Wizard Code

A View on Low-Level Programming DRAFT VERSION 4

Tuomo Petteri Venäläinen

November 27, 2016

Ramblings on hacking low-level and other kinds of code.

Part I

Table of Contents

Contents

Part II

Ideas

I think this book should take a closer look on ARM assembly; ARM processors are very common in systems such as smart phones, and provide a power- and budget-friendly way to get into computing for people such as our children. One page to check out is

http://www.raspberrypi.org/;

the Raspberry Pi seems to be a quite useful, ultra-low cost computer with USB ports, Ethernet, and other modern features. I would strongly recommend it for projects such as learning assembly programming. A big **thank you** for **Jeremy Sturdivant** for providing me a Raspi. :D

Part III

Preface

Draft 4

Draft number 4 contains fixes in implementation of the C library nonlocal goto interface with setjmp() and longjmp() functions. The assembly statements are now declared **__volatile__** and as single assembly statement to keep them safer from enemies such as compiler optimisations as suggested by the GNU C Compiler info page as well as some friendly folks on IRC on Freenode. :) In other words the code should be more robust and reliable now. Notice how elegant the ARM implementation is; ARM assembly is so cool it makes me want to program in assembly, something rare to nonexistent on most PC architectures. I wonder if **Intel and friends** should release stripped-down versions of their instruction sets in new CPU modules and back the plan up with support from C compilers and other software. Books such as the Write Great Code series by Randall Hyde,

see: **http://www.writegreatcode.com/**

could have suggestions for minimal versions of the wide-spread X86 instruction sets; I'd look into using mostly **IA-32** and **X86-64** operations. As per competition, I'd give everything I have for access to **64-bit ARM workstations**; hopefully some shall pop up in the market in due time...

Draft 2

This draft, version 2, has some rewordings, fixes typos and mistakes, and cleans a few small things up. I found some mistakes in the code snippets as well.

Chapter 1

Forewords

This book started as a somewhat-exhaustive paper on computer presentation of integral numeric values. As time went on, I started combining the text with my older and new ideas of how to teach people to know C a bit deeper. I'm in the hopes this will make many of the readers more fluent and capable C programmers.

1.1 First Things

Dedicated to my friends who have put up with me through the times of good and bad and all those great minds behind good software such as classic operating systems and computer games.

```
Be strong in the spirit,
the heart be thy guide,
Love be thy force,
life be thy hack.
```

1.1.1 Thank You

Big thanks to everyone who's helped me with this book in the form of comments and suggestions; if you feel I have forgotten to include your name here, please point it out... As I unfortnately forgot to write some names down when getting nice feedback on IRC, I want to thank the IRC people on Freenode collectively for helping me make the book better.

- Dale 'swishy' Anderson
- Craig 'craig' Butcher
- Ioannis Georgilakis
- Matthew 'kinetik' Gregan
- Hisham 'CodeWarrior' Mardam Bey

- Dennis 'dmal' Micheelsen
- Andrew 'Deimos' Moenk
- Michael 'doomrobo' Rosenberg
- Martin 'bluet' Stensgård
- Jeremy 'jercos' Sturdivant
- Vincent 'vtorri' Torri
- Andrew 'awilcox' Wilcox
- Timo Yletyinen

1.1.2 Preface

Wizard Code

Wizard Code is intended to provide a close look on low-level programming using the ISO C and sometimes machine-dependent assembly languages. The idea is to gather together information it took me years to come by into a single book. I have also included some examples on other types of programming. This book is GNU- and UNIX-friendly.

1.1.3 Goals

One of the goals of this book is to teach not only how to optimise code but mostly how to write fast code to start with. This shows as chapters dedicated to optimisation topics as well as performance measurements and statistics for some of the code in this book.

I hope this book satisfies both budding young programmers and experienced ones. Have fun and keep the curious spirit alive.

1.1.4 Rationale

Why C?

One of the reasons I chose C as the language of preference is that it's getting hard to find good information on it; most new books seem to concentrate on higher-level languages such as Java, Perl, Python, Ruby, and so on - even new books on C++ don't seem to be many these days. Another reason for this book is that I have yet to see a book on the low-level aspects of C as a close-to-machine language.

Know the Machine

I think every programmer will still benefit from knowing how the machine works at the lowest level. As a matter of fact, I consider this knowledge crucial for writing high-performance applications, including but not limited to operating system kernels and system libraries.

GNU & UNIX

Where relevant, the example programs and other topics are UNIX- centric. This is because, even though things are getting a bit awkward in the real world, UNIX is, deep in its heart, a system of notable elegance and simplicity.

As they're practically de facto standards today, I chose to use the GNU C Compiler and other GNU tools for relevant parts of this book. For the record, Linux has been my kernel of choice for one and a half decades. Some of the freely available programming tools such as Valgrind are things you learn to rely on using Linux.

1.1.5 C Language

1.1.5.1 Overview

Personal View

I consider the C language the biggest contribution to software ever.

C is Low-Level

C is a low-level programming language. It was designed to have an efficient memory abstraction - memory addresses are represented as pointers. Basic C operations map to common machine instructions practically one-to-one. More complex operations are consistently implemented in the standard library and system libraries. The runtime requirements are very small. There exists a minimalistic but efficient standard library that is a required part of C implementations. I think it's good to say it shares much of the elegance of early versions of UNIX.

C is Simple

C is a simple language. It doesn't take long to learn the base for the whole language, but it takes a long while to master it; assuming mastering it is a possibility.

C is Powerful

C is a powerful language. Pretty much no other language, short of assembly and C++, lets you do everything C does - for the good and bad.

If you can't do it in C, do it in assembly.

"If you can't do it in assembly, it's not worth doing." C has long been the language of choice for programmers of operating system kernels, scientific applications, computer games, graphical applications, and many other kinds of software. There are plenty of new languages and a bit of everything for everyone around, but C has kept its place as what I call the number one language in the history of computer programming. Whatever it is that you do, you have almost total control of the machine. Mixed with a bit of inline and very rarely - unless you want to - raw assembly lets you do practically everything you can with a microprocessor.

1.1.5.2 History

The Roots

The roots of the C language lead to the legendary AT&T Bell Laboratories around the turn of the 1960's and 1970's. Dennis Ritchie and other system hackers, most notably Ken Thompson, needed a relatively machine-independent language to make it easier to implement their new operating system, UNIX, in a more portable way.

Legend has it that UNIX actually started from a project whose purpose was to create a computer game for an old PDP machine. I guess games aren't all bad for inspiration; another story tells that FreeBSD's Linux emulation started as an attempt to make 'Frisbee' run Linux versions of Doom.

1.1.5.3 Future

System Language

C is a traditional system language, and will very likely - even hopefully - continue its existence as the language of choice for low-level programmers. Operating system kernels can be implemented as stand-alone programs, but still require some assembly programming. Most if not all common compilers allow relatively easy mixing of assembly-language statements with C code; with a bit of care, you can command the machine with its native instructions from C code quite easily and seamlessly.

Minimalistic Approach

With its minimalistic runtime requirements, machine-friendly data and code presentation, and relative ease of implementing custom libraries, C makes a great language for development on embedded systems. Even though C is a great language for programming high-speed applications for the desktop, the crazy amount of CPU horsepower in modern PCs often makes other, higher-level languages more desirable for desktop application development. However, typical embedded devices such as mobile phones have more dire requirements for code speed and size. Hence, I predict a long and prosperous life for the C language, which many of us love - and sometimes hate - with passion. :)

Code Speed

On the desktop, C still has its place in development of multimedia and other applications with extreme speed requirements. C excels at things such as direct hardware access.

System Language

As a system language, C is and, in my opinion, should be an obvious choice for developing operating system kernels and other system software with.

1.1.6 KISS Principle

Simplicity

The KISS principle - Keep It Simple, Silly/Stupid - states that simplicity is a key goal and that complexity should be avoided; in this case, particularly in computer software development. Perhaps it also hints, in this context, at the ease of creating programs no one else can understand.

Elegance

It's not necessarily obvious how much work it takes to find the essence of problems being solved. Software developers had better not be judged in terms of how many code lines they can produce - a much better measure would be how few lines they can solve problems with. What the number of code lines doesn't reveal is how many dead ends and mediocre solutions it required to come up with a hopefully elegant and clean solution.

Do What You Need to

To summarize the KISS principle for software development, do what you need to and only what you need to. The simpler and fewer your operations, the faster the code. The fewer lines of code, the fewer bugs.

1.1.7 Software Development

One-Man Projects

There are as many ways to develop software as there are software developers. I'm keen on one-man projects, perhaps mostly because I'm still learning. In fact, I think one of the really cool things about software development is that you never run out of new things to learn. The field of software is relatively new and still taking form. New ways to solve problems are being invented all the time.

The good things about one-man projects include no communication overhead and the possibility for one person to know the whole system inside-out. Implementation and design can be done simultaneously and mistakes fixed as they emerge.

Philosophy

For software development, as well as all creative work, I suggest following the way of the empty mind. Close out all unnecessary distractions, become one with what you do, think it, feel it, do it. Find total concentration with nothing but the task at hand in your mind.

Art

Software is written expression, information, knowledge - software is art.

1.1.8 Conclusions

Essence

To develop great software, look for the essence of things. Keep your data structures and code simple. Experiment with solutions, learn from the bad ones, try new ones. Even though there may be no perfection, it's still a good thing to reach for.

Statement

C is alive and a-rockin'!

1.2 Suggested Reading

Books

Author(s)	Book & ISBN
Booth, Rick	Inner Loops 0-201-47960-5
Hyde, Randall	WRITE GREAT CODE, Volume 1: Understanding the Machine 1-59327-003-8
Hyde, Randall	WRITE GREAT CODE, Volume. 2: Thinking Low-Level, Writing High-Level 1-59327-003-8
Lamothe, Andre	Black Art of 3D Game Programming 1-57169-004-2
Lions, John	Lions' Commentary on UNIX 6th Edition with Source Code 1-57398-013-7
Maxfield, Clyde	The Definitive Guide to How Computers do Math 0-471-73278-8
Warren, Henry S.	Hacker's Delight 0-201-91465-4

Chapter 2

Overview

A Look at C

This books starts with a look at the C programming language. It's not written to be the first course on C, but instead for programmers with some knowledge of the language. The readers will get a grasp of some aspects of the 'new' version of the language ('C99', **ISO/IEC 9899**) as well as other language basics. You will gain in-depth knowledge of C's **stack-based** execution, i.e. how code operates at machine level. There is a bit of information about using C for system programming (signals, memory model) too, including a reasonably good standard library (malloc-style) dynamic memory allocator. Pointers have their own chapter.

Compiler Optimisations

For most code in this book, the GCC flag **-O** is the best optimisation level to use. At times, the code may run **slower** if you use **-O2** and beyond. Where specific optimisations need to be enabled or disabled, I try to hint you at it. In particular, some routines depend on 'standard' use of frame pointer, hence it's necessary to give GCC the **-fno-omit-frame-pointer** flag for building correct code.

Basic Computer Architecture

Next we shall move on to basic computer architecture, followed by chapters describing how computers represent numerical data internally. I will cover things such as integer overflows and underflows; hopefully this could make spotting some of the more exotic bugs easier. We are going to take a somewhat-quick look at how **IEEE** standard floating point values are represented as well.

C and Assembly

The book continues with lower-level programming. We will see examples of special compiler attributes (e.g. **__packed__**, **__aligned__**) that give us more control on our code's behavior. There's a chapter on i386 machine architecture. We'll learn a bit about i386 assembly in general and using it with GCC and the other GNU tools in particular. We will take a look at inline assembly and learn how to implement the **setjmp()** and **longjmp()** standard library functions; these are one of the trickiest parts to implement in a standard library in some ways.

Code Style

There is a part in this book dedicated to **code style** to emphasize it's an important aspect of software development.

Code Optimisation

Code optimisation, as one of the things I'm keen on about programming, has a dedicated set of chapters. We will first take a quick look at some machine features and then roll our sleeves and start looking at how to write fast code. The **Examples** section has a couple of pretty neat graphics algorithms. We implement simple tools to measure the speed of our code in the section **Zen Timer**; the name Zen timer originally came from **Michael Abrash** who has worked on such classic pieces of software as **Quake**.

A Bag of Tricks

As an easter egg to those of you who enjoy coding tricks, there's a chapter called **A Bag of Tricks**. There we take a look at some often quite-creative small techniques gathered from sources such as the legendary MIT **HAKMEM**, the book **Hacker's Delight** by Henry S. Warren of IBM, as well as other books and the Internet. The implementations are by myself and it would be nice to **get comments** on them.

The i386

Next in this book, we shall get deeper into the world of the i386 as well as take a look at its details from the perspective of kernel programmers.

Author's Comments

All in all, I have written about things I learnt during the course of the last decade or so. Instead of being highly theoretical, I tried to write a book which concentrates on 'practical' things, shows some interesting tricks, and perhaps gives you deeper insight to the world of computers. I hope this book makes some of you better programmers.

With that being said, let's start rockin' and a-rollin'! :)

Part IV

Notes on C

Chapter 3

C Types

This section doesn't attempt to be a primer on C types; instead, I cover aspects I consider to be of importance for low-level programming.

3.1 Base Types

The system-specific limits for these types are defined in **<limits.h>**.

It is noteworthy that **you cannot use sizeof() at preprocessing time**. Therefore, system software that depends on type sizes should use explicit-size types or machine/compiler-dependent declarations for type sizes, whichever strategy is feasible for the situation.

TODO: different data models (LP64, LLP64, ...)

Type	Typical Size	Origin
char	8 bits	C89
short	16 bits	C89
int	32 bits	C89
long	32 or 64 bits	C89
long long	64 bits	**C99**; used widely before

Common Assumptions

Please note that the typical sizes are by no means carved in stone, even though such assumptions are made in too many places. The day someone decides to break these assumptions will be judgment day in the software world.

Native Words

Note that the typical size for long tends to be 32 bits on machines with 32-bit [maximum] native word size, 64 bits on 64-bit architectures. Also note that the i386 CPU and later 32-bit CPUs in the Intel-based architectures do support a way to present 64-bit values using two registers. One particular source of problems when porting from 32-bit to 64-bit platforms is the type **int**. It was originally designed to be a 'fast word', but people have used it as 'machine word' for ages; the reason for so many trouble

is that it tends to be 32-bit on 64-bit architectures (as well as 32-bit). Luckily, that's mostly old news; you can use specified-size types introduced in C99.

Char Signedness

The signedness of **char** is compiler-dependent; it's usually a good idea to use **unsigned char** explicitly. These types can be declared **signed** or **unsigned**, as in **unsigned char**, to request the desired type more explicitly. The type **long long** existed before C99 in many compilers, but was only standardised in C99. One problem of the old days was code that wasn't '8-bit clean' because it represented text as [signed] chars. Non-ASCII text presentations caused problems with their character values greater than 127 (0x7f hexadecimal).

3.2 Size-Specific Types

Fewer Assumptions

One of the great things about C99 is that it makes it easier, or, should I say, realistically possible, to work when you have to know sizes of entities. In the low-level world, you basically do this all the time.

3.2.1 Explicit-Size Types

The types listed here are defined in **<stdint.h>**.

The advent of C99 brought us types with explicit widths. The types are named uintW_t for unsigned, and intW_t for signed types, where **W indicates the width of the types in bits**. These types are optional.

Unsigned	Signed
uint8_t	**int8_t**
uint16_t	**int16_t**
uint32_t	**int32_t**
uint64_t	**int64_t**

These types are declared in **<stdint.h>**. There are also macros to declare 32-bit and 64-bit constants; INT32_C(), UINT32_C(), INT64_C() and UINT64_C(). These macros postfix integral values properly, e.g. typically with ULL or UL for 64-bit words.

3.2.2 Fast Types

The types listed here are defined in **<stdint.h>**

The C99 standard states these types to be specified for the fastest machine-types capable of presenting given-size values. The types below are optional.

Unsigned	Signed
uint_fast8_t	**int_fast8_t**
uint_fast16_t	**int_fast16_t**
uint_fast32_t	**int_fast32_t**
uint_fast64_t	**int_fast64_t**

The numbers in the type names express the desired width of values to be represented in bits.

3.2.3 Least-Width Types

The types listed here are defined in **<stdint.h>**

The C99 standard states these types to be specified for the minimum- size types capable of presenting given-size values. These types are optional.

Unsigned	Signed
uint_least8_t	**int_least8_t**
uint_least16_t	**int_least16_t**
uint_least32_t	**int_least32_t**
uint_least64_t	**int_least64_t**

The numbers in the type names express the desired width of values to be represented in bits.

3.3 Other Types

This section introduces common types; some of them are not parts of any C standards, but it might still help to know about them.

Memory-Related

size_t is used to specify sizes of memory objects in bytes. Note that some older systems are said to define this to be a signed type, which may lead to erroneous behavior.

ssize_t is a signed type used to represent object sizes; it's typically the return type for **read()** and **write()**.

ptrdiff_t is defined to be capable of storing the difference of two pointer values.

intptr_t and **uintptr_t** are, respectively, signed and unsigned integral types defined to be capable of storing numeric pointer values. These types are handy if you do arithmetics beyond addition and subtraction on pointer values.

File Offsets

off_t is used to store and pass around file-offset arguments. The type is signed to allow returning negative values to indicate errors. Traditionally, off_t was 32-bit, which lead to trouble with large files (of 2^{31} or more bytes). As salvation, most systems let you activate 64-bit off_t if it's not the default; following is a list of a few possible macros to do it at compile-time.

```
#define _FILE_OFFSET_BITS 64
#define _LARGEFILE_SOURCE 1
#define _LARGE_FILES      1
```

Alternatively, with the GNU C Compiler, you could compile with something like

```
gcc -D_FILE_OFFSET_BITS=64 -o proggy proggy.c
```

3.4 Wide-Character Types

/* **TODO**: write on these */

- **wchar_t**
- **wint_t**

3.5 Aggregate Types

struct and union

Structures and unions are called aggregates. Note that even though it's possible to declare functions that return aggregates, it often involves copying memory and therefore, should usually be avoided; if this is the case, use pointers instead. Some systems pass small structures by loading all member values into registers; here it might be faster to call pass by value with structures. This is reportedly true for 64-bit PC computers.

Aggregates nest; it's possible to have structs and unions inside structs.

To avoid making your structs unnecessarily big, it's often a good idea to group bitfields together instead of scattering them all over the place. It might also be good to organise the biggest-size members first, paying attention to keeping related data fields together; this way, there's a bigger chance of fetching several ones from memory in a single [cacheline] read operation. This also lets possible alignment requirements lead to using fewer padding bytes.

Compilers such as the GNU C Compiler - GCC - allow one to specify structures to be packed. Pay attention to how you group items; try to align each to a boundary of its own size to speed read and write operations up. This alignment is a necessity on many systems and not following it may have critical impact on runtime on systems which don't require it.

Use of global variables is often a bad idea, but when you do need them, it's a good idea to group them inside structs; this way, you will have fewer identifier names polluting the name space.

3.5.1 Structures

struct is used to declare combinations of related members.

3.5.1.1 Examples

struct Example

```
struct list {
    struct listitem *head;
    struct listitem *tail;
};
```

declares a structure with 2 pointers, whereas

Second Example

```
struct listitem {
    unsigned long    val;
    struct listitem *prev;
    struct listitem *next;
};
```

declares a structure with two [pointer] members and a value member; these could be useful for [bidirectional] linked-list implementations.

Structure members are accessed with the operators . and -> in the following way:

```
struct list      list;

/* assign something to list members */

/* list is struct */
struct listitem *item = list.head;
struct listitem *next;

while (item) {
    next = item->next; /* item is pointer */
}
```

3.5.2 Unions

union Example

union is used to declare aggregates capable of holding one of the specified members at a time. For example,

```
union {
    long lval;
    int  ival;
};
```

can have ival set, but setting lval may erase its value. The . and -> operators apply for unions and union-pointers just like they do for structures and structure-pointers.

3.5.3 Bitfields

Bitfield Example

Bitfields can be used to specify bitmasks of desired widths. For example,

```
struct bitfield {
    unsigned mask1 : 15;
    unsigned mask2 : 17;
}
```

declares a bitfield with 15- and 17-bit members. Padding bits may be added in-between mask1 and mask2 to make things align in memory suitably for the platform. Use your compiler's pack-attribute to avoid this behavior.

Portability

Note that bitfields used to be a portability issue (not present or good on all systems). They still pose issues if not used with care; if you communicate bitfields across platforms or in files, be prepared to deal with bit- and byte-order considerations.

3.6 Arrays

Let's take a very quick look on how to declare arrays. As this is really basic C, I will only explain a 3-dimensional array in terms of how its members are located in memory.

3.6.1 Example

3-Dimensional Example

`int tab[8][4][2];`

would declare an array which is, in practice, a flat memory region of $8*4*2*sizeof(int)$; 64 ints, that is. Now

`tab[0][0][0]`

would point to the very first int in that table,

`tab[0][0][1]`

to the int value right next to it (address-wise),

`tab[7][1][0]`

to the int at offset

$(7*4*2+1*4+0*2) = 60$, i.e. the 59th int in the table.

Here is a very little example program to initialise a table with linearly growing values.

```
#include <stdio.h>

int
main(int argc, char *argv[])
```

```
{
    int tab[8][4][2];
    int i, j, k, ndx = 0;

    for (i = 0 ; i < 8 ; i++) {
        for (j = 0 ; j < 4 ; j++) {
            for (k = 0 ; k < 2 ; k++) {
                tab[i][j][k] = ndx++;
            }
        }
    }
}
```

One thing to notice here is that using **index** (or **rindex**) as an identifier name is a bad idea because many UNIX systems define them as macros; I use **ndx instead of index**.

3.7 typedef

C lets one define aliases for new types in terms of existing ones.

3.7.1 Examples

typedef Example

Note that whereas I am using an **uninitialised value** of w, which is undefined, the value doesn't matter as any value's logical XOR with itself is zero. This is theoretically faster than explicit assignment of 0; the instruction XOR doesn't need to pack a zero-word into the instruction, and therefore the CPU can prefetch more adjacent bytes for better pipeline parallelism.

```
typedef long word_t; /* define word_t to long */

word_t w;

w ^= w; /* set w to 0 (zero). */
```

would define word_t to be analogous to long. This could be useful, for example, when implementing a C standard library. Given that LONG_SIZE is defined somewhere, one could do something like

```
#if (LONG_SIZE == 4)
typedef long uint32_t;
#elif (LONG_SIZE == 8)
typedef long uint64_t;
#else
#error LONG_SIZE not set.
#endif
```

There would be other declarations for **<stdint.h>**, but those are beyond the scope of this section.

3.8 sizeof

The **sizeof** operator lets you compute object sizes at compile-time, except for variable-length arrays.

3.8.1 Example

Zeroing Memory

You can initialise a structure to all zero-bits with

```
#include <stat.h>

struct stat statbuf = { 0 };
```

Note that sizeof returns object sizes in bytes.

3.9 offsetof

C99 Operator

ISO C99 added a handy new operator, offsetof. You can use it to compute offsets of members in structs and unions at compile-time.

3.9.1 Example

offsetof Example

Consider the following piece of code

```
#include <stat.h>

size_t szofs;

struct stat statbuf;

szofs = offsetof(statbuf, st_size);
```

This most likely useless bit of code computes the offset of the **st_size** field from the beginning of the **struct statbuf**. Chances are you don't need this kind of information unless you're playing with C library internals.

3.10 Qualifiers and Storage Class Specifiers

3.10.1 const

The **const** qualifier is used to declare read-only data, which cannot be changed using the identifier given. For example, the prototype of strncpy()

const Example

```
char *strncpy(char *dest, const char *src, size_t n);
```

states that src is a pointer to a string whose data strncpy() is not allowed to change.

On the other hand, the declaration

Another Example

```
char *const str;
```

means that str is a constant pointer to a character, whereas

```
char const *str;
```

would be a pointer to a constant character.

It may help you to better understand constant qualifiers by reading them right to left.

3.10.2 static

File Scope

Global identfiers (functions and variables) declared with the static specifier are only visible within a file they are declared in. This may let the compiler optimise the code better as it knows there will be no access to the entities from other files.

Function Scope

The **static** qualifier, when used with automatic (internal to a function) variables, means that the value is saved across calls, i.e. allocated somewhere other than the stack. In practice, you probably want to initialise such variables when declaring them. For example,

static Example

```
#define INIT_SUCCESS 0

#include <pthread.h>

pthread_mutex_t initmtx = PTHREAD_MUTEX_INITIALIZER;

void
proginit(int argc, char *argv[])
{
    static volatile int initialised = 0;
```

```
    /* only run once */
    pthread_mutex_lock(&initmtx);
    if (initialised) {
        pthread_mutex_unlock(&initmtx);

        return;
    }

    /* critical region begins */
    if (!initialised) {
        /* initialise program state */

        initialised = 1;
    }
    /* critical region ends */
    pthread_mutex_unlock(&initmtx);

    return;
}
```

Comments

The listing shows a bit more than the use of the static qualifier; it includes a **critical region**, for which we guarantee single-thread-at-once access by protecting access to the code with a mutex (mutual exclusion lock).

3.10.3 extern

The **extern** specifier let's you introduce entities in other files. It's often a good idea to avoid totally-global functions and variables; instead of putting the prototypes into global header files, if you declare

```
#include <stdint.h>

uintptr_t baseadr = 0xfe000000;

void
kmapvirt(uintptr_t phys, size_t nbphys)
{
    /* FUNCTION BODY HERE */
}
```

in a source file and don't want to make **baseadr** (if you need to use it from other files, it might be better if not) and **kmapvirt()** global, you can do this in other files

```
#include <stdint.h>

extern uintptr_t baseadr;

extern void kmapvirt(uintptr_t, size_t);
```

Note that you don't need argument names for function prototypes; the types are necessary for compiler checks (even though it may not be required).

3.10.4 volatile

Usage

You should declare variables that may be accessed from signal handlers or several threads with the **volatile** specifier to make the compiler check the value every time it is used (and eliminate assumptions by the optimiser that might break such code).

3.10.5 register

Usage

The register storage specifier is used to make the compiler reserve a register for a variable for its whole scope. Usually, it's better to trust the compiler's register allocator for managing registers

3.11 Type Casts

Possible Bottleneck

With integral types, casts are mostly necessary when casting a value to a smaller-width type. If you're concerned about code speed, pay attention to what you're doing; practice has shown that almost-innocent looking typecasts may cause quite hefty performance bottlenecks, especially in those tight inner loops. Sometimes, when making size assumptions on type, casts may actually break your code.

Sometimes it's necessary to cast pointers to different ones. For example,

```
((uintptr_t)u8ptr - (uintptr_t)u32ptr)
```

would evaluate to the distance between *u8ptr and *u32ptr in bytes. Note that the example assumes that

```
(uintptr_t)u8ptr > (uintptr_t)u32ptr
```

More on pointers and pointer arithmetics in the following chapter.

Chapter 4

Pointers

In practice, C pointers are memory addresses. One of the interesting things about C is that it allows access to pointers as numeric values, which lets one do quite 'creative' things with them. Note that when used as values, arrays decay to pointers to their first element; arrays are by no means identical to pointers.

4.1 void Pointers

void * vs. char *

ISO C defines void pointers, **void ***, to be able to assign any pointer to and from without explicit typecasts. Therefore, they make a good type for function arguments. Older C programs typically use **char *** as a generic pointer type.

It's worth mentioning that one cannot do arithmetic operations on void pointers directly.

4.2 Pointer Basics

Basic Examples

As an example, a pointer to values of type **int** is declared like

```
int *iptr;
```

You can access the value at the address iptr like

```
i = *iptr;
```

and the two consecutive integers right after the address iptr like

```
i1 = iptr[1];
i2 = iptr[2];
```

To make iptr point to iptr[1], do

```
iptr++;
```

or

```
iptr = &iptr[1];
```

4.3 Pointer Arithmetics

The C Language supports scaled addition and subtraction of pointer values.

To make iptr point to iptr[1], you can do

```
iptr++; // point to next item; scaled addition
```

or

```
iptr += 1;
```

Similarly, e.g. to scan an array backwards, you can do

```
iptr--;
```

to make iptr point to iptr[-1], i.e. the value right before the address iptr.

To compute the distance [in given-type units] between the two pointer addresses iptr1 and iptr2, you can do

```
diff = iptr2 - iptr1;
```

Note that the result is scaled so that you get the distance in given units instead of what

```
(intptr_t)iptr2 - (intptr_t)iptr1;
```

would result to. The latter is mostly useful in advanced programming to compute the difference of the pointers in bytes.

If you don't absolutely need negative values, it is better to use

```
(uintptr_t)iptr2 - (uintptr_t)iptr1; // iptr2 > iptr1
```

or things will get weird once you get a negative result (it is going to end up equivalent to a big positive value, but more on this in the sections discussing numerical presentation of integral values. Note, though, that this kind of use of the C language may hurt the maintainability and readability of code, which may be an issue especially on team projects.

It is noteworthy that C pointer arithmetics works on table and aggregate (**struct** and **union**) pointers as well. It's the compiler (and sometimes CPU) who scale the arithmetics of operations such as **++** to work properly.

4.4 Object Size

Memory

Pointer types indicate the size of memory objects they point to. For example,

```
uint32_t u32 = *ptr32;
```

reads a 32-bit [unsigned] value at address ptr32 in memory and assigns it to u32.

Results of arithmetic operations on pointers are scaled to take object size in account. Therefore, it's crucial to use proper pointer types [or cast to proper types] when accessing memory.

```
uint32_t *u32ptr1 = &u32;
uint32_t *u32ptr2 = u32ptr1 + 1;
```

makes u32ptr2 point to the uint32_t value right next to u32 in memory.

In C, any pointer [value] can be assigned to and from **void *** without having to do type-casts. For example,

```
void    *ptr = &u32;
uint8_t *u8ptr = ptr;
uint8_t  u8 = 0xff;

u8ptr[0] = u8ptr[1] = u8ptr[2] = u8ptr[3] = u8;
```

would set all bytes in u32 to 0xff (255). Note that doing it this way is better than

```
size_t   n = sizeof(uint32_t);

while (n--) {
    *u8ptr++ = u8;
}
```

both because it avoids loop overhead and data-dependencies on the value of u8ptr, i.e. the [address of] next memory address doesn't depend on the previous operation on the pointer.

Chapter 5

Logical Operations

5.1 C Operators

TODO - had some LaTeX-messups, will fix later. :)

5.1.1 AND

Truth Table

The logical function AND is true when both of its arguments are. The truth table becomes

Bit #1	Bit #2	AND
0	0	0
0	1	0
1	0	0
1	1	1

5.1.2 OR

Truth Table

The logical function OR is true when one or both of its arguments are. Some people suggest OR should rather be called **inclusive OR**.

The truth table for OR is represented as

Bit #1	Bit #2	OR
0	0	0
0	1	1
1	0	1
1	1	1

5.1.3 XOR

Truth Table

The logical function XOR, exclusive OR, is true when exactly one of its arguments is true (1).

The truth table of XOR is

Bit #1	Bit #2	XOR
0	0	0
0	1	1
1	0	1
1	1	0

5.1.4 NOT

Truth Table

The logical function NOT is true when its argument is false.

Bit	NOT
0	1
1	0

5.1.5 Complement

Complementing a value means turning its 0-bits to ones and 1-bits to zeroes; 'reversing' them.

Bit	Complement
0	1
1	0

Chapter 6

Memory

Table of Bytes

C language has a thin memory abstraction. Put short, you can think of memory as a flat table/series of bytes which appears linear thanks to operating system virtual memory.

6.1 Alignment

Many processors will raise an exception if you try to access unaligned memory addresses [using pointers], and even on the ones which allow it, it tends to be much slower than aligned access. The address addr is said to be aligned to n-byte boundary if

```
(adr % n) == 0 /* modulus with n is zero, */
```

i.e. when adr is a multiple of n.

It's worth mentioning that if you need to make sure the pointer ptr is aligned to a boundary of p2, where p2 is a power of two, it's faster to check that

```
/* low bits zero. */
#define aligned(ptr, p2) \
    (!((uintptr_t)ptr & ((p2) - 1)))
```

The type uintptr_t is defined to one capable of holding a pointer value in the ISO/ANSI C99 standard header **<stdint.h>**.

6.2 Word Access

Whereas the most trivial/obvious implementations of many tasks would access memory a byte at a time, for example

```
nleft = n >> 2;
n -= nleft << 2;
```

```
/* unroll loop by 4; set 4 bytes per iteration. */
while (nleft--) {
    *u8ptr++ = u8;
    *u8ptr++ = u8;
    *u8ptr++ = u8;
    *u8ptr++ = u8;
}
/* set the rest of bytes one by one */
while (n--) {
    *u8ptr++ = u8;
}
```

it's better to do something like

```
n32 = n >> 2;   /* n / 4 */
u32 = u8;     /* u8 */
u32 |= u32 << 8;  /* (u8 << 8) | u8 */
u32 |= u32 << 16; /* (u8 << 24) | (u8 << 16) | (u8 << 8) | u8 */
/* set 32 bits at a time. */
while (n32--) {
    *u32ptr++ = u32;
}
```

or even

```
n32 = n >> 4;   /* n / 16 */
u32 = u8;     /* u8 */
u32 |= u32 << 8;  /* (u8 << 8) | u8 */
u32 |= u32 << 16; /* (u8 << 24) | (u8 << 16) | (u8 << 8) | u8 */
/*
 * NOTE: x86 probably packs the indices as 8-bit immediates
 * - eliminates data dependency on previous pointer value
 *   present when *u32ptr++ is used
 */
for (i = 0 ; i < n32 ; i++) {
    u32ptr[0] = u32;
    u32ptr[1] = u32;
    u32ptr[2] = u32;
    u32ptr[3] = u32;
    u32ptr += 4;
}
```

in order to access memory a [32-bit] word at a time. On a typical CPU, this would be much faster than byte-access! Note, though, that this example is simplified; it's assumed that u32ptr is aligned to 4-byte/32-bit boundary and that n is a multiple of 4. We'll see how to solve this using **Duff's device** later on in this book.

