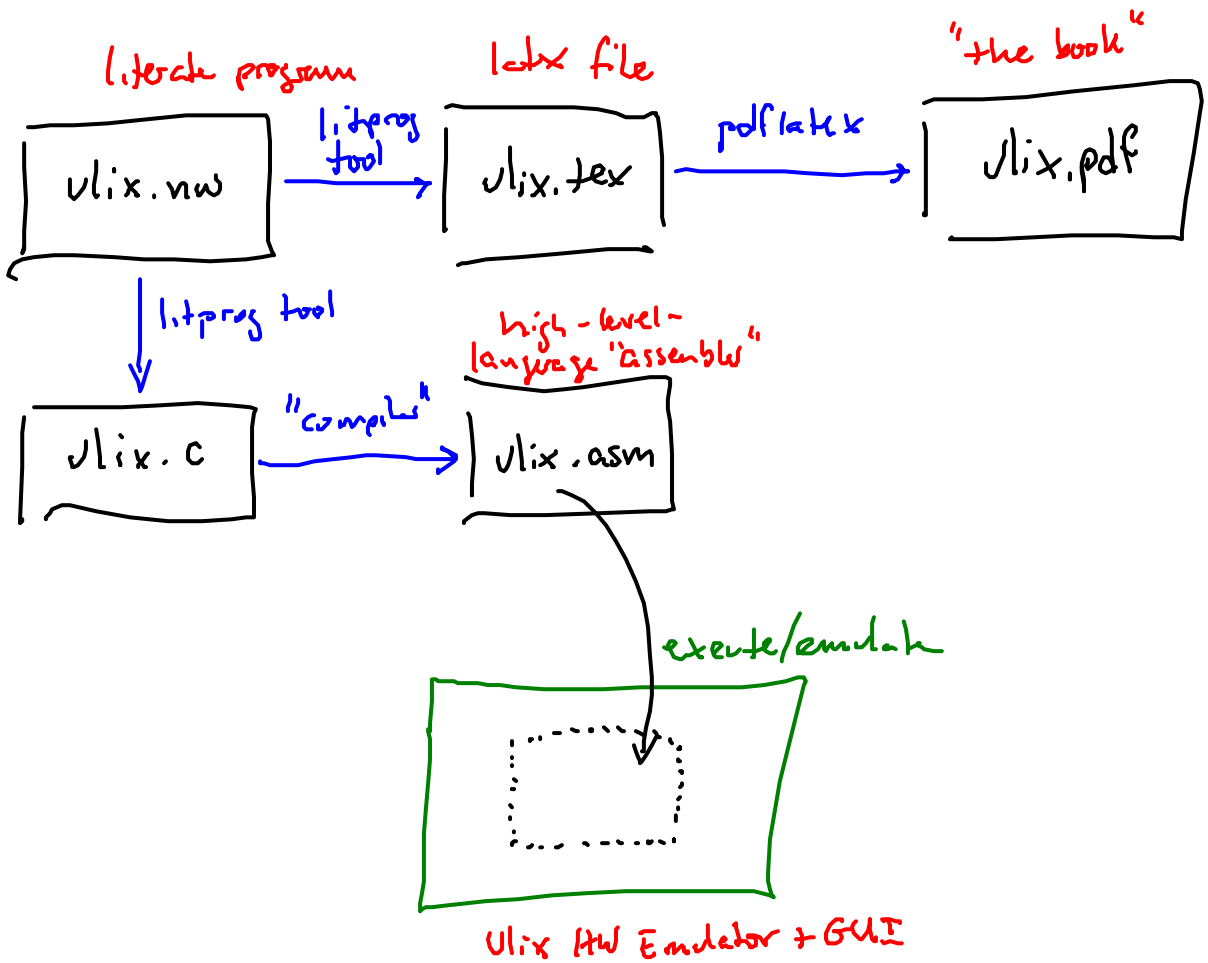
- Segment-based virtual memory
- Input/output stuff (apart from a simple hard disk for swapping)
- File systems
- Most security things (only separation between user/system space and between kernel level threads, no users)

