HELSINKI UNIVERSITY OF TECHNOLOGY
Department of Computer Science and Engineering
Laboratory of Computer and Information Science

Antti Rasinen

Analysis of Linux evolution using aligned source code segments

Master's Thesis submitted in partial fulfillment of the requirements for the
degree of Master of Science in Technology.

Espoo, May 3, 2006

Supervisor: Academy Professor Heikki Mannila

Instructor: Chief Research Scientist Jaakko Hollmén, D. Sc. (Tech.)

HELSINKI UNIVERSITY
OF TECHNOLOGY

ABSTRACT OF THE
MASTER'S THESIS

Author: Antti Rasinen	
Name of the thesis: Analysis of Linux evolution using aligned source code segments	
Date: May 3, 2006	Number of pages: 70
Department: Computer Science and Engineering	Professorship: T-122
Supervisor: Academy Professor Heikki Mannila	
Instructor: Chief Research Scientist Jaakko Hollmén, D. Sc. (Tech.)	
<p>In this thesis, we have examined the structure and evolution of the Linux kernel. We considered the source code of the kernel as ordinary text without any regard to its semantics. After selecting three functionally central modules to study, we performed an initial study of the code segments that have remained the same between versions. We used the results of that study to build a more detailed data set of these aligned segments, or alignments for short. To study this data set, we used different visualisation methods, scatter plots, Sammon's projection, and linear and branching change graphs. We also studied the raw alignments directly to support or disprove hypotheses gained from the visualisations. From these we could detect the locations of important changes both in time and files. Furthermore, we classified the kinds of changes the files have experienced. The methods used here may prove useful in studying large collections of legacy code, whose original maintainers are not available.</p>	
Keywords: data analysis, string alignment, Linux, evolution	

Tekijä:	Antti Rasinen	
Työn nimi:	Linuxin kehityksen analysointi käyttäen lähdekoodin vastaavuuksia	
Päivämäärä:	3. toukokuuta 2006	Sivuja: 70
Osasto:	Tietotekniikan osasto	Professuuri: Tik-122
Työn valvoja:	Akatemiprofessori Heikki Mannila	
Työn ohjaaja:	Johtava tutkija Jaakko Hollmén, TkT	
<p>Tässä työssä tutkittiin Linux-käyttöjärjestelmäytimen rakennetta ja muutoksia sen kehityksen aikana. Lähestyimme ytimen lähdekoodia tavallisen tekstidokumentin tapaan, kiinnittämättä huomiota käytetyn ohjelmointikielen semantiikkaan. Valitsimme tutkimuskohteiksemme kolme tärkeää moduulia, joille suoritimme alustavan analyysin pystyäksemme paremmin tunnistamaan ytimen eri versioiden muuttuneet ja yhteiset osat. Tämän tutkimuksen tulosten perusteella teimme kattavan kartan eri versioiden yhteisistä segmenteistä. Jatkoanalyysissä käytimme erilaisia visualisointimenetelmiä, kuten Sammonin projektiota ja tiedosto- ja segmenttikohtaisia muutostarkkuuksia. Tuimme näistä tehtyjä päätöksiä tutkimalla segmenttejä myös suoraan. Pystyimme selvittämään usean mielenkiintoisen muutostarkkuuden sijainnin sekä ajallisesti, että tiedoston tarkkuudella. Lisäksi pystyimme luokittelemaan tiedostojen kokemat muutokset. Tässä työssä esitettyjä menetelmiä voidaan mahdollisesti käyttää laajojen ja vanhojen ohjelmistojen ymmärtämiseen, joiden alkuperäiset kehittäjät tai ylläpitäjät eivät ole käytettävissä.</p>		
Avainsanat: data-analyysi, merkkijonovastaavuudet, Linux, evoluutio		

Acknowledgements

This Master's thesis was carried out in the Laboratory of Computer and Information Science at the Helsinki University of Technology.

I would like to thank Masters of Science Rasinen and Tatti for their assistance during the first few days of writing the thesis, especially for their help with L^AT_EX. I would also like to thank Heikki Vuolteenaho for the many stimulating lunch-hour discussions, which provided a much-welcomed diversion.

I am greatly indebted to my instructor Jaakko Hollmén, whose guidance and experience removed many of the wrinkles from my brow while making this thesis.

Finally, I would like to thank my family for their selfless work during my childhood, which has brought me to this point in my life.

Most importantly I wish to express my gratitude and love for Anni for her support, motivation¹, and love, that helped me through the most pessimistic periods.

Otaniemi, April 13th, 2006.

Antti Rasinen

¹Do a Google Image search for “Hainan Beach”.

Contents

Symbols and abbreviations	vii
1 Introduction	1
1.1 Structure of the thesis	2
2 Finding aligned segments in source code	3
2.1 Global alignment	4
2.2 Local alignment	8
2.3 Multiple sequence alignment algorithms	10
2.4 Choosing the most suitable algorithm for this work	11
3 Linux source code as data	14
3.1 Introduction to the Linux operating system	14
3.1.1 Versions	16
3.2 Overview of the kernel structure	17
3.3 File comparison graphs	22
3.3.1 File comparison graph reduction	22
3.3.2 Improving results by removing <code>#include</code> -directives	28
4 Analysis of alignment data	29
4.1 Distribution of the length of the aligned blocks	29
4.2 Power laws	32
4.2.1 Our attempt to model the power law behaviour	32
4.3 Visualising the modules with Sammon’s projection	34

4.4	Mapping the relationships between files within a module . . .	37
4.5	Visualising the differences in chains of files	42
4.5.1	Best matching chains	43
4.5.2	Visualisation methods	44
4.5.3	Types of change between files	45
4.6	Overall analysis of changes	51
4.6.1	Analysis of module <code>ipc</code>	51
4.6.2	Analysis of module <code>kernel</code>	51
4.6.3	Analysis of module <code>mm</code>	52
5	Summary and conclusions	54
5.1	Future work	55
5.2	Ethical considerations	55
A	Examples of found alignments	56
	Bibliography	67

Symbols and abbreviations

$A(f_1, f_2)$	The set of alignments between files f_1 and f_2 .
d	A distance function between symbols of an alphabet.
$dist(\cdot, \cdot)$	A distance function between two strings.
ϵ	The empty string.
E	Cost function for multidimensional scaling.
F	A score array used in dynamic programming.
f_1, f_2	A pair of files that share an alignment
k	Scaling coefficient for power law distributions.
$\max\{a_1, a_2, \dots, a_n\}$	The largest element in the set $\{a_1, a_2, \dots, a_n\}$
$O(f(n))$	An asymptotic upper bound for the function f . Used to approximate the required resources given input of size n . For example, $O(n)$ means linear growth and $O(n^2)$ quadratical growth.
$P(X = x)$	The probability of random variable X having the value x .
<i>score</i>	A score function for an alignment.

<i>version/module</i>	A reference to a specific module in Linux source code. <i>Version</i> refers to the version number (one of 1.0, 1.2.0, 2.0.1, 2.2.0, 2.4.0 or 2.6.0) and <i>module</i> refers to either <code>ipc-</code> , <code>kernel-</code> or <code>mm-</code> module.
<i>version/module/filename</i>	A reference to a specific file in Linux source code. The prefix is as above and <i>filename</i> specifies the file.
$\zeta(k)$	The Riemann zeta-function.
I/O	Input/Output
IPC	Interprocess communication
MDS	Multidimensional scaling

Chapter 1

Introduction

BANG! From the Big Bang at the start of the universe to the modern Internet, our history is full of events that have radically changed our lives and our culture. Our success as a species is due to our overcoming the difficulties presented by each information explosion: spoken language, written language, printing press¹ and cheap paper, mass communication, computers, and the Internet. We have come far during the millenia.

Today the pace is still accelerating. It is possible to fill a memory card in a digital camera in a matter of minutes, even though the card has a larger capacity than the hard drives used in personal computers ten years ago. This is an example of the situation where the digital records are very large in terms of storage, megabytes per picture.

Given enough transactions, even small individual records can generate large amounts of data. Visitors at a web site generate only one line in the logfiles, but with a popular site the logs can grow very fast. This has given rise to the practice of data mining, in which vast amounts of data are analysed and distilled into a more usable form, into knowledge. A good introduction to the topic is [13].

The problem we are studying is between these two extremes. The Linux operating system is being developed by hundreds of volunteers, most of who

¹Usually attributed to Gutenberg, the printing press was first invented in China by Bi Shēng in the 11th century.

contribute far more than just one line to the source code. The end result is a textual database of over 200 megabytes², whose size has doubled with the release of each new version.

While this distributed approach is seen as the strength of Linux and the most important reason for its popularity, there are also downsides. There are only a few people in the world who can confidently say they know the the entire kernel, and none of them know it line-by-line. When we also consider the past versions of the kernel, reconstructing the history of Linux requires a great deal of work and study.

In this work we attempt to lessen the work needed. We apply algorithms usually used in genetics and language research to the Linux source code. From this we build visualisations that help us to see “Linux at a glance”.

Our approach was inspired by the synteny maps used to study similarities between different species. See [6] for an introduction.

1.1 Structure of the thesis

In Chapter 2 we present the global and local alignment algorithms with some discussion to their benefits and drawbacks. Next, in Chapter 3 we describe our source material, the Linux kernel. We discuss its history, its structure and our initial work in analysing it.

In Chapter 4 we use the alignment data set to visualise the history of Linux. We also identify and describe the most important changes on a general level.

²July 2005

Chapter 2

Finding aligned segments in source code

Our aim in this work is to study the development of the Linux kernel. To do this, we must be able to recognise which parts of the kernel have changed with time and which remain the same. We solve this problem with alignment algorithms.

Alignment, or string matching, is a problem which surfaces in many areas such as pattern recognition [9] and genetic analysis [1]. In a nutshell, we wish to know differences or similarities between two strings. Most Unix-users have encountered the program `grep`, which applies exact string matching to extract those lines containing the specified string from the input stream¹.

Although exact matches are of limited usefulness to us in this work, we prefer a method that is not sensitive to small differences. For purposes of code compilation, whitespace² is completely irrelevant—yet a simple addition of a line will result in a broken match. Many other textual changes are likewise “neutral”, for example a variable name substitution from `i` to `j`.

Using heuristics or complex lexical analysis is not a very appealing solution. In-

¹To be precise, `grep` allows for *regular expressions*; it is possible to search for more than one string at a time. Nevertheless, this is different from the “fuzzy” methods discussed in this chapter.

²Whitespace means anything that is displayed as “nothing”: spaces, tabs, and new lines.

stead we turn to string alignment algorithms, namely Needleman-Wunsch [22] and Smith-Waterman [26] algorithms, which have been widely applied to genetic sequences. We adopt the terms used in those applications.

We will first examine the two basic algorithms used in alignment and then discuss their strengths and weaknesses. A more comprehensive review with more algorithms is available in [10] and in [9]. There is also a large Web collection of string matching algorithms in [7].

2.1 Global alignment

There is a strong connection between some string similarity measures and global alignment algorithms. Similarity measures that use dynamic programming [5] usually rely on a two-dimensional table to contain solutions to subproblems solved earlier. This table can be reused to highlight the differences between the two strings in linear time.

The basic step of dynamic programming is to split the problem into subproblems. We can then recursively solve these subproblems with dynamic programming and combine the results to obtain the optimal result. In our case, we must assign a score to a pair of strings. Let us use the strings *Antti* and *Anni* as an example.

The match quality is indicated by a characterwise³ score function d . Given a problem, such as $\text{dist}(\text{Antti}, \text{Anni})$, we can identify the subproblems:

- $\text{dist}(\text{Antt}, \text{Ann}) + d(\text{i}, \text{i}),$
- $\text{dist}(\text{Antti}, \text{Ann}) + d(\epsilon, \text{i})$ and
- $\text{dist}(\text{Antt}, \text{Anni}) + d(\text{i}, \epsilon),$

where ϵ denotes the empty string and is used as a shorthand notation for insertions and deletions. The optimal solution is the one, which yields the

³It is also possible to interpret the score function as a special case of the string matching function. However, most actual implementations use a separate score function. We repeat that practice here.

highest score; it depends on both the characters, both of which are *i* in this example, and on the optimal solutions of the subproblems solved at an earlier stage.