The point of this section was just to demonstrate [the importance of] **word-size memory access**; the interesting thing is that this is not the whole story about implementing fast memset() in C; in fact, there's a bunch of more tricks, some with bigger gains than others, to it. We shall explore these in the part **Code Optimisation** of this book.

Chapter 7

System Interface

7.1 Signals

Brief

Signals are a simple form of IPC (Inter-Process Communications). They are asynchronous events used to notify processes of conditions such as arithmetic (SIGFPE, zero-division) and memory access (SIGSEGV, SIGBUS) errors during program execution. Asynchronous means signals may be triggered at any point during the execution of a process. On a typical system, two signals exist for user-defined behavior; SIGUSR1 and SIGUSR2. These can be used as a rough form of communications between processes.

System

Most signal-handling is specific to the host operating system, but due to its widespread use, I will demonstrate UNIX/POSIX signals as well as some simple macro techniques by representing a partial implementation of **<signal.h>**. I will not touch the differences of older signal() and sigaction() here; that belongs to system programming books. As such a book, **Advanced Programming in the UNIX Environment** by late **Richard W. Stevens** is a good text on the topic.

Asynchronosity

It is noteworthy that signal handlers can be triggered at any time; take care to declare variables accessed from them volatile and/or protect them with lock-mechanisms such as mutexes. C has a standard type, **sig_atomic_t**, for variables whose values can be changed in one machine instruction (atomically).

Critical Regions

I will touch the concept of critical regions quickly. A critical region is a piece of code which may be accessed several times at once from different locations (signal handlers or multiple threads). Need arises to protect the program's data structures not to corrupt them by conflicting manipulation (such as linked lists having their head point to wrong item). At this point, it's beyond our interest to discuss how to protect critical regions

beyond using mutexes that are locked when the region is entered, unlocked when left (so as to serialise access to those regions).

Signal Stack

On many UNIX systems, you can set signals to be handled on a separate stack, typically with **sigaltstack()**.

SIGCLD is not SIGCHLD

There is a bit of variation in how signals work on different systems, mostly about which signals are available. This seems mostly irrelevant today, but I'll make one note; the old System V SIGCLD has semantics different from SIGCHLD, so don't mix them (one sometimes sees people redefine SIGCLD as SIGCHLD which should not be done).

7.2 Dynamic Memory

malloc() and Friends

Standard C Library provides a way to manage dynamic memory with the functions malloc(), calloc(), realloc() and free(). As we are going to see in our allocator source code, there's a bunch of other related functions people have developed during the last couple of decades, but I will not get deeper into that here.

Dynamic allocation is a way to ask the system for memory for a given task; say you want to read in a file (for the sake of simplicity, the whole contents of the file). Here's a possible way to do it on a typical UNIX system.

readfile()

```
#include <stdlib.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <fcntl.h>

void *
readfile(char *filename, size_t *sizeret)
{
    void        *buf = NULL;
    struct stat  statbuf;
    size_t       nread;
    size_t       nleft;
    int          fd;

    if (sizeret) {
        *sizeret = 0;
    }
    if (stat(filename, &statbuf) < 0) {

        return NULL;
```

```
    }
    if (!S_ISREG(statbuf.st_mode)) {

        return NULL;
    }
    fd = open(name, O_RDONLY);
    if (fd < 0) {

        return NULL;
    }
    nread = 0;
    nleft = statbuf.st_size;
    if (nleft) {
        buf = malloc(nleft);
        if (buf) {
            while (nleft) {
                nread = read(fd, buf, nleft);
                if (nread < 0) {
                    if (errno == EINTR) {

                        continue;
                    } else {
                        free(buf);

                        return NULL;
                    }
                } else {
                    nleft -= nread;
                }
            }
            if (sizeret) {
                *sizeret = statbuf.st_size;
            }
        }
    }
    close(fd);

    return buf;
}
```

EINTR

to read the file into the just-allocated buffer. Notice how we deal with UNIX-style interrupted system calls by checking the value of errno against EINTR; should it occur that a read system call is interrupted, we'll just continue the loop to read more.

An implementation of malloc() and other related functions is presented later in this book.

7.2.1 Heap

sbrk() and brk()

Heap segment is where the traditional albeit POSIX-undefined brk() (kernel) and sbrk() (C library) dynamic memory interface operates. Note that sbrk() is merely a wrapper to a [simple] **brk system call**.

sbrk() not in POSIX

Back in the old days, dynamic memory was implemented with the brk() system call (often using the sbrk() standard library wrapper). All it really does is adjust the offset of the top of the heap. This seems to be recognised as a somewhat too rudimentary or dated hack by POSIX; they have deliberately excluded sbrk() from their standard. Note that this doesn't mean sbrk() would not be available on your system; it most likely is.

sbrk() + mmap()

In reality, malloc-allocators today use only mmap() or a mix of mmap() and sbrk(). It's a compromise between speed and ease of use, mostly (the kernel does the book-keeping for mmap() and it tends to be thread-safe, i.e. reentrant, too).

7.2.2 Mapped Memory

Files

Modern allocators use mmap() + munmap() [from POSIX] to manage some of their allocations. The C library provides an interface for this as a special case of mapping files; files can be memory-mapped, giving us a fast way to write and read data to and from file-systems. The special case is to allocate zeroed regions not belonging to files.

Anonymous Maps

As a detail, **mmap()** can often be used to allocate anonymous (zeroed) memory. I know of two strategies to how to implement this; map **/dev/zero** or use the **MAP_ANON** flag.

Chapter 8

C Analogous to Assembly

8.1 'Pseudo-Assembly'

Some of us call C pseudo-assembly. C is very close to the machine and most if not all machine language dialects have a bunch of instructions to facilitate fast implementation of C code execution. This is probably not a coincidence.

In this chapter, I shall try to explain the elegant simplicity of C as well as its close relationship with the machine; keep in mind assembly is just symbolic machine language. Hence assembly makes a great tool for explaining how C code utilises the machine.

Here is a simple mapping of common C operations to pseudo-machine instructions.

8.1.1 Pseudo Instructions

Hypothetical Instructions

C Operation	Mnemonic
&	AND
|	OR
^	XOR
~	NOT
++	INC
–	DEC
+	ADD
-	SUB
*	MUL, IMUL
/	DIV, IDIV
%	MOD

Conditional Jumps

C Comparison	Test	Brief
N/A	JMP	Jump Unconditionally
!	JZ	Jump if Zero
(x)	JNZ	Jump if Not Zero
<	JLT	Jump if Less Than
<=	JLTE	Jump if Less Than or Equal
>	JGT	Jump if Greater Than
>=	JGTE	Jump if Greater Than or Equal

8.2 Addressing Memory

Pointer Operations

C Syntax	Function
*	dereference pointer
&	get object address/pointer
[]	access member of array or bitfield using index
.	access member of struct or union
->	access member of struct or union using pointer
variable	value in register or memory
array	contiguous region of memory
constant	register or immediate (in-opcode) values

8.3 C to Assembly/Machine Translation

Here we shall take a look at how C code is translated to assembly by a hypothetical compiler. I will also make notes on some things one can do to speed code up.

Note that in this section, I mix **variable** names and **register** names in the pseudo-assembly code; in real life, this would mean memory locations and registers, but better code would be based purely on using registers. I chose this convention for the sake of readability. Some examples actually do use registers only for a bit closer touch to the real machine.

8.3.1 Branches

8.3.1.1 if - else if - else

It pays to put the most likely test cases first. CPUs do have branch prediction logic, but they still need to fetch code and jump around in it, which defeits their internal execution pipelines and consumes cache memory.

Let's see what's going on in a piece of C code; I will use pseudo-code in a form similar to C and assembly to explain how this code could be translated at machine level.

C Code

```
#1: if (a < b) {
        ;
#2: } else if (b < c && b > d) {
        ;
#3: } else if (c > a || d <= a) {
        ;
#4: } else {
        ;
#5: }
```

Now let's look at what's going on under the hood.

Pseudo Code

```
CMP a, b        ; line #1
JGE step2       ; branch to step2 if !(a < b)

/* code between #1 and #2 */

JMP done
step2:
CMP b, c        ; line #2
JGE step3       ; branch to step3 if !(b < c)
CMP b, d        ; line #2
JLE step3       ; branch to step3 if !(b > d)

/* code between #2 and #3 */

JMP done
step3:
CMP c, a        ; line #3
JLT step4       ; branch to step4 if !(c > a)
CMP d, a        ; #line 3
JGT step4       ; brach to step4 if !(d <= a)

/* code between #3 and #4 */

JMP done
step4:

/* code between #4 and #5 */

done:           ; done
```

8.3.1.2 switch

One useful way to replace switch statements with relatively small sets of integral keys is to use function pointer tables indexed with key values. The construct

```
switch (a) {
    case 0:
```

```
            a++;

            break;
        case 1:
            a += b;

            break;
        default:

            break;
    }
```

could be translated to

```
    TEST a          ; set flags
    JZ label0       ; branch to label0 if (a == 0)
    CMP a, $1       ; compare a and 1
    JE label1       ; branch to label1 if equal
    JMP done        ; default; jump over switch
    label0:
    INC a           ; a++;
    JMP done        ; done
    label1:
    ADD b, a        ; a += b;
    done:           ; done
```

This should often be **faster than if- else if - else** with several test conditionals. With suitable case values (small integrals), a good compiler might know how to convert this construct to a jump table; check the value of a and jump to a location indexed by it.

8.3.2 Loops

8.3.2.1 for

The C snippet

```
    long *tab = dest;

    for (i = 0 ; i < n ; i++) {
        tab[i] = 0;
    }
```

could be translated as

```
    MOV $0, %EAX          ; i = 0;
    loop:
    CMP %EAX,n            ; compare i and n
    JE  done              ; done if (i == n)
    MOV $0, %EAX(tab, 4)  ; tab[i] = 0; scale i by 4
    INC %EAX              ; i++;
    JMP loop              ; iterate
    done:
```

Note that I used a register for loop counter to speed things up, even though I have mixed variables and registers in the pseudo-code freely. This is more likely what a real compiler would do. Some architectures also have prethought looping instructions which might use a specified register for the loop counter, for example. IA-32 has REP-functionality.

Indexed Scaled Addressing

The line

```
MOV $0, %EAX(tab, 4); tab[i] = 0;
```

means "move 0 to the location at tab + %EAX * 4, i.e. i (%EAX) is scaled by 4, i.e. sizeof(long) on 32-bit architecture.

8.3.2.2 while

The C loop construct

```
int i = NBPG >> 2;
long *tab = dest;

while (i--) {
    *tab++ = 0;
}
```

Could work (with 4-byte longs) as

```
        MOV tab, %EAX   ; move address to register
MOV $NBPG, %EBX ; move NBPG to register (i)
SHR $2, %EBX ; shift NBPG right by 2
        loop:
        TEST %EBX
        JZ done
        DEC %EBX
        MOV $0, *%EAX
        ADD $4, %EAX
        JMP loop
        done:
```

In this example, I used registers to contain the memory address and loop counter to speed the code up.

8.3.2.3 do-while

The difference between while and do-while is that do-while always iterates the body of the loop at least once.

The C code

```
int i = NBPG >> 2;
long *tab = dest;
```

```
        do {
            *tab++ = 0;
        } while (--i);
```

can be thought of as

```
        MOV tab, %EAX
        MOV NBPG, %EBX
SHR $2, %EBX
        loop:
        MOV  $0, *%EAX
        ADD  $4, %EAX
        DEC  %EBX
        TEST %EBX
        JNZ  loop
        done:
```

In this example, I allocated registers for all variables like a good compiler should do.

8.3.3 Function Calls

Function call mechanisms used by C are explained in detail in the chapter **C Run Model** in this book.

For now, suffice it to illustrate the i386 calling convention quickly.

A function call of

```
        foo(1, 2, 3);

        int foo(int arg1, int arg2, int arg3) {
            return (arg1 + arg2 + arg3);
        }
```

Can be thought of as

Pseudo Code

```
        /* function call prologue */
        PUSH $3         ; push arguments in reverse order
        PUSH $2
        PUSH $1
        PUSH %EIP       ; push instruction pointer
        PUSH %EBP       ; current frame pointer
        MOV  %EBP, %ESP ; set stack pointer
        JMP  foo

        /* stack
         * -----
         * arg3
         * arg2
         * arg1
         * retadr
```

```
 * prevfp; <- %ESP and %EBP point here
 */

foo:
MOV  8(%ESP), %EAX
MOV  12(%ESP), %EBX
MOV  16(%ESP, %ECX
ADD  %EBX, %EAX
ADD  %ECX, %EAX ; return value in EAX
MOV  %EBP, %ESP ; return frame

/* return from function */
POP  %EBP       ; pop old frame pointer
POP  %EBX       ; pop old EIP value
MOV  %EBP, %ESP ; set stack pointer to old frame
JMP  *%EBX      ; jump back to caller
```

Notes

foo() doesn't have internal variables, so we don't need to adjust stack pointer [or push values] on entry to and at leaving the function.

Call Conventions

Usually, CPUs have specific instructions to construct call frames as well as return from functions. It was done here by hand to demonstrate the actions involved.

Chapter 9

C Run Model

C is a low-level language. As such, it doesn't assume much from the host operating system and other support software. The language and its run model are elegant and simple. C is powerful for low-level software development; situations where you really have to write assembly-code are rare and few, yet C gives one a close interface to low-level machine facilities with reasonable portability to different hardware platforms.

Stack, Memory, Registers

In short, C code execution is typically based on values stored on the stack, elsewhere in memory, and machine registers. Note, however, that stack as well as heaps are not language features, but rather details of [typical] implementations.

Stack is allocated from process virtual memory and from physical memory as dictated by the kernel and page-daemon. Other than move data to registers, there's little you can do to affect stack operation with your code.

Memory is anything from the lowest-level (closest-to-CPU) cache to main physical memory. Utilising caches well tends to pay back in code speed. For one thing, it's good to keep related data on as few cachelines [and pages] as possible.

Registers are the fastest-access way to store and move data around in a computer. All critical variables such as loop counters should have registers allocated for them. It's noteworthy to avoid the **C keyword register**, as that is said to allocate a register for the whole lifetime/scope of the function and is therefore to be considered wasteful. Note, however, that as per C99, the specifier register is only a suggestion "that access to the obejct should be as fast as possible."

Use of the register-specifier may be helpful in situations where few or no compiler optimisations are in effect; on the other hand, it is noteworthy that compiler optimisations often make debugging hard if not impossible.

9.1 Code Execution

9.1.1 Program Segments

Code and Data

I will provide a somewhat-simplified, generalised view to how UNIX systems organise program execution into a few segments.

STACK	execution data, function interface
DYN	mapped regions
BSS	runtime data; allocated; zeroed; **heap**
DATA	initialised data
RODATA	read-only data
TEXT	program code

9.1.1.1 Minimum Segmentation

Here I describe a minimal set of i386 memory segments

Flat Memory Model

- Use one text segment (can be the same for **TEXT and RODATA**).
- Use one read-write data segment (for **DATA, BSS, and DYN** segments.
- Use one read-write, expand-down data segment for **STACK** segment.

This flat model is useful for, e.g., **kernel development**. Most other programmers don't need to be concerned about segmentation.

9.1.2 TEXT Segment

Program Code

The TEXT segment contains program code and should most of the time be set to read-only operation to avoid code mutations as well as tampering with process code (for exploits and other malware as an example).

9.1.3 RODATA Segment

Read-Only Data

Storage for items such as initialised constant variables and perhaps string literals (better not try to overwrite them). Read-only data segment.

9.1.4 DATA Segment

Initialised Data

Storage mostly for initialised global entities and probably static variables. Can be both read and written.

9.1.5 BSS Segment

Allocated Data

The name of the BSS Segment originates from PDP-assembly and stood for Block Started by Symbol. This is where runtime-allocated data structures i.e. uninitialised global structures [outside C functions] are reserved. This is also where dynamic memory allocation (malloc(), calloc()) takes place if you use the traditional system library wrapper **sbrk()** to set so-called program break. In practice, the break is a pointer to the byte right after the current top of heap. This segment should be filled with zero bytes at program load time.

Zeroed Memory

Note that when new physical pages are mapped to virtual space of user processes, it needs to be zeroed not to accidentally slip confidential data such as, in one of the worst cases, uncrypted passwords. This task is frequently done to freed/unmapped pages **by system kernels**.

9.1.6 DYN Segment

Mapped Regions

The implementation details vary system to system, but this segment's purpose here is to emphasize the usage of **mmap()** to map zeroed memory. This could be facilitated by having the heap (**BSS**) live low in memory and DYN far above it to minimalise the risk of the segments overrunning each other. I plan to try to put this segment right below stack and make it start mapping memory from the top in my kernel project.

9.1.7 STACK Segment

Routines & Variables

The stack is a region of memory, usually in the high part of the virtual address space (high memory addresses). Conventionally, the segment 'grows down', i.e. values are pushed into memory and the stack pointer decremented instead of the common linear order memory access where pointers are incremented and dereferenced with linearly growing addresses. This allows the stack to be located high and work in concert with dynamic memory segments lower in address space (the heap grows upwards, address-wise; mapped regions may be located under the stack).

The stack exists to implement function call interface, automatic variables and other aspects of C's run model.

9.2 C Interface

9.2.1 Stack

9.2.1.1 Stack Pointer

Stack pointer is a [register] pointer to the current item on the stack. On i386-based machine architecture, the stack pointer is stored in the ESP-register. When a value is popped from the stack, it is first taken from the address in the stack pointer, and then the stack pointer is incremented by 4 bytes; ESP points to the current location on the stack. Pushing values works by decrementing the stack pointer to point to the next location, then storing the value at the address pointed to by the stack pointer. I'll clarify this by describing the operations in C-like code

```
#define pop()     (*(sp++))
#define push(val) (*(--sp) = (val))
```

9.2.2 Frame Pointer

Frame pointer points to the stack frame of the current function. This is where the frame pointer value for the caller, the return address for the proper instruction in it, and the arguments the current function was called with are located. On i386, the frame pointer is stored in the EBP-register. Many compilers provide an optimisation to omit using the frame pointer; beware that this optimisation can break code that explicitly relies on the frame pointer.

9.2.3 Program Counter aka Instruction Pointer

Program Counter is a traditional term for instruction pointer [register]. This is the address register used to find the next instruction in memory. On i386, this pointer is stored in the EIP- register (and cannot be manipulated without a bit of trickery). At machine level, when exceptions/interrupts such as zero-division occur, EIP may point to the instruction that caused the exception or the instruction right after it in memory (to allow the program to be restarted after handling the interrupt.

9.2.4 Automatic Variables

Variables within function bodies are called automatic because the compiler takes care of their allocation on the stack.

It is noteworthy that unless you initialise (set) these variables to given values, they contain whatever is in that location in [stack] memory and so unlogical program behavior may happen if you use automatic variables without initialising them first. Luckily, compilers can be configured to warn you about use of uninitialised variables if they don't do it by default.

9.2.5 Stack Frame

On the i386, a stack frame looks like (I show the location of function parameters for completeness' sake).

```
/* IA32. */
struct frame {
            /* internal variables */
    int32_t ebp; /* 'top' */
    int32_t eip; /* return address */
            /*
             * function call arguments in
             * reverse order
             */
};
```

Note that in structures, the lower-address members come first; the instruction pointer is stored before (push) and so above the previous frame pointer.

9.2.6 Function Calls

A function call in typical C implementations consists roughly of the following parts. If you need details, please study your particular implementation.

- push function arguments to stack in reverse order (right to left) of declaration.
- store the instruction pointer value for the next instruction to stack.
- push the current value of the frame pointer to stack.
- set frame pointer to value of stack pointer
- adjust stack pointer to reserve space for automatic variables on stack.

In pseudo-code:

```
push(arg3)
push(arg2)
push(arg1)
push(EIP)    // return address
push(EBP)    // callee frame
mov(ESP, EBP)    // set frame pointer
add(sizeof(autovars), ESP) // adjust stack
```

A hypothetical call

```
foo(1, 2, 3);
```

would then leave the bottom of the stack look something like this:

Value	Explanation	Notes
3	argument #3	
2	argument #2	
1	argument #1	
eip	return address to caller	next instruction after return
ebp	frame pointer for caller	EBP points here
val1	first automatic variable on stack	
val2		
val3		ESP points here

It's noteworthy than unless you explicitly initialise them, the values on stack (v1, v2, and v3) contain 'garbage', i.e. any data that has or has not been written to memory addresses after system bootstrap.

9.2.6.1 Function Arguments

TODO: distinction between function arguments and parameters

Stack or Registers

Function arguments can be variables either on the stack or in registers.

As an example, let's take a quick look at how FreeBSD implements system calls.

The first possibility is to push the function arguments [in reverse order] to the call stack. Then one would load the EAX-register with the system call number, and finally trigger INT 80H to make the kernel do its magic.

Alternatively, in addition to EAX being loaded with the system call number, arguments can be passed in EBX, ECX, EDX, ESI, EDI, and EBP.

9.2.6.2 Return Value

EAX:EDX

The traditional i386 register for return values is EAX; 64-bit return values can be implementing by loading EDX with the high 32 bits of the value. FreeBSD is said to store the return value for SYS_fork in EDX; perhaps this is to facilitate the special 'two return-values' nature of fork() without interfering with other use of registers in C libraries and so on.

C-language error indicator, **errno**, can be implemented by returning the proper number in EAX and using other kinds of error indications to tell the C library that an error occurred. The FreeBSD convention is to set the carry flag (the CF-bit in EFLAGS). Linux returns signed values in EAX to indicate errors.

32-bit words are getting small today. Notably, this shows as several versions of seek(); these days, disks and files are large and you may need weird schemes to deal with offsets of more than 31 or 32 bits. Therefore, **off_t** is 64-bit signed to allow bigger file offsets and sizes. Linux **llseek()** passes the seek offset (off_t) in two registers on 32-bit systems.

9.2.6.3 i386 Function Calls

i386 Details

Let's take a look at how C function calls are implemented on the i386.

To make your function accessible from assembly code, tell GCC to give it 'external linkage' by using stack (not registers) to pass arguments.

```
#include <stdio.h>
#include <stdlib.h>

#define ALINK __attribute__ ((regparm(0)))

ALINK
void
hello(char *who, char *prog, int num1, int num2)
{
    int32_t res;

    res = num1 + num2;

    printf("hello %s\n", who);
    printf("%s here\n", prog);
    printf("%d + %d is equal to %d\n",\
           num1, num2, res);

    return;
}

int
main(int argc, char *argv[])
{
    hello(argv[1], argv[0], 1, 2);

    exit(0);
}
```

This program takes a single command-line argument, your name, and uses the convention that the first argument (**argv[0]** is name of executable (including the supplied path) and the second argument **argv[1]** is the first command line argument (the rest would follow if used, but they are ignored as useless).

When hello() is called, before entry to it, GCC arranges equivalent of

```
push num2
push num1
push argv[0]
push argv[1]
```

Note that the arguments are pushed in 'reverse order'. It makes sense thinking of the fact that now you can pop them in 'right' order.

At this stage, the address of the machine instruction right before the system call is stored; in pseudo-code,

```
pushl %retadr
```

Next, the compiler arranges a stack frame.

```
pushl %ebp # frame pointer
movl %ebp, %esp # new stack frame
```

Note that EBP stores the frame pointer needed to return from the function so it's generally not a good idea to use EBP in your code.

Now it is time to allocate automatic variables, i.e. variables internal to a function that are not declared static. The compiler may have done the stack adjustment in the previous listing and this allocation with the ENTER machine instruction, but if not so, it adds a constant to the stack pointer here, for example

```
addl $0x08, %esp # 2 automatic variables
```

Note that the stacks operates in 32-bit mode, so for two automatic (stack) variables, the adjustment becomes 8.

This may be somewhat hairy to grasp, so I will illustrated it C-style.

You can think of the frame as looking like this at this point.

```
/*
 * EBP points to oldfp.
 * ESP is &avar[-2].
 */
struct cframe {
    int32_t avar[0]; // empty; not allocated
    int32_t oldfp; // frame of caller
    int32_t retadr; // return address
    int32_t args[0]; // empty
}
```

Empty Tables

Note the empty tables (size 0), which C99 actually forbids. These are used as place holders (don't use up any room in the struct); they are useful to pass stack addresses in this case.

Illustration

Finally, I will show you how the stack looks like in plain English. Note that in this illustration, memory addresses grow upwards.

Stack	Value	Explanation
num2	0x00000002	value 2
num1	0x00000001	value 1
argv[1]	pointer	second argument
argv[0]	pointer	first argument
retadr	return address	address of next instruction after return
oldfp	caller frame	EBP points here
res	**undefined**	automatic variable

Return Address

If you should need the return address in your code, you can read it from the stack at address EBP + 4. In C and a bit of (this time, truly necessary) assembly, this could be done like this.

```
struct _stkframe {
    int32_t oldfp; // frame of caller
    int32_t retadr; // return address
};

void
dummy(void)
{
    struct _stkframe *frm;

    __asm__ ("movl %%ebp, %0" : "=rm" (frm));
    fprintf(stderr, "retadr is %x\n", frm->retadr);

    return;
}
```

Note that on return from functions the compiler arranges, in addition to the other magic, something like this.

```
popl %ebp
movl %ebp, %esp
ret
```

Callee-Save Registers

By convention, the following registers are 'callee-save', i.e. saved before entry to functions (so you don't need to restore their values by hand).

Registers
EBX
EDI
ESI
EBP
DS
ES
SS

9.3 Nonlocal Goto; setjmp() and longjmp()

In C, **<setjmp.h>** defines the far-jump interface. You declare a buffer of the type **jmp_buf**

```
jmp_buf jbuf;
```

This buffer is initialised to state information needed for returning to the current location in code (the instruction right after the call to setjmp()) like this:

```
if (!setjmp(jbuf)) {
    dostuff();
}
/* continue here after longjmp() */
```

Then, to jump back to call dostuff(), you would just

```
longjmp(jbuf, val);
```

9.3.1 Interface

9.3.1.1 <setjmp.h>

Here's our C library header file **<setjmp.h>**

```
#ifndef __SETJMP_H__
#define __SETJMP_H__

#if defined(__x86_64__) || defined(__amd64__)
#include <x86-64/setjmp.h>
#elif defined(__arm__)
#include <arm/setjmp.h>
#elif defined(__i386__)
#include <ia32/setjmp.h>
#endif

typedef struct _jmpbuf jmp_buf[1];

/* ISO C prototypes. */
int  setjmp(jmp_buf env);
void longjmp(jmp_buf env, int val);

/* Unix prototypes. */
int  _setjmp(jmp_buf env);
void _longjmp(jmp_buf env, int val);

#endif /* __SETJMP_H__ */
```

9.3.2 Implementation

Note that you need to disable some optimisations in order for setjmp() and longjmp() to build and operate correctly. With the GNU C Compiler, this is achieved by using the **-fno-omit-frame-pointer** compiler flag.

9.3.2.1 IA-32 implementation

Following is an implementation of the setjmp() and longjmp() interface functions for the IA-32 architecture. Note that the behavior of non-volatile automatic variables within the caller function of setjmp() may be somewhat hazy and undefined.

ia32/setjmp.h

```
#ifndef __IA32_SETJMP_H__
#define __IA32_SETJMP_H__

#include <stddef.h>
#include <stdint.h>
#include <signal.h>

#include <zero/cdecl.h>

struct _jmpbuf {
    int32_t  ebx;
    int32_t  esi;
    int32_t  edi;
    int32_t  ebp;
    int32_t  esp;
    int32_t  eip;
#if (_POSIX_SOURCE)
    sigset_t sigmask;
#elif (_BSD_SOURCE)
    int      sigmask;
#endif
} PACK();

struct _jmpframe {
    int32_t ebp;
    int32_t eip;
    uint8_t args[EMPTY];
} PACK();

/*
 * callee-save registers: ebx, edi, esi, ebp, ds, es, ss.
 */

#define __setjmp(env)                                                   \
    __asm__ __volatile__ ("movl %0, %%eax\n"                            \
                          "movl %%ebx, %c1(%%eax)\n"                    \
                          "movl %%esi, %c2(%%eax)\n"                    \
                          "movl %%edi, %c3(%%eax)\n"                    \
                          "movl %c4(%%ebp), %%edx\n"                    \
                          "movl %%edx, %c5(%%eax)\n"                    \
                          "movl %c6(%%ebp), %%ecx\n"                    \
                          "movl %%ecx, %c7(%%eax)\n"                    \
```

```
                        "leal %c8(%%ebp), %%edx\n"                  \
                        "movl %%edx, %c9(%%eax)\n"                  \
                        :                                           \
                        : "m" (env),                                \
                        "i" (offsetof(struct _jmpbuf, ebx)),        \
                          "i" (offsetof(struct _jmpbuf, esi)),      \
                          "i" (offsetof(struct _jmpbuf, edi)),      \
                          "i" (offsetof(struct _jmpframe, ebp)),    \
                          "i" (offsetof(struct _jmpbuf, ebp)),      \
                          "i" (offsetof(struct _jmpframe, eip)),    \
                          "i" (offsetof(struct _jmpbuf, eip)),      \
                          "i" (offsetof(struct _jmpframe, args)),   \
                          "i" (offsetof(struct _jmpbuf, esp))       \
                        : "eax", "ecx", "edx")

#define __longjmp(env, val)                                         \
    __asm__ __volatile__ ("movl %0, %%ecx\n"                        \
                        "movl %1, %%eax\n"                          \
                        "cmp $0, %eax\n"                            \
                        "jne 0f\n"                                  \
                        "movl $1, %eax\n"                           \
                        "0:\n"                                      \
                        "movl %c2(%%ecx), %%ebx"                    \
                        "movl %c3(%%ecx), %%esi"                    \
                        "movl %c4(%%ecx), %%edi"                    \
                        "movl %c5(%%ecx), %%ebp"                    \
                        "movl %c6(%%ecx), %%esp"                    \
                        "movl %c7(%%ecx), %%edx"                    \
                        "jmpl *%edx\n"                              \
                        :                                           \
                        : "m" (env),                                \
                          "m" (val),                                \
                          "i" (offsetof(struct _jmpbuf, ebx)),      \
                          "i" (offsetof(struct _jmpbuf, esi)),      \
                          "i" (offsetof(struct _jmpbuf, edi)),      \
                          "i" (offsetof(struct _jmpbuf, ebp)),      \
                          "i" (offsetof(struct _jmpbuf, esp)),      \
                          "i" (offsetof(struct _jmpbuf, eip))       \
                        : "eax", "ebx", "ecx", "edx",               \
                          "esi", "edi", "ebp", "esp")

#endif /* __IA32_SETJMP_H__ */
```

9.3.2.2 X86-64 Implementation

x86-64/setjmp.h

```
/*
 * THANKS
 * ------
 * - Henry 'froggey' Harrington for amd64-fixes
 * - Jester01 and fizzie from ##c on Freenode
 */

#ifndef __X86_64_SETJMP_H__
#define __X86_64_SETJMP_H__

#include <stddef.h>
#include <stdint.h>
//#include <signal.h>
#include <zero/cdecl.h>

//#include <mach/abi.h>


struct _jmpbuf {
    int64_t  rbx;
    int64_t  r12;
    int64_t  r13;
    int64_t  r14;
    int64_t  r15;
    int64_t  rbp;
    int64_t  rsp;
    int64_t  rip;
#if (_POSIX_SOURCE)
//    sigset_t sigmask;
#elif (_BSD_SOURCE)
//    int      sigmask;
#endif
} PACK();

struct _jmpframe {
    int64_t rbp;
    int64_t rip;
    uint8_t args[EMPTY];
} PACK();

/*
 * callee-save registers: rbp, rbx, r12...r15
 */

#define __setjmp(env)                                                   \
    __asm__ __volatile__ ("movq %0, %%rax\n"                            \
                          "movq %%rbx, %c1(%%rax)\n"                    \
                          "movq %%r12, %c2(%%rax)\n"                    \
                          "movq %%r13, %c3(%%rax)\n"                    \
                          "movq %%r14, %c4(%%rax)\n"                    \
```

```
                        "movq %%r15, %c5(%%rax)\n"                  \
                        "movq %c6(%%rbp), %%rdx\n"                  \
                        "movq %%rdx, %c7(%%rax)\n"                  \
                        "movq %c8(%%rbp), %%rcx\n"                  \
                        "movq %%rcx, %c9(%%rax)\n"                  \
                        "leaq %c10(%%rbp), %%rdx\n"                 \
                        "movq %%rdx, %c11(%%rax)\n"                 \
                        :                                           \
                        : "m" (env),                                \
                          "i" (offsetof(struct _jmpbuf, rbx)),      \
                          "i" (offsetof(struct _jmpbuf, r12)),      \
                          "i" (offsetof(struct _jmpbuf, r13)),      \
                          "i" (offsetof(struct _jmpbuf, r14)),      \
                          "i" (offsetof(struct _jmpbuf, r15)),      \
                          "i" (offsetof(struct _jmpframe, rbp)),    \
                          "i" (offsetof(struct _jmpbuf, rbp)),      \
                          "i" (offsetof(struct _jmpframe, rip)),    \
                          "i" (offsetof(struct _jmpbuf, rip)),      \
                          "i" (offsetof(struct _jmpframe, args)),   \
                          "i" (offsetof(struct _jmpbuf, rsp))       \
                        : "rax", "rcx", "rdx")

#define __longjmp(env, val)                                         \
    __asm__ __volatile__ ("movq %0, %%rcx\n"                        \
                        "movq %1, %%rax\n"                          \
                        "movq %c2(%%rcx), %%rbx\n"                  \
                        "movq %c3(%%rcx), %%r12\n"                  \
                        "movq %c4(%%rcx), %%r13\n"                  \
                        "movq %c5(%%rcx), %%r14\n"                  \
                        "movq %c6(%%rcx), %%r15\n"                  \
                        "movq %c7(%%rcx), %%rbp\n"                  \
                        "movq %c8(%%rcx), %%rsp\n"                  \
                        "movq %c9(%%rcx), %%rdx\n"                  \
                        "jmpq *%%rdx\n"                              \
                        :                                           \
                        : "m" (env),                                \
                          "m" (val),                                \
                          "i" (offsetof(struct _jmpbuf, rbx)),      \
                          "i" (offsetof(struct _jmpbuf, r12)),      \
                          "i" (offsetof(struct _jmpbuf, r13)),      \
                          "i" (offsetof(struct _jmpbuf, r14)),      \
                          "i" (offsetof(struct _jmpbuf, r15)),      \
                          "i" (offsetof(struct _jmpbuf, rbp)),      \
                          "i" (offsetof(struct _jmpbuf, rsp)),      \
                          "i" (offsetof(struct _jmpbuf, rip))       \
                        : "rax", "rbx", "rcx", "rdx",               \
                          "r12", "r13", "r14", "r15",               \
                          "rsp")

#endif /* __X86_64_SETJMP_H__ */
```