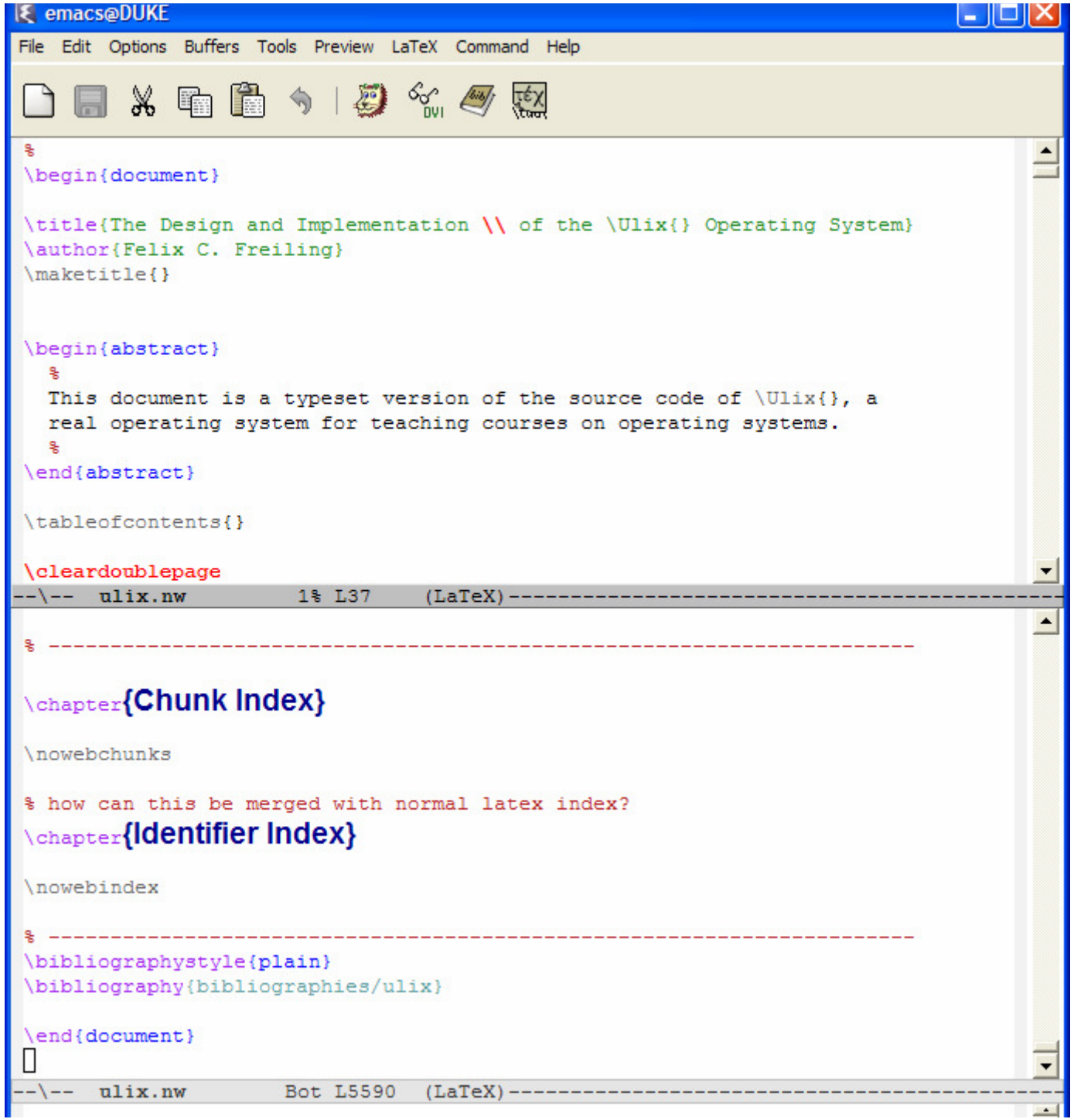
Design of ULIX Kernel

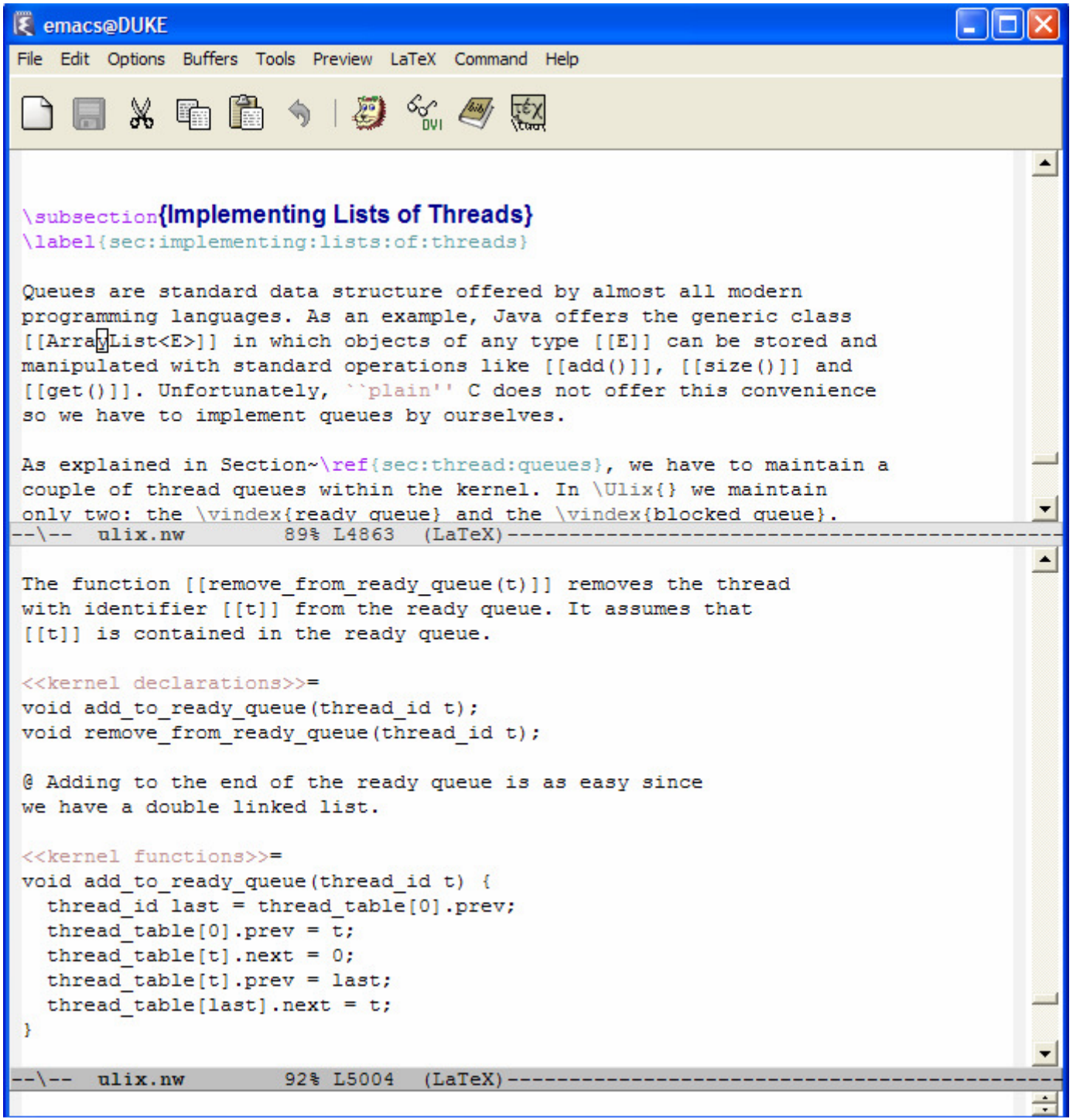
- Services provided in a monolithical kernel



Putting it Together




emacs@DUKE
File Edit Options Buffers Tools Preview LaTeX Command Help
%
\begin{document}
\title{The Design and Implementation \\ of the \Ulix{} Operating System}
\author{Felix C. Freiling}
\maketitle{}
\begin{abstract}
%
This document is a typeset version of the source code of \Ulix{}, a
real operating system for teaching courses on operating systems.
%
\end{abstract}
\tableofcontents{}
\cleardoublepage
--\-- ulix.nw 1% L37 (LaTeX)
% ---
\chapter{Chunk Index}
\nowebchunks
% how can this be merged with normal latex index?
\chapter{Identifier Index}
\nowebindex
% ---
\bibliographystyle{plain}
\bibliography{bibliographies/ulix}
\end{document}
--\-- ulix.nw Bot L5590 (LaTeX)