The cases listed above are known as substitution, insertion, and deletion. The latter two are interchangeable, depending on which input string one considers. Computing their score requires that we also know the value of $\text{dist}(\text{Antt}, \text{Ann})$, which is a subproblem for both insertion and deletion cases. It is inefficient to compute this value more than once.

We can compute the values with a bottom-up strategy, in which we first calculate the simplest subproblems and store their value in an array F . An element F_{ij} of the table contains the answer to the (sub)problem of computing $\text{score}(\text{str1}_{1..i}, \text{str2}_{1..j})$. Using this approach, we need only compute the value of each subproblem once. The table contains the matching score as the last computed value.

The approach presented above describes the Needleman-Wunsch algorithm on a general level. The algorithm was published in [22] and is also known as the Sellers algorithm. The algorithm is listed in Listing 1.

Listing 1 Computation of the Needleman-Wunsch score table F .

Input: A distance function d and strings str1 , str2 .

```

 $m = |\text{str1}|$ ;  $n = |\text{str2}|$ 
for  $i = 0$  to  $m$  do //Initialise the first column
     $F_{i,0} = 0$ 
for  $j = 0$  to  $n$  do //Initialise the first row
     $F_{0,j} = 0$ 
for  $i = 1$  to  $m$  do
    for  $j = 1$  to  $n$  do
         $S_s = F_{i-1,j-1} + d(\text{str1}[i-1], \text{str2}[j-1])$  //Substitution score
         $S_i = F_{i-1,j} + d(\text{str1}[i-1], \epsilon)$  //Insertion score
         $S_d = F_{i,j-1} + d(\epsilon, \text{str2}[j-1])$  //Deletion score
         $F_{ij} = \max\{S_s, S_i, S_d\}$ 
return  $F$ 

```

After filling the table F , it is possible to calculate how the strings align, or in other words how the individual characters or symbols line up. This is achieved by backtracking from the element F_{mn} , where m and n are the lengths of the

aligned strings. For each of the three operations listed above, we can check which of them are consistent with F_{mn} , i.e. which operation gives the same value as F_{mn} .

To find the correct alignment, we know that there are at three possibilities to arrive at a given element in the table and we know what operations those three possibilities represent: substitution, insertion and deletion. We can thus move step by step toward the element F_{00} and build the alignment at each step. We output the alignment as a string with gaps denoted by dashes. The `find_path` algorithm is described in Listing 2.

Listing 2 Computation of the optimal alignment with the score table F using the Needleman-Wunsch algorithm.

Input: A distance function d , strings $str1, str2$ and a score table F

```

    outstr1 =  $\epsilon$ ; outstr2 =  $\epsilon$ 
    while  $i > 0$  and  $j > 0$  do
         $S = F_{ij}$ 
         $S_s, S_i, S_d = \text{scores}(F, i, j)$  //Compute scores for substitution, insertion
        and deletion.
        if  $S = S_s$  then //Substitution
            outstr1 =  $str1[i - 1] + outstr1$ 
            outstr2 =  $str2[j - 1] + outstr2$ 
             $i = i - 1; j = j - 1$ 
        else if  $S = S_i$  then //Insertion
            outstr1 =  $str1[i - 1] + outstr1$ 
            outstr2 = " - " + outstr2
             $i = i - 1$ 
        else
            outstr1 = " - " + outstr1
            outstr2 =  $str2[j - 1] + outstr2$ 
             $j = j - 1$ 
    if  $i == 0$  then
        Add rest of  $str2$  to  $outstr2$  and prepend  $outstr1$  with equal amount of
        dashes.
    else
        Add rest of  $str1$  to  $outstr1$  and prepend  $outstr2$  with equal amount of
        dashes.
    return outstr1, outstr2

```

It is easy to estimate the computational complexity and the memory require-

ments for the algorithm. We denote the lengths of the input strings by m and n . Filling the table is the most computationally challenging operation; it has the time complexity $O(mn)$ whereas `find_path` has only linear complexity $O(\max\{m, n\})$. The memory complexity is also $O(mn)$, as the table is the only significant data structure used.

The most important assumption behind global alignment is that strings are compared in order from the first symbol to the last. The strings are assumed to be similar in structure, containing roughly the same elements in the same order.

In the following demonstration, the sentences are from the United Nations' Declaration of Human Rights in two closely related Norwegian dialects: Nynorsk and Bokmål, respectively. The alignment is over the whole sentence, but it is split here in two for easier presentation.

```

Alle menneske- er fødde til fridom og med
Alle mennesker er fød--t-- fri--e og med

sa-me menneskeverd og menneskere-t----tar.
samme menneskeverd og menneskerettigheter.

```

It is very easy to see the differences between the two dialects. The gaps pinpoint missing or added items; mismatches show differences caused by substitutions. The result also highlights the importance of the score matrix. Different sets of parameters will naturally yield different results. We will use a rather simple set of parameters. A match is a substitution between the same symbols; a mismatch is a substitution, in which the symbols differ. Insertions and deletions are symmetrical and we will call them both gaps. In this example a match gave a score of 2 while mismatches and gaps had a score of -1 . A linguist would surely point out that the words *fødde* and *født* should align better than they do in the example. Increasing the score for a whitespace-whitespace match could possibly suffice.

But global alignments have their limitations. It is possible for a global alignment to become confused when the structures of the two strings differ. A

simple swap will force the algorithm described here, Needleman-Wunsch, to ignore one of the swapped segments.

As an example, consider the strings “Charles Darwin” and “Darwin, Charles”. A human observer is not distracted by the rearrangement. However, observe the results of a global alignment with the same parameters as above.

```

-----Charles Darwin
Darwin, Charles-----

```

The algorithm has sacrificed information about Charles’ surname in order to have an acceptable match.

2.2 Local alignment

Local alignments, as the name implies, isolate similar segments from the input strings. As the local alignment algorithms can choose where to start and end the alignments in both sequences, they are well suited to identifying common elements from two different sources.

In spite of this difference, the two families of algorithms are very closely related. The local alignment algorithm used in this work, the Smith-Waterman algorithm, is a modification of the Needleman-Wunsch algorithm presented in the previous section. The algorithms are nearly identical and their complexity is the same both in time and memory. The Smith-Waterman algorithm was first introduced in [26].

In the Smith-Waterman algorithm the score table F is initialised with almost the same code as in Needleman-Wunsch. The only difference occurs on line 11, where a “cutoff” mechanism is introduced. Instead of $F_{ij} = \max\{S_s, S_i, S_d\}$, we have $F_{ij} = \max\{0, S_s, S_i, S_d\}$. The rationale behind the addition of the cutoff-constant⁴ is that we can stop constructing our alignment when we meet a 0 in the table F .

The difference between the algorithms is more pronounced when we look at finding the best alignment. Instead of walking through the entire table from

⁴The constant need not be 0, but it is a very common choice.

the bottom-right corner to the top-left, we must first identify the maximal element in the table. We can use it as the starting point⁵ and then proceed as with Needleman-Wunsch. The stop-condition must also be modified to account for the possibility of encountering the cutoff value. The `find_path` for Smith-Waterman is presented in Listing 3.

Listing 3 Computation of the optimal alignment with the score table F using the Smith-Waterman algorithm.

```
Input: Strings str1, str2 and score table  $F$  (also distance function  $d$ )
  outstr1 =  $\epsilon$ ; outstr2 =  $\epsilon$ 
   $i, j = \operatorname{argmax}\{F_{ab}\}$ 
   $i_{\max} = i$ ;  $j_{\max} = j$ 
  while  $i > 0$  and  $j > 0$  and  $F_{ij} \neq 0$  do
     $S = F_{ij}$ 
     $S_s, S_i, S_d = \text{SCORES}(F, i, j)$  //Compute scores for substitution, inser-
    tion and deletion.
    if  $S = S_s$  then //Substitution
      outstr1 = str1[ $i - 1$ ] + outstr1
      outstr2 = str2[ $j - 1$ ] + outstr2
       $i = i - 1$ ;  $j = j - 1$ 
    else if  $S = S_i$  then //Insertion
      outstr1 = str1[ $i - 1$ ] + outstr1
      outstr2 = " - " + outstr2
       $i = i - 1$ 
    else
      outstr1 = " - " + outstr1
      outstr2 = str2[ $j - 1$ ] + outstr2
       $j = j - 1$ 
  //For multiple alignments:
  PARTITION( $F, i, i_{\max}, j, j_{\max}$ ) //See Listing 4.
  return outstr1, outstr2
```

The time and memory complexities are practically identical to the Needleman-Wunsch algorithm.

Let us return to the example discussed above. Using the same parameters and input data, we can find the best matching alignment. It should not come as a

⁵Starting from the maximal element is not necessarily the best choice in all circumstances. For example, to a human observer, some alignments might continue beyond the maximal element. However, most of these tails can be found later.

surprise that the best match is the same the global alignment algorithm found, i.e., “Charles”. But unlike global matching, we have not made any compromises.

The next best matching alignment is the string “Darwin”, and the third best match is the whitespace found in both input strings. Only the comma in “Darwin, Charles” was not paired with a segment from the other string.

Smith-Waterman algorithm also inherits the error resistance of Needleman-Wunsch. For example, had our input strings been “Chrles Darwin” and “Durwin, Charles”, we would end up with alignments

```
Ch-rles/Darwin/space
Charles/Durwin/space,
```

where / separates the alignments. Too large a difference, though, will result in the alignment being split into two parts—in the same situation global alignment would report either mismatches, gaps or insertions.

2.3 Multiple sequence alignment algorithms

There are also algorithms that can align more than two sequences simultaneously. They can be divided into three groups: hyperlattice alignments, star alignments and tree alignments. We will give only a brief introduction to these methods; a more complete discussion is in Chapters 6 and 7 of [10] and in [2].

Hyperlattice methods are a simple extension of the pairwise alignment algorithms explained above. Instead of using a two-dimensional array, we must use an n -dimensional array. This also renders the methods unusable except for very short sequences: the space complexity for the array is $O(n^k)$ for k sequences. The time complexity grows even faster as $O(2^k n^k)$.

Tree-based methods, such as the widely-used Clustal (see [8]), first try to determine a *phylogenetic tree* or guide tree. In such a tree, the sequences that are more closely related to each other are also close to each other in the tree. The distance between the sequences must be computed with some distance function; Clustal uses the number of exact matches divided by the sequence length.

The tree is then used to build the alignment. Consider the sequences A , B and C , with the shortest distance between A and B . Then Clustal first aligns A with B . The resulting alignment “ AB ” is then aligned with C to form the final alignment. Tree-based methods are useful in determining conserved regions and evolutionary relationships across species.

Star-based methods are used in biology to align regions of similar function, which closely mirrors our goal in this work. In star alignment one of the input strings is chosen as the center. Next, every other sequence is aligned with the center sequence. The alignments are then aggregated one at a time. If an alignment has a gap, then the resulting alignment will also have a gap at that point. Note that an existing gap may also be lengthened. Gaps are never removed during the aggregation process.

Although multiple sequence alignments are appealing, we must remember that our “species” are the versions of Linux at different times. The versions form a series. This is in contrast with biological evolution, in which several sequences evolve in parallel.

Instead of asking how the six versions of Linux studied in this work are related to each other, we are more interested in finding out how the *consecutive* versions are related. Pairwise algorithms suffice to answer this question. We already know the structure of the species graph and it is neither a tree or a star.

2.4 Choosing the most suitable algorithm for this work

In light of the above discussion, we have chosen to use (pairwise) local alignment and specifically the Smith-Waterman algorithm to analyse the Linux source code. A global alignment is at its core a similarity measure; it does not yield information about the structure of the changes.