9.3.2.3 ARM Implementation

arm/setjmp.h

```
#ifndef __ARM_SETJMP_H__
#define __ARM_SETJMP_H__

#include <stddef.h>
#include <stdint.h>
#include <signal.h>

#include <zero/cdecl.h>

#if 0 /* ARMv6-M */

/* THANKS to Kazu Hirata for putting this code online :) */

struct _jmpbuf {
    int32_t r4;
    int32_t r5;
    int32_t r6;
    int32_t r7;
    int32_t r8;
    int32_t r9;
    int32_t r10;
    int32_t fp;
    int32_t sp;
    int32_t lr;
#if (_POSIX_SOURCE)
    sigset_t sigmask;
#elif (_BSD_SOURCE)
    int      sigmask;
#endif
} PACK();

#define __setjmp(env)                                                   \
    __asm__ __volatile__ ("mov r0, %0\n" : : "r" (env));                \
    __asm__ __volatile__ ("stmia r0!, { r4 - r7 }\n");                  \
    __asm__ __volatile__ ("mov r1, r8\n");                              \
    __asm__ __volatile__ ("mov r2, r9\n");                              \
    __asm__ __volatile__ ("mov r3, r10\n");                             \
    __asm__ __volatile__ ("mov r4, fp\n");                              \
    __asm__ __volatile__ ("mov r5, sp\n");                              \
    __asm__ __volatile__ ("mov r6, lr\n");                              \
    __asm__ __volatile__ ("stmia r0!, { r1 - r6 }\n");                  \
    __asm__ __volatile__ ("sub r0, r0, #40\n");                         \
    __asm__ __volatile__ ("ldmia r0!, { r4, r5, r6, r7 }\n");           \
```

```
    __asm__ __volatile__ ("mov r0, #0\n");                              \
    __asm__ __volatile__ ("bx lr\n")

#define __longjmp(env, val)                                                 \
    __asm__ __volatile__ ("mov r0, %0\n" : : "r" (env));                \
    __asm__ __volatile__ ("mov r1, %0\n" : : "r" (val));                \
    __asm__ __volatile__ ("add r0, r0, #16\n");                         \
    __asm__ __volatile__ ("ldmia r0!, { r2 - r6 }\n");                  \
    __asm__ __volatile__ ("mov r8, r2\n");                              \
    __asm__ __volatile__ ("mov r9, r3\n");                              \
    __asm__ __volatile__ ("mov r10, r4\n");                             \
    __asm__ __volatile__ ("mov fp, r5\n");                              \
    __asm__ __volatile__ ("mov sp, r6\n");                              \
    __asm__ __volatile__ ("ldmia r0!, { r3 }\n");                       \
    __asm__ __volatile__ ("sub r0, r0, #40\n");                         \
    __asm__ __volatile__ ("ldmia r0!, { r4 - r7 }\n");                  \
    __asm__ __volatile__ ("mov r0, r1\n");                              \
    __asm__ __volatile__ ("moveq r0, #1\n");                            \
    __asm__ __volatile__ ("bx r3\n")

#endif /* 0 */

struct _jmpbuf {
    int32_t r4;
    int32_t r5;
    int32_t r6;
    int32_t r7;
    int32_t r8;
    int32_t r9;
    int32_t r10;
    int32_t fp;
    int32_t sp;
    int32_t lr;
    sigset_t sigmask;
} PACK();

#define __setjmp(env)                                                   \
    __asm__ __volatile__ ("movs r0, %0\n"                               \
                          "stmia r0!, { r4-r10, fp, sp, lr }\n"         \
                          "movs r0, #0\n"                               \
                          :                                             \
                          : "r" (env))

#define __longjmp(env, val)                                             \
    __asm__ __volatile__ ("movs r0, %0\n"                               \
                          "movs r1, %1\n"                               \
                          "ldmia r0!, { r4-r10, fp, sp, lr }\n"         \
                          "movs r0, r1\n"                               \
                          "moveq r0, #1\n"                              \
                          "bx lr\n"                                     \
```

```
                        :                                               \
                        : "r" (env), "r" (val))


#endif /* __ARM_SETJMP_H__ */
```

9.3.3 setjmp.c

```
#include <signal.h>
#include <setjmp.h>
#include <zero/cdecl.h>

#if defined(ASMLINK)
ASMLINK
#endif
int
setjmp(jmp_buf env)
{
    __setjmp(env);
    _savesigmask(&env->sigmask);

    return 0;
}

#if defined(ASMLINK)
ASMLINK
#endif
void
longjmp(jmp_buf env,
        int val)
{
    _loadsigmask(&env->sigmask);
    __longjmp(env, val);

    /* NOTREACHED */
}

#if defined(ASMLINK)
ASMLINK
#endif
int
_setjmp(jmp_buf env)
{
    __setjmp(env);

    return 0;
}
```

```
#if defined(ASMLINK)
ASMLINK
#endif
void
_longjmp(jmp_buf env,
         int val)
{
    __longjmp(env, val);

    /* NOTREACHED */
}
```

Part V

Computer Basics

It is time to take a quick look at basic computer architecture.

Chapter 10

Basic Architecture

10.1 Control Bus

For our purposes, we can think of control bus as two signals; RESET and CLOCK.

- RESET is used to trigger system initialisation to a known state to start running the operating system.
- CLOCK is a synchronisation signal that keeps the CPU and its external [memory and I/O] devices in synchronisation.

10.2 Memory Bus

Address Bus

Basically, address bus is where addresses for memory and I/O access are delivered. To simplify things, you might push a memory address on the address bus, perhaps modify it with an index register, and then fetch a value from or store a value to memory.

Data Bus

Data bus works in concert with the address bus; this is where actual data is delivered between the CPU and memory as well as I/O devices.

10.3 Von Neumann Machine

Essentially, von Neumann machines consist of **CPU**, **memory**, and **I/O** (input and output) facilities. Other similar architecture names such as **Harvard** exist for versions with extended memory subsystems, mostly, but as that is beyond our scope, I chose the 'original' name.

10.3.1 CPU

A CPU, central processing unit, is the heart of a computer. Without mystifying and obscuring things too much, let's think about it as a somewhat complex programmable calculator.

For a CPU, the fastest storage form is a register. A typical register set has dedicated registers for integer and floating point numbers; to make life easier, these are just bit-patterns representing values (integer) or approximations of values (floating-point) in some specified formats.

A notable quite-recent trend in CPUs are multicore chips; these have more than one execution unit inside them in order to execute several threads in parallel.

10.3.2 Memory

Memory is a non-persistent facility to store code and data used by the CPU. Whereas the register set tends to be fixed for a given CPU, memory can usually be added to computer systems. From a programmer's point of view, memory is still several times slower than registers; this problem is dealt with [fast] cache memory; most commonly, level 1 (L1) cache is on-chip and L2 cache external to the CPU.

As a rule of thumb, fast code should avoid unnecessary memory access and organize it so that most fetches would be served from cache memory. In general, let's, for now, say that you should learn to think about things in terms of words, cachelines, and pages - more on this fundamental topic later on.

10.3.3 I/O

I/O, input and output, is used for storing and retrieving external data. I/O devices tend to be orders of magnitude slower than memory; a notable feature, though, is that they can be persistent. For example, data written on a disk will survive electric outages instead of being wiped like most common types of memory would.

In addition to disks, networks have become a more-and-more visible form of I/O. In fact, whereas disks used to be faster than [most] networks, high-speed networks are fighting hard for the speed king status.

10.4 Conclusions

Simplified a bit, as a programmer it's often safe to think about storage like this; fastest first:

- registers
- memory
- disks
- network

- removable media

Note, though, that high-speed networks may be faster than your disks.

Part VI

Numeric Values

In this part, we take a look at computer presentation of numeric values. Deep within, computer programming is about moving numeric values between memory, registers, and I/O devices, as well as doing mathematical operations on them.

Chapter 11

Machine Dependencies

11.1 Word Size

Before the advent of C99 explicit-size types such as **int8_t** and **uint8_t**, programmers had to 'rely on' sizes of certain C types. I will list [most of] the assumptions made here, not only as a historic relic, but also to aid people dealing with older code with figuring it out.

Note that the sizes are listed in bytes.

Type	Typical Size	Notes
char	8-bit	may be signed; unsigned char for 8-bit clean code
short	16-bit	
int	32-bit	'fast integer'; typically 32-bit
long	32- or 64-bit	typically machine word
long long	64	standardised in ISO C99
float	32	single-precision floating point
double	64	double precision floating point
long double	80 or 128	extended precision floating point

One could try to check for these types with either GNU **Autoconf** (the **configure** scripts present in most open source projects these days use this strategy) or perhaps with something like

```
#include <limits.h>

#if (CHAR_MAX == 0x7f)
#define CHAR_SIGNED
#define CHAR_SIZE 1
#elif (CHAR_MAX == 0xff)
#define CHAR_UNSIGNED
#define CHAR_SIZE 1
#endif

#if (SHRT_MAX == 0x7fff)
```

```
#define SHORT_SIZE 2
#endif

#if (INT_MAX == 0x7fffffff)
#define INT_SIZE 4
#endif

#if (LONG_MAX == 0x7fffffff)
#define LONG_SIZE 4
#elif (LONG_MAX == 0x7fffffffffffffffULL)
#define LONG_SIZE 8
#endif
```

Notes

This code snippets is just an example, not totally portable.

Note that in this listing, sizes are defined in octets (i.e., 8-bit bytes). It's also noteworthy that **sizeof** cannot be used with preprocessor directives such as #if; therefore, when you have to deal with type sizes, you need to use some other scheme to check for and declare them.

11.2 Byte Order

Most of the time, when working on the local platform, the programmer does not need to care about byte order; the concern kicks in when you start communicating with other computers using storage and network devices.

Byte order is machine-dependent; so-called little endian machines have the lowest byte at the lowest memory address, and big endian machines vice versa. For example, the i386 is little endian (lowest byte first), and PowerPC CPUs are big endian.

Typical UNIX-like systems have **<endian.h>** or **<sys/endian.h>** to indicate their byte order. Here's an example of how one might use it; I define a structure to extract the 8-bit components from a 32-bit ARGB-format pixel value.

```
#include <stdio.h>
#include <stdint.h>
#include <endian.h> /* <sys/endian.h> on some systems (BSD) */

#if (_BYTE_ORDER == _LITTLE_ENDIAN)
struct _argb {
    uint8_t b;
    uint8_t g;
    uint8_t r;
    uint8_t a;
};
#elif (_BYTE_ORDER == _BIG_ENDIAN)
struct _argb {
    uint8_t a;
    uint8_t r;
```

```
    uint8_t g;
    uint8_t b;
};
#endif
#define aval(u32p) (((struct _argb *)(u32p))->a)
#define rval(u32p) (((struct _argb *)(u32p))->r)
#define gval(u32p) (((struct _argb *)(u32p))->g)
#define bval(u32p) (((struct _argb *)(u32p))->b)
```

Chapter 12

Unsigned Values

With the concepts of word size and byte order visited briefly, let's dive into how computers represent numerical values internally. Let us make our lives a little easier by stating that voltages, logic gates, and so on are beyond the scope of this book and just rely on the bits 0 and 1 to be magically present.

12.1 Binary Presentation

Unsigned numbers are presented as binary values. Each bit corresponds to the power of two its position indicates, for example

```
01010101
```

is, from right to left,

```
\(1 * 2^0 + 0 * 2^1 + 1 * 2^2 + 0 * 2^3 + 1 * 2^4 + 0 * 2^5 + 1 * 2^6 + 0 * 2^7\)
```

which, in decimal, is

```
\(1 + 0 + 4 + 0 + 16 + 0 + 64 + 0 == 85\).
```

Note that C doesn't allow one to specify constants in binary form directly; if you really want to present the number above in binary, you can do something like

```
#define BINVAL ((1 << 6) | (1 << 4) | (1 << 2) | (1 << 0))
```

As was pointed to me, the GNU C Compiler (GCC) supports, as an extension to the C language, writing binary constants like

```
uint32_t val = 0b01010101; /* binary value 01010101. */
```

12.2 Decimal Presentation

Decimal presentation of numeric values is what we human beings are used to. In C, decimal values are presented just like we do everywhere else, i.e.

```
int x = 165; /* assign decimal 165. */
```

12.3 Hexadecimal Presentation

Hexadecimal presentation is based on powers of 16, with the digits 0..9 representing, naturally, values 0 through 9, and the letters a..f representing values 10 through 15.

For example, the [8-bit] unsigned hexadecimal value

```
0xff
```

corresponds, left to right, to the decimal value

```
\(15 * 16^1 + 15 * 16^0 == 240 + 15 == 255\).
```

A useful thing to notice about hexadecimal values is that each digit represents 4 bits. As machine types tend to be multiples of 4 bytes in size, it's often convenient to represent their numeric values in hexadecimal. For example, whereas

```
4294967295
```

doesn't reveal it intuitively, it's easy to see from its hexadecimal presentation that

```
0xffffffff
```

is the maximum [unsigned] value a 32-bit type can hold, i.e. all 1-bits.

A useful technique is to represent flag-bits in hexadecimal, so like

```
#define BIT0  0x00000001
#define BIT1  0x00000002
#define BIT2  0x00000004
#define BIT3  0x00000008
#define BIT4  0x00000010
#define BIT5  0x00000020
/* BIT6..BIT30 not shown */
#define BIT31 0x80000000
```

where a good thing is that you can see the [required] size of the flag-type easier than with

```
#define BIT0  (1U << 0)
#define BIT1  (1U << 1)
/* BIT2..BIT30 not shown */
#define BIT31 (1U << 31)
```

Hexadecimal Character Constants

Hexadecimal notation can be used to represent character constants by prefixing them with
x within single quotes, e.g. '\xff'

12.4 Octal Presentation

Octal presentation is based on powers of 8. Each digit corresponds to 3 bits to represent values 0..7. Constant values are prefixed with a '0' (zero), e.g.

```
01234 /* octal 1234 */
```

Octal Character Constants

One typical use of octal values is to represent numerical values of 7- or 8-bit characters, of which some have special syntax. In C, octal integer character constants are enclosed within a pair of single quotes "'" and prefixed with a backslash "\".

A char can consist of up to three octal digits; for example '\1', '\01', and '\001' would be equal.

For some examples in ASCII:

Char	Octal	C	Notes
NUL	000'	0'	string terminator
BEL	007'	a'	bell
BS	010'	b'	backspace
SPACE	040'	' '	space/empty
A	101	'A'	upper case letter A
B	102	'B'	upper case letter B
a	141	'a'	lower case letter a
b	142	'b'	lower case letter b
\	134	'\\'	escaped with another backslash

12.5 A Bit on Characters

A noteworthy difference between DOS-, Mac- and UNIX-text files in ASCII is that whereas UNIX terminates lines with a linefeed '\n', DOS uses a carriage return + linefeed pair "\r\n", and Mac (in non-CLI mode) uses '\r'. Mac in CLI-mode uses '\n' just like UNIX.

Implementations of C supporting character sets other than the standard 7-/8-bit let you use a special notation for characters that cannot be represented in the char-type:

```
L'x' /* 'x' as a wide character (wchar_t) */
```

12.6 Zero Extension

Zero extension just means filling high bits with 0 (zero). For example, to convert a 32-bit unsigned integral value to 64 bits, you would just fill the high 32 bits 32..63 with zeroes, and the low 32 bits with the original value.

12.7 Limits

The minimum 32-bit unsigned number in C is, naturally,

```
#define UINT32_MIN 0x00000000U /* all zero-bits. */
```

The maximum for 32-bit unsigned number is

```
#define UINT32_MAX 0xffffffffU /* all 1-bits */
```

12.8 Pitfalls

12.8.1 Underflow

Note that with unsigned types, subtractions with negative results lead to big values. For example, with 32-bit unsigned types,

```
uint32_t u32 = 0;

--u32; /* u32 becomes 0xffffffff == UINT32_MAX! */
```

Therefore, if u32 can be zero, never do something like this:

```
while (--u32) {
    /* do stuff. */
}
```

Instead, do

```
if (u32) {
    while (--u32) {
        /* do stuff. */
    }
}
```

or

```
do {
    /* do stuff. */
} while (u32-- > 0);
```

12.8.2 Overflow

With unsigned types, additions with results bigger than maximum value lead to small values. For example, with 32-bit unsigned types,

```
uint32_t u32 = 0xffffffff; /* maximum value */

++u32; /* u32 becomes 0! */
```

One hazard here is constructs such as

```
uint16_t u16;

for (u16 = 0 ; u16 <= UINT16_MAX ; u16++) {
    /* do stuff. */
}
```

because adding to the maximum value causes an overflow and wraps the value to zero, the loop would never terminate.

Chapter 13

Signed Values

By introducing the sign [highest] bit, we can represent negative values to make life more interesting.

13.1 Positive Values

Positive values 0..M, where M is the maximum value of a signed type, are represented just like in unsigned presentation.

13.2 Negative Values

Let us see what is going on with negative [signed] values.

13.2.1 2's Complement

This section discusses the dominant 2's complement presentation of negative signed integral values.

For signed types, negative values are represented with the highest bit (the sign-bit) set to 1.

The rest of the bits in a negative signed value are defined so that the signed n-bit numeric value i is presented as

\(2^n - 1\)

Note that 0 (zero) is presented as an unsigned value (sign-bit zero).

As an example, 32-bit -1, -2, and -3 would be represented like this:

```
#define MINUS_ONE   0xffffffff
#define MINUS_TWO   0xfffffffe
#define MINUS_THREE 0xfffffffd
```

To negate a 2's complement value, you can use this algorithm:

- invert all bits
- add one, ignoring any overflow

13.2.2 Limits

The minimum 32-bit signed number is

```
#define INT32_MIN (-0x7fffffff - 1)
```

whereas the maximum 32-bit signed number is

```
#define INT32_MAX 0x7fffffff
```

13.2.3 Sign Extension

Sign extension means filling high bits with the sign-bit. For example, a 32-bit signed value would be converted to a 64-bit one by filling the high 32 bits 32..63 with the sign-bit of the 32-bit value, and the low 32 bits with the original value.

13.2.4 Pitfalls

Note that you can't negate the smallest negative value. Therefore, the result of abs(type_max), i.e. absolute value, is undefined.

13.2.4.1 Underflow

On 2's complement systems, the values of subtractions with results smaller than the type-minimum are undefined.

13.2.4.2 Overflow

On 2's complement systems, the values of additions with results greater than the type-maximum are undefined.

Chapter 14

Floating Point Numeric Presentation

TODO: comparison

http://docs.sun.com/source/806-3568/ncg_goldberg.html

http://randomascii.wordpress.com/2012/09/09/game-developer-magazine-floating-point

This chapter explains the IEEE 754 Standard for floating point values.

14.1 Basics

Floating-point numbers consist of two parts; **significand** and **exponent**.

As a typical scientific constant, the speed of light can be represented, in decimal base, as

```
299792.458 meters/second
```

or equivalently

```
\(2.99792458 x 10^5\).
```

In C notation, the latter form would be

```
2.99792458e5 /* speed of light. */
```

Here the mantissa is 2.99792458 and the exponent in base 10 is 5.

The new versions of the C Language, starting from C99, i.e. ISO/IEC 9899:1999, introduced hexadecimal presentation of floating point literals. For example,

```
0x12345e5
```

would be equal to

```
\((1 * 16^4 + 2 * 16^3 + 3 * 16^2 + 4 * 16^1 + 5 * 16^0) * 16^5\)
```

which would be equal to

\((1 * 65536 + 2 * 4096 + 3 * 256 + 4 * 16 + 5 * 1) * 1048576\)

= \((65536 + 8192 + 768 + 64 + 5) * 1048576\)

= \(74565 * 1048576\)

= \(7.818706 * 10^{10}\)

in decimal notation. This is more convenient to write than

78187060000.

14.2 IEEE Floating Point Presentation

The IEEE 754 Standard is the most common representation for floating point values.

14.2.1 Significand; 'Mantissa'

Significand and coefficient are synonymous to [the erroneously used term] mantissa.

The significand in IEEE 754 floating-point presentation doesn't have an explicit radix point; instead, it is implicitly assumed to always lie in a certain position within the significand. The length of the significand determines the precision of numeric presentation.

Note that using the term mantissa for significand is not exactly correct; when using logarithm tables, mantissa actually means the logarithm of the significand.

14.2.2 Exponent

Scale and characteristic are synonymous to exponent.

It should be sufficient to mention that a floating point value is formed from its presentation with the formula

\(f = significand * base^{exponent}\)

14.2.3 Bit Level

IEEE 754 Floating Point Presentation, at bit-level, works like this

- The highest bit is the sign bit; value 1 indicates a negative value.
- The next highest bits, the number of which depends on precision, store the exponent.
- The low bits, called fraction bits, store the significand.

Let's illustrate this simply (in most significant bit first) as

(SIGN)|(EXPONENT)|(FRACTION).

- The sign bit determines the sign of the number, i.e. the sign of the the significand.
- The exponent bits encode the exponent; this will be explained in the next, precision-specific sections.
- W_{part} is used to denote the width of part in bits.
- Conversion equations are given in a mix of mathematical and C notation, most notably I use the asterisk ('*') to denote multiplication.

TODO: CONVERSIONS TO AND FROM DECIMAL?

14.2.4 Single Precision

W_{value} is 32,

of which

W_{sign} is 1
$W_{exponent}$ is 8
$W_{significand}$ is 23.

To get the true exponent, presented in 'offset binary representation', the value **0x7f** (127) needs to be subtracted from the stored exponent. Therefore, the value 0x7f is used to store actual zero exponent and the minimum efficient exponent is -126 (stored as 0x01).

14.2.4.1 Zero Significand

Exponent **0x00** is used to represent 0.0 (zero).

In this case, the relatively uninteresting conversion equation to convert to decimal (base-10) value is

`\((-1)^{sign} * 2^{-126} * 0.significand\).`

Exponents 0x01 through 0xfe are used to represent 'normalised values', i.e. the equation becomes

`\((-1)^{sign} * 2^{exponent - 127} * 1.significand\).`

Exponent **0xff** is used to represent [signed] infinite values.

14.2.4.2 Non-Zero Significand

- Exponent **0x00** is used to represent subnormal values.
- Exponents 0x01 through 0xfe represent normalised values, just like with the significand of zero.
- Exponent **0xff** is used to represent special Not-a-Number (NaN) values.

14.2.5 Double Precision

W_{value} is 64,

of which

W_{sign} is 1
$W_{exponent}$ is 11
$W_{significand}$ is 52.

To get the true exponent, presented in 'offset binary representation', the value **0x2ff** (1023) needs to be subtracted from the stored exponent.

14.2.5.1 Special Cases

0 (zero) is represented by having both the exponent and fraction all zero-bits.

The exponent **0x7ff** is used to present positive and negative infinite values (depending on the sign bit) when the fraction is zero, and NaNs when it is not.

With these special cases left out, the conversion to decimal becomes

`\((-1)^{sign} * 2^{exponent - 1023} x 1.significand\).`

14.2.6 Extended Precision

The most common extended precision format is the 80-bit format that originated in the Intel i8087 mathematics coprocessor. It has been standardised by IEEE.

This extended precision type is usually supported by the C compilers via the long double type. These values should be aligned to 96-bit boundaries, which doesn't make them behave very nicely when 64-bit wide memory access is used; therefore, you may want to look into using 128-bit long doubles. The Gnu C Compiler (GCC) does allow this with the

`-m128bit-long-double`

compiler flag.

Intel SIMD-architectures starting from **SSE** support the **MOVDQA** machine instruction to move aligned 128-bit words between SSE-registers and memory. I tell this as something interesting to look at for those of you who might be wishing to write, for example, fast memory copy routines.

14.2.6.1 80-Bit Presentation

W_{value} is 80,

of which

W_{sign} is 1
$W_{exponent}$ is 15
$W_{significand}$ is 64.

14.3 i387 Assembly Examples

14.3.1 i387 Header

```
#ifndef __I387_MATH_H__
#define __I387_MATH_H__

#define getnan(d)                                                       \
    (dsetexp(d, 0x7ff), dsetmant(d, 0x000fffffffffffff), (d))
#define getsnan(d) \
    (dsetsign(d), dsetexp(d, 0x7ff), dsetmant(d, 0x000fffffffffffff), (d))
#define getnanf(f)                                                      \
    (fsetexp(f, 0x7ff), fsetmant(f, 0x007fffff), (f))
#define getsnanf(f)                                                     \
    (fsetsign(f), fsetexp(f, 0x7ff), fsetmant(f, 0x007fffff), (f))
#define getnanl(ld)                                                      \
    (ldsetexp(f, 0x7fff), ldsetmant(ld, 0xffffffffffffffff), (ld))
#define getsnanl(ld) \
    (ldsetsign(ld), ldsetexp(ld, 0x7fff), ldsetmant(ld, 0xffffffffffffffff), (ld))

#endif /* __I387_MATH_H__ */
```

14.3.2 i387 Source

```
#include <features.h>
#include <fenv.h>
#include <errno.h>
#include <math.h>
#include <i387/math.h>
#include <zero/trix.h>

__inline__ double
sqrt(double x)
{
    double retval;

    if (isnan(x) || fpclassify(x) == FP_ZERO) {
        retval = x;
    } else if (!dgetsign(x) && fpclassify(x) == FP_INFINITE) {
        retval = dsetexp(retval, 0x7ff);
    } else if (x < -0.0) {
        errno = EDOM;
        feraiseexcept(FE_INVALID);
        if (dgetsign(x)) {
            retval = getsnan(x);
        } else {
            retval = getnan(x);
```

```
        }
    } else {
        __asm__ __volatile__ ("fldl %0\n" : : "m" (x));
        __asm__ __volatile__ ("fsqrt\n");
        __asm__ __volatile__ ("fstpl %0\n"
                              "fwait\n"
                              : "=m" (retval));
    }

    return retval;
}

__inline__ double
sin(double x)
{
    double retval;

    if (isnan(x)) {
        retval = x;
    } else if (fpclassify(x) == FP_INFINITE) {
        errno = EDOM;
        feraiseexcept(FE_INVALID);
        if (dgetsign(x)) {
            retval = getsnan(x);
        } else {
            retval = getnan(x);
        }
    } else {
        __asm__ __volatile__ ("fldl %0\n" : : "m" (x));
        __asm__ __volatile__ ("fsin\n");
        __asm__ __volatile__ ("fstpl %0\n"
                              "fwait\n"
                              : "=m" (retval));
    }

    return retval;
}

__inline__ double
cos(double x)
{
    double retval;

    __asm__ __volatile__ ("fldl %0\n" : : "m" (x));
    __asm__ __volatile__ ("fcos\n");
    __asm__ __volatile__ ("fstpl %0\n"
                          "fwait\n"
                          : "=m" (retval));

    return retval;
```

```
}

__inline__ double
tan(double x)
{
    double tmp;
    double retval;

    if (isnan(x)) {
        retval = x;
    } else if (fpclassify(x) == FP_INFINITE) {
        errno = EDOM;
        feraiseexcept(FE_INVALID);
        if (dgetsign(x)) {
            retval = getsnan(x);
        } else {
            retval = getnan(x);
        }
    } else {
        __asm__ __volatile__ ("fldl %0\n" : : "m" (x));
        __asm__ __volatile__ ("fptan\n");
        __asm__ __volatile__ ("fstpl %0\n" : "=m" (tmp));
        __asm__ __volatile__ ("fstpl %0\n"
                              "fwait\n"
                              : "=m" (retval));
        if (dgetsign(retval) && isnan(retval)) {
            retval = 0.0;
        }
    }

    return retval;
}

#if ((_BSD_SOURCE) || (_SVID_SOURCE) || _XOPEN_SOURCE >= 600            \
    || (_ISOC99_SOURCE) || _POSIX_C_SOURCE >= 200112L)

__inline__ float
sinf(float x)
{
    float retval;

    __asm__ __volatile__ ("flds %0\n" : : "m" (x));
    __asm__ __volatile__ ("fsin\n");
    __asm__ __volatile__ ("fstps %0\n"
                          "fwait\n"
                          : "=m" (retval));

    return retval;
}
```

```
__inline__ float
cosf(float x)
{
    float retval;

    __asm__ __volatile__ ("flds %0\n" : : "m" (x));
    __asm__ __volatile__ ("fcos\n");
    __asm__ __volatile__ ("fstps %0\n"
                          "fwait\n"
                          : "=m" (retval));
    return retval;
}

__inline__ float
tanf(float x)
{
    float tmp;
    float retval;

    if (isnan(x) || fpclassify(x) == FP_ZERO) {
        retval = x;
    } else if (fpclassify(x) == FP_INFINITE) {
        if (dgetsign(x)) {
            retval = -M_PI * 0.5;
        } else {
            retval = M_PI * 0.5;
        }
    } else {
        __asm__ __volatile__ ("flds %0\n" : : "m" (x));
        __asm__ __volatile__ ("fptan\n");
        __asm__ __volatile__ ("fstps %0\n" : "=m" (tmp));
        __asm__ __volatile__ ("fstps %0\n"
                              "fwait\n"
                              : "=m" (retval));
        if (fgetsign(retval) && isnan(retval)) {
            retval = 0.0;
        }
    }

    return retval;
}

__inline__ long double
sinl(long double x)
{
    long double retval;

    __asm__ __volatile__ ("fldt %0\n" : : "m" (x));
    __asm__ __volatile__ ("fsin\n");
    __asm__ __volatile__ ("fstpt %0\n"
```

```
                        "fwait\n"
                        : "=m" (retval));

    return retval;
}

__inline__ long double
cosl(long double x)
{
    long double retval;

    __asm__ __volatile__ ("fldt %0\n" : : "m" (x));
    __asm__ __volatile__ ("fcos\n");
    __asm__ __volatile__ ("fstpt %0\n"
                        "fwait\n"
                        : "=m" (retval));

    return retval;
}

#endif

#if (_GNU_SOURCE)
void
sincos(double x, double *sin, double *cos)
{
    __asm__ __volatile__ ("fldl %0\n" : : "m" (x));
    __asm__ __volatile__ ("fsincos\n");
    __asm__ __volatile__ ("fstpl %0\n"
                        "fwait\n"
                        : "=m" (*cos));
    __asm__ __volatile__ ("fstpl %0\n"
                        "fwait\n"
                        : "=m" (*sin));

    return;
}

void
sincosf(float x, float *sin, float *cos)
{
    __asm__ __volatile__ ("flds %0\n" : : "m" (x));
    __asm__ __volatile__ ("fsincos\n");
    __asm__ __volatile__ ("fstps %0\n"
                        "fwait\n"
                        : "=m" (*cos));
    __asm__ __volatile__ ("fstps %0\n"
                        "fwait\n"
                        : "=m" (*sin));
```

```
    return;
}

void
sincosl(long double x, long double *sin, long double *cos)
{
    __asm__ __volatile__ ("fldt %0\n" : : "m" (x));
    __asm__ __volatile__ ("fsincos\n");
    __asm__ __volatile__ ("fstpt %0\n"
                          "fwait\n"
                          : "=m" (*cos));
    __asm__ __volatile__ ("fstpt %0\n"
                          "fwait\n"
                          : "=m" (*sin));

    return;
}
#endif
```

Part VII

Machine Level Programming

Chapter 15

Machine Interface

15.1 Compiler Specification

15.1.1 <cdecl.h>

```
#ifndef __CDECL_H__
#define __CDECL_H__

/*
 * EMPTY     - used to define 0-byte placeholder tables a'la
 *
 * int tab[EMPTY];
 */
#if defined(__STDC_VERSION__) && (__STDC_VERSION__ >= 199901L)
#   define EMPTY
#else
#   define EMPTY    0
#endif

/*
 * ALIGN(a)        - align to boundary of a.
 * PACK               - pack structures, i.e. don't pad for alignment (DIY).
 * REGARGS(n)        - call with n register arguments.
 * ASMLINK        - external linkage; pass arguments on stack, not registers
 * FASTCALL        - use as many register arguments as feasible
 *                (for system calls).
 */
#define ALIGN(a)   __attribute__ ((__aligned__(a)))
#define PACK       __attribute__ ((__packed__))
#define REGARGS(n) __attribute__ ((regparm(n)))
#define ASMLINK    __attribute__ ((regparm(0)))
#if defined(__i386__)
#define FASTCALL   REGARGS(3)
```

```
#endif

#endif /* __CDECL_H__ */
```

15.2 Machine Definition

15.2.1 <mach.h>

```
#ifndef __MACH_H__
#define __MACH_H__

#include <stdint.h>

#define NBWORD   4    /* native CPU word size */
#define NBCL     32   /* cacheline size */
#define NBPAGE   4096 /* page size */
#define NADDRBIT 32   /* number of significant address bits in pointers */

#include "cdecl.h"

/* call frame used by the compiler */
struct m_cframe {
    uint8_t avar[EMPTY]; /* automatic variables */
    int32_t ebp;            /* frame pointer to caller */
    int32_t eip;            /* return address to caller */
    uint8_t args[EMPTY]; /* placeholder for function arguments */
} _PACK;

/* stack structure used for interrupt returns (or other use of IRET) */
struct m_iret {
    int32_t eip;
    int32_t cs;
    int32_t eflags;
    int32_t esp;
    int32_t ss;
};

#endif /* __MACH_H__ */
```

Chapter 16

IA-32 Register Set

Note that EBP and ESP are usually considered general purpose registers; I deliberately chose to put them under Special Registers as I feel that's a better way to think of them.