emacs@DUKE
File Edit Options Buffers Tools Preview LaTeX Command Help
\subsection{Implementing Lists of Threads}
\label{sec:implementing:lists:of:threads}
Queues are standard data structure offered by almost all modern
programming languages. As an example, Java offers the generic class
[[ArrayList<E>]] in which objects of any type [[E]] can be stored and
manipulated with standard operations like [[add()]], [[size()]] and
[[get()]]. Unfortunately, ``plain'' C does not offer this convenience
so we have to implement queues by ourselves.
As explained in Section~\ref{sec:thread:queues}, we have to maintain a
couple of thread queues within the kernel. In \Ulix{} we maintain
only two: the \vindex{ready queue} and the \vindex{blocked queue}.
--\-- ulix.nw 89% L4863 (LaTeX)
The function [[remove_from_ready_queue(t)]] removes the thread
with identifier [[t]] from the ready queue. It assumes that
[[t]] is contained in the ready queue.
<<kernel declarations>>=
void add_to_ready_queue(thread_id t);
void remove_from_ready_queue(thread_id t);
@ Adding to the end of the ready queue is as easy since
we have a double linked list.
<<kernel functions>>=
void add_to_ready_queue(thread_id t) {
thread_id last = thread_table[0].prev;
thread_table[0].prev = t;
thread_table[t].next = 0;
thread_table[t].prev = last;
thread_table[last].next = t;
}
--\-- ulix.nw 92% L5004 (LaTeX)

An example of the semantics of the `prev` and `next` entries in the thread table is shown in Figure 5.3. It shows that the thread identifier 0 is used as an "end marker" for the lists. It also shows that the `prev` entry of the first entry in the queue points to the last element in the queue. In this way, it is easily possible to navigate through the queues in any way which is convenient.

Figure 5.3 also shows a small implementation trick. The thread identifier of the thread itself is always equal to the index of the thread in the thread table. Given a TID of `t`, then `thread_table[t]` is the thread control block of that thread. This also means that the entry `tid` in the thread control block is more or less superfluous.

Now since we are using the value 0 to mark the end of a list, the entry 0 in the thread table has become more or less useless to store thread information. We use it instead as the "anchor" of the ready queue. So to access the first element in the ready queue, we just need to look into:

```
thread_table[0].next
```

The last entry in the ready queue can similarly be accessed using the following expression:

```
thread_table[0].prev
```

The ready queue in the figure contains threads 1, 4, and 7 (in this order).

Recalling the simple state model of threads in Section 5.4.1, every thread is in exactly one state at any time. This means that a thread is either running, ready or blocked. This also means that a thread can be in at most one queue at a time. In case the thread is blocked instead of ready, we can re-use the `prev` and `next` entries in the thread table to implement the blocked list. We only need to have an anchor for this blocked list. This anchor will be a structure similar to the 0-th entry in the thread table.

79a ⟨kernel declarations 38⟩+≡

```
struct blocked_queue {
  thread_id next; // id of the ''next'' thread
  thread_id prev; // id of the ''previous'' thread
};
```

Uses thread_id 74b.

So assume `b` is a variable of type `blocked_queue` representing a blocked list. If both entries in `b` are 0, then the list is empty. If not, then using the thread table we can now find the first, second etc. element by following the `next` pointers. Following the `next` pointers in this way, we can traverse the entire list until we reach an entry in which `next==0`. That's the end of the list. Looking again at Figure 5.3, the blocked queue contains threads 2, 5 and 6 (in this order).

Finally, here's a useful function to initialize a blocked queue. This is just to encapsulate the semantics of "emptiness".

79b ⟨kernel functions 39a⟩+≡

```
void initialize_blocked_queue(blocked_queue* q) {
  q.prev = 0;
  q.next = 0;
}
```

Defines:
  initialize_blocked_queue, never used.

Implementing the Ready Queue

We now provide some convenient functions to add and remove threads from the queues. We start with the ready queue. The function `add_to_ready_queue(t)` adds the thread with identifier `t` to the end of the ready queue. It assumes that the TCB of thread `t` has been set up and initialized already.

The function `remove_from_ready_queue(t)` removes the thread with identifier `t` from the ready queue. It assumes that `t` is contained in the ready queue.

80a ⟨kernel declarations 38⟩+≡

```
void add_to_ready_queue(thread_id t);
void remove_from_ready_queue(thread_id t);
```

Uses add_to_ready_queue 80b, remove_from_ready_queue 80c, and thread_id 74b.

Adding to the end of the ready queue is as easy since we have a double linked list.

80b ⟨kernel functions 39a⟩+≡

```
void add_to_ready_queue(thread_id t) {
  thread_id last = thread_table[0].prev;
  thread_table[0].prev = t;
  thread_table[t].next = 0;
  thread_table[t].prev = last;
  thread_table[last].next = t;
}
```

Defines:
  add_to_ready_queue, used in chunks 80 and 82.
Uses thread_id 74b and thread_table 75c.

Removing is similarly easy.