The ordinary version of the Smith-Waterman -algorithm only computes the alignment with the largest score. In this work we modified the algorithm to

recompute the weights in order to obtain the next largest score and so on. To ensure meaningful alignments that do not overlap, we must also mark the visited rows and columns of the score matrix as unusable. The resulting algorithm is presented in Listing 4.

Listing 4 Partition of the score table F for recomputation.

Input: The score table F , a collection of subtables S and the coordinates for the latest found alignment $i_{min}, i_{max}, j_{min}, j_{max}$.
 // S can be thought of being a part of F which is why we have omitted it in `find_path`.
 // Clear the left and bottom borders of the removed area.
for all i **do**
 $F_{i, j_{max}} = 0$
for all j **do**
 $F_{i_{max}, j} = 0$
for all $s \in S$ such that s contains a column between i_{min} and i_{max} or contains a row between j_{min} and j_{max} **do**
 Remove s from S // s can be divided into at most four parts, $p_1, \dots, p_4$
 for all p_n such that $\max(p_n) > \text{cutoff}$ **do**
 Recompute the scores for p_n with Smith-Waterman algorithm.
 Add p_n to S

The memory requirements for the modified algorithm do not change except for the small overhead required to keep track of the subtables. Estimating the time complexity is easier, if we assume the lengths of both input sequences to be equal. We denote this length by N .

The upper limit for the number of operations is obtained when only one row and column are removed from the table. We can then calculate the number of required operations as

$$N_{ops} = \sum_{k=1}^N k^2 = \frac{n(n+1)(2n+1)}{6}.$$

Thus the worst case scenario gives us the time complexity $O(N^3)$.

In practice this estimate is too pessimistic. For example, we are not interested in very small alignments; a single character is simply not descriptive enough when trying to compare C language source code. We can use this to ignore

any subtables, which contain scores below our minimum limit. In this work we used a minimum limit of 15 with a score 3 for a match. This requires that any alignment must be at least five characters long, which excludes the most common variable names and reserved words.

Chapter 3

Linux source code as data

The algorithms used in the previous chapter are applicable to any data that can be represented as a string. Their most common use is in genetics, where they and their derivatives are used to identify common subsequences. In this work we applied the algorithm to the source code of the Linux operating system with a goal of recognising related segments in different versions. As Linux has been in development for over a decade, the files have experienced a large number of rewrites and reorganisations.

To construct the alignments, we had to first preprocess the data. We chose three modules which we felt best captured the development of the most crucial parts of the Linux kernel. We performed simple analysis to reduce the computational work load to a manageable level and then computed the alignments.

3.1 Introduction to the Linux operating system

Linux is an open-source operating system developed by volunteering participants over the world. It is one of the major successes of the so called free or open software movement¹.

¹These two movements are actually distinct, but there is significant overlap. Both camps seem to lay equal claim to the success of Linux. The reader is advised to study [27] and [23] for further information on their subtle differences.

Linux began as a hobby project for a Finnish computer science student Linus Torvalds. His aim was to create a free clone of the expensive commercial Unix variants that would run on a PC. He released the early versions in late 1991 under the GNU General Public License, which helped to attract other developers to improve the kernel. The early development is partially documented in the messages posted to the newsgroup `comp.os.minix`.

The project took a few years to mature and Linux 1.0 was released in March 1994. This is the first version we will be using in this work. Another significant moment in the history of Linux was the release of version 2.0 in June 1996, after which Linux started to gain popularity quickly.

Linux is a modern operating system with practically every feature expected of a desktop computer operating system. It supports multitasking, multiple users, and multiple processors. The memory management system supports virtual memory, paging, and protected mode. A very detailed look at Linux can be found in Linus Torvalds' Master's Thesis [30] or in [4] and [28].

In addition to these features, Linux is also highly scalable. Although it was initially developed for a desktop computer, it can now run on a variety of platforms starting from PDAs and mobile phones up to large multicomputer server installations. This is due to a high level of architecture independence, which allows the kernel to be ported easily to new hardware².

The kernel is written in the C programming language with some architecture-dependent parts written in assembler language. Our work will only examine source code from the C source files. We expect the reader to know some of the basic concepts of the language, such as header files, reserved words, functions *et cetera*. We do not go into details of the language here. The reader is advised to study [17] if necessary.

²Portability was not a design target for the early versions. In one of the first newsgroup messages (see [29]), Linus Torvalds writes (*sic*) "It is NOT portable (uses 386 task switching etc), and it probably never will support anything other than AT-harddisks."

3.1.1 Versions

Since its inception in 1991 Linux has used three different version number schemas. Before version 1.1.0, the versions were numbered somewhat informally: 0.01, 0.11, 0.12, 0.95, 0.95a etc. until version 1.0.

Since 1.1.0 Linux version numbers had a third component. The second and third numbers are known as major and minor version numbers, respectively. Furthermore, the development versions had an odd major number and stable releases had an even number. This system was used between years 1994–2005. Release dates for versions 1.2.0–2.6.10 are shown in Figure 3.1.

Figure 3.1: Release dates of Linux kernels. Kernels with an odd minor number are unstable development versions (marked with plus signs), which are then frozen into stable versions (marked with dots). Note that stable versions have much longer lifespans compared to the development versions: in February 2004 all four stable 2.x kernels were updated.

Since 2.6.10 in March 2005, a fourth number is added for bugfixes that are too

small to warrant a minor number change. For example, 2.6.11 was followed by 2.6.11.1. The bugfix-number 0, though, is omitted. The next minor release was thus called 2.6.12, followed by 2.6.12.1.

For this work we have chosen to include the first (or nearly the first) version of each stable release. In terms of functionality, these versions are sufficiently different from each other. The versions selected are 1.0, 1.2.0, 2.0.1, 2.2.0, 2.4.0 and 2.6.0.

3.2 Overview of the kernel structure

The most recent version at the time of writing, July 2005, was 2.6.12.5. We will use this version to illustrate the structure of the Linux kernel.

The root directory for the source code tree contains 16 subdirectories. These are listed in Table 3.1. Although there has not been much change in the basic organisation of kernel source code, we have summarised the changes in Table 3.2. The most notable periods of change have been between versions 1.0 and 1.2.0, and 2.4.0 and 2.6.0³. As a general rule, these subdirectories contain machine-independent routines that take advantage of the architecture-specific code in the subdirectory `arch`.

The total size for this kernel is 228 megabytes. Each release has increased the size exponentially as shown by Figure 3.2.

The distribution of the directory sizes is rather surprising. The pie-chart in Figure 3.3 illustrates this. The largest directory, `drivers`, has a size of 99 megabytes, which corresponds to 43% of the total size. Next four largest directories, `arch`, `include`, `fs`, and `sound` account for 106 megabytes.

In comparison, the size of Linux 2.4.0 is only 100 megabytes. The `drivers`-directory is once again the largest subdirectory with 54 megabytes. The larger relative size (54%) is explained by the fact that the sound-subsystem was moved out of `drivers` to its own subdirectory in 2.6. Another indication of the continuing growth of Linux is the fact that between versions 2.4.0 and

³As this work neared completion in February 2006, the subdirectory `block` was added. This subdirectory contains drivers for block-based I/O devices.

Table 3.1: The subdirectories of Linux 2.6.12.5. The second column shows the earliest version that contains the directory. This data is from the source code, its documentation and (for version 2.0) [24]. We are limiting ourselves to those versions examined in this work. The version can be seen as the first stable occurrence of the subdirectory.

Directory	Since version	Description
arch	1.2.0	Code specific to a certain architecture. This subdirectory contains architecture-dependent versions of several subdirectories also present in the main directory, such as <code>lib</code> , <code>mm</code> and <code>kernel</code> .
crypto	2.6.0	Code for cryptographic applications. Includes several common algorithms such as AES, SHA-1 and Blowfish.
Documentation	2.0	Various documentation related to kernel development and use.
drivers	1.0	Code for basic buses, devices and protocols.
fs	1.0	Code for file systems.
include	1.0	Header files needed to build the kernel and its subsystems.
init	1.0	Functionality for starting the kernel.
ipc	1.0	Code for interprocess communication.
kernel	1.0	Most important kernel functionality (e.g. important system calls).
lib	1.0	The library code for the kernel.
mm	1.0	Code for memory management.
net	1.0	Network-related code.
scripts	2.0	Scripts used when the kernel is configured.
security	2.6.0	Security framework for Linux.
sound	2.6.0	Code and drivers for audio devices.
usr	2.6.0	Code for very basic I/O operations and the RAM filesystem.

Figure 3.2: Comparison of growth rates for the full distribution, modules studied in this work (`ipc`, `mm` and `kernel`) and the kernel module alone. The full distribution includes the source code, header files, documentation etc.; the modules contain only the C-language source code. Note how the combined sizes of `ipc`, `kernel` and `mm` modules follows the general exponential, while `kernel` module, when examined alone, remains almost constant. Note that sizes are plotted on a logarithmic scale.

Figure 3.3: Relative directory sizes in Linux 2.6.12.5. The section OTHER contains the subdirectories `crypto`, `init`, `ipc`, `kernel`, `lib`, `mm`, `scripts`, `security`, and `usr`.

Table 3.2: The subdirectories not found in Linux 2.6.12.5. We are limiting ourselves to those versions examined in this work. This data is from the source code and its documentation.

Directory	Occurs in
boot	1.0
ibcs	1.0
tools	1.0
zBoot	1.0
modules	1.2.0, 2.0

2.6.12.5, the number of people mentioned in the CREDITS-file⁴ rose from 375 to 468.

We can see that different device drivers are responsible for the large size of the kernel. These files do not interest us, as device drivers do not tend to change much. They are also usually self-contained; for example, a driver for a sound card chip is completely independent of other sound card chip drivers. The only exceptions are the chips from the same manufacturer or direct clones.

We eliminated these directories from further considerations. Upon closer inspection, we found three suitable directories: `mm`, `ipc` and `kernel`. These three directories have several appealing features:

- small file sizes and total sizes,
- changes occurring in files, not just new files,
- no or few subdirectories, and
- the purpose of is code easy to understand.

⁴The CREDITS-file is “at least a partial credits-file of people that have contributed to the Linux project.” It is maintained by Linus Torvalds himself.

3.3 File comparison graphs

In theory, we could simply concatenate all the files in a given Linux release together and then perform our analysis on the resulting string. However, as Chapter 2 shows, both the time and memory requirements grow as $O(mn)$, where n and m are the lengths of the input strings. For two 100 megabyte strings, we would require something in the order of 10 petabytes of memory. This is not within our possibilities at the time of writing.

To reduce these requirements, we approach the problem with a two-level strategy⁵. First we must operate on the level of modules and select such pairs of files in two different versions that they are sufficiently similar. We can then concentrate our analysis on these files and not waste time on a fruitless search.

The results of such pruning can be interpreted as a bipartite graph. The source nodes are files from version X, module A (or X/A for brevity) and the target nodes are from Y/B. An edge in this graph denotes our interest to examine this pair of files closer. An example graph for 2.0.1/mm and 2.2.0/mm is shown in Figure 3.4. We call these graphs *file comparison graphs*.

3.3.1 File comparison graph reduction

The initial file comparison graphs were computed using a modified version of the Smith-Waterman algorithm. We wished only to know if the score of any alignment between a pair of files exceeds a prespecified cutoff value. This allowed us to limit the amount of memory used by the algorithm.

Since we only needed the score, we could discard most of the score table F . The scores are computed row by row, and only the previous row is required for the computation of the current row. Thus the memory use drops from $O(n^2)$ to $O(n)$.

This modification pruned out files that have no significant⁶ alignment be-

⁵Technically we should also select the correct mapping between modules of different versions. This is not a problem in this work, as the modules `ipc`, `kernel` and `mm` have remained unchanged during the years.

⁶Significance is defined here as “having a higher score than the cutoff value”.