16.1 General Purpose Registers

Register	Special Use
EAX	32-bit return value
EBX	data pointer
ECX	string and loop counter
EDX	I/O pointer, high bits of 64-bit return value
ESI	data pointer, string destination
EDI	stack data pointer

16.2 Special Registers

Register	Purpose
EBP	frame pointer
ESP	stack pointer
EIP	instruction pointer
EFLAGS	machine status flags

16.3 Control Registers

Register	Purpose
CR3	PDBR (page directory page register)

Chapter 17

Assembly

17.1 AT&T vs. Intel Syntax

The very first thing to notice for Intel-based platform assembly programmers is that the Intel syntax

```
MNEMONIC dest, src
```

is not valid using the GNU tools GCC and GNU Assembler;

instead, you need to use AT&T syntax, i.e.

```
MNEMONIC src, dest
```

to me, as I'm not an old school assembly programmer, this latter syntax makes more sense. Your mileage may vary. Either way, as it turns out, C is so flexible and fast that we actually have to resort to assembly very rarely; mostly we need it for certain machine specific, usually kernel-level, operations as well as in the extreme case, for code speedups.

Registers are named starting with a '%', e.g.

```
%eax.
```

When mixing register names with other arguments which need the '%' prefix, you need to make the register names start with '%%'.

In AT&T syntax, immediate operands are prefixed with $, e.g.

```
$0x0fc0
```

would represent the hexadecimal value that would be **0fc0h** in Intel syntax; note that the h suffix is replaced with the C-style 0x prefix.

One more thing to notice is that AT&T syntax assembly encodes the operand size as a prefix into the opcode, so instead of

```
mov al, byte ptr val
```

you need to write

`movb val, %al`

so the

`byte ptr, word ptr, and dword ptr`

memory operands change to opcode postfixes

`'b', 'w', and 'l'.`

For some quadword (64-bit) operations you'd use 'q' and in some rare cases, for 128-bit double quadword operations, 'dq'.

For memory addressing using base register, Intel syntax uses

`'[' and ']'`

to enclose the register name; AT&T syntax uses

`'(' and ')'.`

so the Intel syntax for indirect memory references, which would be

`[base + index * scale + displacement]`

becomes

`displacement(base, index, scale)`

in the AT&T syntax.

Now this may all be so confusing that I'd better sum it up with a few examples of the difference of the Intel and AT&T syntaxes.

17.1.1 Syntax Differences

Intel	AT&T
mov eax, 8	movl $8, %eax
mov ebx, abh	movl $0xab, %ebx
int 80h	int $0x80
mov eax, ebx	movl %ebx, %eax
mov eax, [ebx]	movl (%ebx), %eax
mov eax, [ebx + 5]	movl 5(%ebx), %eax
mov eax, [ecx + 40h]	movl 0x40(%ecx), %eax
add eax, [ebx + ecx * 4h]	addl (%ebx, %ecx, 0x4), %eax
lea eax, [ebx + edx]	leal (%ebx, %edx), %eax
add eax, [ebx + edx * 8h - 40h]	addl -0x40(%ebx, %edx, 0x8), %eax

Now let's take a look at what assembly programs look like.

17.1.2 First Linux Example

This example demonstrates function calls and how to exit a process on Linux using the exit() system call.

Source Code

```
# simple example program
# - implements exit(0)

.text

.globl main

main:
        call        func
        call        linexit

        # dummy function to demonstrate stack interaction of C programs
func:
        # compiler and CALL set up return stack frame
        pushl        %ebp
        movl        %esp, %ebp

        # DO YOUR THING HERE

        leave
        ret

        # simple function to make an exit() function call on Linux.
linexit:
        movl        $0x00, %ebx        # exit code
        movl        $0x01, %eax        # system call number (sys_exit)
        int        $0x80                # trigger system call
```

17.1.3 Second Linux Example

Here I implement **false** in assembly using the exit() system call in a bit different fashion than in the previous example.

Source Code

```
# simple educational implementation of false
# - implements exit(1) using the Linux EXIT system call

.text

.globl main

main:
        movl        $0x01, %eax
        movl        %eax, sysnum
        call        lindosys

lindosys:
        pushl        %ebp
```

```
        movl        %esp, %ebp

        movl        $sysframe, %esp
        popl        %ebx
        popl        %ecx
        popl        %edx
        pop        %eax
        int        $0x80

        leave
        ret
.data

sysframe:
_exitval:
        .long        0x00000001        # %ebx
        .long        0x00000000        # %ecx
        .long        0x00000000        # %edx
sysnum:
        .long        0x00000001        # linux system call number
```

17.1.3.1 Stack Usage

Let us take a closer look at how the code above uses the stack.

There's a label called **sysframe** in the DATA segment. As the comments suggest, this contains the register arguments for triggering a Linux system call with. The actual system call is triggered with

```
int \$0x80
```

In this example, the system call number of 0x01 (**sys_exit**) is passed in **sysnum**; this use is equivalent to assigning a global variable in C code.

Our function **lindosys** starts with the typical prologue for a function with no automatic (internal/stack) variables, i.e.

```
pushl %ebp # push frame pointer
movl %esp, %ebp # save stack pointer
```

After that, it sets the stack pointer to point to **sysframe**, pops 3 system calls arguments into the **EBX**, **ECX** and **EDX** registers, copies the system call number from **sysnum** into **EAX** and finally fires the system call by generating an interrupt.

After this, in case of system calls other than **exit**, it would return to the calling function with the standard

```
leave
ret
```

Function prologue. First, **leave** sets the stack pointer **ESP** to the current value of the frame pointer **EBP**, then pops the earlier value of frame pointer for the calling function.

After this, **ret** pops the return address and returns.

Note that the **_exitval** label is used as an alias to store the first system call argument to be popped from the stack.

TODO: FINISH... REF: http://www.ibiblio.org/gferg/ldp/GCC-Inline-Assembly-HOWTO.html

Chapter 18

Inline Assembly

18.1 Syntax

GNU Inline assembly uses the following format

```
__asm__ (TEMPLATE : OUTPUT : INPUT : CLOBBERED);
```

TEMPLATE contains the instructions and refers to the optional operands in the OUTPUT and INPUT fields. CLOBBERED lists the registers whose values are affected by executing the instruction in TEMPLATE. As we are going to see a bit later, you can specify **"memory"** as a special case in the CLOBBERED field. Also, if the instructions in TEMPLATE can chance condition code registers, you need to include **"cc"** in the CLOBBERED list. Note also that if the code affects memory locations not listed in the constraints you need to declare your assembly volatile like

```
__asm__ __volatile__ ("cli\n"); // disable interrupts

/* insert code here */

__asm__ __volatile__ ("sti\n");
```

the volatile attribute also helps you in the way that the compiler will not try to move your instructions to try and optimise/reschedule them.

18.1.1 rdtsc()

Let's see how we can read the timestamp [clock-cycle] counter using the rdtsc assembly instruction.

```
#include <stdint.h>

union _rdtsc {
    uint32_t u32[2];
    uint64_t u64;
};
```

```
typedef union _rdtsc rdtsc_t;

static __inline__ uint64_t
rdtsc(void)
{
    rdtsc_t tsval;

    __asm__ ("rdtsc\n"
             "movl %%eax, %0\n"
             "movl %%edx, %1\n"
             : "=m" (tsval.u32[0]),
               "=m" (tsval.u32[1])
             : /* no INPUT field */
             : "eax", "edx");

    return tsval.u64;
}
```

We try to inline this in-header function not visible to other files (declared **static**). Inlining has its own section in part **Code Optimisation** of this book.

In the listing above, the INPUT field consists of the RDTSC instruction, which takes no operands, and two movl operations. RDTSC returns the low 32 bits of its 64-bit return value in EAX and the high 32 bits in EDX. We use a trick with the union to make the compiler return the combination of these two 32-bit values as a 64-bit one. Chances are it uses the same two registers and can optimise what our code may seem to do.

Notice how the OUTPUT field uses **"=m"**; output operands are prefixed with '**=**' to denote they are **assigned/written**. The 'm' means these have to be memory operands (IA-32 has no 64-bit integer registers). The '=' means this an output (write-only) operand.

The CLOBBERED field says we pollute the registers EAX and EDX. All fields except TEMPLATE are optional. Every optional field that lists more than one operand uses commas to separate them.

18.2 Constraints

18.2.1 IA-32 Constraints

Identifier	Possible Registers or Values
a	%eax, %ax, %al
b	%ebx, %bx, %bl
c	%ecx, %cx, %cl
d	%edx, %dx, %dl
S	%esi, %si
D	%edi, %di
q	registers a, b, c, or d
I	constant between 0 and 31 (32-bit shift count)
J	constant between 0 and 63 (64-bit shift count)
K	0xff
L	0xffff
M	constant between 0 and 3 (lea instruction shift count)
N	constant between 0 and 255 (out instruction output value)
f	floating point register
t	first floating point register (top of stack)
u	second floating point register
A	register a or d; 64-bit return values with high bits in d and low bits in a

18.2.2 Memory Constraint

The example above used the memory constraint **"=m"** for output. You would use **"m"** for input operands.

18.2.3 Register Constraints

If you want to let the compiler pick a register to use, use **"r"** (input) or **"=r"** (output). As an example, if you don't care if a register or memory is used, you can use the combination of "rm" or "=rm" for input and output operands, respectively. The 'r' in them might speed the operation up, but leave it out if you want a memory location to be updated.

18.2.4 Matching Constraints

Sometimes, a single variable serves both as input and output. You can do this by specifying matching (digit) constraints.

18.2.4.1 Example; incl

```
__asm__ ("incl %0" : "=a" (reg) : "0" (reg));
```

18.2.5 Other Constraints

Constraint	Rules
m	memory operand with any kind of address
o	offsettable memory operand; adding a small offset gives valid address
V	memory operand which is not offsettable
i	immediate integer operand (a constant), including a symbolic constant
n	immediate integer operand with a known numeric value
g	any general register, memory or immediate integer operand

Use 'n' instead of 'i' for operands less than a word wide if the system cannot support them as assembly-time constants.

18.2.6 Constraint Modifiers

Constraint	Meaning
=	write only; replaced by output data
&	early-clobber operand; modified before instruction finished; cannot be use elsewhere

18.3 Clobber Statements

It is to be considered good form to list registers you clobber in the clobber statement. Sometimes, you may need to add **m̈emory¨** to your clobber statement. Use of the **__volatile__** keyword and making assembly operations single statements is often necessary to keep the compiler [optimiser] from doing hazardous things such as reordering instructions for you.

18.3.1 Memory Barrier

Where memory access needs to be serialised, you can use memory barriers like this

```
#define membar() \
    __asm__ __volatile__ ("" : : : "memory")
```

Chapter 19

Interfacing with Assembly

19.1 alloca()

alloca() is used to allocate space within the stack frame of the current function. Whereas this operation is known to be potentially unsafe, I use it to demonstrate how to interface with assembly code from our C programs.

19.1.1 Implementation

Here is our header file to declare alloca().

<alloca.h>

```
#ifndef __ALLOCA_H__
#define __ALLOCA_H__

#include <stddef.h>

#if defined(__GNUC__)
#define alloca(size) __builtin_alloca(size)
#else
void * alloca(size_t size);
#endif

#endif /* __ALLOCA_H__ */
```

Let us take a look at the **x86-64** version of the alloca() routine.

alloca.S

```
#if defined(__x86_64__) || defined(__amd64__) && !defined(__GNUC__)

.globl alloca

.text 64

/*
 * registers at call time
 * ----------------------
 * rdi        - size argument
 *
 * stack at call time
 * ------------------
 * return address <- stack pointer
 */
alloca:
        subq        $8, %rdi        // adjust for popped return address
        movq        %rsp, %rax        // copy stack pointer
        subq        %rdi, %rax        // reserve space; return value is in RAX
        ret                         // return

#endif
```

Notes

- After linking with assembled alloca object, alloca can be triggered from C code just like typical C code by calling **alloca()**.

- Our alloca() function is disabled with the GNU C Compiler in favor of **__builtin_alloca()**.

19.1.2 Example Use

The code snippet below demonstrates calling alloca() from C code.

```
#include <string.h>
#include <alloca.h>

#define STUFFSIZE 128

int
dostuff(long cmd)
{
    int   retval;
    void *ptr;

    /* allocate and initialise stack space */
    ptr = alloca(STUFFSIZE);
    memset(ptr, 0, STUFFSIZE);
    /* process cmd */
    retval = stuff(cmd, ptr);

    return retval;
}
```

Part VIII

Code Style

Chapter 20

A View on Style

20.1 Concerns

Readability

Good code should document what it does reasonably well. Comments should not concentrate on how something is done (unless it's obscure), but rather on what is being done. Good code is easy to read and understand to a point. It is good to hide the more obscure things; for example, clever macro tricks should be put in header files not to pollute the code with hard-to-read parts. It's highly recommended to have style guides to keep code from several programmers as consistent as possible.

Maintainability

Good code is easy to modify; for example, to add new features to. Although the C preprocessor is sometimes considered a bit of a curse, macros (and, of course, functions for bigger code pieces) are good to avoid code repetition. Try to recognise code patterns and not repeat them; this way, it's easier to fix possible bugs as they are only in one place.

Reusability

Good code should be commented and documented otherwise. Good and precise design specifications help especially team projects. I suggest splitting projects into logical parts, defining the interfaces between them, and then implementing the modules. This kind of approach is called **divide and conquer** by some. There are also top-down aspects with this approach (you try to see the big picture and start working towards the details), but the programmers working on the modules may just as well use bottom-up at lower level. It's the end-result, the produced code and its quality, that counts. Keep in mind software development is art and everyone has their own style which is good to respect as long as it doesn't interfere with other parts of projects.

20.2 Thoughts

To Each Their Own

I feel the need to point out that this section is a personal view on things. Team projects may have style guides and policies very different from these; individual programmers tend to have their own preferences. To each their own - consider this section food for thought.

Simplicity

Good code should be as self-documenting as possible. Readability, clarity, intuitivity, logicality, and such factors contribute to the quality of code. Simplicity makes things easy to test, debug, and fix. Errare humanum est; the fewer lines of code you have, the fewer bugs you are likely to find.

Brevity

To make code faster to type, read, and hopefully grasp, I suggest using relatively brief identifier names.

Here is a somewhat extreme example

```
/* 'bad' code commented out */
#if 0
#define pager_get_page_offset(adr) ((uintptr_t)(adr) & 0xfff)
#endif

/* clarified version */
#define pageofs(adr) ((uintptr_t)(adr) & 0xfff)
```

Note how much easier it is to read the latter one and how little if any information we hid from the programmers exploring the code.

Mixed Case

As a personal protest at mixing case, let me point out a couple of observations.

Here's a few different ways for naming a macro like above

It is faster to type and easier to read

```
pageofs(ptr);
```

than

```
page_ofs(ptr);
```

or

```
PageOffset(ptr);
```

or

```
pageOffset(ptr);
```

or even

```
PageOfs(ptr);
```

You don't need to hold the shift key for the first one, plus it seems both the most compact and clearest version to read to me.

As a rule of thumb, **the smaller the scope** of your variables and functions, **the shorter the names** can be.

Consistency

Style is, well, a matter of taste. What ever kind of style you choose, be uniform and consistent about it. A bit of thought and cleverness will take your code forward a long way in terms of readability; it's good for both modifiability and maintainability.

Uniformity

Bigger projects, especially those with several people working on them, benefit miraculously if all programmers use uniform style. Should you be starting such a project, I suggest you look into writing a style guide one of the very first things.

20.3 Conventions

K & R

This chapter lists some basic conventions I'm used to follow. Many of these originate from sources such as Kernighan & Ritchie (the creators of the C language), old UNIX hackers, and so forth.

20.3.1 Macro Names

Constant Values

One often sees macros named in all uppercase; the C language and UNIX themselves use this convention repeatedly; SIGABRT, SIGSEGV, SIGILL, EILSEQ, EINTR, EAGAIN, etc. all fit my convention of naming constant value macros in all upper case.

Even though I have done so, I tend not to name function-like macros (those with arguments evaluating to a non-constant value) in upper case; instead, I do things such as

```
#define SIGMSK   0x3f // signals 0 through 63
#define SIGRTMSK 0x50 // realtime signals; above 31

/* u is unsigned; 0 is not a valid signal number */
#define _sigvalid(u) ((u) && !((u) & ~_SIGMSK))
```

Note that it's good form to document what you have done using macros because you can't take pointers to macros (could be resolved with wrapper functions).

Comments

Right these days, I'm adopting to the convention of using 'C++ style' comments (starting with "//") introduced to C in the 1999 ISO standard for end-of-line comments and comments enclosed between '/*' and '*/' for comment-only lines. A multiline comment I write like

```
/*
 * HAZARD: dirty programming trick below.
 */
#define ptrisaln(ptr, p2) \
    (!((uintptr_t)(ptr) & ((p2) - 1)))
```

Many nice editors such as Emacs can do code highlights, so for example on my setup, comments show as red text. Such highlight features can also help you debug code; think of an unclosed comment making a long piece of text red and how easy it makes to spot the missing comment closure.

20.3.2 Underscore Prefixes

Note that C reserves identifiers whose name begin with an underscore ('_') for system use. Read on, though.

An old convention is to prefix names of identifiers with narrow scope (not visible to all files) with underscores. It's negotiable if one should use a single underscore or two - or use them at all. Chances are you should introduce such identifiers within the file or in a header used in relatively few places. A seasoned hacker may barf on you if you make them globally visible. =)

20.3.3 Function Names

Even though I'm not a big fan of all object-oriented naming schemes, I do attest seeing the name of the module (file) where a function is implemented from its name is a good thing. Whether you should or should not use underscores as word delimiters depends on what you feel is the best way. ;) However, my vote still goes for brevity; remember my earlier rant about why I would use **pageofs** as a macro name over a few alternatives.

20.3.4 Variable Names

I will tell you a few examples of how I'm planning to name variables in my ongoing kernel project.

- As always, try to be brief and descriptive.
- Leave the shortest names, such as 'x', 'y', 'z', to automatic variables or, when there's little risk of being misunderstood, aggregate fields to avoid namespace collisions.
- Use parameter names for function prototypes in header files.
- Use longer and more descriptive names for global variables (if you need to use them) and function arguments, again to avoid polluting the name space.
- Use names starting with 'u' for unsigned types to make it easier to predict variable behavior, especially when doing more creative arithmetics (e.g., dealing with overflows by hand) on them.

20.3.5 Abbreviations

I am of the opinion that abbreviations are a good thing when used with care; uniformly and consistently. However, naming schemes for them vary so much that I thought a bit of documentation on them would be good.

Examples

Abbreviation	Explanation
adr	address; numerical pointer value
arg	argument
atr	attribute
aln	alignment; addresses
auth	authentication; authorisation
blk	block; I/O
buf	buffer; I/O
cbrt	cubic root
cl	cache line
con	console
cpu	central processor unit
ctx	context
cur	current [item]
decr	decrement
fp	frame pointer
fpu	floating point unit
frm	frame
func	function; function pointers
gpu	graphics processor unit
lst	list
mem	memory
mod	module
nam	name
num	number; numerical identication
hst	host
htab	hash table
hw	hardware
id	[possibly numerical] identification
incr	increment
intr	interrupt
lg	logarithm
mtx	mutex (mutual exclusion lock)
ndx	index
num	number; numerical ID
nod	node
pc	program counter; instruction pointer
perm	permission
phys	physical; address
pnt	point
ppu	physics processor unit
proc	process; processor
prot	protection
proto	protocol
pt	part; point
ptr	pointer
rbt	red-black tree
reg	register
ret	return
rtn	routine
sem	semaphore
shm	shared memory
sp	stack pointer
sqrt	square root
stk	stack
str	string
tab	table; array
thr	thread
tmp	temporary variable
val	[probably numerical] value

I suggest the laziness of not thinking beyond names such as **xyzzy**, **foo**, **bar**, **foobar**, etc. for only the very quickest [personal] hacks. There it can be lots of fun though, and it may be humorous to people with similar hacker mindset. =D

20.4 Naming Conventions

- prefix machine-specific names with **m_** (machine dependent), for example struct **m_iret**
- prefix floating point variable names with **f**
- name loop iteration counts like **i** (int) or **l** (long); for nested loops, use successive single-letter names (**j**, **k**, etc. or **m**, **n** and so forth
- use mathematical symbols such as **x**, **y**, and **z** where relevant
- **n** for count variables; alone or as a prefix
- name functions and function-like macros descriptively like **pagemap()**, **pagezero()**
- prefix file-global entities (ones outside functions) with module (file) or other logical names; for example, a global page map in mem.c could be **memphysmap** or **membufmap**
- prefix names of program globals (such as structs) with program or some other conventional name such as **k** or **kern** in a kernel
- name constant-value macros with all upper case like the C language often does (e.g. **EILSEQ**)
- brevity over complexity; why name a function **kernel_alloc_memory** when **kmalloc** works just as well; is easier to read and actually C-library style/conventional

Here is an example. **TODO**: better/bigger example

```
#include "mem.h"

/* initialise i386 virtual address space */
pageinit(uint32_t *map, uint32_t base, uint32_t size);

#define KVMBASE 0xc0000000 // virtual memory base
#define NPDE    1024

uint32_t mempagedir[NPDE];
```

Globals

Note that use of globals entities (those beyond file scope), should generally be avoided. When you have to do it, consider using file-scope structures which you put members into and passing pointers them to your functions. This will keep the namespace cleaner and confusions fewer.

This code snippet reflects how I tend to, to a point, organise things in files. The order is usually something like described below.

Source Files

- #include statements
- globals
- function implementations

Header Files

- #include statements
- typedef statements
- function prototypes
- function-like macros
- constant macros
- aggregate type (struct and union) declarations

20.5 Other Conventions

- use comments to tell what the code does without getting too detailed
- use narrow scope; use **static** for local scope (used within a file) identifiers; try to stick close to one file per module (or perhaps a **source file + header**), e.g. **mem.c** and **mem.h** for memory management
- use **macros**; hiding peculiar things such as creative bit operations makes them easier to reuse (and if put into header files, keeps them from hurting your eyes when reading the actual code) :)
- avoid '**magic numbers**'; define macros for constants in code for better maintainability
- use **typedef** sparingly; keep things easier to grasp at first sight
- avoid code repetition and deep nesting; use macros; pay attention to program flow
- use parentheses around macro arguments in macro bodies to avoid hard-to-find mistakes with macro evaluation
- enclose macros which use **if** inside do /* macro body */ while (0) to avoid unexpected behavior with else and else if

do ... while (0)

To illustrate the last convention, it is good to use

```
#define mymacro(x) \
do { \
    if (x) printf("cool\n") else printf("bah\n"); \
} while (0)
```

or, perhaps better still

```
#define mymacro(x) \
do { \
    if (x) { \
        printf("cool\n"); \
    } else { \
      printf("bah\n"); \
    }
} while (0)
```

instead of

```
#define mymacro(x) \
if (x) printf("cool\n") else printf("bah\n")
```

Part IX

Code Optimisation

Chapter 21

Execution Environment

21.1 CPU Internals

In this section, we shall take a quick look on some hardware-level optimisation techniques which processors use commonly.

21.1.1 Prefetch Queue

Prefetch queues are used to read chunks of instruction data at a time. It's a good idea **not to use many branching constructs**, i.e. jump around in code, to keep the CPU from not having to flush its prefetch queue often.

21.1.2 Pipelines

Processor units use pipelining to execute several operations in parallel. These operations, micro-ops, are parts of actual machine instructions. A common technique to make code 'pipeline' better, i.e. run faster, is to avoid data dependencies in adjacent operations. This means that the target operands should not be source operands for the next instruction (or operation at 'high' level such as C code). Ordering operations properly reduces pipeline stalls (having to wait for other operations to complete to continue), therefore making code execute more in parallel and faster.

21.1.3 Branch Prediction

TODO

Chapter 22

Optimisation Techniques

Even though careful coding will let you avoid having to apply some of these techniques, it is still good to know about them for the cases where you deal with code either written by other people or by yourself earlier; one learns and becomes better all the time by doing; in this case, a better programmer by programming.

22.1 Data Dependencies

A Few Words on Memory

Note that memory has traditionally been, and still is to a point, much slower to access than registers. Proper memory access works word by word within alignment requirements. Memory traversal such as zeroing pages should benefit from

Removing Data Dependency on Pointer

```
        while (i--) {
            ptr[0] = 0;
            ptr[1] = 0;
            ptr[2] = 0;
            ptr[3] = 0;
    ptr += 4;
        }
```

over

```
        while (i--) {
            *ptr++ = 0;
            *ptr++ = 0;
            *ptr++ = 0;
            *ptr++ = 0;
        }
```

because the next memory transfer does not depend on a new pointer value. It often pays to organise memory access in code, just like it's good to organise instructions so

as to do something creative before using the last computation's results. This technique is called data dependency elimination.

22.2 Recursion Removal

Here is the modified first example program for a hypothetical programming game we are developing codenamed Cyberhack. :)

```
void
start(void)
{
    run(rnd(memsz));
}

void
run(unsigned int adr)
{
    int myid = 0x04200420;
    int *ptr = (int *)adr;

    ptr[0] = myid;
    ptr[1] = myid;
    ptr[2] = myid;
    ptr[3] = myid;

    run(rnd(memsz));
}
```

As the experienced eye should see, this would lead to quite a bit of stack usage; **run()** calls itself tail-recursively (recursion at end). Every call will generate a new stack frame, which leads to indefinitely growing stack.

Stack Bloat After N Calls to run()

retadr **prevfp**	return address to caller caller's frame pointer
> ... <	
retadr **prevfp**	**N**th stack frame **N**th stack frame

You should, instead, use something like

```
void
start(void)
{
    for ( ; ; ) {
        run(rnd(memsz));
    }
}

void
```

```
run(unsigned int adr)
{
    int myid = 0x04200420;
    int *ptr = (int *)adr;

    ptr[0] = myid;
    ptr[1] = myid;
    ptr[2] = myid;
    ptr[3] = myid;

    return;
}
```

22.3 Code Inlining

inline-keyword

C99 introduced (and systems widely used it before) the keyword **inline**. This is a hint to the compiler to consider inlining functions.

Let's look at the example of C in the end of the previous chapter. We call **run()** once per loop iteration from **start()**. Instead, it's a good idea to use

```
void
run(void)
{
    int myid = 0x04200420;

    for ( ; ; ) {
        int *ptr = (int *)rnd(memsz);

        ptr[0] = myid;
        ptr[1] = myid;
        ptr[2] = myid;
        ptr[3] = myid;
    }
}
```

In this final example, we get by with only one stack frame for **run()**.

Inlining Code

The idea of inlining code is to avoid function calls, especially for small operations and ones that are done often (say, from inside loops).

Macros can be used for extreme portability, but **__inline__** and related attributes have been around for so long already (not in C89) that they are often a better bet; macros are next to impossible to debug.

Here is a GCC example. rdtsc() was first introduced in the chapter **Inline Assembly** elsewhere in this book. This is an example that uses inline in concert with static in header files, so the declaration is only visible in one file at a time.

```
static __inline__ uint64_t
rdtsc(void)
{
    rdtsc_t tsval;

    __asm__ ("rdtsc\n"
             "movl %%eax, %0\n"
             "movl %%edx, %1\n"
             : "=m" (tsval.u32[0]),
               "=m" (tsval->u32[1])
             : /* no INPUT field */
             : "eax", "edx");

    return tsval.u64;
}
```

Here is the same code changed to a macro. This one works even with compilers not supporting the use of keywords such as **inline** or **__inline__**.

```
/* write RDTSC to address tp in memory */
#define rdtsc(tp)                    \
            __asm__ ("rdtsc\n"     \
                     "movl %%eax, %0\n     \
                     "movl %%edx, %1\n"     \
                     : "=m" ((tp)->u32[0])  \
                       "=m" ((tp)->u32[1])  \
                     : /* no INPUT field */ \
                     : "eax", "edx")
```

22.4 Unrolling Loops

This section describes a technique which good compilers utilise extensively.

Chances are you don't need to unroll by hand, but I think it's good to see how to do it and even a good compiler might not do it when you want to.

This section represents use of so-called **Duff's device**.

22.4.1 Basic Idea

The idea of loop unrolling is to run the code for several loop iterations during one. This is to avoid loop-overhead, mostly of checking if the loop is to be reiterated, and perhaps, with modern CPUs, to utilise pipeline-parallelism better.

I will illustrate loop unrolling with a simplified piece of code to set memory to zero (a common operation to initialise global data structures as well as those one gets from malloc(); the latter can be asked to be zeroed explicitly by using calloc()). This one assumes sizeof(long); for a better version, see section **Duff's Device** below.

Source Code

```
void
pagezero(void *addr, size_t len)
{
    long *ptr = addr;
    long val = 0;
    long incr = 4;

    len >>= 4;
    while (len--) {
        ptr[0] = val;
        ptr[1] = val;
        ptr[2] = val;
        ptr[3] = val;
        ptr += incr;
    }
}
```

22.5 Branches

As a rule of thumb, if and when you have to use branches, order them so that the most likely ones are listed first. This way, you will do fewer checks for branches not to be taken.

22.5.1 if - else if - else

It pays to put the most likely branches (choices) to be taken as far up in the flow as possible.

22.5.2 switch

Switch is useful, e.g. for implementing Duff's devices.

22.5.2.1 Duff's Device

Duff's Device

Duff's device can best be demonstrated by how to use it. Here is a version of our function pagezero() above programmed using one. Pay attention to how the switch falls through when not using **break** to terminate it. This example also attempts to figure out sizeof(long) at compile-time by consulting compiler implementation's type limits.

Source Code

```
/* use Duff's device for unrolling loop */

#include <stdint.h>
#include <limits.h>

/* determine size of long */
#if (ULONG_MAX == 0xffffffffUL)
#define LONGBITS 32
#elif (ULONG_MAX == 0xffffffffffffffffUL)
#define LONGBITS 64
#else
#error pagezero not supported for your word-size        /* don't compile */
#endif

void
pagezero(void *addr, size_t len)
{
    long *ptr = addr;
    long val = 0;
    long incr = 4;
#if (LONGBITS == 32)
    long mask = 0x0000000fL;
#elif (LONGBITS == 64)
    long mask = UINT64_C(0x000000000000000ffL);
#endif

#if (LONGBITS == 32)
    len >>= 4;
#elif (LONGBITS == 64)
    len >>= 5;
#endif
    while (len) {
        /* Duff's device */
        switch (len & mask) {
            case 0:
                ptr[3] = val;
            case 1:
                ptr[2] = val;
            case 2:
                ptr[1] = val;
            case 3:
                ptr[0] = val;
        }
        ptr += incr;
        len--;
    }
}
```

22.5.3 Jump Tables

Sometimes there's a way to get around branching with if - elseif - else or switch statements by making close observations on the values you decide branch targets on.

As an example, I'll show you how to optimise event loops, which practically all X11 clients (application programs) use.

Here, the thing to notice is that instead of the possibly worst case of doing something like

```
XEvent ev;

XNextEvent(disp, &event);
if (ev.type == Expose) {
    /* handle Expose events */
} else if (ev.type == ButtonPress) {
    /* handle ButtonPress events */
} else if (ev.type == ButtonRelease) {
    /* handle ButtonRelease events */
} /* and so forth. */
```

which can easily grow into a dozen else if branches or more,

one could do something like

```
/* ... */
switch (ev.type) {
    case Expose:
        /* handle Expose events */

        break;
    case ButtonPress:
        /* handle ButtonPress events */

        break;
    case ButtonRelease:
        /* handle ButtonRelease events */

        break;
    /* and so on */
    default:

        break;
}
```

which a good compiler might find a way to optimise to a jump table, it's worth one's attention to take a look at event **number** definitions in <X11/X.h>

```
/* excerpts from <X11/X.h> */
/* ... */
#define ButtonPress 4
#define ButtonPress 5
```

```
/* ... */
#define Expose      12
/* ... */
#define LASTEvent   36 /* must be bigger than any event # */
```

As we can see, not only are event numbers small integral constants greater than 0 (0 and 1 are reserved for protocol errors and replies), but an upper limit for them is also defined. Therefore, it is possible, for standard (i.e. non-extension) Xlib events, to do something like

```
#include <X11/X.h>
/* event handlers take event pointer argument */
typedef void evfunc_t(XEvent *);

evfunc_t  evftab[LASTEvent]; /* zeroed at startup */
evfunc_t *evfptr;
XEvent ev;

XNextEvent(disp, &ev);
evfptr = evftab[ev->type]
if (evfptr) {
    evfptr(&ev);
}
```

Function Pointers

In short, we typedef (for convenience) a new type for event handler **function pointers** and use event numbers to index a table of them. In case we find a non-zero (non-NULL) pointer, there is a handler set for the event type and we will call it, passing a pointer to our just-read event to it. Not only is the code faster than the earlier versions, but it is also cleaner and more elegant if you ask me.

Dynamic Approach

It is also possible to extend this scheme to handle extension events if you allocate the handler function pointer table dynamically at run time.

22.6 Bit Operations

In these examples, the following conventions are used

Variable	Requirements	Notes
p2	power of two	one 1-bit, the rest are zero
l2	base-2 logarithm	
w	integral value	

```
w ^ w equals 0.

w ^ 0xffffffff equals ~w.

if l2 raised by 2 is p2 and w is unsigned,
```

```
w >> l2 is equal to w / p2 and

w << l2 is equal to w * l2.

if and only if p2 is power of two,

p2 % (p2 - 1) equals 0.

~0x00000000L equals 0xffffffffL [equals (-1L)].
```

Notes

- ISO C standard states results of **right shifts of negative values** are undefined. The C standard also doesn't specify whether right shifts are logical (fill with zero) or arithmetic (fill high bits with sign).