80c ⟨kernel functions 39a⟩+≡

```
void remove_from_ready_queue(thread_id t) {
  thread_id prev_thread = thread_table[t].prev;
  thread_id next_thread = thread_table[t].next;
  thread_table[prev_thread].next = next_thread;
  thread_table[next_thread].prev = previous_thread;
}
```

Defines:
  remove_from_ready_queue, used in chunks 80 and 82a.
Uses thread_id 74b and thread_table 75c.

We initialize the ready queue to be empty.

80d ⟨initialize kernel global variables 57d⟩+≡

```
thread_table[0].prev = 0;
thread_table[0].next = 0;
```

Uses thread_table 75c.

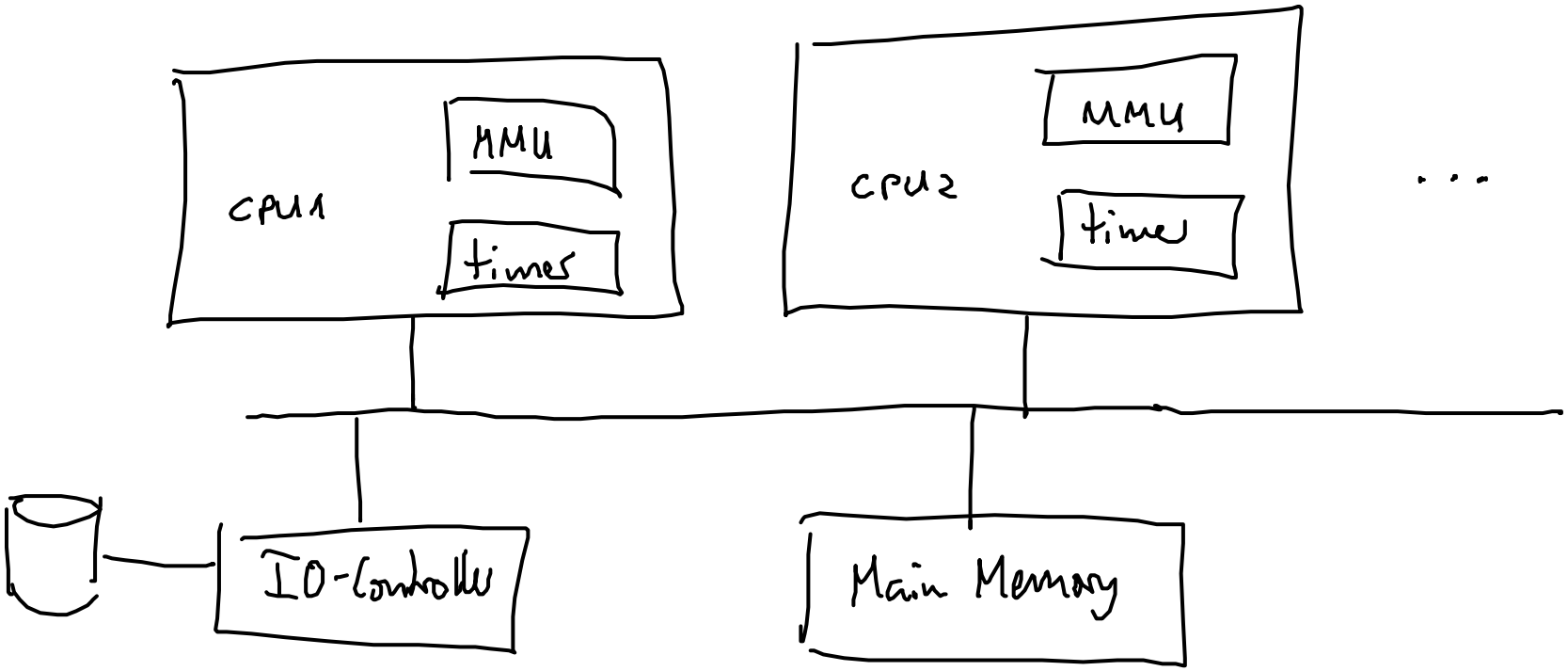
Überblick

- Warum ULIX?
- Literarisches Programmieren
- Grobarchitektur von Laufzeitsystemen
- Entwurf von ULIX
- ULIX-Hardware Überblick

ULIX-Hardware Überblick

Hardware Layout

no caches!