Figure 3.4: An example file comparison graph for modules 2.0.1/mm and 2.2.0/mm. The files on the graph are connected if the score of the longest alignment between the files is higher than a pre-specified cutoff value. The cutoff value here has been 500 and the scores are 1 for matches and -1 for gaps and mismatches.

tween them. To speed things up, we implemented the algorithm in C and used a simple distance function, with score 1 for matches or -1 for mismatches, insertions and deletions. The parameters are chosen for their simplicity. We decided to use similar parameters throughout the work; using a more detailed score function was not deemed worth the effort.

As we can see from Figure 3.4, the results needed some fine-tuning. The example graph shows a very strongly connected component, containing eg. `vmalloc.c` and `memory.c`. Although raising the cutoff would reduce the amount of edges in the graph, too high a cutoff value might prune out some interesting pairs. To determine a suitable value, we turned to the data.

We used two different datasets to determine suitable values for the distance function parameters and the cutoff value: pairs that should match and pairs that likely do not match. To see how the intuitively matching files compare with files that do not match, we picked 10 pairs of files randomly, listed in Table 3.3. For a reference pair, we picked 10 pairs of matching files by using human judgement on similarity. These pairs are listed in Table 3.4.

We calculated the alignments for each pair and computed their lengths. Figure 3.5 shows the distribution for the alignments. For purposes of clarity we have limited the x -axis to show only the relevant parts of the two distributions. The longest alignments in the randomly chosen pairs are roughly 100 characters long, while the tail is much longer with handpicked pairs.

The difference between the distributions is evident. Alignments for randomly chosen pairs are in general very short, as is to be expected. The case is even clearer when we consider only the longest alignments. These are listed in Tables 3.3 and 3.4.

Based on these observations we set the cutoff limit to be 150 characters; that is, every pair of files must have at least one alignment whose length in characters exceeds the cutoff limit. Using a match-coefficient of 2, this roughly translates to a score of 300. This is a tighter condition, but we feel there is enough margin for error.

Our reduction resulted in 769 pairs when comparing every version against every other version. The unreduced version would be 10724 pairs. The details

Table 3.3: Ten randomly chosen pairs of files used in suitable cutoff value estimation and the longest alignment for each pair measured in characters.

Pair		Longest alignment
2.6.0/kernel/uid16.c	2.6.0/mm/fremap.c	19
2.2.0/kernel/time.c	2.4.0/ipc/sem.c	82
2.6.0/mm/madvise.c	2.6.0/kernel/configs.c	28
2.4.0/kernel/sys.c	2.4.0/kernel/sysctl.c	73
2.6.0/kernel/posix-timers.c	2.0.1/mm/vmscan.c	55
1.2.0/kernel/dma.c	2.4.0/kernel/fork.c	27
2.4.0/kernel/pm.c	2.4.0/mm/page_alloc.c	71
2.2.0/kernel/exit.c	2.2.0/kernel/exec_domain.c	101
2.6.0/kernel/module.c	2.4.0/mm/mmap.c	87
2.0.1/kernel/info.c	2.4.0/mm/mlock.c	44

Table 3.4: Ten handpicked reference pairs used in suitable cutoff value estimation and the longest alignment for each pair.

Pair		Longest alignment
1.0/ipc/msg.c	2.6.0/ipc/msg.c	288
1.0/ipc/sem.c	2.2.0/ipc/sem.c	2690
2.0.1/mm/vmalloc.c	2.4.0/mm/vmalloc.c	2840
2.0.1/kernel/sys.c	2.2.0/kernel/sys.c	6359
2.2.0/mm/memory.c	2.4.0/mm/memory.c	9097
2.0.1/mm/mmap.c	2.2.0/mm/mmap.c	3978
2.0.1/mm/mmap.c	2.2.0/mm/mmap_avl.c	12810
2.4.0/kernel/sched.c	2.6.0/kernel/sched.c	1229
2.4.0/kernel/module.c	2.6.0/kernel/intermodule.c	4885
2.4.0/mm/mlock.c	2.6.0/mm/mlock.c	3055

Figure 3.5: The distribution of the alignment lengths for 10 randomly chosen and handpicked pairs of files. The x -axis is limited to 200 for clarity; for hand-picked files there exist longer alignments than 200. Also note the logarithmic y -axis.

of the alignment data are discussed in Chapter 4.

3.3.2 Improving results by removing `#include`-directives

While fine-tuning the file comparison graph computation, we noticed several false positives in the file comparison graphs. Many such pairs of files that according to our intuition had nothing to do with each other were marked as interesting.

We computed the alignments for some of these pairs. It turned out that the preprocessor directive `#include` was to blame. In C, `#include` is used to insert the contents of another file into the file at compile time. This is used to include the C header files, which contain information about constants and functions defined in other files.

Let us consider a typical `#include` directive found in Linux kernel sources. It has the form `#include <linux/name.h>\n`, where *name* is the name of the header file without the extension `.h`. For Linux 2.6, the average length for *name* is 7.14 with the minimum and maximum length being 2 and 20 characters, respectively.

Furthermore assume that a match between two characters increases the score by 1 and mismatches or gaps decrease the score by the same amount. The common parts in an `#include` directive are worth 19 points.

The average computed above is not the mean over every `#include` statement in the source files. As the most common header files also have the shortest names, such as `mm.h`, `fs.h`, `tty.h` etc., the true mean is likely to be less than the average.

If we use the average, then each `#include <linux... line` results in a net gain of 11.86 points in the total score — regardless of the name of the included file. As C files routinely include 10–20 headers per file, the `#include` statements rapidly inflate the score without yielding any true insights.

We thus removed the `#include` statements from our data. These amounted to 1–3% of the data, depending on the Linux version. This also significantly reduced the amount of arcs in our file comparison graphs.

Chapter 4

Analysis of alignment data

In this chapter we take a closer look at the alignments obtained from the Linux source code. We begin with general observations and then continue to refine our analysis of the data.

Our approach can be seen as one of many methods of static program analysis. In static analysis, the source code is analysed without running the program. Legacy code analysis, evolution and maintenance are discussed in [32] or in Finnish in [15].

Using the modified Smith-Waterman algorithm with the file comparison graphs introduced in Chapter 3 resulted in 98909 alignments in 769 pairs of files. Slightly more than a third of these alignments are between consecutive versions; 37414 alignments to be precise.

The runs were performed on six two-processor PC computers running at 2.2 GHz. The computations were finished in less than twelve hours. The memory consumption was roughly 8 gigabytes with the largest score table.

4.1 Distribution of the length of the aligned blocks

As part of the initial exploratory study, we plotted the number of alignments against their score. Our initial expectation was that there would be significantly more short alignments than long ones, which was confirmed. The distribution

also had a very noticeable shape; on a log-log scale the points formed a line. This is indicative of a *power law*. We shall introduce them in more detail in Section 4.2.

The data follows a distribution that is almost a textbook case on power law. Figure 4.1 shows the the observed count of alignments with a given score.

Figure 4.1: Number of alignments as a function of alignment scores. The results fall almost on a straight line, suggesting a power law. Note the logarithmic scale. The bifurcation at the low end of the plot is due to the score function d ; the algorithm gives even scores slightly more often than odd scores when the alignments are very short.

This is further broken down in Figure 4.2, where we have studied smaller subsets of the data. There are four sets: alignments between versions 1.2 and 2.0, alignments between version 1.0 and all other versions, alignments between versions 1.0 and 2.6.0 and alignments between all versions restricted to the module `mm`.

Figure 4.2: The power law structure is found on all levels, even when we consider only a fraction of the alignments. The plots describe four different ways of restricting which alignments we use.

4.2 Power laws

When a random variable X is said to follow a power law the resulting distribution is known as the zeta distribution in the discrete case. We will not discuss the continuous power law distribution¹ as we do not need it in this work. The relationship is expressed as

$$P(X = x) = \frac{x^{-k}}{\zeta(k)},$$

where $\zeta(k)$ is the Riemann zeta-function. Often the scaling coefficient is ignored and we write $P(X = x) \propto x^{-k}$. Note that the distribution is not defined for $k = 1$.

Relationships such as these occur in a variety of problems and they arise naturally in the context of scale-free systems. For this reason they are often used with Internet or WWW analysis, as discussed in [3] and [11]. Other applications include yeast protein networks, sounds of fracturing paper and populations of cities ([19], [25] and [12], respectively).

It is very easy to recognize a power law. Taking a logarithm of the Equation 4.2 we obtain

$$\log y = -k \log x + \log a,$$

where $a = -\zeta(k)$. The transformed variables $\log y$ and $\log x$ have a linear relationship. It is simple to detect this from a scatter plot. See Figure 4.2 for an example.

4.2.1 Our attempt to model the power law behaviour

Due to the ubiquitous nature of power law relationships, their discovery in the data set was not a huge surprise. However, the cause behind them was not clear. Unlike the more common Gaussian distribution and the related limit theorems, there is no single mechanism responsible for a power law distribution.

In [21] there is a collection of the most common reasons explaining a power

¹The Pareto distribution.

law: preferential attachment (World Wide Web) and Mandelbrot's optimisation principle. There are also discussion whether a log-normal distribution would be better in some cases claimed to exhibit power law behaviour, for example in multiplicative processes. None of the examples described our situation perfectly, so we attempted to form a simple model for the data.

Starting from the assumption that the alignments were essentially scale-free, we constructed a data-generating model. We assume that the length of an alignment is selected from an uniform distribution over $(0, \kappa b)$, where b is the size of the file and κ is an adjustable parameter. After generating a block of size r , we set $b \leftarrow b - r$ and generate another block.

After b reaches a small enough value, we end the process. To better model the discrete nature of the real alignment data, we round the values of $r = \kappa b$ to an integer and end the process when $b = 1$.

After completing the model, we tried to fit κ to the data obtained from the alignments. According to the data a good first guess for κ is between 0.1 and 0.01; our attempts showed that for the best fit $\kappa < 0.001$. Even then the fit was clearly not a good one. This discrepancy led us to examine our initial assumptions more closely.

Our model rests on there being an upper limit on the relative size of the blocks. Examining the data we discovered that the earlier blocks are proportionally larger than the later blocks. The relationship between relative size was not clearly as simple as we had hoped as the graphs showed non-linear behaviour on regular and log-plots. We did not investigate the power law further.

This study of the power law is, however, useful as a verification. As the plots in Figure 4.2 show, we can see the straight line in even very small subsets of the data. If we recomputed the alignments and their lengths would not fall on the power law line, we would suspect an error in the data or in the algorithm.

Another underlying assumption was that the alignments were similar at every scale. A closer look at the data showed us that this was too simple an approximation. Very short alignments are very noisy; almost anything aligns with anything else, when the alignment length is under ten characters. The "meaningfulness" of the alignments increases as their length increases. For this

reason several of the following experiments ignore alignments with a score less than 150. Examples of the found alignments are found in Appendix A.

4.3 Visualising the modules with Sammon’s projection

In this work our goal was to understand the development of the kernel as time progresses. We did this by visualising the data on different levels and based our analysis on the visualisations. We encountered these levels already when preprocessing the data; we can see the source code as character-based, or as token-based², file-based, and ultimately module-based.

We already studied the modules in the kernel already in Chapter 3. The next level in detail are the individual files. We wished to gain some information on the “closeness” of the files based on the computed alignments. We considered each file in each version separately, that is `1.0/mm/swap.c` is a different data point from `2.6.0/mm/swap.c`. To map these files onto a plane, we turned to multidimensional scaling (MDS).

MDS algorithms take a set of data points that have distances defined between them and project the points to a low-dimensional space, most often the two-dimensional plane. This operation allows us to recognise the relations between the files, even though the algorithms can not (in usual circumstances) capture all the information in the original data set.