22.6.1 Karnaugh Maps

TODO: show how to use Karnaugh maps to optimise Boolean stuff.

22.6.2 Techniques and Tricks

I will start this section with what seems a somewhat rarely used trick.

A double-linked list item typically has two pointers in each item; prev and next, which point to the previous and next item in a list, respectively. With a little bit magic and knowing one of the pointers at access time, we can pack two pointers into one integral value (probably of the standard type uintptr_t).

```
pval = (uintptr_t)ptr1 ^ (uintptr_t)ptr2;

/* do something here */

/* pval properties */
p1 = (void *)(pval ^ (uintptr_t)ptr2);
p2 = (void *)(pval ^ (uintptr_t)ptr1);
```

We can also remove one of the value by XORing the value of their XOR with the other one, so

```
op1 = (void *)(pval ^ p2); // op1 == ptr1
op2 = (void *)(pval ^ p1); // op2 == ptr2
```

would give us the original values of ptr1 and ptr2.

In other words, the XOR logical function is used so that XORing the packed value with one pointer evaluates to the integral value of the second one.

Note that you can't remove items from the middle of a list implemented using this technique if you don't know the address of either the previous or next item. Hence, you should only use it for lists when you operate by traversing them in order. This could be useful for a kernel pager LRU queues; the list would allow us to add (push)

page item just allocated or paged in front of the queue and remove (dequeue) pages to be written out from the back. The structure would then serve as a stack as well as a simplified two-end list.

This looks fruitful; a trivial structure for implementing such a list would look like

```
struct page {
    uintptr_t    adr;
    struct page *prev;
    struct page *next;
};
```

This would be changed to

```
struct page {
    uintptr_t adr;
    uintptr_t xptr; // XOR of prev and next.
};
```

If we have a table of such structures, we may not even need the address field; the address space is linear and if there is a structure for every page starting from the address zero and **pagetab is a pointer to the table**, we can do

```
#define PTRPGBITS 12 // i386

/* calculate page address from structure offset */
#define pageadr(pp) \
    ((uintptr_t)((pp) - (pagetab)) << PTRPGBITS)

/* minimal page structure */
struct page {
    uintptr_t xptr; // XOR of prev and next.
};
```

The i386 has virtual address space of 1024 * 1024 pages, so the savings compared to the first version are (1024 * 1024 * 64 bits) which is 8 megabytes; we'd only use 4 megabytes instead of 12 for the page structure table, and even the second version would use 8.

22.7 Small Techniques

22.7.1 Constant Folding

22.7.2 Code Hoisting

Loop invariant motion

Taking everything unnecessary out of loops, especially inner ones, can pay back nicely. A good compiler should know to do this, but it's still good to know what is going on.

```
        do {
            *ptr++ = 0;
```

```
} while (ptr < (char *)dest + nb);
```

We can hoist the addition out of the loop continuation test.

```
char *lim = (char *)dest + nb;

do {
    *ptr++ = 0;
} while (ptr < lim);
```

22.8 Memory Access

C programmers see memory as flat table of bytes. It is good to access memory in as big units as you can; this is about words, cachelines, and ultimately pages.

22.8.1 Alignment

As a rule of thumb, align to the size of the item aligned; e.g.

Alignment	Common Types
1-byte	int8_t, uint8_t, char, unsigned char
2-byte	int16_t, uint16_t, short
4-byte	int32_t, uint32_t, int, long **for 32-bit**
8-byte	int64_t, uint64_t, long on **64-bit**, long long
16-byte	long double if **128-bit**

Assumed Type Sizes

The table above lists sized-types (recommended), but also common assumptions to make it easier for you to read existing code or write code for older pre-C99 compilers.

22.8.2 Access Size

Try to keep memory access sizes aligned to word, cacheline, and page boundaries. Keep closely-related data close in memory not to use too many cachelines. Access memory in words rather than bytes where possible (**alignment!**).

22.8.2.1 Alignment

Many systems raise a signal on unaligned word access of memory, and even the ones that don't will need to read two words and combine the result. Therefore, keep your **word access aligned to word boundaries** at all times.

```
if p2 is power of two, a pointer is
aligned to p2-boundary if

    ((uintptr_t)ptr & ((p2) - 1)) == 0
```

This leads to the macro

```
#define aligned(p, p2) \
    (((uintptr_t)(p) & ((p2) - 1)) == 0)
```

Which can also be written as

```
#define aligned(p, p2) \
    (!((uintptr_t)(p) & ((p2) - 1)))
```

Which one of these two forms is more readable is a matter of taste.

22.8.3 Cache

Typical microprocessors have 2-3 levels of cache memory running at different speeds. The L1 (on-die) cache is the fastest. Memory is read into cache a **cacheline or stride** at a time; on a typical IA-32 architecture, the cacheline is **32 bytes**, i.e. 256 bits. By using local cache parameters and word-access wisely, you can have good wins in code run speeds.

22.8.3.1 Cache Prewarming

Pentium Writeback Trick

Interestingly, it looks like some Pentium-class systems such as my AMD Athlon XP, seem to write cachelines faster if they read the first item of the cacheline to be written into a register first. For example, see the sections on **pagezero()** below. The trick is to make sure the cacheline is in cache memory to avoid writing to main memory directly with the Pentium writeback semantics. It depends on the application whether this usage of the cache speeds things up.

22.9 Code Examples

22.9.1 pagezero()

Here I make a few assumptions to simplify things. This could be used verbatim at kernel-level as the name of the function, **pagezero**, suggests.

The requirements (which make all but the trivial version unuseful as implementations of **memset()**, an implementation of which is found elsewhere in this book), for this function are

TODO: fix item #3

- The region to be zeroed must be **aligned** to a boundary of long, i.e. its address is an even multiple of sizeof(long).
- The size of the region is a multiple of sizeof(long).
- In the unrolled versions, the size of the region must be a multiple of 4 * sizeof(long).

Note that even though some of these implementations may seem silly, I have seen most if not all of them reading code. Everyone makes mistakes and has to end improving things if, say, deadlines are to be met. After all, computer programming is an ongoing learning process which is one of the reasons it can be so satisfactory. It also seems good to look at slower code to see how it can be improved.

22.9.1.1 Algorithms

pagezero1()

In short, we set memory to 0 a **long** at a time. This is the trivial and slowest version.

Source Code

```
/* we assume sizeof(long) is 4 */

void
pagezero1(void *adr, size_t len)
{
    long *ptr = adr;

    len >>= 2;
    while (len--) {
        *ptr++ = 0;
    }
}
```

pagezero2()

Let us unroll the loop to make the code run faster.

Source Code

```
void
pagezero2(void *adr, size_t len)
{
    long *ptr = adr;

    len >>= 2;
    while (len) {
        *ptr++ = 0;
        *ptr++ = 0;
        *ptr++ = 0;
        *ptr++ = 0;
        len -= 4;
    }
}
```

pagezero3()

Let us, without thinking of it twice, replace the subtraction of 4 from len because INC

(decrement by one) might be a faster machine instruction than SUB (generic subtraction).

Source Code

```
void
pagezero3(void *adr, size_t len)
{
    long *ptr = adr;

    len >>= 4;
    while (len--) {
        *ptr++ = 0;
        *ptr++ = 0;
        *ptr++ = 0;
        *ptr++ = 0;
    }
}
```

As DIV (division) tends to be a very slow operation and 4 is a power of two, I also used

```
len >> 4;
```

instead of

```
len /= 16;
```

or, better

```
len /= 4 * sizeof(long);
```

which a good compiler should do as well.

This may be a bit better, but still quite pathetic.

pagezero4()

There's a data dependency on **ptr**, whose value changes right after we use it and so right before we use it again. Fortunately, it is easy to eliminate this speed issue.

Let's try

Source Code

```
void
pagezero4(void *adr, size_t len)
{
    long *ptr = adr;
    size_t ofs = 0;

    len >>= 4;
    while (len--) {
        ptr[ofs] = 0;
        ptr[ofs + 1] = 0;
        ptr[ofs + 2] = 0;
        ptr[ofs + 3] = 0;
```

```
        ptr += 4;
    }
}
```

pagezero5()

Again, this looks better. However, as you can see, we are doing unnecessary calculations adding constants to **ofs**. Time to change the code again. As it turns out, we don't need the variable ofs at all.

Source Code

```
void
pagezero5(void *adr, size_t len)
{
    long *ptr = adr;

    len >>= 4;
    while (len--) {
        ptr[0] = 0;
        ptr[1] = 0;
        ptr[2] = 0;
        ptr[3] = 0;
        ptr += 4;
    }
}
```

There's at least one more reason why this should be better than the previous version in addition to the fact that we eliminated a variable and a bunch of addition operations; IA-32 supports indexed addressing with immediate 8-bit index constants (embedded to machine instructions), and a good compiler should make this version use 8-bit immediate indices.

pagezero6()

There is still one more thing a good compiler should do that I will show for the sake of your knowledge. Let us eliminate the possibility of a non-optimising compiler (or optimising one running with the optimisations turned off, which is common practice when compiling code to be debuggable) doing the memory writes by replicating a MOV with the constant zero as immediate operand.

Source Code

```
void
pagezero6(void *adr, size_t len)
{
    long *ptr = adr;
    long val = 0;

    len >>= 4;
    while (len--) {
        ptr[0] = val;
```

```
        ptr[1] = val;
        ptr[2] = val;
        ptr[3] = val;
        ptr += 4;
    }
}
```

Now, with a bit of luck, val is assigned a register, the instructions [without the immediate operands] shorter and so the loop more likely to use less code cache to reduce 'trashing' it, and last but not least, the size of the compiled binary should be smaller.

pagezero7()

As one more change, let's try replacing the increment constant 4 within the loop with a variable (hopefully register). Note that most of the time, the **register** keyword should not be used because it forces compilers to allocate a register for the whole runtime of the function, therefore making the set of available registers for other computations smaller.

Source Code

```
void
pagezero7(void *adr, size_t len)
{
    long *ptr = adr;
    long val = 0;
    long incr = 4;

    len >>= 4;
    while (len--) {
        ptr[0] = val;
        ptr[1] = val;
        ptr[2] = val;
        ptr[3] = val;
        ptr += incr;
    }
}
```

pagezero8()

This version of the routine adds a bit of portability. Note that you can't use **sizeof(long)** to define **LONGBITS**; this makes the code need to be modified for different systems; not a hard thing to port.

pagezero8() also moves the loop counter decrement [by one] operation to the end of the loop; it doesn't need to be executed right after checking it in the beginning.

Source Code

```
#define LONGBITS 32
void
pagezero8(void *adr, size_t len)
{
```

```
    long *ptr = adr;
    long val = 0;
    long incr = 4;

#if (LONGBITS == 32)
    len >>= 4;
#elif (LONGBITS == 64)
    len >>= 5;
#endif
    while (len) {
        ptr[0] = val;
        ptr[1] = val;
        ptr[2] = val;
        ptr[3] = val;
        len--;
        ptr += incr;
    }
}
```

pagezero9() and test

I read the good book **Inner Loops** by **Rick Booth** to learn what my old friend **Eric B. Mitchell** calls **cache warming**; Rick explains how Pentiums use writeback cache in a way that they write directly to main memory if the cacheline being written is not in cache. This is probably the reason a cacheline read before writing the cacheline dropped pagezero()'s runtime from 12 microseconds for pagezero8() to 9 for pagezero9() on the system I tested them on. A worthy speedup. Note also how I let memory access settle for a bit by moving other operations in between reading memory and writing it. As a **Pentium-detail**, the beast has 8 data buses to cache, one for each 4-byte entity of the cacheline, so writes here should use all 8 buses and work fast. With the Pentium parameters of 32-byte cache lines and 32-bit long words, this loop writes a single cache line of zeroes each loop iteration.

Some of the header files, such as **zen.h** needed to build the examples in this book are represented in the part **Code Examples**.

Source Code

```
#include <stdio.h>
#include <stdlib.h>
#include "cdecl.h"
#include "zen.h"

/* we assume sizeof(long) is 4 */
#define LONGBITS 32

uint8_t pagetab[1024 * 1024] __attribute__((__aligned__(4096)));

unsigned long
profzen(void (*routine)(void *, size_t), char *str)
{
```

```
    zenclk_t      clk;
    unsigned long nusec;
    unsigned long mintime;
    long          l;

    memset(pagetab, 0xff, sizeof(pagetab));
    sleep(1);
    for (l = 0 ; l < 1024 ; l++) {
        zenstartclk(clk);
        routine(pagetab, 65536);
        zenstopclk(clk);
        nusec = zenclkdiff(clk);
        if (nusec < mintime) {
            mintime = nusec;
        }
    }
    fprintf(stderr, "%s took %lu microseconds\n", str, mintime);

    return nusec;
}

void
pagezero9(void *adr, size_t len)
{
    long *next = adr;
    long *ptr;
    long val = 0;
    long incr = 8;
    long tmp;

#if (LONGBITS == 32)
    len >>= 5;
#elif (LONGBITS == 64)
    len >>= 6;
#endif
    while (len) {
        tmp = *next;
        len--;
        ptr = next;
        ptr[0] = val;
        ptr[1] = val;
        ptr[2] = val;
        ptr[3] = val;
        next += incr;
        ptr[4] = val;
        ptr[5] = val;
        ptr[6] = val;
        ptr[7] = val;
    }
}
```

```
int
main(int argc, char *argv[])
{
    profzen(pagezero9, "pagezero9");

    exit(0);
}
```

22.9.1.2 Statistics

Let's take a look at the speed of our different versions of the pagezero routine and look at how to measure execution timer using the Zen timer represented in its own chapter elsewhere in this book.

Here is a test program; I have included the routines to let you not have to skim this book back and forth to see how they work, therefore making it easy to compare the impact of the changes on the run speed.

Note that the tests are run on a multitasking system (without not much other activity, though). I take the minimum of 1024 runs so I can eliminate the impact of the process possibly getting scheduled out, i.e. put to sleep, in the middle of the tests. I also try to avoid this by sleeping (to let the kernel schedule us out, then back in) before I start running the routine to be tested.

Source Code

TODO: include stats for pagezero8() and pagezero9()

```
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include "cdecl.h"
#include "zen.h"
#include "zenia32.h"

#define LONGSIZE     4
#define LONGSIZELOG2 2

void
pagezero0(void *adr, size_t len)
{
    char *ptr = adr;

    while (len--) {
        *ptr++ = 0;
    }
}

void
```

```
pagezero1(void *adr, size_t len)
{
    long *ptr = adr;

    len >>= LONGSIZELOG2;
    while (len--) {
        *ptr++ = 0;
    }
}

void
pagezero2(void *adr, size_t len)
{
    long *ptr = adr;

    len >>= LONGSIZELOG2;
    while (len) {
        *ptr++ = 0;
        *ptr++ = 0;
        *ptr++ = 0;
        *ptr++ = 0;
        len -= 4;
    }
}

void
pagezero3(void *adr, size_t len)
{
    long *ptr = adr;

    len >>= 2 + LONGSIZELOG2;
    while (len--) {
        *ptr++ = 0;
        *ptr++ = 0;
        *ptr++ = 0;
        *ptr++ = 0;
    }
}

void
pagezero4(void *adr, size_t len)
{
    long *ptr = adr;
    size_t ofs = 0;

    len >>= 2 + LONGSIZELOG2;
    while (len--) {
        ptr[ofs] = 0;
        ptr[ofs + 1] = 0;
        ptr[ofs + 2] = 0;
```

```
        ptr[ofs + 3] = 0;
        ptr += 4;
    }
}

void
pagezero5(void *adr, size_t len)
{
    long *ptr = adr;

    len >>= 2 + LONGSIZELOG2;
    while (len--) {
        ptr[0] = 0;
        ptr[1] = 0;
        ptr[2] = 0;
        ptr[3] = 0;
        ptr += 4;
    }
}

void
pagezero6(void *adr, size_t len)
{
    long *ptr = adr;
    long val = 0;

    len >>= 2 + LONGSIZELOG2;
    while (len--) {
        ptr[0] = val;
        ptr[1] = val;
        ptr[2] = val;
        ptr[3] = val;
        ptr += 4;
    }
}

void
pagezero7(void *adr, size_t len)
{
    long *ptr = adr;
    long val = 0;
    long incr = 4;

    len >>= 2 + LONGSIZELOG2;
    while (len--) {
        ptr[0] = val;
        ptr[1] = val;
        ptr[2] = val;
        ptr[3] = val;
        ptr += incr;
```

```
    }
}

//uint8_t pagetab[4096] __attribute__((__aligned__(4096)));
uint8_t *pagetab[128];

unsigned long
profzen(void (*routine)(void *, size_t), char *str)
{
#if (PROFCLK)
    zenclk_t      clk;
    unsigned long cnt;
    unsigned long mintime = 0;
    long          l;
#elif (PROFTICK)
    zentick_t     tick;
#if (LONGSIZE == 8)
    long          cnt;
    long          mintime = 0x7fffffffffffffffL;
#else
    long long     cnt;
    long long     mintime = 0x7fffffffffffffffLL;
#endif
    long          l;
#endif

    sleep(1);
    for (l = 0 ; l < 128 ; l++) {
        pagetab[l] = malloc(4096);
#if (PROFCLK)
        zenstartclk(clk);
#elif (PROFTICK)
        zenstarttick(tick);
#endif
        routine(pagetab[l], 4096);
#if (PROFCLK)
        zenstopclk(clk);
        cnt = zenclkdiff(clk);
#elif (PROFTICK)
        zenstoptick(tick);
        cnt = zentickdiff(tick);
#endif
        if (cnt < mintime) {
            mintime = cnt;
        }
    }
    for (l = 0 ; l < 128 ; l++) {
        free(pagetab[l]);
    }
#if (PROFCLK)
```

```
    fprintf(stderr, "%s took %lu microseconds\n", str, mintime);
#elif (PROFTICK)
    fprintf(stderr, "%s took %lld cycles\n", str, mintime);
#endif

    return cnt;
}

void
pagezero8(void *adr, size_t len)
{
    long *ptr = adr;
    long val = 0;
    long incr = 4;

    len >>= 2 + LONGSIZELOG2;
    while (len) {
        ptr[0] = val;
        ptr[1] = val;
        ptr[2] = val;
        ptr[3] = val;
        ptr += incr;
        len--;
    }
}

void
pagezero9(void *adr, size_t len)
{
    long *next = adr;
    long *ptr;
    long val = 0;
    long incr = 8;
    long tmp;

    len >>= 3 + LONGSIZELOG2;
    while (len) {
        tmp = *next;
        len--;
        ptr = next;
        ptr[0] = val;
        ptr[1] = val;
        ptr[2] = val;
        ptr[3] = val;
        next += incr;
        ptr[4] = val;
        ptr[5] = val;
        ptr[6] = val;
        ptr[7] = val;
    }
```

```
}

void
pagezero10(void *adr, size_t len)
{
    long *next1 = adr;
    long *next2 = (long *)((uint8_t *)adr + (len >> 1));
    long *ptr1;
    long *ptr2;
    long val = 0;
    long incr = 8;
    long tmp1;
    long tmp2;

    len >>= 4 + LONGSIZELOG2;
    while (len) {
        __builtin_prefetch(next1);
        __builtin_prefetch(next2);
//        tmp1 = *next1;
//        tmp2 = *next2;
        len--;
        ptr1 = next1;
        ptr2 = next2;
        ptr1[0] = val;
        ptr2[0] = val;
        ptr1[1] = val;
        ptr2[1] = val;
        ptr1[2] = val;
        ptr2[2] = val;
        ptr1[3] = val;
        ptr2[3] = val;
        next1 += incr;
        next2 += incr;
        ptr1[4] = val;
        ptr2[4] = val;
        ptr1[5] = val;
        ptr2[5] = val;
        ptr1[6] = val;
        ptr2[6] = val;
        ptr1[7] = val;
        ptr2[7] = val;
    }
}

void
pagezero11(void *adr, size_t len)
{
    long *next1 = adr;
    long *next2 = (long *)((uint8_t *)adr + 2048);
    long *next3 = (long *)((uint8_t *)adr + 8 * sizeof(long));
```

```
long *next4 = (long *)((uint8_t *)adr + 2048 + 8 * sizeof(long));
long *ptr1;
long *ptr2;
long val = 0;
long incr = 8;

len >>= 4 + LONGSIZELOG2;
while (--len) {
    __builtin_prefetch(next1);
    __builtin_prefetch(next2);
    __builtin_prefetch(next3);
    __builtin_prefetch(next4);
    ptr1 = next1;
    ptr2 = next2;
    ptr1[0] = val;
    ptr2[0] = val;
    ptr1[1] = val;
    ptr2[1] = val;
    ptr1[2] = val;
    ptr2[2] = val;
    ptr1[3] = val;
    ptr2[3] = val;
    next1 += incr;
    next2 += incr;
    next3 += incr;
    next4 += incr;
    ptr1[4] = val;
    ptr2[4] = val;
    ptr1[5] = val;
    ptr2[5] = val;
    ptr1[6] = val;
    ptr2[6] = val;
    ptr1[7] = val;
    ptr2[7] = val;
}
ptr1 = next1;
ptr2 = next2;
ptr1[0] = val;
ptr2[0] = val;
ptr1[1] = val;
ptr2[1] = val;
ptr1[2] = val;
ptr2[2] = val;
ptr1[3] = val;
ptr2[3] = val;
ptr1[4] = val;
ptr2[4] = val;
ptr1[5] = val;
ptr2[5] = val;
ptr1[6] = val;
```

```
    ptr2[6] = val;
    ptr1[7] = val;
    ptr2[7] = val;
}

void
pagezero12(void *adr, size_t len)
{
    long *next1 = adr;
    long *next2 = (long *)((uint8_t *)adr + 2048);
    long *next3 = (long *)((uint8_t *)adr + 8 * sizeof(long));
    long *next4 = (long *)((uint8_t *)adr + 2048 + 8 * sizeof(long));
    long *ptr1;
    long *ptr2;
    long val = 0;
    long incr = 8;

    len >>= 4 + LONGSIZELOG2;
    while (--len) {
        __builtin_prefetch(next1);
        ptr1 = next1;
        ptr2 = next2;
        __builtin_prefetch(next2);
        ptr1[0] = val;
        ptr2[0] = val;
        ptr1[1] = val;
        ptr2[1] = val;
        __builtin_prefetch(next3);
        ptr1[2] = val;
        ptr2[2] = val;
        ptr1[3] = val;
        ptr2[3] = val;
        __builtin_prefetch(next4);
        next1 += incr;
        next2 += incr;
        next3 += incr;
        next4 += incr;
        ptr1[4] = val;
        ptr2[4] = val;
        ptr1[5] = val;
        ptr2[5] = val;
        ptr1[6] = val;
        ptr2[6] = val;
        ptr1[7] = val;
        ptr2[7] = val;
    }
    ptr1 = next1;
    ptr2 = next2;
    ptr1[0] = val;
    ptr2[0] = val;
```

```
    ptr1[1] = val;
    ptr2[1] = val;
    ptr1[2] = val;
    ptr2[2] = val;
    ptr1[3] = val;
    ptr2[3] = val;
    ptr1[4] = val;
    ptr2[4] = val;
    ptr1[5] = val;
    ptr2[5] = val;
    ptr1[6] = val;
    ptr2[6] = val;
    ptr1[7] = val;
    ptr2[7] = val;
}

#if 0 /* BROKEN CODE */
void
pagezero13(void *adr, size_t len)
{
    long *next1 = adr;
    long *next2 = (long *)((uint8_t *)adr + 4096);
    long *next3 = (long *)((uint8_t *)adr + 8 * sizeof(long));
    long *next4 = (long *)((uint8_t *)adr + 4096 + 8 * sizeof(long));
    long *ptr1;
    long *ptr2;
    long val = 0;
    long incr = 8;

    len >>= 4 + LONGSIZELOG2;
    while (--len) {
        __builtin_prefetch(next1);
        ptr1 = next1;
        ptr2 = next2;
        __builtin_prefetch(next2);
        ptr1[0] = val;
        ptr2[0] = val;
        ptr1[1] = val;
        ptr2[1] = val;
        __builtin_prefetch(next3);
        ptr1[2] = val;
        ptr2[2] = val;
        ptr1[3] = val;
        ptr2[3] = val;
        __builtin_prefetch(next4);
        next1 += incr;
        next2 += incr;
        next3 += incr;
        next4 += incr;
        ptr1[4] = val;
```

```
        ptr2[4] = val;
        ptr1[5] = val;
        ptr2[5] = val;
        ptr1[6] = val;
        ptr2[6] = val;
        ptr1[7] = val;
        ptr2[7] = val;
    }
    ptr1 = next1;
    ptr2 = next2;
    ptr1[0] = val;
    ptr2[0] = val;
    ptr1[1] = val;
    ptr2[1] = val;
    ptr1[2] = val;
    ptr2[2] = val;
    ptr1[3] = val;
    ptr2[3] = val;
    ptr1[4] = val;
    ptr2[4] = val;
    ptr1[5] = val;
    ptr2[5] = val;
    ptr1[6] = val;
    ptr2[6] = val;
    ptr1[7] = val;
    ptr2[7] = val;
}

#endif /* BROKEN CODE */

int
main(int argc, char *argv[])
{
    profzen(pagezero0, "pagezero0");
    profzen(pagezero1, "pagezero1");
    profzen(pagezero2, "pagezero2");
    profzen(pagezero3, "pagezero3");
    profzen(pagezero4, "pagezero4");
    profzen(pagezero5, "pagezero5");
    profzen(pagezero6, "pagezero6");
    profzen(pagezero7, "pagezero7");
    profzen(pagezero8, "pagezero8");
    profzen(pagezero9, "pagezero9");
    profzen(pagezero10, "pagezero10");
    profzen(pagezero11, "pagezero11");
    profzen(pagezero12, "pagezero12");
//    profzen(pagezero13, "pagezero13");

    exit(0);
}
```

Here are the minimum run times (those are the ones that count here) I saw running the test several times. They seem consistent; I ran the tests several times. These times came from the program compiled with compiler optimisations on (**-O** flag with GCC).

```
pagezero1 took 32 microseconds
pagezero2 took 11 microseconds
pagezero3 took 11 microseconds
pagezero4 took 14 microseconds
pagezero5 took 11 microseconds
pagezero6 took 11 microseconds
pagezero7 took 12 microseconds
```

This shows that setting words instead of bytes pays back in a marvelous way. Let's look at the times without compiler optimisations to see if anything else made difference; compilers may do things such as unroll loops themselves.

```
pagezero1 took 110 microseconds
pagezero2 took 102 microseconds
pagezero3 took 102 microseconds
pagezero4 took 81 microseconds
pagezero5 took 63 microseconds
pagezero6 took 63 microseconds
pagezero7 took 63 microseconds
```

The results are interesting - note how big the impact of compiler optimisations is. The changes we applied have gained a bit of speed, but it really becomes noticeable only after we compile with the GCC **-O** flag. The latter statistics do show, though, that what we did was somewhat fruitful. The big speedups were word-size access to memory and unrolling the loop. As you can see, you should turn optimisations off or lower if you really want to measure your own code's execution times in a way to be somewhat trustworthy.

22.10 Data Examples

22.10.1 Bit Flags

TODO: bitset(), setbit(), clrbit(), getbits(), tagged pointers, examples.

22.10.2 Lookup Tables

Sometimes it's good to cache (relatively small) sets of computed values into tables and fetch them based on the operands of such computation. This technique is used later in this chapter; look at the section **Fade In/Out Effects**.

TODO: packing character attribute bit flags into tables.

22.10.3 Hash Tables

TODO

22.10.4 The V-Tree

Here is a hybrid data structure I came up with when investigating **van Emde Boas trees**. Even though it is not suitable for sparsely scattered key values (it would eat all memory in the universe), it's interesting for what it can be used; the plan is to look into using it to drive kernel buffer cache. Its main use would be relatively linear key spaces with no key collisions.

Highlights of this data structure in comparison with hash tables are:

- Not counting the (relatively rare) table allocations of this dynamic data structure, **INSERT**, **FIND**, and **DELETE** operations work in 'constant'/**predictable** time. The biggest interference with run-time are occasional allocations and deallocations of internal tables.
- The structure can be iterated in key order.
- It is relatively easy to implement lookups for the next and previous valued keys.

22.10.4.1 Example Implementation

The listing below has embedded tests to make it easier to explore.

The example implementation below shows using 2-level trees (of tables) and demonstrates using bitmaps to speed implementations of FINDNEXT and FINDPREV up; it is noteworthy that with 8-bit level indices, a 256-bit lookup bitmap will fit a single i386 cacheline. It then takes 8 32-bit zero comparisons to spot an empty subtree, which is much faster than 256 32-bit [pointer] comparisons.