CPU

- Can run in two modes: system and user
- Set of general purpose registers
- Special registers
  - Program counter PC
  - Program status word PSW with
    - MODE bit (system or user)
    - Interrupt masking registers
  - Stack pointers for user and system mode USP, SSP
  - Interrupt Vector Table register IVT
- Usual machine language instructions ...
  - Load, Store, JMP, Branch ...
- Standard instruction cycle

Instruction Set

- JSR: jump to subroutine
  - push return address to current stack
  - JMP to
- RTS: return from subroutine
  - pop PC from current stack
- TRAP: jump to interrupt handler
  - like JSR, but saves to system stack, saves PSW too and switches to system mode
- RTI: return from interrupt handler
  - like RTS, but restores mode from system stack

Main Memory/MMU

- Main memory modeled as a plain array of bytes
- External devices are controlled through memory-mapped IO
- Memory management unit (MMU) implements an address translation
  - Based on tables in main memory
  - MMU has a base register for the page tables
  - MMU has a command register for switching paging on/off

Interrupts

- Use an interrupt vector at well-known place in main memory
- Several interrupt levels ("importance" of interrupts)
- Types of Interrupts:
  - Synchronous interrupts (caused by the processor or MMU):
    - TRAP: Controlled jumps to the OS
    - During command execution (divide by 0, page fault)
  - Asynchronous interrupts (caused by external devices):
    - IO transfer complete, timer expired, etc.

Interrupt Requests

- CPU has in the PSW
  - an interrupt request register IRR
  - an interrupt enable register IER
- Devices (including the processor) wishing to interrupt set IRR to a specified level
- Service interrupt only if IRR is larger than IER
  - Can use this to mask interrupts

Timer Unit

- Can be used to send regular interrupts to CPU
- Programmed through registers:
  - Duration register
  - Start/Stop register

Secondary Storage and DMA

- Simple hard disk
- Basically second type of main memory (just slower)
- No file system, just raw sectors
- Asynchronous IO:
  - Instructions to start a transfer between a buffer in main memory and a sector in secondary storage (DMA)
  - Secondary storage raises interrupt upon completion

Zusammenfassung

- Warum ULIX?
- Literarisches Programmieren
- Grobarchitektur von Laufzeitsystemen
- Entwurf von ULIX
- ULIX-Hardware Überblick

Appendix

References and Resources

References

- Felix Freiling: The design of the ULIX operating system. Unpublished, January 17, 2008 (available upon request)
- Marshall Kirk McKusick, George V. Neville-Neil: The Design and Implementation of the FreeBSD Operating System. Addison-Wesley, 2005. (The FreeBSD Book)
- Andrew S. Tanenbaum, Albert S. Woodhull: Operating Systems: Design and Implementation. Person, 3rd ed., 2006. (The MINIX Book)
- Jürgen Nehmer, Peter Sturm: Systemsoftware: Grundlagen moderner Betriebssysteme. dPunkt, 1998.
- Donald Knuth: TeX: The Program. Addison-Wesley, 1986.
- Donald Knuth: Literate Programming. In: The Computer Journal, 27(2):97-111, 1984.
- Norman Ramsey: Literate Programming Simplified. IEEE Software, 11(5):97-105, September 1994.
- Andrew L. Johnson and Brad C. Johnson. Literate programming using noweb. Linux Journal, 64-69, October 1997

Resources


Literate Programming - Mozilla Firefox
Datei Bearbeiten Ansicht Chronik Lesezeichen Extras Hilfe
http://www.literateprogramming.com/
literate programming
literateprogramming.com
Quotes
Literate Programming
Software Documentation
Design Documentation
Agile Documentation
Source Code Comments
Software Aging
Quotes - Printer Friendly
Category
CWEB
Articles
Tools
Links
Community
Feedback
Literate Programming
Donald Knuth. "Literate Programming (1984)" in Literate Programming. CSLI, 1992, pg. 99.
I believe that the time is ripe for significantly better documentation of programs, and that we can best achieve this by considering programs to be works of literature. Hence, my title: "Literate Programming."
Let us change our traditional attitude to the construction of programs: Instead of imagining that our main task is to instruct a computer what to do, let us concentrate rather on explaining to human beings what we want a computer to do.
The practitioner of literate programming can be regarded as an essayist, whose main concern is with exposition and excellence of style. Such an author, with thesaurus in hand, chooses the names of variables carefully and explains what each variable means. He or she strives for a program that is comprehensible because its concepts have been introduced in an order that is best for human understanding, using a mixture of formal and informal methods that reinforce each other.
Fertig

Noweb Resources and Tips

- Noweb home page: http://www.eecs.harvard.edu/nr/noweb/
- Convenient installation needs Icon programming language: http://www.cs.arizona.edu/icon/
- For Windows - install using Cygwin: http://www.cygwin.com/