There are several MDS algorithms available. Each emphasises different features of the data set depending on the cost function and optimisation algorithm used. An overview is available in [16]. We chose to apply Sammon’s projection (see [20]), which attempts to preserve small distances between points better than

²A *token* is the basic unit of structured text. The most common tokens in programming languages are reserved words, variables, and symbols. We do not employ token-based analysis in this work, but we recognise its usefulness.

large distances. The cost function for Sammon's projection is

$$E = \sum_{k \neq l} \frac{[d(k, l) - d'(k, l)]^2}{d(k, l)},$$

where d is the original metric and d' is the metric in the low-dimensional space. The algorithm tries to minimise the cost function with an iterative approach.

The results of the process are shown in Figures 4.3, 4.4 and 4.5. The files that have the same name are connected with lines. The first version of the files is marked with a short character mnemonic. The last version is marked with a circle and a dot. The circle is omitted, if the first and last files are the same.

Figure 4.3: Sammon's projection for the files in module `ipc`. The letters denote the first version in a sequence of identically named files and the circled dot represents the last version. There does not seem to be a consistent pattern.

Unfortunately the figures are not very clear. The files tended to cluster around the middle, making their interpretation difficult. The plots of modules `kernel` and `mm` reveal a complex closely related system. Some outliers are however visible. For example, `bootmem.c` (denoted by b in the plot) and `mmmap.c` (denoted by mm) are separate from the main group. Such outliers may represent a file that has remained mostly unchanged.

Figure 4.4: Sammon's projection for the files in module `kernel`. The files are mostly clustered together, although some outliers such as `softirq.c` and `sys.c` (denoted by `so` and `sy`, respectively) are visible.

Figure 4.5: Sammon's projection for the files in module `mm`. The files are more sparsely distributed compared to `kernel` and there are several outliers visible.

4.4 Mapping the relationships between files within a module

The preceding analysis has several drawbacks. First of all, we have made the assumption that the relationship between files is decided by their name alone. However, there are important exceptions. We will discuss this in more detail later in Section 4.5.1.

Secondly, there is little or no information about the progress of time. We want to answer questions about where³ and when the most significant changes occur. Finally, the magnitude of the changes is not clear in the data mapped with Sammon’s projection. We chose to develop a visualisation method that performed better with regards to these three points.

As a first step we assigned a similarity measure for files in consecutive versions. The similarity was computed by summing together the scores of all the alignments that have a score over the cutoff value of 150, that is

$$S(f_1, f_2) = \sum_{\substack{a \in A(f_1, f_2) \\ \text{score}(a) > 150}} \text{score}(a),$$

where $A(f_1, f_2)$ is the set of alignments between files f_1 and f_2 .

Note that in this visualization, we considered only alignments between consecutive versions; there are 37414 such alignments out of total of 98909, i.e. approximately 37.8%. The ratio stays does not change significantly when we consider those alignments whose score is above 150; we retained 871 out of 2330 alignments or 37.4%.

We plotted the files as boxes with one box per file per version. Files in a given version are restricted to lie on the same rank. Version numbers are shown on the left-hand side.

A line between two boxes denotes nonzero similarity. The similarity values quantized to four levels and indicated in the graph by increasing line thickness. The scores in the lowest quartile are plotted with dotted lines, scores in

³“Where” refers to the specific source code files in this instance.

the next lowest with dashed lines, and the scores in the second highest and highest quartiles with solid and bold lines, respectively. Note that as the plotting algorithm tries to keep the stronger connections as short as possible, the ordering of files differs between ranks.

Figure 4.6: The relations between files in module `ipc`.

Figure 4.7: The relations between files in module `kernel`. Note how the links between the files are more linear. A file has usually only one parent and one child, which all share the same name.

Figure 4.8: The relations between files in module mm. Note the complex interactions of the central “core”.

The results are shown in Figures 4.6, 4.7 and 4.8. It is remarkable how much the modules differ from each other. We will now present a short analysis based on the charts and, when necessary, source code inspection. A unified analysis can be found in Section 4.6.

For the module `ipc`, the chart revealed a closely knit group of files (`shm.c`, `sem.c` and `msg.c`) and one isolated file, `util.c`. Despite the high level of connectedness, it is worth noting that the strongest links are always between a file of the same name.

The three files are closely related, but it is not due to the code itself being moved in and out of the files. An examination of the files showed us that the files have similar data structures and functions to manipulate them. The alignment process was not sensitive enough to spot their minute differences.

The module `kernel` has also followed a distinctly conservative path. Files tend to form long chains and splits are rare. There is, however, a clearly identifiable rearrangement of code between versions 2.4.0 and 2.6.0. The chain structure allows us a convenient way to analyse files on closer level as explained in Section 4.5.

The most noticeable features on this chart are the splits of `ksyms.c`, `pm.c`, `sched.c` and `sys.c` in version 2.2.0 and the notable increase of the number of files starting after version 2.0.1.

The situation is very different with module `mm`. Isolating distinct chains, as we could do with `kernel` is practically impossible. Also unlike `ipc`, code is constantly reorganised in the the strongly connected “core”. However, there are some recognisable groups of files. We will analyse these in more detail in Section 4.6.3.

4.5 Visualising the differences in chains of files

As mentioned above, especially module `kernel` presents strong chaining; a file usually has only one major parent and/or descendant. We can gain insight into the evolution of the source code itself by plotting these chains. This approach

reveals both the location of the alignments and shows how file sizes have changed from version 1.0 to version 2.6.0.

Our approach consists of finding a sequence of files such that they best describe the movement of the code segments. These sequences are called best matching chains and they are described in more detail in the next subsection. As with the previous analysis, we also study the source code to confirm our interpretation in ambiguous or unclear situations.

4.5.1 Best matching chains

Forming chains out of the files has the virtue of extreme simplicity. We assume that each file has only one descendant that contains the best match in terms of the code itself. This then simplifies both the visualisation and the similarity function.

The assumption is not always a correct one, especially with the module `mm`. If a file has been split into two or more files, we can only find the best match. The true shape of the “chain” is more accurately described by the graphs in Figures 4.6, 4.7 and 4.8 in the previous section. Visualising them satisfactorily was difficult, and we chose the simpler linear chains. In this however, we ended up with 57 distinct chains, from which we could fix some of the suspicious chains by hand. Splits and other types of change are described below.

To form a chain, we start with a file and then determine its best match by finding the file that maximises the similarity function. We repeat this for every file in our data set and then combine these pairs into chains.

There are several ways to select the similarity function. The naive function, used previously with Sammon’s projection, would use the filenames themselves. Remarkably, this would be correct for the vast majority of files studied—but the most interesting changes are those when the filenames differ. Since the code performs the same function regardless of the file it resides in⁴, the similarity

⁴Compare this with the genome, where a chromosome may split into two and the genes still retain their functionality. Other datasets are not as tolerant; the paragraph revealing the identity of the murderer in a whodunnit-novel should preferably be at the end of the book.

should be a function of the alignments.

We compared two simple candidates, *max*- and *sum*-variants, for the similarity function. The *max*-variant looked at the maximum score between alignments of two files, i.e. the best match would be a file that contains the longest common sequence. The *sum*-variant looked at the sum of the alignment scores—the best match would contain the most common “code mass”.

Both approaches had their drawbacks. While the latter option does seem better on paper, the results were contaminated by the large proportion of low score alignments that matched only keywords or very short code segments. The *max*-variant was more robust, but could be fooled in some circumstances, especially splits. In the end we chose to use the *max*-variant and fix suspicious chains by hand.

After studying the results, we can propose a synthesis of the two approaches as the most satisfying similarity measure. If we compute the sum of only the highest scoring alignments⁵, we get most of the benefits of both variants. We did not follow this path in our work, as the results were good enough, c.f. the situation with multiple descendants described above.

4.5.2 Visualisation methods

The main goal for the chain visualisation was that we now wanted to see the alignments at their correct places. Without looking inside the files our understanding of the overall evolution process would be incomplete. As a side effect, placing the alignments in the right context also allowed us to see how file *sizes* had changed during the years.

We plotted the files in the chains as equally spaced vertical bars. The differences in the height of the bars represented the differences in the sizes of the corresponding files. Each chain was scaled separately, so any two chains are not directly comparable.

The bars were vertically split into two halves; the side facing another version

⁵We can choose the alignments by a prespecified percentile, a cutoff score, or by a fixed number N .

would contain the alignments with that version. The alignments were represented as bars and connected to the equivalent alignment with a line. To minimise clutter and to better understand the large scale changes, we only plotted alignments with a score higher than 150.

This method has been developed for this work. Superficially it is similar to the HistoryFlow visualisation presented in [31]. Our method is more suitable in this context as it emphasizes the role of the files as containers of shared alignments and it is more suited to the short chains encountered in this work.

We will show results of the visualisation in the following subsections. There is some literature available in addition to those mentioned at the beginning of the chapter. See for example [14] or [18].

4.5.3 Types of change between files

We identified four different classes of change between different versions. These are gradual change, rewrite, split and fusion/inclusion. As these changes occur between two versions, a chain can (and often will) contain more than one type of change.

Gradual change

When the changes from one version to another remain limited, almost the entire file is marked as an alignment. The files may still differ somewhat as Smith-Waterman algorithm allows gaps and mismatches. You can see this in Figure 4.9, as the file sizes grow slightly, but the alignments manage to cover these changes.

An entire chain may be stable during the timespan from version 1.0 to version 2.6.0. Such chains are quite common, accounting for over a fifth of all chains (12/57).

The file need not be stagnant. File `1.0/ipc/sem.c` is less than half the size of file `2.6.0/ipc/sem.c`. Their sizes are 500 and 1260 lines of code, respectively. We can see their gradual evolution as new code is added. A look at the source

Figure 4.9: The file `sem.c` has grown steadily. Despite the growth its internal organization has remained stable and new additions have not caused the alignments to split into two or more components. A possible explanation is given by version 2.4.0. The unaligned segment at the end of version 2.4.0 suggests that most new code has been inserted after the existing code.

Figure 4.10: The small size of `swap.c` in version 2.2.0 triples in version 2.4.0. Even more significant difference is found between versions 2.4.0 and 2.6.0, where the files share only a fraction of their code. Only a few conserved segments remain suggestive of the relationship between the files.

code confirms our initial assessment that the file has steadily grown.

Rewrite

A *rewrite* is the opposite of gradual change. The file is rewritten either completely from scratch or only a select group of functions is carried over to the new file. Detecting a full rewrite would be impossible with alignments alone; luckily most rewrites conserve at least a part of the code or the comments. Another way to detect them is to study the file names and look for unexpected changes.

The example in Figure 4.10 shows a remarkable rewrite in the file `swap.c` especially between versions 2.2.0 and 2.4.0 and between 2.4.0 and 2.6.0. Starting in version 2.0.1, most of the contents in `swap.c` were moved to `swapfile.c`. The file grew rapidly in version 2.4.0, where it gained several new functions. Even so, most of the code was discarded in the next version.

Detecting rewrites is easier when we use other evidence as well. Figure 4.8 also supports the rewrite hypothesis for 2.4.0/mm/swap.c. In other cases comments in the source code are also very helpful, e.g. in in 2.4.0/mm/swap_state.c:

```
/*
 *  linux/mm/swap_state.c
 *
 *  Copyright (C) 1991, 1992, 1993, 1994  Linus Torvalds
 *  Swap reorganised 29.12.95, Stephen Tweedie
 *
 *  Rewritten to use page cache, (C) 1998 Stephen Tweedie
 */
```

The first reorganisation refers to the development branch 1.3 and the second to the branch 2.1.

Split

In a *split*, a source file is divided into two different files in the next version. A split is usually the result of code restructuring. The new files are usually smaller than the original file and focus more clearly on a single function. For example, the timer-related functionality was removed from 2.4.0/kernel/sched.c, leaving sched.c responsible for process scheduling only. Another example of a split is 1.2.0/mm/swap.c, which radiated into at least five significant parts, including some segments that were subsumed into other files.

From the perspective of a kernel developer, these changes help to understand the overall structure of the module better. The new files are easier to understand than the previous version and they fill their architectural niches better.