Source Code

```
/*
 * Copyright (C) 2008-2010 Tuomo Petteri Venäläinen. All rights reserved.
 */

#define NTESTKEY (64 * 1024)

#define TEST     1
#define CHK      0
#define PROF     1

#include <stdint.h>
#include <stdlib.h>
#include <limits.h> /* CHAR_BIT */
#include <string.h>
#if (PROF)
#include <unistd.h>
```

```
#include "cdecl.h"
#include "zen.h"
#endif

#define bitset(p, b) (((uint8_t *)(p))[(b) >> 3] & (1U << ((b) & 0x07)))
#define setbit(p, b) (((uint8_t *)(p))[(b) >> 3] |= (1U << ((b) & 0x07)))
#define clrbit(p, b) (((uint8_t *)(p))[(b) >> 3] &= ~(1U << ((b) & 0x07)))

#if (TEST)
#include <stdio.h>
#endif

#define VAL_SIZE    2
#define KEY_SIZE    2

#if (VAL_SIZE <= 4)
typedef uint32_t vtval_t;
#elif (VAL_SIZE <= 8)
typedef uint64_t vtval_t;
#endif
#if (KEY_SIZE <= 4)
typedef uint32_t vtkey_t;
#elif (KEY_SIZE <= 8)
typedef uint64_t vtkey_t;
#endif
typedef vtval_t _VAL_T;

#define _NKEY           (1U << _NBIT)
#define _NBIT           (KEY_SIZE * CHAR_BIT)
#define _NLVLBIT        (_NBIT >> 1)
#define _EMPTYBYTE      0xff
#define _EMPTYVAL       (~(vtval_t)0)
#define _EMPTYKEY       (~(vtkey_t)0)

#define _hi(k, n)       ((k) >> (n))
#define _lo(k, n)       ((k) & ((1U << (n)) - 1))
#define _calloc(n, t)   calloc(1 << (n), sizeof(t))
#define _alloc(n, t)    malloc((1 << (n)) * sizeof(t))
#if (PROF)
#define _memset(p, b, n) do { *(p)++ = (b); } while (--(n))
#define _flush(p, n)    _memset(p, 0xff, n);
#endif
#define _clrtab(p, n)   memset(p, _EMPTYBYTE,                          \
                               (((1) << (n)) * sizeof(_VAL_T)))

struct  _item {
    _VAL_T   *tab;
    vtkey_t   minkey;
    vtkey_t   maxkey;
    uint32_t  bmap[1U << (_NLVLBIT - 5)];
```

```
} PACK;

struct _tree {
    struct _item *tab;
    vtval_t      *reftab;
    vtkey_t       nbit;
    uint32_t      himap[1U << (_NLVLBIT - 5)];
};

static vtval_t vtins(struct _tree *tree, vtkey_t key, vtval_t val);
static vtval_t vtdel(struct _tree *tree, vtkey_t key);
static vtval_t vtfind(struct _tree *tree, vtkey_t key);
static vtval_t vtprev(struct _tree *tree, vtkey_t key);
static vtval_t vtnext(struct _tree *tree, vtkey_t key);

struct _tree *
mkveb(int nkeybit)
{
    struct _tree  *tree = malloc(sizeof(struct _tree));
    vtkey_t        nbit = nkeybit >> 1;
    unsigned long  n = 1U << nbit;
    unsigned long  ndx = 0;
    size_t         tabsz;
    void          *ptr;
    struct _item  *item;

    if (tree) {
        tree->nbit = nbit;
        tabsz = n * sizeof(struct _item);
        ptr = malloc(tabsz);
        if (ptr) {
            tree->tab = ptr;
            item = ptr;
            memset(ptr, _EMPTYBYTE, tabsz);
            ptr = NULL;
            while (ndx < n) {
                item->tab = ptr;
                ndx++;
                item++;
            }
            tabsz = n * sizeof(vtval_t);
            ptr = calloc(1, tabsz);
            if (ptr) {
                tree->reftab = ptr;
            } else {
                free(tree->tab);
                free(tree);
                tree = NULL;
            }
        } else {
```

```
            free(tree);
            tree = NULL;
        }
    }

    return tree;
}

static vtval_t
vtins(struct _tree *tree, vtkey_t key, vtval_t val)
{
    vtkey_t       nbit = tree->nbit;
    vtkey_t       hi = _hi(key, nbit);
    vtkey_t       lo = _lo(key, nbit);
    struct _item *treep = &tree->tab[hi];
    _VAL_T       *tabp = treep->tab;
    vtkey_t       tkey = _EMPTYKEY;
    vtval_t       retval = _EMPTYVAL;

    if (!tabp) {
        treep->minkey = treep->maxkey = tkey;
        treep->tab = tabp = _alloc(nbit, _VAL_T);
#if (_EMPTYBYTE != 0)
        _clrtab(tabp, nbit);
#endif
    }
    setbit(tree->himap, hi);
    setbit(treep->bmap, lo);
    if (tabp) {
        tree->reftab[hi]++;
        if (lo < treep->minkey || treep->minkey == tkey) {
            treep->minkey = lo;
        }
        if (lo > treep->maxkey || treep->maxkey == tkey) {
            treep->maxkey = lo;
        }
        tabp[lo] = val;
        retval = val;
    }

    return retval;
}

static vtval_t
vtdel(struct _tree *tree, vtkey_t key)
{
    vtkey_t       nbit = tree->nbit;
    vtkey_t       hi = _hi(key, nbit);
    vtkey_t       lo = _lo(key, nbit);
    struct _item *treep = &tree->tab[hi];
```

```
_VAL_T       *tabp = treep->tab;
_VAL_T       *valp;
vtval_t       tval = _EMPTYVAL;
vtkey_t       tkey = _EMPTYKEY;
vtkey_t       lim = 1U << nbit;
vtval_t       retval = tval;
vtval_t       val;

if (tabp) {
    clrbit(treep->bmap, lo);
    if (tabp) {
        valp = &tabp[lo];
        retval = *valp;
        if (retval != tval) {
            if (!--tree->reftab[hi]) {
                clrbit(tree->himap, hi);
                free(tabp);
                treep->tab = NULL;
                treep->minkey = treep->maxkey = tkey;
            } else {
                *valp = tval;
                if (lo == treep->minkey) {
                    val = _EMPTYVAL;
                    do {
                        ;
                    } while ((++lo < lim)
                             && ((val = valp[lo]) == _EMPTYVAL));
                    if (valp[lo] == _EMPTYVAL) {
                        treep->minkey = tkey;
                    } else {
                        treep->minkey = lo;
                    }
                }
                if (lo == treep->maxkey) {
                    val = _EMPTYVAL;
                    do {
                        ;
                    } while ((lo--)
                             && ((val = valp[lo]) == _EMPTYVAL));
                    if (val == _EMPTYVAL) {
                        treep->maxkey = tkey;
                    } else {
                        treep->maxkey = lo;
                    }
                }
            }
        }
    }
}
```

```
    return retval;
}

static vtval_t
vtfind(struct _tree *tree, vtkey_t key)
{
    vtkey_t       nbit = tree->nbit;
    vtkey_t       hi = _hi(key, nbit);
    vtkey_t       lo = _lo(key, nbit);
    struct _item *treep = &tree->tab[hi];
    _VAL_T       *tabp = treep->tab;
    vtval_t       retval = _EMPTYVAL;

    if (!tabp) {

        return retval;
    }
    retval = tabp[lo];

    return retval;
}

static vtval_t
vtprev(struct _tree *tree, vtkey_t key)
{
    vtkey_t       nbit = tree->nbit;
    vtkey_t       hi = _hi(key, nbit);
    vtkey_t       lo = _lo(key, nbit);
    struct _item *treep = &tree->tab[hi];
    _VAL_T       *tabp = treep->tab;
    _VAL_T       *valp;
    vtkey_t       kval;
    vtval_t       tval = _EMPTYVAL;
    vtval_t       retval = tval;
    uint32_t     *himap = tree->himap;
    uint32_t     *lomap = (treep) ? treep->bmap : NULL;

    if (!tabp || treep->minkey == _EMPTYKEY) {

        return retval;
    }
    if (lo > treep->minkey) {
        valp = tabp;
        do {
            ;
        } while (lo-- > 0 && !bitset(lomap, lo));
        retval = valp[lo];
    } else {
        do {
            ;
```

```
        } while (hi-- > 0 && !bitset(himap, hi));
        treep = &tree->tab[hi];
        kval = treep->maxkey;
        tabp = treep->tab;
        retval = tabp[kval];
    }

    return retval;
}

static vtval_t
vtnext(struct _tree *tree, vtkey_t key)
{
    vtkey_t       nbit = tree->nbit;
    vtkey_t       hi = _hi(key, nbit);
    vtkey_t       lo = _lo(key, nbit);
    vtkey_t       lim = 1U << nbit;
    struct _item *treep = &tree->tab[hi];
    _VAL_T       *tabp = treep->tab;
    _VAL_T       *valp;
    vtkey_t       kval;
    vtval_t       tval = _EMPTYVAL;
    vtval_t       retval = tval;
    uint32_t     *himap = tree->himap;
    uint32_t     *lomap = (treep) ? treep->bmap : NULL;

    if (!tabp || treep->maxkey == _EMPTYKEY) {

        return retval;
    }
    if (lo < treep->maxkey) {
        valp = tabp;
        do {
            ;
        } while (++lo < lim && !bitset(lomap, lo));
        retval = valp[lo];
    } else {
        do {
            ;
        } while (++hi < lim && !bitset(himap, hi));
        treep = &tree->tab[hi];
        kval = treep->minkey;
        tabp = treep->tab;
        retval = tabp[kval];
    }

    return retval;
}

#if (TEST)
```

```
#if (PROF)
static uint8_t _mtab[2 * 1048576]; // TODO: fix PAGEALIGN;
#define START_PROF(id) sleep(1); p = _mtab, n = 2 * 1048576; _flush(p, n); zenstartclk(id)
#define STOP_PROF(id, str)  zenstopclk(id); fprintf(stderr, "%s\t%lu usecs\n", \
                                                    str, zenclkdiff(id))
#else
#define START_PROF(id)
#define STOP_PROF(id, str)
#endif

void
test(void)
{
    uint8_t      *sysheap = (uint8_t *)sbrk(0);
    struct _tree *tree = mkveb(_NBIT);
    uint8_t      *curheap;
    int val;
    int i;
#if (PROF)
    uint8_t *p;
    int n;
    zenclk_t clock;
#endif

    START_PROF(clock);
    for (i = 0 ; i < NTESTKEY - 1 ; i++) {
        val = vtins(tree, i, i);
#if (CHK)
        if (val != i) {
            fprintf(stderr, "insert(1) failed - %d should be %d\n", val, i);

            abort();
        }
#endif
    }
    STOP_PROF(clock, "insert\t");

    START_PROF(clock);
    for (i = 0 ; i < NTESTKEY - 1 ; i++) {
        val = vtfind(tree, i);
#if (CHK)
        if (val != i) {
            fprintf(stderr, "lookup(1) failed - %d should be %d\n", val, i);

            abort();
        }
#endif
    }
    STOP_PROF(clock, "lookup\t");
```

```
    START_PROF(clock);
    for (i = 1 ; i < NTESTKEY - 1 ; i++) {
        if (i) {
            val = vtprev(tree, i);
#if (CHK)
            if (val != i - 1) {
                fprintf(stderr, "vtprev(%x) failed (%x)\n", i, val);

                abort();
            }
#endif
        }
    }
    STOP_PROF(clock, "findprev");

    START_PROF(clock);
    for (i = 0 ; i < NTESTKEY - 2 ; i++) {
        val = vtnext(tree, i);
#if (CHK)
        if (val != i + 1) {
            fprintf(stderr, "vtnext(%x) failed (%x)\n", i, val);

            abort();
        }
#endif
    }
    STOP_PROF(clock, "findnext");

    START_PROF(clock);
    for (i = 0 ; i < NTESTKEY - 1 ; i++) {
        val = vtdel(tree, i);
#if (CHK)
        if (val != i) {
            fprintf(stderr, "vtdel(%x) failed (%x)\n", i, val);

            abort();
        }
#endif
    }
    STOP_PROF(clock, "delete\t");

    curheap = (uint8_t *)sbrk(0);
    fprintf(stderr, "HEAP:\t\t%u bytes\n", curheap - sysheap);
    fprintf(stderr, "RANGE:\t\t%x..%x\n", 0, NTESTKEY - 1);

    for (i = 0 ; i < NTESTKEY - 1 ; i++) {
        val = vtfind(tree, i);
        if (val != _EMPTYVAL) {
            fprintf(stderr, "lookup(2) failed\n");
```

```
            abort();
        }
    }


    return;
}

int
main(int argc,
     char *argv[])
{
    test();

    exit(0);
}

#endif /* TEST */
```

22.11 Graphics Examples

In this section we shall look at some simple graphical operations.

First, some basic pixel definitions for ARGB32. We use the de facto standard ARGB32 pixel format (32-bit, 8 bits for each of ALPHA, RED, GREEN, and BLUE).

Bytefields

One thing to notice in the following listing is the difference of **alphaval()** and **alphaval_p()**. The first one is used when you have a pixel packed into a 32-bit word; the second one lets you fetch individual component bytes from memory to avoid fetching a whole pixel and doing bitshifts. I decided to call **struct argb32** a **bytefield**. Note that you have to ask the compiler not to try and align struct argb32 better with some kind of an attribute; we use PACK, which is defined for GCC in **<cc.h>**.

Source Code

```
#include "cc.h"

#define ALPHAOFS 24
#define REDOFS   16
#define GREENOFS 8
#define BLUEOFS  0

typedef int32_t argb32_t;

struct argb32 {
    uint8_t bval;
```

```
    uint8_t gval;
    uint8_t rval;
    uint8_t aval;
};

/* pix is 32-bit word */
#define alphaval(pix) ((pix) >> ALPHAOFS)                // alpha component
#define redval(pix)   (((pix) >> REDOFS) & 0xff)        // red component
#define greenval(pix) (((pix) >> GREENOFS) & 0xff)       // green component
#define blueval(pix)  (((pix) >> BLUEOFS) & 0xff)        // blue component

/* pointer version; faster byte-fetches from memory */
#define alphaval_p(p) (((struct argb32 *)(p))->aval)
#define redval_p(p)   (((struct argb32 *)(p))->rval)
#define greenval_p(p) (((struct argb32 *)(p))->gval)
#define blueval_p(p)  (((struct argb32 *)(p))->bval)

/* approximation for c / 0xff */
#define div255(c)                                                \
    ((((c) << 8) + (c) + 256) >> 16)
/* simple division per 256 by bitshift */
#define div256(c)                                                \
    ((c) >> 8)
#define alphablendc(src, dest, a)                                \
    ((dest) + div255(((src) - (dest)) * (a)))
#define alphablendc2(src, dest, a)                               \
    ((dest) + div256(((src) - (dest)) * (a)))
#define alphablendcf(src, dest, a)                               \
    ((dest) + (((src) - (dest)) * (a)) / 255.0)

/* compose pixel value from components */
#define mkpix(a, r, g, b)                                        \
    (((a) << ALPHAOFS) | ((r) << REDOFS) | ((g) << GREENOFS) | ((b) << BLUEOFS))
#define setpix_p(p, a, r, g, b)                                  \
    (((struct argb32 *)(p))->aval = (a),                         \
     ((struct argb32 *)(p))->rval = (r),                         \
     ((struct argb32 *)(p))->gval = (g),                         \
     ((struct argb32 *)(p))->bval = (b))
```

22.11.1 Alpha Blending

Alpha blending is a technique to combine two pixel values so that the source pixel is drawn on top of the destination pixel using the alpha value (translucency level) from the source. This is how modern desktop software implements 'translucent' windows and such.

We are going to see some serious bit-twiddling acrobacy; I got the algorithm from **Carsten 'Rasterman' Haitzler**, but all I know of its origins is that it came from some fellow hacker called Jose.

Note that these routines were implemented as macros to make it easy to drop them into loops without using (slow) function calls every iteration.

Performance-wise, Jose's algorithm seems to be the fastest. It took about 3.3 seconds for a crossfade operation of two 1024x768 images using the first alpha blend routine. The second one took about 2.9 seconds to execute. Jose's took about 2.6 seconds. For the record, the initial floating point routine took a bit over 6 seconds to run.

Towards the end of this section, we take a closer look at how alpha blending works as well as examine vector programming by developing a couple of MMX versions of our routines.

22.11.1.1 C Routines

Pixels are alpha blended a component, i.e. one of RED, GREEN or BLUE, at a time. The formula for computing blended components is

$$DEST = DEST + (((SRC - DEST) * ALPHA)/255)$$

where DEST, SRC, and ALPHA are 8-bit component values in the range 0 to 255. One thing to notice is the divide operation; this tends to be slow for microprocessors to accomplish, but luckily we have ways around it; note though, that those ways aren't 100 percent accurate so chances are you don't want to use them for professional quality publications and such applications. In this book, we concentrate on their use on on-screen graphics/images.

Here is an exact floating point algorithm. Note that it uses a divide operation, which tends to be slow. This one takes about double the time to run in comparison to the integer routines.

Source Code

```
#include "pix.h"
#include "blend.h"

#define alphablendf(src, dest, aval)                                        \
    do {                                                                     \
        float _a = (aval);                                                   \
        float _sr = redval_p(src);                                           \
        float _sg = greenval_p(src);                                         \
        float _sb = blueval_p(src);                                          \
        float _dr = redval_p(dest);                                          \
        float _dg = greenval_p(dest);                                        \
        float _db = blueval_p(dest);                                         \
                                                                             \
        _dr = alphablendcf(_sr, _dr, _a);                                    \
        _dg = alphablendcf(_sg, _dg, _a);                                    \
        _db = alphablendcf(_sb, _db, _a);                                    \
        setpix_p((dest), 0, (uint8_t)_dr, (uint8_t)_dg, (uint8_t)_db);  \
    } while (FALSE)
```

Eliminating the divide operation, the runtime drops by around 25 percent for my test runs. Still quite a bit slower than the integer routines, but may give more exact output.

Source Code

```
/* t is table of 256 floats (0 / 0xff through 255.0 / 0xff) */
#define alphablendcf2(src, dest, a, t)                                      \
    ((dest) + (((src) - (dest)) * (a)) * (t)[(a)])

#define alphablendf2(src, dest, aval)                                       \
    do {                                                                    \
        argb32_t _a = (aval);                                               \
        float    _sr = redval_p(src);                                       \
        float    _sg = greenval_p(src);                                     \
        float    _sb = blueval_p(src);                                      \
        float    _dr = redval_p(dest);                                      \
        float    _dg = greenval_p(dest);                                    \
        float    _db = blueval_p(dest);                                     \
                                                                            \
        _dr = alphablendcf2(_sr, _dr, _a);                                  \
        _dg = alphablendcf2(_sg, _dg, _a);                                  \
        _db = alphablendcf2(_sb, _db, _a);                                  \
        setpix_p((dest), 0, (uint8_t)_dr, (uint8_t)_dg, (uint8_t)_db);      \
    } while (FALSE)
```

Here is the first integer algorithm. This one should be reasonably good in terms of output quality. Notice how the macros hide the somewhat tedious pixel component calculations and make the code easier to digest.

Source Code

```
#include "pix.h"
#include "blend.h"

#define alphablendhiq(src, dest, aval)                                      \
    do {                                                                    \
        argb32_t _a = (aval);                                               \
        argb32_t _sr = redval(src);                                         \
        argb32_t _sg = greenval(src);                                       \
        argb32_t _sb = blueval(src);                                        \
        argb32_t _dr = redval(dest);                                        \
        argb32_t _dg = greenval(dest);                                      \
        argb32_t _db = blueval(dest);                                       \
                                                                            \
        _dr = alphablendc(_sr, _dr, _a);                                    \
        _dg = alphablendc(_sg, _dg, _a);                                    \
        _db = alphablendc(_sb, _db, _a);                                    \
        (dest) = mkpix(0, _dr, _dg, _db);                                   \
    } while (FALSE)

#define alphablendhiq_p(src, dest, aval)                                    \
```

```
    do {                                                    \
        argb32_t _a = (aval);                               \
        argb32_t _sr = redval_p(src);                       \
        argb32_t _sg = greenval_p(src);                     \
        argb32_t _sb = blueval_p(src);                      \
        argb32_t _dr = redval_p(dest);                      \
        argb32_t _dg = greenval_p(dest);                    \
        argb32_t _db = blueval_p(dest);                     \
                                                            \
        _dr = alphablendc(_sr, _dr, _a);                    \
        _dg = alphablendc(_sg, _dg, _a);                    \
        _db = alphablendc(_sb, _db, _a);                    \
        setpix_p((dest), 0, _dr, _dg, _db);                 \
    } while (FALSE)
```

The next listing is the previous routine modified to use a faster approximation for divide-by-0xff operations; we simply divide by 256 doing bitshifts. It would seem to cut a bit over 10 percent off the runtime of our code under some tests.

Source Code

```
#include "pix.h"
#include "blend.h"

#define alphablendloq(src, dest, aval)                      \
    do {                                                    \
        argb32_t _a = (aval);                               \
        argb32_t _sr = redval(src);                         \
        argb32_t _sg = greenval(src);                       \
        argb32_t _sb = blueval(src);                        \
        argb32_t _dr = redval(dest);                        \
        argb32_t _dg = greenval(dest);                      \
        argb32_t _db = blueval(dest);                       \
                                                            \
        _dr = alphablendc2(_sr, _dr, _a);                   \
        _dg = alphablendc2(_sg, _dg, _a);                   \
        _db = alphablendc2(_sb, _db, _a);                   \
        (dest) = mkpix(0, _dr, _dg, _db);                   \
    } while (FALSE)

#define alphablendloq_p(src, dest, aval)                    \
    do {                                                    \
        argb32_t _a = (aval);                               \
        argb32_t _sr = redval_p(src);                       \
        argb32_t _sg = greenval_p(src);                     \
        argb32_t _sb = blueval_p(src);                      \
        argb32_t _dr = redval_p(dest);                      \
        argb32_t _dg = greenval_p(dest);                    \
        argb32_t _db = blueval_p(dest);                     \
                                                            \
```

```
        _dr = alphablendc2(_sr, _dr, _a);                                   \
        _dg = alphablendc2(_sg, _dg, _a);                                   \
        _db = alphablendc2(_sb, _db, _a);                                   \
        *(dest) = mkpix(0, _dr, _dg, _db);                                  \
    } while (FALSE)
```

The algorithm I told about above; the one that came from Jose. This is about 10 percent faster than the bitshifting version.

Source Code

```
#include "pix.h"
#include "blend.h"

/* Jose's fast alphablend-algorithm */

#define alphablendpix(c0, c1, a)                                            \
    (((((((c0) >> 8) & 0xff00ff) - (((c1) >> 8) & 0xff00ff)) * (aval)) \
       + ((c1) & 0xff00ff00)) & 0xff00ff00)                                 \
     + (((((((c0) & 0xff00ff) - ((c1) & 0xff00ff)) * (aval)) >> 8)      \
         + ((c1) & 0xff00ff)) & 0xff00ff))

#define alphablendfast(src, dest, aval)                                     \
    do {                                                                    \
        uint64_t _rbmask = 0x00ff00ff00ff00ffULL;                           \
        argb32_t _gamask = 0xff00ff00ff00ff00ULL;                           \
        argb32_t _srcrb;                                                    \
        argb32_t _destrb;                                                   \
        argb32_t _destag;                                                   \
        argb32_t _val1;                                                     \
        argb32_t _val2;                                                     \
                                                                            \
        _srcrb = (src);                                                     \
        _destrb = (dest);                                                   \
        _destag = (dest);                                                   \
        _srcrb &= _rbmask;                                                  \
        _destrb &= _rbmask;                                                 \
        _destag &= _gamask;                                                 \
        _val1 = (src);                                                      \
        _val2 = _destag;                                                    \
        _val1 >>= 8;                                                        \
        _val2 >>= 8;                                                        \
        _val1 &= _rbmask;                                                   \
        _srcrb -= _destrb;                                                  \
        _val1 -= _val2;                                                     \
        _srcrb *= (aval);                                                   \
        _val1 *= (aval);                                                    \
        _srcrb >>= 8;                                                       \
        _val1 += _destag;                                                   \
        _srcrb += _destrb;                                                  \
```

```
        _val1 &= _gamask;                                               \
        _srcrb &= _rbmask;                                              \
        _val1 += _srcrb;                                                \
        (dest) = _val1;                                                 \
    } while (0)
```

22.11.1.2 MMX Routines

The basic idea of vectorisation is to work on multiple data operands in parallel; the term SIMD (Single Instruction, Multiple Data) is commonly used for this.

Remember that alpha blending works by doing computations on pixel components. These components are 8 bits each. We did find ways to get around the division, but we still need to do multiplications, which means we can't do our calculations in 8-bit registers; there will be overflows. The way MMX comes to the rescue is that we can represent the 4 components of a pixel as 16-bit values in a 64-bit register (technically, the alpha component wouldn't be needed) and effectively do the substraction, multiplication, and addition operations on all those components (as 16-bit subcomponents) in parallel. We'll get away with fewer machine operations.

The first listing uses Intel compiler intrinsics for MMX as a way to avoid assembly. As the intrinsics at the time of writing this don't cover everything we need (64-bit move using the MOVQ machine instruction for) and this is a book on low level programming, we shall next rewrite the routine using inline assembly.

It is noteworthy that one needs to exit MMX mode with the assembly instruction **emms** to make the floating-point unit (**i387**) work correctly. Therefore, every time you stop using MMX instructions, do something like

```
__asm__ __volatile__ ("emms\n");
```

Here is the intrinsics version of our second integer alpha blending routine.

Source Code

```
#include <mmintrin.h>        /* MMX compiler intrinsics */

#include "pix.h"
#include "blend.h"

/* NOTE: leaves destination ALPHA undefined */
#define alphablendloq_mmx(src, dest, aval)                              \
    do {                                                                \
        __m64 _mzero;                                                   \
        __m64 _msrc;                                                    \
        __m64 _mdest;                                                   \
        __m64 _malpha;                                                  \
        __m64 _mtmp;                                                    \
                                                                        \
        _mzero = _mm_cvtsi32_si64(0);             /* 0000000000000000 */ \
        _malpha = _mm_cvtsi32_si64(aval);         /* 00000000000000AA */ \
```

```
        _mtmp = _mm_slli_si64(_malpha, 16);         /* 0000000000AA0000 */ \
        _malpha = _mm_or_si64(_mtmp, _malpha);      /* 0000000000AA00AA */ \
        _mtmp = _mm_slli_si64(_malpha, 32);         /* 00AA00AA00000000 */ \
        _malpha = _mm_or_si64(_malpha, _mtmp);      /* 00AA00AA00AA00AA */ \
        _msrc = _mm_cvtsi32_si64(src);              /* S:00000000AARRGGBB */ \
        _mdest = _mm_cvtsi32_si64(dest);            /* D:00000000AARRGGBB */ \
        _msrc = _mm_unpacklo_pi8(_msrc, _mzero);    /* S:00AA00RR00GG00BB */ \
        _mdest = _mm_unpacklo_pi8(_mdest, _mzero); /* D:00AA00RR00GG00BB */ \
        _msrc = _mm_sub_pi16(_msrc, _mdest);        /* S - D */           \
        _msrc = _mm_mullo_pi16(_msrc, _malpha);     /* T = (S - D) * A */ \
        _msrc = _mm_srli_pi16(_msrc, 8);            /* T >> 8 */          \
        _mdest = _mm_add_pi8(_msrc, _mdest);        /* D = D + T */        \
        _mdest = _mm_packs_pu16(_mdest, _mzero);    /* D:00000000??RRGGBB */ \
        (dest) = _mm_cvtsi64_si32(_mdest);          /* DEST = D */        \
    } while (FALSE)
```

In tests, this routine turned out to run about 10 percent faster than the C versions. It is noteworthy though that Jose's C algorithm runs faster - good work! :)

Now let's rewrite this using inline assembly.

Source Code

```
#include "pix.h"
#include "blend.h"

#define alphablendloq_mmx_asm(src, dest, aval)                          \
    do {                                                                \
        __asm__ ("pxor %mm0, %mm0\n");                                  \
        __asm__ ("movd %0, %%mm1\n" : : "rm" (src));                    \
        __asm__ ("movd %0, %%mm2\n" : : "rm" (dest));                   \
        __asm__ ("movd %0, %%mm3\n" : : "rm" (aval));                   \
        __asm__ ("punpcklbw %mm0, %mm1\n");                             \
        __asm__ ("movq %mm3, %mm5\n");                                  \
        __asm__ ("punpcklbw %mm0, %mm2\n");                             \
        __asm__ ("psllq $16, %mm5\n");                                  \
        __asm__ ("pxor %mm5, %mm3\n");                                  \
        __asm__ ("movq %mm3, %mm5\n");                                  \
        __asm__ ("psllq $32, %mm5\n");                                  \
        __asm__ ("pxor %mm5, %mm3\n");                                  \
        __asm__ ("psubw %mm2, %mm1\n");                                 \
        __asm__ ("movq %mm1, %mm4\n");                                  \
        __asm__ ("pmullw %mm3, %mm4\n");                                \
        __asm__ ("psrlw $8, %mm4\n");                                   \
        __asm__ ("paddb %mm4, %mm2\n");                                 \
        __asm__ ("packuswb %mm0, %mm2\n");                              \
        __asm__ __volatile__ ("movd %%mm2, %0\n" : "=rm" (dest));       \
    } while (FALSE)
```

This version turned out to be a very little bit faster than Jose's algorithm implemented

in C. What's more interesting, though, is that it cut the runtime of the intrinsics version down from about 2.8 seconds to 2.6 under a crossfade test of about 100 alphablend operations of 1024x768 resolution images. Notice that the final MOVD operation must be declared **__volatile__**. Also beware that it's not a good idea to mix use of regular variables/registers with MMX code.

22.11.1.3 Cross-Fading Images

As an easter egg to those of you who have kept reading, I will show how to crossfade an image to another one (fade the first one out and gradually expose the second one on top of it) using the alphablend routines we have implemented.

In real life, you most likely need to synchronise graphics display after each step; the details of this are platform-dependent.

```
#include "pix.h"

#define STEP 0x0f

/* cross-fade from src1 to src2; dest is on-screen data */
void
crossfade(argb32_ *src1, argb32_t *src2, argb32_t *dest,
      size_t len)
{
    argb32_t val;
    size_t   nleft;

    nleft = len;
    while (nleft--) {
        for (val = 0 ; val <= 0xff ; val += STEP) {
            alphablendfast(src1, dest, 0xff - val);
    alphablendfast(src2, dest, val);
        }
        /* synchronise screen here */
    }
    /* copy second image intact */
            memcpy(dest, src2, len * sizeof(argb32_t));
}
```

22.11.2 Fade In/Out Effects

Here is a simple way to implement graphical fade in and fade out effects. To use this, you would loop over graphical data with the val argument to the macros ranging from 0 to 0xff similarly to what we did in the previous code snippet.

I will use the chance of demonstrating a couple of simple optimisation techniques for this routine. First, it has a division operation and those tend to be slow. That can be emulated by introducing a table of 256 floats to look the desired value up from. This made my test run time drop from about 19000 microseconds to around 17000.

Another way to cut a little bit of the runtime off is to eliminate the (floating point) multiplication operations as well as the casts between float and argb32_t. both **_fmul** and pixel/color component values are 8-bit and so can have 256 different values. This gives us a table of 256 * 256 values of the type **uint8_t** (no need for full pixel values), that is 65536 values. This table uses 64 kilobytes of memory (8-bit values). Chances are you don't want to do this at all; I don't see you needing this routine in games or other such programs which need the very last bit of performance torn out of the machine, but you may have other uses for lookup tables so I'll show you how to do it.

Source Code

```
#include "pix.h"

/* basic version */
#define fadein1(src, dest, val)                                          \
    do {                                                                 \
        argb32_t _rval;                                                  \
        argb32_t _gval;                                                  \
        argb32_t _bval;                                                  \
        float    _ftor;                                                  \
                                                                         \
        _ftor = (float)val / 0xff;                                       \
        _rval = (argb32_t)(_ftor * _gfx_red_val(src));                   \
        _gval = (argb32_t)(_ftor * _gfx_green_val(src));                 \
        _bval = (argb32_t)(_ftor * _gfx_blue_val(src));                  \
        mkpix(dest, 0, _rval, _gval, _bval);                             \
    } while (FALSE)

#define fadeout1(src, dest, val)                                         \
    do {                                                                 \
        argb32_t _rval;                                                  \
        argb32_t _gval;                                                  \
        argb32_t _bval;                                                  \
        float    _ftor;                                                  \
                                                                         \
        _ftor = (float)(0xff - val) / 0xff;                              \
        _rval = (argb32_t)(_ftor * _gfx_red_val(src));                   \
        _gval = (argb32_t)(_ftor * _gfx_green_val(src));                 \
        _bval = (argb32_t)(_ftor * _gfx_blue_val(src));                  \
        mkpix(dest, 0, _rval, _gval, _bval);                             \
    } while (FALSE)

/* use lookup table to eliminate division _and_ multiplication + typecasts */

/*
 * initialise lookup table
 * u8p64k points to 65536 uint8_t values like in
 * uint8_t fadetab[256][256];
 */
#define initfade1(u8p64k)                                                \
    do {                                                                 \
```

```
        long  _l, _m;                                                   \
        float _f;                                                       \
        for (_l = 0 ; _l <= 0xff ; _l++) {                              \
            f = (float)val / 0xff;                                      \
            for (_m = 0 ; _m <= 0xff ; _m++) {                          \
                (u8p64k)[_l][_m] = (uint8_t)(_f * _m);                  \
            }                                                           \
        }                                                               \
    } while (0)

#define fadein2(src, dest, val, tab)                                    \
    do {                                                                \
        _rval = (tab)[val][redval(src)];                                \
        _gval = (tab)[val][greenval(src)];                              \
        _bval = (tab)[val][blueval(src)];                               \
        mkpix(dest, 0, _rval, _gval, _bval);                            \
    } while (FALSE)

#define fadeout(src, dest, val)                                         \
    do {                                                                \
        val = 0xff - val;                                               \
        _rval = (tab)[val][redval(src)];                                \
        _gval = (tab)[val][greenval(src)];                              \
        _bval = (tab)[val][blueval(src)];                               \
        mkpix(dest, 0, _rval, _gval, _bval);                            \
    } while (FALSE)
```

Part X

Code Examples

Chapter 23

Zen Timer

First of all, greetings to **Michael Abrash**; 'sorry' for stealing the name Zen timer, I just thought it sounded good and wanted to pay you respect. ;)

Zen Timer implements timers for measuring code execution in microseconds as well as, currently for IA-32 machines, clock cycles.

23.1 Implementation

23.1.1 Generic Version; gettimeofday()

```
/*
 * Copyright (C) 2005-2010 Tuomo Petteri Venäläinen. All rights reserved.
 */

#ifndef __ZEN_H__
#define __ZEN_H__

#include <stdint.h>
#include <sys/time.h>

typedef volatile struct timeval zenclk_t[2];

#define _tvdiff(tv1, tv2)                       \
    (((tv2)->tv_sec - (tv1)->tv_sec) * 1000000 \
     + ((tv2)->tv_usec - (tv1)->tv_usec))

#define zenzeroclk(id)                          \
    memset(id, 0, sizeof(id))
#define zenstartclk(id)                         \
    gettimeofday(&id[0], NULL)
#define zenstopclk(id)                          \
    gettimeofday(&id[1], NULL)
```

```
#define zenclkdiff(id)                          \
    _tvdiff(&id[0], &id[1])

#endif /* __ZEN_H__ */
```

23.1.2 IA32 Version; RDTSC

```
/*
 * Copyright (C) 2005-2010 Tuomo Petteri Venäläinen. All rights reserved.
 */

#ifndef __ZENIA32_H__
#define __ZENIA32_H__

#include <stdint.h>

union _tickcnt {
    uint64_t u64val;
    uint32_t u32vals[2];
};

typedef volatile union _tickcnt zentick_t[2];

#define _rdtsc(ptr)                                                     \
    __asm__ __volatile__("rdtsc\n"                                      \
                         "movl %%eax, %0\n"                             \
                         "movl %%edx, %1\n"                             \
                         : "=m" ((ptr)->u32vals[0]), "=m" ((ptr)->u32vals[1]) \
                         :                                              \
                         : "eax", "edx");

#define zenzerotick(id)                                                 \
    memset(id, 0, sizeof(id))
#define zenstarttick(id)                                                \
    _rdtsc(&id[0])
#define zenstoptick(id)                                                 \
    _rdtsc(&id[1])
#define zentickdiff(id)                                                 \
    (id[1].u64val - id[0].u64val)

#endif /* __ZENIA32_H__ */
```

Chapter 24

C Library Allocator

malloc() et al

This section shows a sample implementation of a decent, somewhat scalable, thread-safe standard library allocator.

POSIX Threads

The allocator in this listing demonstrates simple thread-techniques; one thing to pay attention to is the use of **__thread** to declare thread-local storage (TLS), i.e. data that is only visible to a single thread. This is used to store thread IDs to allow multiple ones to access the allocator at the same time with less lock contention. **pthread_key_create()** is used to specify a function to reclaim arenas when threads terminate; an arena is reclaimed when there are no more threads attached to it.

For this piece of code, I want to thank **Dale Anderson** and **Matthew Gregan** for their input and Matthew's nice stress-test routines for the allocator. Cheers New Zealand boys! :) There are a few other thank yous in the code comments, too.

The allocator should be relatively fast, thread-safe, and scale nicely. It has not been discontinued, so chances are a thing or a few will change. My main interest is in the runtime-tuning of allocator behavior which has been started in a simple way (see the macro **TUNEBUF**).

24.1 Design

24.1.1 Buffer Layers

Diagram

The following is a simple ASCII diagram borrowed from allocator source.

```
/*
 *        malloc buffer layers
 *        --------------------
 *
 *                --------
 *                | mag  |
 *                --------
 *                    |
 *                --------
 *                | slab |
 *                --------
 *        --------  |  |   -------
 *        | heap |--|  |---| map |
 *        --------         -------
 *
 *        mag
 *        ---
 *        - magazine cache with allocation stack of pointers into the slab
 *          - LIFO to reuse freed blocks of virtual memory
 *
 *        slab
 *        ----
 *        - slab allocator bottom layer
 *        - power-of-two size slab allocations
 *          - supports both heap and mapped regions
 *
 *        heap
 *        ----
 *        - process heap segment
 *          - sbrk() interface; needs global lock
 *
 *        map
 *        ---
 *        - process map segment
 *          - mmap() interface; thread-safe
 */
```

24.1.2 Details

Magazines

The allocator uses a so-called 'Bonwick-style' **buffer** ('magazine') layer on top of a traditional slab allocator. The magazine layer implements allocation stacks [of pointers] for sub-slab regions.

Slabs are power-of-two-size regions. To reduce the number of system calls made, allocations are buffered in magazines. Using pointer stacks for allocations makes reuse of

allocated blocks more likely.

TODO: analyse cache behavior here - with Valgrind?

sbrk() and mmap()

The Zero allocator uses sbrk() to expand process heap for smaller allocations, whereas mmap() is used to allocate bigger chunks of [zeroed] memory. Traditionally, sbrk() is not thread-safe, so a global lock is necessary to protect global data structures; one reason to avoid too many calls to sbrk() (which triggers the 'brk' system call on usual Unix systems). On the other hand, mmap() is thread safe, so we can use a bit finer-grained locking with it.