As we can see in Figure 4.11, file 2.2.0/kernel/sched.c is very neatly split into two. Note that both 2.4.0/kernel/timer.c and 2.4.0/kernel/sched.c share an alignment at the start of the file; this is in fact a comment in the source code and not actual code.

Figure 4.11: A split in 2.2.0/kernel/sched.c. The file is split into sched.c and timer.c, above and below respectively. Notice that the scale is different in the two visualisations.

Figure 4.12: The `sys_sysinfo` function is moved into `timer.c` in version 2.6.0.

Fusion or inclusion

Two source files are merged together in a *fusion*. This happens rarely in practice. Moreover these few cases could be more accurately described as inclusion, where a smaller file is incorporated into a larger file. We found two such chains. In some split processes parts of the split file are included another file, in which case we also call fusion *inclusion*.

The rarity of fusions is most likely due to the ever expanding nature of the development process. As discussed above, source code files are easier to understand when they are as focused and as small as possible⁶. Combining two files would break this rule of thumb, unless significant parts of the previous code are discarded. While this occasionally happens, it is more often accompanied by a large rewrite.

The file `2.4.0/kernel/info.c` was subsumed into `2.6.0/kernel/timer.c` as shown in figure 4.12. The original file was only 69 lines of code in version 2.4.0 and contained only one function, `sys_sysinfo`.

⁶But not smaller. Apologies to Albert Einstein.

4.6 Overall analysis of changes

Based on the preceding visualisations, we could recognise major landmarks in Linux evolution. Although the visualisations do not reveal us all the details of the differences, they do give us valuable information at a glance. We will discuss the major changes in brief.

4.6.1 Analysis of module `ipc`

The module `ipc`, responsible for interprocess communication, has remained quite stable from version 1.0 to version 2.6.0. The module consists of four files: `msg.c`, `sem.c`, `shm.c` and `util.c`. All of these were included in version 1.0, and no other files have been added since.

The most dramatic change was the sudden growth of `util.c` between versions 2.2.0 and 2.4.0, when the size of the file almost tripled. The new file, however, contained almost everything from the previous version and the change was caused by the addition of new code into the file.

During the same time period, `msg.c` received some other new functions as well, and parts of `shm.c` were discarded. To conclude, the only significant `ipc`-event within our time window has been the rework between 2.2.0 and 2.4.0.

4.6.2 Analysis of module `kernel`

The `kernel`-module is hugely more varied than the `ipc`-module. There are 35 files and one subdirectory in 2.6.0/`kernel`. The number has nearly doubled from version 1.0, where there were 19 files in the `kernel`-module.

The leap from version 1.0 to version 1.2.0 has been a significant moment of transition. Four files (`ldt.c`, `mktime.c`, `traps.c` and `vsprintf.c`) disappear between 1.0 and 1.2.0. Also four files (`fork.c`, `ioport.c`, `irq.c`, and `signal.c`) are rewritten or large parts are moved to architecture-specific modules and thus out of our study.

The evolution of the versions 2.x is a great deal more difficult to analyse.

Despite the increased complexity, it is possible to spot some general trends. The following results are more quantitative than previous ones and perhaps somewhat more subjective.

As in all modules, there are files in the `2.x/kernel` -module that remain stable during their entire history. Some examples of this are `dma.c`, `itimer.c`, and `panic.c`. Roughly a third of the files falls into this category. Furthermore, the transition from 1.2.0 to 2.0.1 is very smooth in practically all cases. There are no significant rewrites and only two marginally interesting cases, `resource.c` and `sched.c`.

After that two different classes emerge. As a rule of thumb, a file that has been rewritten or otherwise changed radically, is not be altered significantly in the next version. Furthermore, a change in 2.2 often indicates a change in 2.6 as well. Both of the these findings make sense when seen in a developmental context. It is easier to focus efforts on a subset of a module instead of a full rewrite⁷. Also the problems of a previous rewrite may not be apparent until another module has been refactored.

An exception proves the rule. A few files have led very active lives and have changed significantly in 2.2, 2.4 and 2.6. The two most prominent examples are `resource.c` and `sched.c`.

4.6.3 Analysis of module `mm`

The module `mm` is responsible for memory management. Its growth has been the greatest of the three studied modules. In version 1.0, `mm` consisted of only 5 files; in version 2.6.0, the count has climbed up to 30. This rapid expansion presents some difficulties in the analysis that we have not encountered in the previous modules.

The simple linear chain-based method does not work well with `mm`. The frequency of splits, occasionally into more than two files, increases the number of chains needed to completely study the changes. For this reason we will first look at Figure 4.8.

⁷In other and less exact words, the difficulty of a rewrite increases faster than linearly.

The largest component, the so called “core”, is a web of files starting in version 1.2.0 with `filemap.c`, `vmalloc.c`, `memory.c`, `mprotect.c`, `swap.c` and `kmalloc.c`.

Within the core there are two initially almost independent groups. The first one is composed of the first four files mentioned above and parts of `swap.c`. The other is `kmalloc.c` and the rest of `swap.c`. The former group deals mostly with general memory management, including virtual memory, while the latter is concerned with *swapping* and *paging*.

The two branches are united in version 2.4.0. While the swap-related files still continue to exist, a comparison of the code files reveals that they have practically nothing in common. The only significant alignments are comments at the beginning of the source code files, some of them acknowledging the rewrite. We can thus consider the “New Swap” to be an independent element in the evolution of `mm`.

Another independent group is built around the file `mmap.c` of version 1.2.0 and its descendants. This group consists of at most three files per version, two of which are `mmap.c` and `mremap.c`. There is, however, a connection to the core group in version 2.6.0, when `nommu.c` contains elements of both `mmap.c` and `memory.c`.

In fairness, one must not assume that every file in `mm` has been a victim of repeated drive-by-rewrites. As with the `kernel`-module, there are files that have been conserved through the years. Naturally their proportion is smaller; on visual inspection the fraction is around 20%–25%. Memory management in Linux has been and continues to be heavily developed.

The files that have been introduced rather late in the process have not been subjected to rewrites or reorganisations. It is important to recognise that they are not *functionally* isolated from the other files. For example, the seemingly lone `slab.c` contains the source code for the so called *slab memory allocator*, which is at the heart of Linux memory management. Note that this warning also applies to other modules discussed above.

Chapter 5

Summary and conclusions

In this work we have presented how large collections of source code can be analysed without deep and specific understanding of the application or its history. Some domain knowledge is still necessary, as indicated by the necessity to remove `#include`-statements from the source code.

Using three modules from the Linux operating system, we computed the common subsequences (or alignments) between the files in the modules with modified Smith-Waterman algorithm. The files were then analysed on a large scale. We found that for this case short alignments were mostly meaningless to a human observer. A suitable minimum length should be determined depending on the case. We also found a power law relationship between the number of alignments and their length, although we could not devise a model to explain it.

We used three major visualisation methods for this work. The results from the first method, Sammon's projection, were not very revealing and require further refinement. The other two methods, in which we plotted the files and their explicit relationships to other files proved to be more useful. They allowed us to develop a rudimentary analysis of the changes in the Linux kernel.

5.1 Future work

The largest problem is the huge amount memory needed by the Smith-Waterman algorithm. It would be preferable to concatenate an entire module into one string and use that as the basic unit of analysis. Unfortunately this just is not practical¹ as the memory requirements climb into hundreds and thousands of gigabytes.

A related issue is the time complexity of the algorithm. We had to prune several pairs of files to reduce the computation time to a more manageable level. The Smith-Waterman algorithm is an exact, but not a very fast method. Faster alternatives, such as BLAST [1], that first compute only partial alignments, might prove useful in this regard.

If we consider analysing legacy code in general, there is also much we could do to incorporate our knowledge about the project or the programming language used. For example, we could use the header files (.h files) to infer some changes, especially on the level of functions. Another possible approach is to study the call graphs that connect the calling function with the called function.

5.2 Ethical considerations

This work for the most part lies outside the sphere of morals and ethics; the methods presented here should not have unintended social consequences. The work does not require the manufacture of a product liable to have social or environmental side effects. We do, however, suggest ways to shorten the time spent on understanding an software project. This can be seen as useful, especially against the backdrop of increasing use of software in the society and the need to maintain or update it.

¹Yet.

Appendix A

Examples of found alignments

Here we present some examples of the alignments.

Very short alignments (score < 25)

1.0/mm/swap.c	2.2.0/mm/page_alloc.c
ounte	ount)

1.0/ipc/sem.c	2.2.0/ipc/shm.c
curr-	curre

1.2.0/mm/mprotect.c	2.6.0/mm/vmalloc.c
vma->vm_start	map_vm_are

1.2.0/mm/swap.c	2.6.0/mm/swap.c
lockm	lock_

1.2.0/mm/memory.c	2.2.0/mm/filemap.c
_area;	_page;

Short alignments ($25 \leq \text{score} < 150$)

2.2.0/mm/filemap.c

}

/*

* Shared mappings need to be able to d

2.2.0/kernel/fork.c

cache = NULL;

1.0/ipc/util.c

d ipc_init (void)

{

sem_init();

msg_init();

shm_init();

return;

}

2.0.1/mm/filemap.c

flags & VM_MAYWRITE)) {

ops

2.4.0/mm/memory.c

address);

address = (address + PGDIR_SIZE) & PGDIR_MASK;

dir++;

} while (address && (addre

2.4.0/mm/swapfile.c

}

/*

* Page lock needs to be held in a

2.6.0/kernel/fork.c

cache = NULL;

m

2.4.0/ipc/util.c

t ipc_init (void)

{

sem_init();

msg_init();

shm_init();

return;

}

2.4.0/mm/swapfile.c

flags & SWAP_FLAG_PREFER) {

p-

2.6.0/mm/vmalloc.c

>addr;

if (addr >= vaddr + tmp->size - PAGE_SIZE)

continue;

while (addr < vaddr)

Alignments above the cutoff ($150 \leq \text{score} < 500$)

1.2.0/mm/swap.c and 2.6.0/mm/swapfile.c

1.2.0/mm/swap.c

```

    }
    if (offset < p->lowest_bit)
        p->lowest_bit = offset;
    if (offset > p->highest_bit)
        p->highest_bit = offset;
    if (!p->swap_map[offset])

```

2.6.0/mm/swapfile.c

```

    if (offset < p->lowest_bit)
        p->lowest_bit = offset;
    if (offset > p->highest_bit)
        p->highest_bit = offset;
    nr_swap_pages++;
    p->inu

```

2.2.0/mm/mremap.c and 2.4.0/mm/mmap.c

2.2.0/mm/mremap.c

```

f (vma->vm_flags & VM_LOCKED) {
    unsigned long locked = current->mm->locked_vm << PAGE_SHIFT;
    locked += new_len - old_len;
    ret = -EAGAIN;
    if (locked > current->rlim[RLIMIT_MEMLOCK].rlim_cur)
        goto out;
}
r

```

2.4.0/mm/mmap.c

```

f (mm->def_flags & VM_LOCKED) {
    unsigned long locked = mm->locked_vm << PAGE_SHIFT;
    locked += len;
    if (locked > current->rlim[RLIMIT_MEMLOCK].rlim_cur)
        return -EAGAIN;
}
/

```

2.0.1/kernel/sched.c and 2.2.0/kernel/itimer.c

2.0.1/kernel/sched.c

```

}

```

2.2.0/kernel/itimer.c

```

}

```



```
static void process_timeout(unsigned long __data)
{
    struct task_struct * p = (struct task_struct *) __data;
```

```
void it_real_fn(unsigned long __data)
{
    struct task_struct * p = (struct task_struct *) __data;
    u
```

2.2.0/kernel/signal.c and 2.6.0/kernel/signal.c

```
2.2.0/kernel/signal.c
signal. */
    if (sig < SIGRTMIN) {
        /* XXX: As an extension, support queueing exactly
           one non-rt signal if SA_SIGINFO is set, so that
           we can get more detailed information about the
           cause of the signal. */
```

```
2.6.0/kernel/signal.c
signals. */
    if (sig_ignored(t, sig))
        goto out;

    /* Support queueing exactly one non-rt signal, so that we
       can get more detailed information about the cause of
       the signal. */
```

2.2.0/mm/slab.c and 2.6.0/mm/slab.c

```
2.2.0/mm/slab.c
TATS_INC_ALLOCED(cachep);
    SLAB_STATS_INC_ACTIVE(cachep);
    SLAB_STATS_SET_HIGH(cachep);
    slabp->s_inuse++;
    bufp = slabp->s_freep;
    slabp->s_freep = bufp->buf_nextp;
    if (slabp->s_freep) {
ret_obj:
        if (!slabp->s_index) {
            bufp->buf_slabp = slabp;
```

```
2.6.0/mm/slab.c
TATS_INC_ALLOCED(cachep);
    STATS_INC_ACTIVE(cachep);
    STATS_SET_HIGH(cachep);

    /* get obj pointer */
    ac_entry(ac)[ac->avail++] = slabp->s_mem + slabp->free*cachep->objsize;

    slabp->inuse++;
    next = slab_bufctl(slabp)[slabp->free];
    #if DEBUG
```

Long and very long alignments (score ≥ 500)

1.2.0/mm/mmap.c and 2.0.1/mm/mmap.c

```

1.2.0/mm/mmap.c
*
* linux/mm/mmap.c
*
* Written by obz.
*/

static int anon_map(struct inode *, struct file *, struct vm_area_struct *);

/*
 * description of effects of mapping type and prot in current implementation.
 * this is due to the limited x86 page protection hardware. The expected
 * behavior is in parens:
 *
 * map_type prot
 * PROT_NONE PROT_READ PROT_WRITE PROT_EXEC
 * MAP_SHARED r: (no) no r: (yes) yes r: (no) yes r: (no) yes
 * w: (no) no w: (no) no w: (yes) yes w: (no) no
 * x: (no) no x: (no) yes x: (no) yes x: (yes) yes
 *
 * MAP_PRIVATE r: (no) no r: (yes) yes r: (no) yes r: (no) yes
 * w: (no) no w: (no) no w: (copy) copy w: (no) no
 * x: (no) no x: (no) yes x: (no) yes x: (yes) yes
 */

pgprot_t protection_map[16] = {
    __P000, __P001, __P010, __P011, __P100, __P101, __P110, __P111,
    __S000, __S001, __S010, __S011, __S100, __S101, __S110, __S111
}
2.0.1/mm/mmap.c
*
* linux/mm/mmap.c
*
* Written by obz.
*/

/*
 * description of effects of mapping type and prot in current implementation.
 * this is due to the limited x86 page protection hardware. The expected
 * behavior is in parens:
 *
 * map_type prot
 * PROT_NONE PROT_READ PROT_WRITE PROT_EXEC
 * MAP_SHARED r: (no) no r: (yes) yes r: (no) yes r: (no) yes
 * w: (no) no w: (no) no w: (yes) yes w: (no) no
 * x: (no) no x: (no) yes x: (no) yes x: (yes) yes
 *
 * MAP_PRIVATE r: (no) no r: (yes) yes r: (no) yes r: (no) yes
 * w: (no) no w: (no) no w: (copy) copy w: (no) no
 * x: (no) no x: (no) yes x: (no) yes x: (yes) yes
 */

pgprot_t protection_map[16] = {
    __P000, __P001, __P010, __P011, __P100, __P101, __P110, __P111,
    __S000, __S001, __S010, __S011, __S100, __S101, __S110, __S111
}

```

1.0/mm/memory.c and 2.4.0/mm/memory.c

```

1.0/mm/memory.c
;
return 0;

2.4.0/mm/memory.c
;
return error;

```

```

}

/*
 * maps a range of physical memory into the requested pages. the old
 * mappings are removed. any references to nonexistent pages results
 * in null mappings (currently treated as "copy-on-access")
 */
int remap_page_range(unsigned long from, unsigned long to, unsigned long size, int mask)|static inline void remap_pte_range(pte_t * pte, unsigned long address, unsigned long size,
{
    unsigned long *page_table, *dir;
    unsigned long poff, pcnt;
    unsigned long page;

    {
        unsigned long end;

        a

```

1.0/kernel/time.c and 2.0.1/kernel/time.c

1.0/kernel/time.c

```

    tv->tv_usec += do_gettimeoffset();
    if (tv->tv_usec >= 1000000) {
        tv->tv_usec -= 1000000;
        tv->tv_sec++;
    }
    sti();
#else /* not __i386__ */
    cli();
    *tv = xtime;
    sti();
#endif /* not __i386__ */
}

asmlinkage int sys_gettimeofday(struct timeval *tv, struct timezone *tz)
{
    int error;

    if (tv) {
        struct timeval ktv;
        error = verify_area(VERIFY_WRITE, tv, sizeof *tv);
        if (error)
            return error;
        do_gettimeofday(&ktv);
        put_fs_long(ktv.tv_sec, (unsigned long *) &tv->tv_sec);
        put_fs_long(ktv.tv_usec, (unsigned long *) &tv->tv_usec);
    }

```

2.0.1/kernel/time.c

```

    value = get_user(tptr);
    cli();
    xtime.tv_sec = value;
    xtime.tv_usec = 0;
    time_state = TIME_ERROR;
    time_maxerror = MAXPHASE;
    time_esterror = MAXPHASE;
    sti();
    return 0;
}

#endif

asmlinkage int sys_gettimeofday(struct timeval *tv, struct timezone *tz)
{
    int error;

    if (tv) {
        struct timeval ktv;
        error = verify_area(VERIFY_WRITE, tv, sizeof *tv);
        if (error)
            return error;
        do_gettimeofday(&ktv);
        memcpy_tofs(tv, &ktv, sizeof(ktv));
    }

```

```

    }
    if (tz) {
        error = verify_area(VERIFY_WRITE, tz, sizeof *tz);
        if (error)
            return error;
        put_fs_long(sys_tz.tz_minuteswest, (unsigned long *) tz);
        put_fs_long(sys_tz.tz_dsttime, ((unsigned long *) tz)+1);
    }
    return 0;
}

/*
 * Adjust the time obtained from the CMOS to be GMT time instead of
 * local time.
 *
 * This is ugly, but preferable to the alternatives. Otherwise we
 * would either need to write a program to do it in /etc/rc (and risk
 * confusion if the program gets run more than once; it would also be
 * hard to make the program warp the clock precisely n hours) or
 * compile in the timezone information into the kernel. Bad, bad....
 *
 * XXX Currently does not adjust for daylight savings time. May not
 * need to do anything, depending on how smart (dumb?) the BIOS
 * is. Blast it all.... the best thing to do not depend on the CMOS
 * clock at all, but get the time via NTP or timed if you're on a
 * network.... - TYT, 1/1/92
 */
inline static void warp_clock(void)
{
    cli();
    xtime.tv_sec += sys_tz.tz_minuteswest * 60;
    sti();
}

/*
 * The first time we set the timezone, we will warp the clock so that
 * it is ticking GMT time instead of local time. Presumably,
 * if someone is setting the timezone then we are running in an
 * environment where the programs understand about timezones.
 * This should be done at boot time in the /etc/rc script, as
 * soon as possible, so that the clock can be set right. Otherwise,
 * various programs will get confused when the clock gets warped.
 */
asmlinkage int sys_settimeofday(struct timeval *tv, struct timezone *tz)
{
    if (tz) {
        error = verify_area(VERIFY_WRITE, tz, sizeof *tz);
        if (error)
            return error;
        memcpy_tofs(tz, &sys_tz, sizeof(sys_tz));
    }
    return 0;
}

/*
 * Adjust the time obtained from the CMOS to be UTC time instead of
 * local time.
 *
 * This is ugly, but preferable to the alternatives. Otherwise we
 * would either need to write a program to do it in /etc/rc (and risk
 * confusion if the program gets run more than once; it would also be
 * hard to make the program warp the clock precisely n hours) or
 * compile in the timezone information into the kernel. Bad, bad....
 *
 * - TYT, 1992-01-01
 *
 * The best thing to do is to keep the CMOS clock in universal time (UTC)
 * as real UNIX machines always do it. This avoids all headaches about
 * daylight saving times and warping kernel clocks.
 */
inline static void warp_clock(void)
{
    cli();
    xtime.tv_sec += sys_tz.tz_minuteswest * 60;
    sti();
}

/*
 * In case for some reason the CMOS clock has not already been running
 * in UTC, but in some local time: The first time we set the timezone,
 * we will warp the clock so that it is ticking UTC time instead of
 * local time. Presumably, if someone is setting the timezone then we
 * are running in an environment where the programs understand about
 * timezones. This should be done at boot time in the /etc/rc script,
 * as soon as possible, so that the clock can be set right. Otherwise,
 * various programs will get confused when the clock gets warped.
 */
asmlinkage int sys_settimeofday(struct timeval *tv, struct timezone *tz)
{
    static int firsttime = 1;

```



```

static int firsttime = 1;

if (!suser())
    return -EPERM;
if (tz) {
    sys_tz.tz_minuteswest = get_fs_long((unsigned long *) tz);
    sys_tz.tz_dsttime = get_fs_long((unsigned long *) tz)+1);
    if (firsttime) {
        firsttime = 0;
        if (!tv)
            warp_clock();
    }
}
if (tv) {
    int sec, usec;

    sec = get_fs_long((unsigned long *)tv);
    usec = get_fs_long((unsigned long *)tv)+1);

    cli();
    /* This is revolting. We need to set the xtime.tv_usec
     * correctly. However, the value in this location is
     * is value at the last tick.
     * Discover what correction gettimeofday
     * would have done, and then undo it!
     */
    usec -= do_gettimeoffset();

    if (usec < 0)
    {
        usec += 1000000;
        sec--;
    }
    xtime.tv_sec = sec;
    xtime.tv_usec = usec;
    time_status = TIME_BAD;
    time_maxerror = 0x70000000;
    time_esterror = 0x70000000;
    sti();
}
return 0;
}

/* adjtimex mainly allows reading (and writing, if superuser) of
 * kernel time-keeping variables. used by xntpd.

struct timeval new_tv;
struct timezone new_tz;

if (!suser())
    return -EPERM;
if (tv) {
    int error = verify_area(VERIFY_READ, tv, sizeof(*tv));
    if (error)
        return error;
    memcpy_fromfs(&new_tv, tv, sizeof(*tv));
}
if (tz) {
    int error = verify_area(VERIFY_READ, tz, sizeof(*tz));
    if (error)
        return error;
    memcpy_fromfs(&new_tz, tz, sizeof(*tz));
}
if (tz) {
    sys_tz = new_tz;
    if (firsttime) {
        firsttime = 0;
        if (!tv)
            warp_clock();
    }
}
if (tv)
    do_settimeofday(&new_tv);
return 0;
}

long pps_offset = 0; /* pps time offset (us) */
long pps_jitter = MAXTIME; /* time dispersion (jitter) (us) */

long pps_freq = 0; /* frequency offset (scaled ppm) */
long pps_stabil = MAXFREQ; /* frequency dispersion (scaled ppm) */

long pps_valid = PPS_VALID; /* pps signal watchdog counter */

int pps_shift = PPS_SHIFT; /* interval duration (s) (shift) */

long pps_jitcnt = 0; /* jitter limit exceeded */
long pps_calcnt = 0; /* calibration intervals */
long pps_errcnt = 0; /* calibration errors */
long pps_stbcnt = 0; /* stability limit exceeded */