Thread Safety

Zero allocator uses mutexes to guarantee thread-safety; threads running simultaneously are not allowed to modify global data structures without locking them.

Scalability

The allocator has [currently a fixed number of] arenas. Every thread is given an arena ID to facilitate running several threads doing allocation without lower likeliness of lock contention, i.e. without not having to wait for other threads all the time. Multiprocessor machines are very common today, so this scalability should be good on many, possibly most new systems. Indeed the allocator has shown good performance with multithreaded tests; notably faster than more traditional slab allocators. Kudos to **Bonwick et al** from Sun Microsystems for inventing the magazine layer. :)

24.2 Implementation

24.2.1 UNIX Interface

POSIX/UNIX

On systems that support it, you can activate POSIX system interface with

```
#define _POSIX_SOURCE   1
#define _POSIX_C_SOURCE 199506L
```

In addition to these, you need the **-pthread** compiler/linker option to build POSIX-compliant multithread-capable source code.

Header File

Here is a header file I use to compile the allocator - it lists some other feature macros found on UNIX-like systems.

```
/*
 * Copyright (C) 2007-2008 Tuomo Petteri Venäläinen. All rights reserved.
 */

#ifndef __ZERO_UNIX_H__
#define __ZERO_UNIX_H__
```

```
#if 0
/* system feature macros. */
#if !defined(_ISOC9X_SOURCE)
#define _ISOC9X_SOURCE      1
#endif

#if !defined(_POSIX_SOURCE)
#define _POSIX_SOURCE       1
#endif
#if !defined(_POSIX_C_SOURCE)
#define _POSIX_C_SOURCE     199506L
#endif

#if !defined(_LARGEFILE_SOURCE)
#define _LARGEFILE_SOURCE   1
#endif
#if !defined(_FILE_OFFSET_BITS)
#define _FILE_OFFSET_BITS   64
#endif
#if !defined(_LARGE_FILES)
#define _LARGE_FILES        1
#endif
#if !defined(_LARGEFILE64_SOURCE)
#define _LARGEFILE64_SOURCE 1
#endif
#endif /* 0 */

#include <stdint.h>
#include <signal.h>

/* posix standard header. */
#include <unistd.h>

/* i/o headers. */
#include <fcntl.h>
#include <sys/types.h>
#include <sys/uio.h>
#include <sys/stat.h>
#include <sys/mman.h>

#define _SBRK_FAILED        ((void *)-1L)

#define _MMAP_DEV_ZERO      0 /* set mmap to use /dev/zero. */

/* some systems may need MAP_FILE with MAP_ANON. */
#ifndef MAP_FILE
#define MAP_FILE            0
#endif
#if !defined(MAP_FAILED)
#define MAP_FAILED          ((void *)-1L)
```

```
#endif
#if (defined(MMAP_DEV_ZERO) && MMAP_DEV_ZERO)
#define mapanon(fd, size)                    \
    mmap(NULL, size, PROT_READ | PROT_WRITE, \
         MAP_PRIVATE | MAP_FILE,             \
         fd,                                 \
         0)
#else
#define mapanon(fd, size)                                          \
    mmap(NULL,                                                     \
         size,                                                     \
         PROT_READ | PROT_WRITE,                                   \
         MAP_PRIVATE | MAP_ANON | MAP_FILE,                        \
         fd,                                                       \
         0)
#endif
#define unmapanon(ptr, size)                                       \
    munmap(ptr, size)

#define growheap(ofs) sbrk(ofs)

#endif /* __ZERO_UNIX_H__ */
```

24.2.2 Source Code

Allocator Source

```
/*
 * Copyright (C) 2008-2012 Tuomo Petteri Venäläinen. All rights reserved.
 *
 * See the file LICENSE for more information about using this software.
 */

/*
 *        malloc buffer layers
 *        --------------------
 *
 *                --------
 *                | mag  |
 *                --------
 *                    |
 *                --------
 *                | slab |
 *                --------
 *        --------  |  |   -------
 *        | heap |--|  |---| map |
 *        --------         -------
 *
```

```
 *        mag
 *        ---
 *        - magazine cache with allocation stack of pointers into the slab
 *          - LIFO to reuse freed blocks of virtual memory
 *
 *        slab
 *        ----
 *        - slab allocator bottom layer
 *        - power-of-two size slab allocations
 *          - supports both heap and mapped regions
 *
 *        heap
 *        ----
 *        - process heap segment
 *          - sbrk() interface; needs global lock
 *
 *        map
 *        ---
 *        - process map segment
 *          - mmap() interface; thread-safe
 */

#define INTSTAT 0
#define HACKS   0
#define ZEROMTX 1
#define STAT    0

#define SPINLK  0
/* NOT sure if FreeBSD still needs spinlocks */
#if defined(__FreeBSD__)
#undef SPINLK
#define SPINLK  1
#endif

#ifdef _REENTRANT
#ifndef MTSAFE
#define MTSAFE  1
#endif

/*
 * TODO
 * ----
 * - tune nmbuf() and other behavior
 * - implement mallopt()
 * - improve fault handling
 */

/*
 * THANKS
 * ------
```

```
 * - Matthew 'kinetik' Gregan for pointing out bugs, giving me cool routines to
 *   find more of them, and all the constructive criticism etc.
 * - Thomas 'Freaky' Hurst for patience with early crashes, 64-bit hints, and
 *   helping me find some bottlenecks.
 * - Henry 'froggey' Harrington for helping me fix issues on AMD64.
 * - Dale 'swishy' Anderson for the enthusiasism, encouragement, and everything
 *   else.
 * - Martin 'bluet' Stensgård for an account on an AMD64 system for testing
 *   earlier versions.
 */

#include <features.h>
#include <errno.h>
#include <stddef.h>
#include <stdlib.h>
#include <stdint.h>
#include <stdio.h>

#define SBRK_FAILED ((void *)-1L)

static void   initmall(void);
static void   relarn(void *arg);
static void * getmem(size_t size, size_t align, long zero);
static void   putmem(void *ptr);
static void * _realloc(void *ptr, size_t size, long rel);

/* red-zones haven't been implemented completely yet... some bugs. */
#define RZSZ     0
#define markred(p) (*(uint64_t *)(p) = UINT64_C(0xb4b4b4b4b4b4b4b4))
#define chkred(p)                                                       \
    ((*(uint64_t *)(p) == UINT64_C(0xb4b4b4b4b4b4b4b4))                 \
     ? 0                                                                \
     : 1)

#define LKDBG    0
#define SYSDBG   0
#define VALGRIND 0

#include <string.h>
#if (MTSAFE)
#define PTHREAD  1
#include <pthread.h>
#endif
#endif
#if (ZEROMTX)
#include <zero/mtx.h>
typedef long            LK_T;
#elif (SPINLK)
#include <zero/spin.h>
typedef long            LK_T;
```

```
#elif (PTHREAD)
typedef pthread_mutex_t LK_T;
#endif
#if (VALGRIND)
#include <valgrind/valgrind.h>
#endif
#include <zero/param.h>
#include <zero/cdecl.h>
//#include <mach/mach.h>
#include <zero/trix.h>
#include <zero/unix.h>
//#include <mach/param.h>

#define TUNEBUF 0
/* experimental */
#if (PTRBITS > 32)
#define TUNEBUF 1
#endif

/* basic allocator parameters */
#if (HACKS)
#define BLKMINLOG2    5  /* minimum-size allocation */
#define SLABTEENYLOG2 12 /* little block */
#define SLABTINYLOG2  16 /* small-size block */
#define SLABLOG2      19 /* base size for heap allocations */
#define MAPMIDLOG2    21
#define MAPBIGLOG2    22
#else
#define BLKMINLOG2    5  /* minimum-size allocation */
#define SLABTEENYLOG2 12 /* little block */
#define SLABTINYLOG2  16 /* small-size block */
#define SLABLOG2      20 /* base size for heap allocations */
#define MAPMIDLOG2    22
#endif
#define MINSZ         (1UL << BLKMINLOG2)
#define HQMAX         SLABLOG2
#define NBKT          (8 * PTRSIZE)
#if (MTSAFE)
#define NARN          8
#else
#define NARN          1
#endif

/* lookup tree of tables */

#if (PTRBITS > 32)

#define NL1KEY     (1UL << NL1BIT)
#define NL2KEY     (1UL << NL2BIT)
#define NL3KEY     (1UL << NL3BIT)
```

```
#define L1NDX      (L2NDX + NL2BIT)
#define L2NDX      (L3NDX + NL3BIT)
#define L3NDX      SLABLOG2
#define NL1BIT     16

#if (PTRBITS > 48)

#define NL2BIT     16
#define NL3BIT     (PTRBITS - SLABLOG2 - NL1BIT - NL2BIT)
#else

#define NL2BIT     (PTRBITS - SLABLOG2 - NL1BIT)
#define NL3BIT     0

#endif /* PTRBITS > 48 */

#endif /* PTRBITS <= 32 */

/* macros */

#if (TUNEBUF)
#define isbufbkt(bid)     ((bid) <= 24)
#define nmagslablog2(bid) (_nslabtab[(bid)])
#else
#define isbufbkt(bid)     0
#define nmagslablog2(bid) (ismapbkt(bid) ? nmaplog2(bid) : nslablog2(bid))
#define nslablog2(bid)    0
#define nmaplog2(bid)     0
#define nslablog2(bid)    0
#define nmaplog2(bid)     0
#endif

#if (TUNEBUF)
/* adjust how much is buffered based on current use */
#define nmagslablog2up(m, v, t)                                          \
    do {                                                                 \
        if (t >= (v)) {                                                  \
            for (t = 0 ; t < NBKT ; t++) {                               \
                _nslabtab[(t)] = m(t);                                   \
            }                                                            \
        }                                                                \
    } while (0)
#if (HACKS)
#define nmagslablog2init(bid) 0
#define nmagslablog2m64(bid)                                             \
    ((ismapbkt(bid))                                                     \
     ? (((bid) <= MAPBIGLOG2)                                            \
        ? 2                                                              \
        : 1)                                                             \
     : (((bid) <= SLABTEENYLOG2)                                         \
```

```
        ? 0                                                     \
        : (((bid) <= SLABTINYLOG2)                              \
           ? 1                                                  \
           : 2)))
#define nmagslablog2m128(bid)                                   \
    ((ismapbkt(bid))                                            \
     ? (((bid) <= MAPBIGLOG2)                                   \
        ? 2                                                     \
        : 1)                                                    \
     : (((bid) <= SLABTEENYLOG2)                                \
        ? 0                                                     \
        : (((bid) <= SLABTINYLOG2)                              \
           ? 0                                                  \
           : 1)))
#define nmagslablog2m256(bid)                                   \
    ((ismapbkt(bid))                                            \
     ? (((bid) <= MAPBIGLOG2)                                   \
        ? 2                                                     \
        : 1)                                                    \
     : (((bid) <= SLABTEENYLOG2)                                \
        ? 0                                                     \
        : (((bid) <= SLABTINYLOG2)                              \
           ? 0                                                  \
           : 0)))
#define nmagslablog2m512(bid)                                   \
    ((ismapbkt(bid))                                            \
     ? (((bid) <= MAPBIGLOG2)                                   \
        ? 1                                                     \
        : 0)                                                    \
     : (((bid) <= SLABTEENYLOG2)                                \
        ? 0                                                     \
        : 0))
#else
#define nmagslablog2init(bid)                                   \
    ((ismapbkt(bid))                                            \
     ? (((bid) <= 23)                                           \
        ? 2                                                     \
        : 1)                                                    \
     : (((bid) <= SLABTEENYLOG2)                                \
        ? 1                                                     \
        : (((bid) <= SLABTINYLOG2)                              \
           ? 1                                                  \
           : 2)))
#define nmagslablog2m64(bid)                                    \
    ((ismapbkt(bid))                                            \
     ? 0                                                        \
     : (((bid) <= SLABTEENYLOG2)                                \
        ? 0                                                     \
        : (((bid) <= SLABTINYLOG2)                              \
           ? 1                                                  \
```

```
           : 2)))
#define nmagslablog2m128(bid)                                   \
    ((ismapbkt(bid))                                            \
     ? (((bid) <= 23)                                           \
        ? 1                                                     \
        : 0)                                                    \
     : (((bid) <= SLABTEENYLOG2)                                \
        ? 1                                                     \
        : (((bid) <= SLABTINYLOG2)                              \
           ? 1                                                  \
           : 2)))
#define nmagslablog2m256(bid)                                   \
    ((ismapbkt(bid))                                            \
     ? (((bid) <= 24)                                           \
        ? 1                                                     \
        : 0)                                                    \
     : (((bid) <= SLABTEENYLOG2)                                \
        ? 1                                                     \
        : (((bid) <= SLABTINYLOG2)                              \
           ? 1                                                  \
           : 2)))
#define nmagslablog2m512(bid)                                   \
    ((ismapbkt(bid))                                            \
     ? (((bid) <= 24)                                           \
        ? 1                                                     \
        : 0)                                                    \
     : (((bid) <= SLABTEENYLOG2)                                \
        ? 0                                                     \
        : (((bid) <= SLABTINYLOG2)                              \
           ? 1                                                  \
           : 2)))
#endif
#endif
#define nblklog2(bid)                                           \
    ((!(ismapbkt(bid))                                          \
      ? (SLABLOG2 - (bid))                                      \
      : nmagslablog2(bid)))
#define nblk(bid)         (1UL << nblklog2(bid))
#define NBSLAB            (1UL << SLABLOG2)
#define nbmap(bid)        (1UL << (nmagslablog2(bid) + (bid)))
#define nbmag(bid)        (1UL << (nmagslablog2(bid) + SLABLOG2))

#if (PTRBITS <= 32)
#define NSLAB             (1UL << (PTRBITS - SLABLOG2))
#define slabid(ptr)       ((uintptr_t)(ptr) >> SLABLOG2)
#endif
#define nbhdr()           PAGESIZE
#define NBUFHDR           16

#define thrid()           ((_aid >= 0) ? _aid : (_aid = getaid()))
```

```
#define blksz(bid)        (1UL << (bid))
#define usrsz(bid)        (blksz(bid) - RZSZ)
#define ismapbkt(bid)     (bid > HQMAX)
#define magfull(mag)      (!(mag)->cur)
#define magempty(mag)     ((mag)->cur == (mag)->max)
#if (ALNSTK)
#define nbstk(bid)        max(nblk(bid) * sizeof(void *), PAGESIZE)
#define nbalnstk(bid)     nbstk(bid)
#else
#define nbstk(bid)        max((nblk(bid) << 1) * sizeof(void *), PAGESIZE)
#endif
#define mapstk(n)         mapanon(_mapfd, ((n) << 1) * sizeof(void *))
#define unmapstk(mag)     unmapanon((mag)->bptr, mag->max * sizeof(void *))
#define putblk(mag, ptr)                                                \
    ((gt2(mag->max, 1)                                                  \
      ? (((void **)(mag)->bptr)[--(mag)->cur] = (ptr))                  \
      : ((mag)->cur = 0, (mag)->adr = (ptr))))
#define getblk(mag)                                                     \
    ((gt2(mag->max, 1)                                                  \
      ? (((void **)(mag)->bptr)[(mag)->cur++])                          \
      : ((mag)->cur = 1, ((mag)->adr))))
#define NPFBIT BLKMINLOG2
#define BPMASK (~((1UL << NPFBIT) - 1))
#define BDIRTY 0x01UL
#define BALIGN 0x02UL
#define clrptr(ptr)       ((void *)((uintptr_t)(ptr) & BPMASK))
#define setflg(ptr, flg)  ((void *)((uintptr_t)(ptr) | (flg)))
#define chkflg(ptr, flg)  ((uintptr_t)(ptr) & (flg))
#define blkid(mag, ptr)                                                 \
    ((mag)->max + (((uintptr_t)(ptr) - (uintptr_t)(mag)->adr) >> (mag)->bid))
#define putptr(mag, ptr1, ptr2)                                         \
    ((gt2((mag)->max, 1))                                               \
     ? (((void **)(mag)->bptr)[blkid(mag, ptr1)] = (ptr2))              \
     : ((mag)->bptr = (ptr2)))
#define getptr(mag, ptr)                                                \
    ((gt2((mag)->max, 1))                                               \
     ? (((void **)(mag)->bptr)[blkid(mag, ptr)])                        \
     : ((mag)->bptr))

#if (STAT)
#include <stdio.h>
#endif

/* synchonisation */

#if (ZEROMTX)
#define mlk(mp)           mtxlk(mp, _aid + 1)
#define munlk(mp)         mtxunlk(mp, _aid + 1)
#define mtylk(mp)         mtxtrylk(mp, _aid + 1)
#elif (SPINLK)
```

```
#define mlk(sp)           spinlk(sp)
#define munlk(sp)         spinunlk(sp)
#define mtrylk(sp)        spintrylk(sp)
#elif (MTSAFE)
#if (PTHREAD)
#define mlk(sp)           pthread_mutex_lock(sp)
#define munlk(sp)         pthread_mutex_unlock(sp)
#define mtrylk(sp)        pthread_mutex_trylock(sp)
#else
#define mlk(sp)           spinlk(sp)
#define munlk(sp)         spinunlk(sp)
#define mtrylk(sp)        spintrylk(sp)
#endif
#else
#define mlk(sp)
#define munlk(sp)
#define mtrylk(sp)
#endif
#define mlkspin(sp)       spinlk(sp)
#define munlkspin(sp)     spinunlk(sp)
#define mtrylkspin(sp)    spintry(sp)

/* configuration */

#define CONF_INIT 0x00000001
#define VIS_INIT  0x00000002
struct mconf {
    long        flags;
#if (MTSAFE)
    LK_T        initlk;
    LK_T        arnlk;
    LK_T        heaplk;
#endif
    long        scur;
    long        acur;
    long        narn;
};

#define istk(bid)                                                   \
    ((nblk(bid) << 1) * sizeof(void *) <= PAGESIZE)
struct mag {
    long        cur;
    long        max;
    long        aid;
    long        bid;
    void       *adr;
    void       *bptr;
    struct mag *prev;
    struct mag *next;
    struct mag *stk[EMPTY];
```

```
};

#define nbarn() (blksz(bktid(sizeof(struct arn))))
struct arn {
    struct mag  *btab[NBKT];
    struct mag  *ftab[NBKT];
    long         nref;
    long         hcur;
    long         nhdr;
    struct mag **htab;
    long         scur;
    LK_T         lktab[NBKT];
};

struct mtree {
#if (MTSAFE)
    LK_T         lk;
#endif
    struct mag **tab;
    long         nblk;
};

/* globals */

#if (INTSTAT)
static uint64_t        nalloc[NARN][NBKT];
static long            nhdrbytes[NARN];
static long            nstkbytes[NARN];
static long            nmapbytes[NARN];
static long            nheapbytes[NARN];
#endif
#if (STAT)
static unsigned long   _nheapreq[NBKT] ALIGNED(PAGESIZE);
static unsigned long   _nmapreq[NBKT];
#endif
#if (TUNEBUF)
static long            _nslabtab[NBKT];
#endif
#if (MTSAFE)
static LK_T            _flktab[NBKT];
#endif
static struct mag     *_ftab[NBKT];
#if (HACKS)
static long            _fcnt[NBKT];
#endif
static void          **_mdir;
static struct arn    **_atab;
static struct mconf    _conf;
#if (MTSAFE) && (PTHREAD)
static pthread_key_t   _akey;
```

```
static __thread long   _aid = -1;
#else
static long            _aid = 0;
#endif
#if (TUNEBUF)
static int64_t         _nbheap;
static int64_t         _nbmap;
#endif
static int             _mapfd = -1;

/* utility functions */

static __inline__ long
ceil2(size_t size)
{
    size--;
    size |= size >> 1;
    size |= size >> 2;
    size |= size >> 4;
    size |= size >> 8;
    size |= size >> 16;
#if (LONGSIZE == 8)
    size |= size >> 32;
#endif
    size++;

    return size;
}

static __inline__ long
bktid(size_t size)
{
    long tmp = ceil2(size);
    long bid;

#if (LONGSIZE == 4)
    tzero32(tmp, bid);
#elif (LONGSIZE == 8)
    tzero64(tmp, bid);
#endif

    return bid;
}

#if (MTSAFE)
static long
getaid(void)
{
    long        aid;
```

```
    mlk(&_conf.arnlk);
    aid = _conf.acur++;
    _conf.acur &= NARN - 1;
    pthread_setspecific(_akey, _atab[aid]);
    munlk(&_conf.arnlk);

    return aid;
}
#endif

static __inline__ void
zeroblk(void *ptr,
        size_t size)
{
    unsigned long *ulptr = ptr;
    unsigned long  zero = 0UL;
    long           small = (size < (LONGSIZE << 3));
    long           n = ((small)
                        ? (size >> LONGSIZELOG2)
                        : (size >> (LONGSIZELOG2 + 3)));
    long           nl = 8;

    if (small) {
        while (n--) {
            *ulptr++ = zero;
        }
    } else {
        while (n--) {
            ulptr[0] = zero;
            ulptr[1] = zero;
            ulptr[2] = zero;
            ulptr[3] = zero;
            ulptr[4] = zero;
            ulptr[5] = zero;
            ulptr[6] = zero;
            ulptr[7] = zero;
            ulptr += nl;
        }
    }

    return;
}

/* fork() management */

#if (MTSAFE)

static void
prefork(void)
{
```

```
    long        aid;
    long        bid;
    struct arn *arn;

    mlk(&_conf.initlk);
    mlk(&_conf.arnlk);
    mlk(&_conf.heaplk);
    aid = _conf.narn;
    while (aid--) {
        arn = _atab[aid];
        for (bid = 0 ; bid < NBKT ; bid++) {
            mlk(&arn->lktab[bid]);
        }
    }

    return;
}

static void
postfork(void)
{
    long        aid;
    long        bid;
    struct arn *arn;

    aid = _conf.narn;
    while (aid--) {
        arn = _atab[aid];
        for (bid = 0 ; bid < NBKT ; bid++) {
            munlk(&arn->lktab[bid]);
        }
    }
    munlk(&_conf.heaplk);
    munlk(&_conf.arnlk);
    munlk(&_conf.initlk);

    return;
}

static void
relarn(void *arg)
{
    struct arn *arn = arg;
#if (HACKS)
    long        n = 0;
#endif
    long        nref;
    long        bid;
    struct mag *mag;
    struct mag *head;
```

```
    nref = --arn->nref;
    if (!nref) {
        bid = NBKT;
        while (bid--) {
            mlk(&arn->lktab[bid]);
            head = arn->ftab[bid];
            if (head) {
#if (HACKS)
                n++;
#endif
                mag = head;
                while (mag->next) {
#if (HACKS)
                    n++;
#endif
                    mag = mag->next;
                }
                mlk(&_flktab[bid]);
                mag->next = _ftab[bid];
                _ftab[bid] = head;
#if (HACKS)
                _fcnt[bid] += n;
#endif
                munlk(&_flktab[bid]);
                arn->ftab[bid] = NULL;
            }
            munlk(&arn->lktab[bid]);
        }
    }

    return;
}

#endif /* MTSAFE */

/* statistics */

#if (STAT)
void
printstat(void)
{
    long l;

    for (l = 0 ; l < NBKT ; l++) {
        fprintf(stderr, "%ld\t%lu\t%lu\n", l, _nheapreq[l], _nmapreq[l]);
    }

    exit(0);
}
```

```
#elif (INTSTAT)
void
printintstat(void)
{
    long aid;
    long bkt;
    long nbhdr = 0;
    long nbstk = 0;
    long nbheap = 0;
    long nbmap = 0;

    for (aid = 0 ; aid < NARN ; aid++) {
        nbhdr += nhdrbytes[aid];
        nbstk += nstkbytes[aid];
        nbheap += nheapbytes[aid];
        nbmap += nmapbytes[aid];
        fprintf(stderr, "%lx: hdr: %ld\n", aid, nhdrbytes[aid] >> 10);
        fprintf(stderr, "%lx: stk: %ld\n", aid, nstkbytes[aid] >> 10);
        fprintf(stderr, "%lx: heap: %ld\n", aid, nheapbytes[aid] >> 10);
        fprintf(stderr, "%lx: map: %ld\n", aid, nmapbytes[aid] >> 10);
        for (bkt = 0 ; bkt < NBKT ; bkt++) {
            fprintf(stderr, "NALLOC[%lx][%lx]: %lld\n",
                    aid, bkt, nalloc[aid][bkt]);
        }
    }
    fprintf(stderr, "TOTAL: hdr: %ld, stk: %ld, heap: %ld, map: %ld\n",
            nbhdr, nbstk, nbheap, nbmap);
}
#endif

#if (X11VIS)
#include <X11/Xlibint.h>
#include <X11/Xatom.h>
#include <X11/Xutil.h>
#include <X11/Xmd.h>
#include <X11/Xlocale.h>
#include <X11/cursorfont.h>
#include <X11/keysym.h>
#include <X11/Xlib.h>

static LK_T  x11visinitlk;
#if 0
static LK_T  x11vislk;
#endif
long         x11visinit = 0;
Display     *x11visdisp = NULL;
Window       x11viswin = None;
Pixmap       x11vispmap = None;
GC           x11visinitgc = None;
GC           x11visfreedgc = None;
```

```
GC           x11visusedgc = None;
GC           x11visresgc = None;

#define x11vismarkfreed(ptr)                                                \
    do {                                                                    \
        if (x11visinit) {                                                   \
            int y = ((uintptr_t)(ptr) >> (BLKMINLOG2 + 10)) & 0x3ff;        \
            int x = ((uintptr_t)(ptr) >> BLKMINLOG2) & 0x3ff;               \
            XDrawPoint(x11visdisp, x11vispmap, x11visfreedgc, x, y);        \
        }                                                                   \
} while (0)
#define x11vismarkres(ptr)                                                  \
    do {                                                                    \
        if (x11visinit) {                                                   \
            int y = ((uintptr_t)(ptr) >> (BLKMINLOG2 + 10)) & 0x3ff;        \
            int x = ((uintptr_t)(ptr) >> BLKMINLOG2) & 0x3ff;               \
            XDrawPoint(x11visdisp, x11vispmap, x11visresgc, x, y);          \
        }                                                                   \
    } while (0)
#define x11vismarkused(ptr)                                                 \
    do {                                                                    \
        if (x11visinit) {                                                   \
            int y = ((uintptr_t)(ptr) >> (BLKMINLOG2 + 10)) & 0x3ff;        \
            int x = ((uintptr_t)(ptr) >> BLKMINLOG2) & 0x3ff;               \
            XDrawPoint(x11visdisp, x11vispmap, x11visusedgc, x, y);         \
        }                                                                   \
    } while (0)

void
initx11vis(void)
{
    XColor col;
    XGCValues gcval;

//    mlk(&x11vislk);
    mlk(&x11visinitlk);
    if (x11visinit) {
        munlk(&x11visinitlk);

        return;
    }
    XInitThreads();
    x11visdisp = XOpenDisplay(NULL);
    if (x11visdisp) {
        x11viswin = XCreateSimpleWindow(x11visdisp,
                                        DefaultRootWindow(x11visdisp),
                                        0, 0,
                                        1024, 1024, 0,
                                        BlackPixel(x11visdisp,
                                                   DefaultScreen(x11visdisp)),
```

```
                                    WhitePixel(x11visdisp,
                                               DefaultScreen(x11visdisp)));
    if (x11viswin) {
        XEvent ev;

        x11vispmap = XCreatePixmap(x11visdisp,
                                   x11viswin,
                                   1024, 1024,
                                   DefaultDepth(x11visdisp,
                                                DefaultScreen(x11visdisp)));

        gcval.foreground = WhitePixel(x11visdisp,
                                      DefaultScreen(x11visdisp));
        x11visinitgc = XCreateGC(x11visdisp,
                                 x11viswin,
                                 GCForeground,
                                 &gcval);

        XFillRectangle(x11visdisp,
                       x11vispmap,
                       x11visinitgc,
                       0, 0,
                       1024, 1024);

        col.red = 0x0000;
        col.green = 0x0000;
        col.blue = 0xffff;
        if (!XAllocColor(x11visdisp,
                         DefaultColormap(x11visdisp,
                                         DefaultScreen(x11visdisp)),
                         &col)) {

            return;
        }
        gcval.foreground = col.pixel;
        x11visfreedgc = XCreateGC(x11visdisp,
                                  x11viswin,
                                  GCForeground,
                                  &gcval);

        col.red = 0xffff;
        col.green = 0x0000;
        col.blue = 0x0000;
        if (!XAllocColor(x11visdisp,
                         DefaultColormap(x11visdisp,
                                         DefaultScreen(x11visdisp)),
                         &col)) {

            return;
        }
```

```
            gcval.foreground = col.pixel;
            x11visusedgc = XCreateGC(x11visdisp,
                                     x11viswin,
                                     GCForeground,
                                     &gcval);

            col.red = 0x0000;
            col.green = 0xffff;
            col.blue = 0x0000;
            if (!XAllocColor(x11visdisp,
                             DefaultColormap(x11visdisp,
                                             DefaultScreen(x11visdisp)),
                             &col)) {

                return;
            }
            gcval.foreground = col.pixel;
            x11visresgc = XCreateGC(x11visdisp,
                                    x11viswin,
                                    GCForeground,
                                    &gcval);

            XSelectInput(x11visdisp, x11viswin, ExposureMask);
            XMapRaised(x11visdisp, x11viswin);
            do {
                XNextEvent(x11visdisp, &ev);
            } while (ev.type != Expose);
            XSelectInput(x11visdisp, x11viswin, NoEventMask);
        }
    }
    x11visinit = 1;
    munlk(&x11visinitlk);
//    munlk(&x11vislk);
}
#endif

static void
initmall(void)
{
    long        bid = NBKT;
    long        aid = NARN;
    long        ofs;
    uint8_t    *ptr;

    mlk(&_conf.initlk);
    if (_conf.flags & CONF_INIT) {
        munlk(&_conf.initlk);

        return;
    }
```

```
#if (STAT)
    atexit(printstat);
#elif (INTSTAT)
    atexit(printintstat);
#endif
#if (_MMAP_DEV_ZERO)
    _mapfd = open("/dev/zero", O_RDWR);
#endif
#if (MTSAFE)
    mlk(&_conf.arnlk);
    _atab = mapanon(_mapfd, NARN * sizeof(struct arn **));
    ptr = mapanon(_mapfd, NARN * nbarn());
    aid = NARN;
    while (aid--) {
        _atab[aid] = (struct arn *)ptr;
        ptr += nbarn();
    }
    aid = NARN;
    while (aid--) {
        for (bid = 0 ; bid < NBKT ; bid++) {
#if (ZEROMTX)
            mtxinit(&_atab[aid]->lktab[bid]);
#elif (PTHREAD) && !SPINLK
            pthread_mutex_init(&_atab[aid]->lktab[bid], NULL);
#endif
        }
        _atab[aid]->hcur = NBUFHDR;
    }
    _conf.narn = NARN;
    pthread_key_create(&_akey, relarn);
    munlk(&_conf.arnlk);
#endif
#if (PTHREAD)
    pthread_atfork(prefork, postfork, postfork);
#endif
#if (PTHREAD)
    while (bid--) {
#if (ZEROMTX)
        mtxinit(&_flktab[bid]);
#elif (PTHREAD) && !SPINLK
        pthread_mutex_init(&_flktab[bid], NULL);
#endif
    }
#endif
    mlk(&_conf.heaplk);
    ofs = NBSLAB - ((long)growheap(0) & (NBSLAB - 1));
    if (ofs != NBSLAB) {
        growheap(ofs);
    }
    munlk(&_conf.heaplk);
```

```
#if (PTRBITS <= 32)
    _mdir = mapanon(_mapfd, NSLAB * sizeof(void *));
#else
    _mdir = mapanon(_mapfd, NL1KEY * sizeof(void *));
#endif
#if (TUNEBUF)
    for (bid = 0 ; bid < NBKT ; bid++) {
        _nslabtab[bid] = nmagslablog2init(bid);
    }
#endif
    _conf.flags |= CONF_INIT;
    munlk(&_conf.initlk);
#if (X11VIS)
    initx11vis();
#endif

    return;
}

#if (MTSAFE)
#if (PTRBITS > 32)
#define l1ndx(ptr) getbits((uintptr_t)ptr, L1NDX, NL1BIT)
#define l2ndx(ptr) getbits((uintptr_t)ptr, L2NDX, NL2BIT)
#define l3ndx(ptr) getbits((uintptr_t)ptr, L3NDX, NL3BIT)
#if (PTRBITS > 48)
static struct mag *
findmag(void *ptr)
{
    uintptr_t  l1 = l1ndx(ptr);
    uintptr_t  l2 = l2ndx(ptr);
    uintptr_t  l3 = l3ndx(ptr);
    void       *ptr1;
    void       *ptr2;
    struct mag *mag = NULL;

    ptr1 = _mdir[l1];
    if (ptr1) {
        ptr2 = ((void **)ptr1)[l2];
        if (ptr2) {
            mag = ((struct mag **)ptr2)[l3];
        }
    }

    return mag;
}

static void
addblk(void *ptr,
       struct mag *mag)
{
```

```
    uintptr_t    l1 = l1ndx(ptr);
    uintptr_t    l2 = l2ndx(ptr);
    uintptr_t    l3 = l3ndx(ptr);
    void        *ptr1;
    void        *ptr2;
    void       **pptr;
    struct mag **item;

    ptr1 = _mdir[l1];
    if (!ptr1) {
        _mdir[l1] = ptr1 = mapanon(_mapfd, NL2KEY * sizeof(void *));
        if (ptr1 == MAP_FAILED) {
#ifdef ENOMEM
            errno = ENOMEM;
#endif

            exit(1);
        }
    }
    pptr = ptr1;
    ptr2 = pptr[l2];
    if (!ptr2) {
        pptr[l2] = ptr2 = mapanon(_mapfd, NL3KEY * sizeof(struct mag *));
        if (ptr2 == MAP_FAILED) {
#ifdef ENOMEM
            errno = ENOMEM;
#endif

            exit(1);
        }
    }
    item = &((struct mag **)ptr2)[l3];
    *item = mag;

    return;
}
#else
static struct mag *
findmag(void *ptr)
{
    uintptr_t   l1 = l1ndx(ptr);
    uintptr_t   l2 = l2ndx(ptr);
    void       *ptr1;
    struct mag *mag = NULL;

    ptr1 = _mdir[l1];
    if (ptr1) {
        mag = ((struct mag **)ptr1)[l2];
    }
```

```
    return mag;
}

static void
addblk(void *ptr,
       struct mag *mag)
{
    uintptr_t    l1 = l1ndx(ptr);
    uintptr_t    l2 = l2ndx(ptr);
    void        *ptr1;
    struct mag **item;

    ptr1 = _mdir[l1];
    if (!ptr1) {
        _mdir[l1] = ptr1 = mapanon(_mapfd, NL2KEY * sizeof(struct mag *));
        if (ptr1 == MAP_FAILED) {
#ifdef ENOMEM
            errno = ENOMEM;
#endif

            exit(1);
        }
    }
    item = &((struct mag **)ptr1)[l2];
    *item = mag;

    return;
}
#endif
#else
#define findmag(ptr)     (_mdir[slabid(ptr)])
#define addblk(ptr, mag) (_mdir[slabid(ptr)] = (mag))
#endif
#endif

static struct mag *
gethdr(long aid)
{
    struct arn  *arn;
    long         cur;
    struct mag **hbuf;
    struct mag  *mag = NULL;
    uint8_t     *ptr;

    arn = _atab[aid];
    hbuf = arn->htab;
    if (!arn->nhdr) {
        hbuf = mapanon(_mapfd, roundup2(NBUFHDR * sizeof(void *), PAGESIZE));
        if (hbuf != MAP_FAILED) {
#if (INTSTAT)
```

```
            nhdrbytes[aid] += roundup2(NBUFHDR * sizeof(void *), PAGESIZE);
#endif
            arn->htab = hbuf;
            arn->hcur = NBUFHDR;
            arn->nhdr = NBUFHDR;
        }
    }
    cur = arn->hcur;
    if (gte2(cur, NBUFHDR)) {
        mag = mapanon(_mapfd, roundup2(NBUFHDR * nbhdr(), PAGESIZE));
        if (mag == MAP_FAILED) {
#ifdef ENOMEM
            errno = ENOMEM;
#endif

            return NULL;
        } else {
#if (VALGRIND)
            if (RUNNING_ON_VALGRIND) {
                VALGRIND_MALLOCLIKE_BLOCK(mag, PAGESIZE, 0, 0);
            }
#endif
        }
        ptr = (uint8_t *)mag;
        while (cur) {
            mag = (struct mag *)ptr;
            *hbuf++ = mag;
            mag->bptr = mag->stk;
            cur--;
            ptr += nbhdr();
        }
    }
    hbuf = arn->htab;
#if (SYSDBG)
    _nhbuf++;
#endif
    mag = hbuf[cur++];
    arn->hcur = cur;

    return mag;
}

#if (TUNEBUF)
static void
tunebuf(long val)
{
    static long tunesz = 0;
    long        nb = _nbheap + _nbmap;

    return;
```

```
    if (!tunesz) {
        tunesz = val;
    }
    if (val == 64 && nb >= 64 * 1024) {
        nmagslablog2up(nmagslablog2m64, val, nb);
    } else if (val == 128 && nb >= 128 * 1024) {
        nmagslablog2up(nmagslablog2m128, val, nb);
    } else if (val == 256 && nb >= 256 * 1024) {
        nmagslablog2up(nmagslablog2m256, val, nb);
    } else if (val == 512 && nb >= 512 * 1024) {
        nmagslablog2up(nmagslablog2m512, val, nb);
    }

    return;
}
#endif

static void *
getslab(long aid,
        long bid)
{
    uint8_t     *ptr = NULL;
    long         nb = nbmag(bid);
#if (TUNEBUF)
    unsigned long tmp;
    static long tunesz = 0;
#endif

    if (!ismapbkt(bid)) {
        mlk(&_conf.heaplk);
        ptr = growheap(nb);
        munlk(&_conf.heaplk);
        if (ptr != SBRK_FAILED) {
#if (INTSTAT)
            nheapbytes[aid] += nb;
#endif
#if (TUNEBUF)
            _nbheap += nb;
#if (STAT)
            _nheapreq[bid]++;
#endif
#endif
        }
    } else {
        ptr = mapanon(_mapfd, nbmap(bid));
        if (ptr != MAP_FAILED) {
#if (INTSTAT)
            nmapbytes[aid] += nbmap(bid);
#endif
```

```
#if (STAT)
            _nmapreq[bid]++;
#endif
        }
    }
#if (TUNEBUF)
    if (ptr != MAP_FAILED && ptr != SBRK_FAILED) {
        tmp = _nbmap + _nbheap;
        if (!tunesz) {
            tunesz = 64;
        }
        if ((tmp >> 10) >= tunesz) {
            tunebuf(tunesz);
        }
    }
#endif

    return ptr;
}

static void
freemap(struct mag *mag)
{
    struct arn  *arn;
    long         cur;
    long         aid = mag->aid;
    long         bid = mag->bid;
    long         bsz = blksz(bid);
    long         max = mag->max;
    struct mag **hbuf;

    arn = _atab[aid];
    mlk(&arn->lktab[bid]);
    cur = arn->hcur;
    hbuf = arn->htab;
//#if (HACKS)
//    if (!cur || _fcnt[bid] < 4) {
//#else
    if (!cur) {
//#endif
        mag->prev = NULL;
        mlk(&_flktab[bid]);
        mag->next = _ftab[bid];
        _ftab[bid] = mag;
#if (HACKS)
        _fcnt[bid]++;
#endif
        munlk(&_flktab[bid]);
    } else {
        if (!unmapanon(clrptr(mag->adr), max * bsz)) {
```

```
#if (VALGRIND)
            if (RUNNING_ON_VALGRIND) {
                VALGRIND_FREELIKE_BLOCK(clrptr(mag->adr), 0);
            }
#endif
#if (INTSTAT)
            nmapbytes[aid] -= max * bsz;
#endif
#if (TUNEBUF)
            _nbmap -= max * bsz;
#endif
            if (gt2(max, 1)) {
                if (!istk(bid)) {
#if (INTSTAT)
                    nstkbytes[aid] -= (mag->max << 1) << sizeof(void *);
#endif
                    unmapstk(mag);
                    mag->bptr = NULL;
#if (VALGRIND)
                    if (RUNNING_ON_VALGRIND) {
                        VALGRIND_FREELIKE_BLOCK(mag, 0);
                    }
#endif
                }
            }
            mag->adr = NULL;
            hbuf[--cur] = mag;
            arn->hcur = cur;
        }
    }
    munlk(&arn->lktab[bid]);

    return;
}

#define blkalnsz(sz, aln)                                                   \
    (((aln) <= MINSZ)                                                       \
     ? max(sz, aln)                                                         \
     : (sz) + (aln))
static void *
getmem(size_t size,
       size_t align,
       long zero)
{
    struct arn  *arn;
    long         aid;
    long         sz = blkalnsz(max(size, MINSZ), align);
    long         bid = bktid(sz);
    uint8_t     *retptr = NULL;
    long         bsz = blksz(bid);
```

```
    uint8_t     *ptr = NULL;
    long         max = nblk(bid);
    struct mag  *mag = NULL;
    void       **stk;
    long         l;
    long         n;
    long         get = 0;

    if (!(_conf.flags & CONF_INIT)) {
        initmall();
    }

    aid = thrid();
    arn = _atab[aid];
    mlk(&arn->lktab[bid]);
    mag = arn->btab[bid];
    if (!mag) {
        mag = arn->ftab[bid];
    }
    if (!mag) {
        mlk(&_flktab[bid]);
        mag = _ftab[bid];
        if (mag) {
            mag->aid = aid;
            _ftab[bid] = mag->next;
            mag->next = NULL;
#if (HACKS)
            _fcnt[bid]--;
#endif
        }
        munlk(&_flktab[bid]);
        if (mag) {
            if (gt2(max, 1)) {
                mag->next = arn->btab[bid];
                if (mag->next) {
                    mag->next->prev = mag;
                }
                arn->btab[bid] = mag;
            }
        }
    } else if (mag->cur == mag->max - 1) {
        if (mag->next) {
            mag->next->prev = NULL;
        }
        arn->btab[bid] = mag->next;
        mag->next = NULL;
    }
    if (!mag) {
        get = 1;
        if (!ismapbkt(bid)) {
```

```
            ptr = getslab(aid, bid);
            if (ptr == (void *)-1L) {
                ptr = NULL;
            }
        } else {
            ptr = mapanon(_mapfd, nbmap(bid));
            if (ptr == MAP_FAILED) {
                ptr = NULL;
            }
#if (INTSTAT)
            else {
                nmapbytes[aid] += nbmap(bid);
            }
#endif
        }
        mag = gethdr(aid);
        if (mag) {
            mag->aid = aid;
            mag->cur = 0;
            mag->max = max;
            mag->bid = bid;
            mag->adr = ptr;
            if (ptr) {
                if (gt2(max, 1)) {
                    if (istk(bid)) {
                        stk = (void **)mag->stk;
                    } else {
                        stk = mapstk(max);
                    }
                    mag->bptr = stk;
                    if (stk != MAP_FAILED) {
#if (INTSTAT)
                        nstkbytes[aid] += (max << 1) << sizeof(void *);
#endif
#if (VALGRIND)
                        if (RUNNING_ON_VALGRIND) {
                            VALGRIND_MALLOCLIKE_BLOCK(stk, max * sizeof(void *), 0, 0
                        }
#endif
                        n = max << nmagslablog2(bid);
                        for (l = 0 ; l < n ; l++) {
                            stk[l] = ptr;
                            ptr += bsz;
                        }
                        mag->prev = NULL;
                        if (ismapbkt(bid)) {
                            mlk(&_flktab[bid]);
                            mag->next = _ftab[bid];
                            _ftab[bid] = mag;
#if (HACKS)
```

```
                            _fcnt[bid]++;
#endif
                        } else {
                            mag->next = arn->btab[bid];
                            if (mag->next) {
                                mag->next->prev = mag;
                            }
                            arn->btab[bid] = mag;
                        }
                    }
                }
            }
        }
    }
    if (mag) {
        ptr = getblk(mag);
        retptr = clrptr(ptr);
#if (VALGRIND)
        if (RUNNING_ON_VALGRIND) {
            if (retptr) {
                VALGRIND_MALLOCLIKE_BLOCK(retptr, bsz, 0, 0);
            }
        }
#endif
        if ((zero) && chkflg(ptr, BDIRTY)) {
            zeroblk(retptr, bsz);
        }
        ptr = retptr;
#if (RZSZ)
        markred(ptr);
        markred(ptr + RZSZ + size);
#endif
        if (retptr) {
#if (RZSZ)
            retptr = ptr + RZSZ;
#endif
            if (align) {
                if ((uintptr_t)(retptr) & (align - 1)) {
                    retptr = (uint8_t *)roundup2((uintptr_t)ptr, align);
                }
                ptr = setflg(retptr, BALIGN);
            }
            putptr(mag, retptr, ptr);
            addblk(retptr, mag);
        }
    }
    if ((get) && ismapbkt(bid)) {
        munlk(&_flktab[bid]);
    }
    munlk(&arn->lktab[bid]);
```

```
#if (X11VIS)
//    mlk(&x11vislk);
    if (x11visinit) {
//        ptr = clrptr(ptr);
        ptr = retptr;
        if (ptr) {
            long     l = blksz(bid) >> BLKMINLOG2;
            uint8_t *vptr = ptr;

            while (l--) {
                x11vismarkres(vptr);
                vptr += MINSZ;
            }
        }
        if (retptr) {
            long     l = sz >> BLKMINLOG2;
            uint8_t *vptr = retptr;

            while (l--) {
                x11vismarkused(ptr);
                vptr += MINSZ;
            }
        }
        XSetWindowBackgroundPixmap(x11visdisp,
                                   x11viswin,
                                   x11vispmap);
        XClearWindow(x11visdisp,
                     x11viswin);
        XFlush(x11visdisp);
    }
//    munlk(&x11vislk);
#endif
#ifdef ENOMEM
    if (!retptr) {
        errno = ENOMEM;
        fprintf(stderr, "%lx failed to allocate %ld bytes\n", aid, 1UL << bid);

        abort();
    }
#if (INTSTAT)
    else {
        nalloc[aid][bid]++;
    }
#endif
#endif

    return retptr;
}

static void
```

```
putmem(void *ptr)
{
#if (RZSZ)
    uint8_t    *u8p = ptr;
#endif
    struct arn *arn;
    void       *mptr;
    struct mag *mag = (ptr) ? findmag(ptr) : NULL;
    long        aid = -1;
    long        tid = thrid();
    long        bid = -1;
    long        max;
    long        glob = 0;
    long        freed = 0;

    if (mag) {
#if (VALGRIND)
        if (RUNNING_ON_VALGRIND) {
            VALGRIND_FREELIKE_BLOCK(ptr, 0);
        }
#endif
        aid = mag->aid;
        if (aid < 0) {
            glob++;
            mag->aid = aid = tid;
        }
        bid = mag->bid;
        max = mag->max;
        arn = _atab[aid];
        mlk(&arn->lktab[bid]);
        if (gt2(max, 1) && magempty(mag)) {
            mag->next = arn->btab[bid];
            if (mag->next) {
                mag->next->prev = mag;
            }
            arn->btab[bid] = mag;
        }
        mptr = getptr(mag, ptr);
#if (RZSZ)
        if (!chkflg(mptr, BALIGN)) {
            u8p = mptr - RZSZ;
            if (chkred(u8p) || chkred(u8p + blksz(bid) - RZSZ)) {
                fprintf(stderr, "red-zone violation\n");
            }
            ptr = clrptr(mptr);
        }
#endif
        if (mptr) {
            putptr(mag, ptr, NULL);
            mptr = setflg(mptr, BDIRTY);
```

```
            putblk(mag, mptr);
            if (magfull(mag)) {
                if (gt2(max, 1)) {
                    if (mag->prev) {
                        mag->prev->next = mag->next;
                    } else {
                        arn->btab[bid] = mag->next;
                    }
                    if (mag->next) {
                        mag->next->prev = mag->prev;
                    }
                }
                if (!isbufbkt(bid) && ismapbkt(bid)) {
                    freed = 1;
                } else {
                    mag->prev = mag->next = NULL;
                    mlk(&_flktab[bid]);
                    mag->next = _ftab[bid];
                    _ftab[bid] = mag;
#if (HACKS)
                    _fcnt[bid]++;
#endif
                    munlk(&_flktab[bid]);
                }
            }
        }
        munlk(&arn->lktab[bid]);
        if (freed) {
            freemap(mag);
        }
#if (X11VIS)
//    mlk(&x11vislk);
        if (x11visinit) {
            ptr = mptr;
            if (ptr) {
                if (freed) {
                    long     l = nbmap(bid) >> BLKMINLOG2;
                    uint8_t *vptr = ptr;

                    while (l--) {
                        x11vismarkfreed(vptr);
                        vptr += MINSZ;
                    }
                } else {
                    long     l = blksz(bid) >> BLKMINLOG2;
                    uint8_t *vptr = ptr;

                    while (l--) {
                        x11vismarkfreed(vptr);
                        vptr += MINSZ;
```

```
                    }
                }
            }
            XSetWindowBackgroundPixmap(x11visdisp,
                                       x11viswin,
                                           x11vispmap);
            XClearWindow(x11visdisp,
                         x11viswin);
            XFlush(x11visdisp);
        }
//    munlk(&x11vislk);
#endif
    }

    return;
}

/* STD: ISO/POSIX */

void *
malloc(size_t size)
{
    void *ptr = getmem(size, 0, 0);

    return ptr;
}

void *
calloc(size_t n, size_t size)
{
    size_t  sz = n * (size + (RZSZ << 1));
    void   *ptr = getmem(sz, 0, 1);

    return ptr;
}

void *
_realloc(void *ptr,
         size_t size,
         long rel)
{
    void       *retptr = ptr;
    long        sz = blkalnsz(max(size + (RZSZ << 1), MINSZ), 0);
    struct mag *mag = (ptr) ? findmag(ptr) : NULL;
    long        bid = bktid(sz);
    uintptr_t   bsz = (mag) ? blksz(mag->bid) : 0;

    if (!ptr) {
        retptr = getmem(size, 0, 0);
    } else if ((mag) && mag->bid != bid) {
```

```
        retptr = getmem(size, 0, 0);
        if (retptr) {
            memcpy(retptr, ptr, min(sz, bsz));
            putmem(ptr);
            ptr = NULL;
        }
    }
    if ((rel) && (ptr)) {
        putmem(ptr);
    }

    return retptr;
}

void *
realloc(void *ptr,
        size_t size)
{
    void *retptr = _realloc(ptr, size, 0);

    return retptr;
}

void
free(void *ptr)
{
    if (ptr) {
        putmem(ptr);
    }

    return;
}

#if (_ISOC11_SOURCE)
void *
aligned_alloc(size_t align,
              size_t size)
{
    void *ptr = NULL;
    if (!powerof2(align) || (size % align)) {
        errno = EINVAL;
    } else {
        ptr = getmem(size, align, 0);
    }

    return ptr;
}

#endif
```

```
#if (_POSIX_C_SOURCE >= 200112L || _XOPEN_SOURCE >= 600)
int
posix_memalign(void **ret,
               size_t align,
               size_t size)
{
    void *ptr = getmem(size, align, 0);
    int   retval = -1;

    if (!powerof2(align) || (size % sizeof(void *))) {
        errno = EINVAL;
    } else {
        ptr = getmem(size, align, 0);
        if (ptr) {
            retval ^= retval;
        }
    }

    *ret = ptr;

    return retval;
}
#endif

/* STD: UNIX */

#if ((_BSD_SOURCE)                                                      \
    || (_XOPEN_SOURCE >= 500 || ((_XOPEN_SOURCE) && (_XOPEN_SOURCE_EXTENDED))) \
    && !(_POSIX_C_SOURCE >= 200112L || _XOPEN_SOURCE >= 600))
void *
valloc(size_t size)
{
    void *ptr = getmem(size, PAGESIZE, 0);

    return ptr;
}
#endif

void *
memalign(size_t align,
         size_t size)
{
    void *ptr = NULL;

    if (!powerof2(align)) {
        errno = EINVAL;
    } else {
        ptr = getmem(size, align, 0);
    }
```

```
    return ptr;
}

#if (_BSD_SOURCE)
void *
reallocf(void *ptr,
         size_t size)
{
    void *retptr = _realloc(ptr, size, 1);

    return retptr;
}
#endif

#if (_GNU_SOURCE)
void *
pvalloc(size_t size)
{
    size_t  sz = roundup2(size, PAGESIZE);
    void   *ptr = getmem(sz, PAGESIZE, 0);

    return ptr;
}
#endif

void
cfree(void *ptr)
{
    if (ptr) {
        free(ptr);
    }

    return;
}

size_t
malloc_usable_size(void *ptr)
{
    struct mag *mag = findmag(ptr);
    size_t      sz = usrsz(mag->bid);

    return sz;
}

size_t
malloc_good_size(size_t size)
{
    size_t rzsz = RZSZ;
    size_t sz = usrsz(bktid(size)) - (rzsz << 1);
```

```
    return sz;
}

size_t
malloc_size(void *ptr)
{
    struct mag *mag = findmag(ptr);
    size_t      sz = (mag) ? blksz(mag->bid) : 0;

    return sz;
}
```