```

```

/*
asmlinkage int sys_adjtimex(struct timex *txc_p)
{
    long ltemp, mtemp, save_adjust;
    int error;

    /* Local copy of parameter */
    struct timex txc;

    error = verify_area(VVERIFY_WRITE, txc_p, sizeof(struct timex));
    if (error)
        return error;

    /* Copy the user data space into the kernel copy
     * structure. But bear in mind that the structures
     * may change
     */
    memcpy_fromfs(&txc, txc_p, sizeof(struct timex));

    /* In order to modify anything, you gotta be super-user! */
    if (txc.mode && !user())
        return -EPERM;

    /* Now we validate the data before disabling interrupts
     */

    if (txc.mode & ADJ_OFFSET)
        /* Microsec field limited to -131000 .. 131000 usecs */
        if (txc.offset <= -(1 << (31 - SHIFT_UPDATE))
            | txc.offset >= (1 << (31 - SHIFT_UPDATE)))
            return -EINVAL;

    /* time_status must be in a fairly small range */
    if (txc.mode & ADJ_STATUS)
        if (txc.status < TIME_OK || txc.status > TIME_BAD)
            return -EINVAL;

    /* if the quartz is off by more than 10% something is VERY wrong ! */
    if (txc.mode & ADJ_TICK)
        if (txc.tick < 900000/HZ || txc.tick > 1100000/HZ)
            return -EINVAL;

    cli();

    /* Save for later - semantics of adjtime is to return old value */

/* hook for a loadable hardpps kernel module */
void (*hardpps_ptr)(struct timeval *) = (void (*)(struct timeval *))0;

/* adjtimex mainly allows reading (and writing, if superuser) of
 * kernel time-keeping variables. used by xntpd.
 */
asmlinkage int sys_adjtimex(struct timex *txc_p)
{
    long ltemp, mtemp, save_adjust;
    int error;

    /* Local copy of parameter */
    struct timex txc;

    error = verify_area(VVERIFY_WRITE, txc_p, sizeof(struct timex));
    if (error)
        return error;

    /* Copy the user data space into the kernel copy
     * structure. But bear in mind that the structures
     * may change
     */
    memcpy_fromfs(&txc, txc_p, sizeof(struct timex));

    /* In order to modify anything, you gotta be super-user! */
    if (txc.modes && !user())
        return -EPERM;

    /* Now we validate the data before disabling interrupts
     */

    if (txc.modes != ADJ_OFFSET_SINGLESHOT && (txc.modes & ADJ_OFFSET))
        /* adjustment Offset limited to +- .512 seconds */
        if (txc.offset <= - MAXPHASE || txc.offset >= MAXPHASE )
            return -EINVAL;

    /* if the quartz is off by more than 10% something is VERY wrong ! */
    if (txc.modes & ADJ_TICK)
        if (txc.tick < 900000/HZ || txc.tick > 1100000/HZ)
            return -EINVAL;

    cli();

    /* Save for later - semantics of adjtime is to return old value */
    save_adjust = time_adjust;

```

```

save_adjust = time_adjust;

/* If there are input parameters, then process them */
if (txc.mode)
{
    if (time_status == TIME_BAD)
        time_status = TIME_OK;

    if (txc.mode & ADJ_STATUS)
        time_status = txc.status;

    if (txc.mode & ADJ_FREQUENCY)
        time_freq = txc.frequency << (SHIFT_KF - 16);

    if (txc.mode & ADJ_MAXERROR)
        time_maxerror = txc.maxerror;

    if (txc.mode & ADJ_ESTERROR)
        time_esterror = txc.esterror;

    if (txc.mode & ADJ_TIMECONST)
        time_constant = txc.time_constant;

    if (txc.mode & ADJ_OFFSET)
        if (txc.mode == ADJ_OFFSET_SINGLESOT)
        {
            time_adjust = txc.offset;
        }
        else /* XXX should give an error if other bits set */
        {
            time_offset = txc.offset << SHIFT_UPDATE;
            mtemp = xtime.tv_sec - time_reftime;
            time_reftime = xtime.tv_sec;
            if (mtemp > (MAXSEC+2) || mtemp < 0)
                mtemp = 0;

            if (txc.offset < 0)
                time_freq -= (-txc.offset * mtemp) >>
                    (time_constant + time_constant);
            else
                time_freq += (txc.offset * mtemp) >>
                    (time_constant + time_constant);

            ltemp = time_tolerance << SHIFT_KF;

/* If there are input parameters, then process them */
if (txc.modes)
{
    if (time_state == TIME_BAD)
        time_state = TIME_OK;

    if (txc.modes & ADJ_STATUS)
        time_status = txc.status;

    if (txc.modes & ADJ_FREQUENCY)
        time_freq = txc.freq;

    if (txc.modes & ADJ_MAXERROR)
        time_maxerror = txc.maxerror;

    if (txc.modes & ADJ_ESTERROR)
        time_esterror = txc.esterror;

    if (txc.modes & ADJ_TIMECONST)
        time_constant = txc.constant;

    if (txc.modes & ADJ_OFFSET)
        if ((txc.modes == ADJ_OFFSET_SINGLESOT)
            || !(time_status & STA_PLL))
        {
            time_adjust = txc.offset;
        }
        else if ((time_status & STA_PLL) || (time_status & STA_PPSTIME))
        {
            ltemp = (time_status & STA_PPSTIME &&
                    time_status & STA_PPSSIGNAL) ?
                pps_offset : txc.offset;

/*
 * Scale the phase adjustment and
 * clamp to the operating range.
 */
            if (ltemp > MAXPHASE)
                time_offset = MAXPHASE << SHIFT_UPDATE;
            else if (ltemp < -MAXPHASE)
                time_offset = -(MAXPHASE << SHIFT_UPDATE);
            else
                time_offset = ltemp << SHIFT_UPDATE;

```

```

        if (time_freq > ltemp)
            time_freq = ltemp;
        else if (time_freq < -ltemp)
            time_freq = -ltemp;
    }
    if (txc.mode & ADJ_TICK)
        tick = txc.tick;

}
txc.offset      = save_adjust;
txc.frequency    = ((time_freq+1) >> (SHIFT_KF - 16));
txc.maxerror     = time_maxerror;
txc.esterror     = time_esterror;
txc.status       = time_status;
txc.time_constant = time_constant;
txc.precision    = time_precision;
txc.tolerance    = time_tolerance;
txc.time         = xtime;
txc.tick         = tick;

sti

```

```

/*
 * Select whether the frequency is to be controlled and in which
 * mode (PLL or FLL). Clamp to the operating range. Ugly
 * multiply/divide should be replaced someday.
 */

if (time_status & STA_FREQHOLD || time_reftime == 0)
    time_reftime = xtime.tv_sec;
mtemp = xtime.tv_sec - time_reftime;
time_reftime = xtime.tv_sec;
if (time_status & STA_FLL)
{
    if (mtemp >= MINSEC)
    {
        ltemp = ((time_offset / mtemp) << (SHIFT_USEC -
            SHIFT_UPDATE));

        if (ltemp < 0)
            time_freq -= -ltemp >> SHIFT_KH;
        else
            time_freq += ltemp >> SHIFT_KH;
    }
}

```


Bibliography

- [1] Stephen F. Altschul, Warren Gish, Webb Miller, Eugene W. Myers, and David J. Lipman. Basic local alignment search tool. *Journal of Molecular Biology*, 215:403–410, 1990.
- [2] Stephen F. Altschul and David J. Lipman. Trees, stars, and multiple biological sequence alignment. *SIAM Journal on Applied Mathematics*, 49(1):197–209, February 1989.
- [3] Pierre Baldi, Paolo Frasconi, and Padhraic Smyth. *Modeling the Internet and the Web*. John Wiley & Sons, 2003.
- [4] M. Beck, H. Böhme, M. Dziadzka, U. Kunitz, R. Magnus, C. Schröter, and D. Verworner. *Linux Kernel Programming, 3rd ed.* Pearson Education Ltd., 2002.
- [5] Richard Bellman. On the theory of dynamic programming. In *Proceedings of the National Academy of Sciences of the United States of America*, volume 38, pages 716–719, Aug 1952.
- [6] Ethan A. Carver and Lisa Stubbs. Zooming in on the human-mouse comparative map: Genome conservation re-examined on a high-resolution scale. *Genome Research*, 7(12):1123–1137, December 1997.
- [7] Sam Chapman. String similarity metrics for information integration. Published in Web at <http://www.dcs.shef.ac.uk/%7esam/stringmetrics.html>. Last visited on March 20th 2006.
- [8] Ramu Chenna, Hideaki Sugawara, Tadashi Koike, Rodrigo Lopez, Toby J. Gibson, Desmond G. Higgins, and Julie D. Thompson. Multiple sequence

- alignment with the Clustal series of programs. *Nucleic Acids Research*, 31(13):3497–3500, July 2003.
- [9] Richard O. Duda, Peter E. Hart, and David G. Stork. *Pattern Classification*. John Wiley & Sons, 2001.
- [10] R. Durbin, S. Eddy, A. Krogh, and G. Mitchinson. *Biological sequence analysis*. Cambridge University Press, 1998.
- [11] M. Faloutsos, P. Faloutsos, and C. Faloutsos. On power-law relationships of the internet topology. *Computer Communication Review*, 29(4):251–262, 1999.
- [12] Xavier Gabaix. Zipf’s law and the growth of cities. *The American Economic Review*, 89(2):129–132, May 1999.
- [13] David J. Hand, Heikki Mannila, and Padhraic Smyth. *Principles of Data Mining*. MIT Press, 2001.
- [14] Maarit Harsu. *Re-engineering legacy software through language conversion*. PhD thesis, University of Tampere, Jun 2000.
- [15] Maarit Harsu. *Ohjelmien ylläpito ja uudistaminen*. Talentum Media Oy, 2003.
- [16] S. Kaski. *Data exploration using self-organizing maps*. PhD thesis, Helsinki University of Technology, Mar 1997. Pages 15–19.
- [17] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall, 2nd. edition, 1988.
- [18] M. M. Lehman and L. A. Belady. *Program evolution: processes of software change*. Academic Press, 1985.
- [19] Sergei Maslov and Kim Sneppen. Specificity and stability in topology of protein networks. *Science*, 296:910, 2002.
- [20] J. W. Sammon, Jr. A nonlinear mapping for data structure analysis. *IEEE Transactions on Computers*, C-18(5):401–409, May 1969.

- [21] Michael Mitzenmacher. A brief history of lognormal and power law distributions. In *Proceedings of the 39th Annual Allerton Conference on Communication, Control, and Computing*, volume 39, pages 182–191, 2001.
- [22] Saul B. Needleman and Christian D. Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology*, 48:443–453, March 1970.
- [23] Open Source Initiative OSI. Why “free” software is too ambiguous. Published in Web at <http://www.opensource.org/advocacy/free-notfree.php>. Last visited on March 1st 2006.
- [24] David A. Rusling. The Linux kernel. Published in Web at <http://www.tldp.org/LDP/tlk/tlk.html>. Last visited on March 25th 2006.
- [25] L.I. Salminen and M.J. Alava A.I. Tolvanen. Analysis of long crack lines in paper webs. *Phys. Rev. Lett.*, 89(18), Oct 2002.
- [26] T. F. Smith and M. S. Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, 147:195–197, 1981.
- [27] Richard M. Stallman. *Free Software, Free Society: Selected Essays of Richard M. Stallman*, chapter 6, pages 55–60. GNU Press, 2002.
- [28] Andrew S. Tanenbaum. *Modern operating systems*, chapter 10, pages 671–757. Prentice Hall, 2nd. edition, 2001.
- [29] Linus Torvalds. What would you like to see most in minix? In Usenet news group `comp.os.minix`. Message-ID: <1991Aug25.205708.9541@klaava.Helsinki.FI>.
- [30] Linus Torvalds. Linux: a portable operating system. Master’s thesis, University of Helsinki, 1997.
- [31] Fernanda B. Viégas, Martin Wattenberg, and Kushal Dave. Studying cooperation and conflict between authors with history flow visualizations. In *Conference on Human Factors in Computing Systems*, 2004.

- [32] Nelson H. Weiderman, John K. Bergey, Dennis B. Smith, and Scott R. Tilley. Approaches to legacy system evolution. Technical Report CMU/SEI-97-TR-014, Software Engineering Institute, Carnegie Mellon University, Dec 1997.