Appendix A

Cheat Sheets

A.1 C Operator Precedence and Associativity

Precedence

The table below lists operators in descending order of evaluation (precedence).

Operators	Associativity
() [] -> .	left to right
! ~ + + - - + - * (cast) sizeof	right to left
* / %	left to right
+ -	left to right
<< >>	left to right
< <= > >=	left to right
== !=	left to right
&	left to right
^	left to right
|	left to right
&&	left to right
?:	right to left
= += -= *= /= %= &= ^= |= <<= >>=	right to left
,	left to right

Notes

- Unary (single operand) +, -, and * have higher precedence than the binary ones

TODO: (ARM?) assembly, Dijkstra's Shunting yard, cdecl

Appendix B

A Bag of Tricks

trix.h

```
#ifndef __ZERO_TRIX_H__
#define __ZERO_TRIX_H__

/*
 * this file contains tricks I've gathered together from sources such as MIT
 * HAKMEM and the book Hacker's Delight
 */

#define ZEROABS 1

#include <stdint.h>
#include <limits.h>
#include <zero/param.h>

#if (LONGSIZE == 4)
#define tzerol(u, r) tzero32(u, r)
#define lzerol(u, r) lzero32(u, r)
#elif (LONGSIZE == 8)
#define tzerol(u, r) tzero64(u, r)
#define lzerol(u, r) lzero64(u, r)
#endif

/* get the lowest 1-bit in a */
#define lo1bit(a)           ((a) & -(a))
/* get n lowest and highest bits of i */
#define lobits(i, n)        ((i) & ((1U << (n)) - 0x01))
#define hibits(i, n)        ((i) & ~((1U << (sizeof(i) * CHAR_BIT - (n))) - 0x01))
/* get n bits starting from index j */
#define getbits(i, j, n)    (lobits((i) >> (j), (n)))
/* set n bits starting from index j to value b */
#define setbits(i, j, n, b) ((i) |= (((b) << (j)) & ~(((1U << (n)) << (j)) - 0x01)))
#define bitset(p, b)        (((uint8_t *)(p))[(b) >> 3] & (1U << ((b) & 0x07)))
```

```
/* set bit # b in *p */
#define setbit(p, b)        (((uint8_t *)(p))[(b) >> 3] |= (1U << ((b) & 0x07)))
/* clear bit # b in *p */
#define clrbit(p, b)        (((uint8_t *)(p))[(b) >> 3] &= ~(1U << ((b) & 0x07)))
/* m - mask of bits to be taken from b. */
#define mergebits(a, b, m)  ((a) ^ (((a) ^ (b)) & (m)))
/* m - mask of bits to be copied from a. 1 -> copy, 0 -> leave alone. */
#define copybits(a, b, m) (((a) | (m)) | ((b) & ~(m)))

/* compute minimum and maximum of a and b without branching */
#define min(a, b)                                                       \
    ((b) + (((a) - (b)) & -((a) < (b))))
#define max(a, b)                                                       \
    ((a) - (((a) - (b)) & -((a) < (b))))
/* compare with power-of-two p2 */
#define gt2(u, p2)  /* true if u > p2 */                                \
    ((u) & ~(p2))
#define gte2(u, p2) /* true if u >= p2 */                               \
    ((u) & -(p2))
/* swap a and b without a temporary variable */
#define swap(a, b)     ((a) ^= (b), (b) ^= (a), (a) ^= (b))
/* compute absolute value of integer without branching; PATENTED in USA :( */
#if (ZEROABS)
#define zeroabs(a)                                                      \
    (((a) ^ (((a) >> (CHAR_BIT * sizeof(a) - 1))))                      \
     - ((a) >> (CHAR_BIT * sizeof(a) - 1)))
#define abs(a)         zeroabs(a)
#define labs(a)        zeroabs(a)
#define llabs(a)       zeroabs(a)
#endif

/* true if x is a power of two */
#define powerof2(x)     (!((x) & ((x) - 1)))
/* align a to boundary of (the power of two) b2. */
//#define align(a, b2)   ((a) & ~((b2) - 1))
//#define align(a, b2)    ((a) & -(b2))
#define mod2(a, b2)     ((a) & ((b2) - 1))

/* round a up to the next multiple of (the power of two) b2. */
//#define roundup2a(a, b2) (((a) + ((b2) - 0x01)) & ~((b2) + 0x01))
#define roundup2(a, b2) (((a) + ((b2) - 0x01)) & -(b2))

/* round down to the previous multiple of (the power of two) b2 */
#define rounddown2(a, b2) ((a) & ~((b2) - 0x01))

/* compute the average of a and b without division */
#define uavg(a, b)      (((a) & (b)) + (((a) ^ (b)) >> 1))

#define divceil(a, b)   (((a) + (b) - 1) / (b))
#define divround(a, b)  (((a) + ((b) / 2)) / (b))
```

```
#define haszero_2(a)    (~(a))
#define haszero_32(a)   (((a) - 0x01010101) & ~(a) & 0x80808080)

#define onebits_32(u32, r)                                              \
    ((r) = (u32),                                                       \
     (r) -= ((r) >> 1) & 0x55555555,                                    \
     (r) = (((r) >> 2) & 0x33333333) + ((r) & 0x33333333),              \
     (r) = ((((r) >> 4) + (r)) & 0x0f0f0f0f),                           \
     (r) += ((r) >> 8),                                                 \
     (r) += ((r) >> 16),                                                \
     (r) &= 0x3f)
#define onebits_32b(u32, r)                                             \
    ((r) = (u32),                                                       \
     (r) -= ((r) >> 1) & 0x55555555,                                    \
     (r) = (((r) >> 2) & 0x33333333) + ((r) & 0x33333333),              \
     (r) = (((((r) >> 4) + (r)) & 0x0f0f0f0f) * 0x01010101) >> 24)

#define bytepar(b, r)                                                   \
    do {                                                                \
        unsigned long _tmp1;                                            \
                                                                        \
        _tmp1 = (b);                                                    \
        _tmp1 ^= (b) >> 4;                                              \
        (r) = (0x6996 >> (_tmp1 & 0x0f)) & 0x01;                        \
    } while (0)
#define bytepar2(b, r)                                                  \
    do {                                                                \
        unsigned long _tmp1;                                            \
        unsigned long _tmp2;                                            \
                                                                        \
        _tmp1 = _tmp2 = (b);                                            \
        _tmp2 >>= 4;                                                    \
        _tmp1 ^= _tmp2;                                                 \
        _tmp2 = 0x6996;                                                 \
        (r) = (_tmp2 >> (_tmp1 & 0x0f)) & 0x01;                         \
    } while (0)
#define bytepar3(b) ((0x6996 >> (((b) ^ ((b) >> 4)) & 0x0f)) & 0x01)

/* count number of trailing zero-bits in u32 */
#define tzero32(u32, r)                                                 \
    do {                                                                \
        uint32_t __tmp;                                                 \
        uint32_t __mask;                                                \
                                                                        \
        (r) = 0;                                                        \
        __tmp = (u32);                                                  \
        __mask = 0x01;                                                  \
        if (!(__tmp & __mask)) {                                        \
            __mask = 0xffff;                                            \
```

```
            if (!(__tmp & __mask)) {                                    \
                __tmp >>= 16;                                           \
                (r) += 16;                                              \
            }                                                           \
            __mask >>= 8;                                               \
            if (!(__tmp & __mask)) {                                    \
                __tmp >>= 8;                                            \
                (r) += 8;                                               \
            }                                                           \
            __mask >>= 4;                                               \
            if (!(__tmp & __mask)) {                                    \
                __tmp >>= 4;                                            \
                (r) += 4;                                               \
            }                                                           \
            __mask >>= 2;                                               \
            if (!(__tmp & __mask)) {                                    \
                __tmp >>= 2;                                            \
                (r) += 2;                                               \
            }                                                           \
            __mask >>= 1;                                               \
            if (!(__tmp & __mask)) {                                    \
                (r) += 1;                                               \
            }                                                           \
        }                                                               \
    } while (0)

/*
 * count number of leading zero-bits in u32
 */
#if 0
#define lzero32(u32, r)                                                 \
    ((u32) |= ((u32) >> 1),                                             \
     (u32) |= ((u32) >> 2),                                             \
     (u32) |= ((u32) >> 4),                                             \
     (u32) |= ((u32) >> 8),                                             \
     (u32) |= ((u32) >> 16),                                            \
     CHAR_BIT * sizeof(u32) - onebits_32(u32, r))
#endif
#define lzero32(u32, r)                                                 \
    do {                                                                \
        uint32_t __tmp;                                                 \
        uint32_t __mask;                                                \
\
        (r) = 0;   \
        __tmp = (u32);                                                  \
        __mask = 0x01;                                                  \
        __mask <<= CHAR_BIT * sizeof(uint32_t) - 1;                     \
        if (!(__tmp & __mask)) {                                        \
            __mask = 0xffffffff;                                        \
            __mask <<= 16;                                              \
```

```
            if (!(__tmp & __mask)) {                                    \
                __tmp <<= 16;                                           \
                (r) += 16;                                              \
            }                                                           \
            __mask <<= 8;                                               \
            if (!(__tmp & __mask)) {                                    \
                __tmp <<= 8;                                            \
                (r) += 8;                                               \
            }                                                           \
            __mask <<= 4;                                               \
            if (!(__tmp & __mask)) {                                    \
                __tmp <<= 4;                                            \
                (r) += 4;                                               \
            }                                                           \
            __mask <<= 2;                                               \
            if (!(__tmp & __mask)) {                                    \
                __tmp <<= 2;                                            \
                (r) += 2;                                               \
            }                                                           \
            __mask <<= 1;                                               \
            if (!(__tmp & __mask)) {                                    \
                (r)++;                                                  \
            }                                                           \
        }                                                               \
    } while (0)

/* 64-bit versions */

#define tzero64(u64, r)                                                 \
    do {                                                                \
        uint64_t __tmp;                                                 \
        uint64_t __mask;                                                \
                                                                        \
        (r) = 0;                                                        \
        __tmp = (u64);                                                  \
        __mask = 0x01;                                                  \
        if (!(__tmp & __mask)) {                                        \
            __mask = 0xffffffff;                                        \
            if (!(__tmp & __mask)) {                                    \
                __tmp >>= 32;                                           \
                (r) += 32;                                              \
            }                                                           \
            __mask >>= 16;                                              \
            if (!(__tmp & __mask)) {                                    \
                __tmp >>= 16;                                           \
                (r) += 16;                                              \
            }                                                           \
            __mask >>= 8;                                               \
            if (!(__tmp & __mask)) {                                    \
                __tmp >>= 8;                                            \
```

```
            (r) += 8;                                                   \
        }                                                               \
        __mask >>= 4;                                                   \
        if (!(__tmp & __mask)) {                                        \
            __tmp >>= 4;                                                \
            (r) += 4;                                                   \
        }                                                               \
        __mask >>= 2;                                                   \
        if (!(__tmp & __mask)) {                                        \
            __tmp >>= 2;                                                \
            (r) += 2;                                                   \
        }                                                               \
        __mask >>= 1;                                                   \
        if (!(__tmp & __mask)) {                                        \
            (r) += 1;                                                   \
        }                                                               \
    }                                                                   \
} while (0)

#define lzero64(u64, r)                                                 \
    do {                                                                \
        uint64_t __tmp;                                                 \
        uint64_t __mask;                                                \
\
        (r) = 0;   \
        __tmp = (u64);                                                  \
        __mask = 0x01;                                                  \
        __mask <<= CHAR_BIT * sizeof(uint64_t) - 1;                     \
        if (!(__tmp & __mask)) {                                        \
            __mask = 0xffffffff;                                        \
            __mask <<= 32;                                              \
            if (!(__tmp & __mask)) {                                    \
                __tmp <<= 32;                                           \
                (r) += 32;                                              \
            }                                                           \
            __mask <<= 16;                                              \
            if (!(__tmp & __mask)) {                                    \
                __tmp <<= 16;                                           \
                (r) += 16;                                              \
            }                                                           \
            __mask <<= 8;                                               \
            if (!(__tmp & __mask)) {                                    \
                __tmp <<= 8;                                            \
                (r) += 8;                                               \
            }                                                           \
            __mask <<= 4;                                               \
            if (!(__tmp & __mask)) {                                    \
                __tmp <<= 4;                                            \
                (r) += 4;                                               \
            }                                                           \
```

```
            __mask <<= 2;                                               \
            if (!(__tmp & __mask)) {                                    \
                __tmp <<= 2;                                            \
                (r) += 2;                                               \
            }                                                           \
            __mask <<= 1;                                               \
            if (!(__tmp & __mask)) {                                    \
                (r)++;                                                  \
            }                                                           \
        }                                                               \
    } while (0)

static __inline__ uint32_t
ceil2_32(uint64_t x)
{
    x--;
    x |= x >> 1;
    x |= x >> 2;
    x |= x >> 4;
    x |= x >> 8;
    x |= x >> 16;
    x++;

    return x;
}

static __inline__ uint64_t
ceil2_64(uint64_t x)
{
    x--;
    x |= x >> 1;
    x |= x >> 2;
    x |= x >> 4;
    x |= x >> 8;
    x |= x >> 16;
    x |= x >> 32;
    x++;

    return x;
}

/* internal macros. */
#define _ftoi32(f)     (*((int32_t *)&(f)))
#define _ftou32(f)     (*((uint32_t *)&(f)))
#define _dtoi64(d)     (*((int64_t *)&(d)))
#define _dtou64(d)     (*((uint64_t *)&(d)))
/* FIXME: little-endian. */
#define _dtohi32(d)    (*(((uint32_t *)&(d)) + 1))
/*
 * IEEE 32-bit
```

```
 * 0..22  - mantissa
 * 23..30 - exponent
 * 31     - sign
 */
/* convert elements of float to integer. */
#define fgetmant(f)       (_ftou32(f) & 0x007fffff)
#define fgetexp(f)        ((_ftou32(f) >> 23) & 0xff)
#define fgetsign(f)       (_ftou32(f) >> 31)
#define fsetmant(f, mant) (_ftou32(f) |= (mant) & 0x007fffff)
#define fsetexp(f, exp)   (_ftou32(f) |= ((exp) & 0xff) << 23)
#define fsetsign(f)       (_ftou32(f) | 0x80000000)
/*
 * IEEE 64-bit
 * 0..51  - mantissa
 * 52..62 - exponent
 * 63     - sign
 */
/* convert elements of double to integer. */
#define dgetmant(d)       (_dtou64(d) & UINT64_C(0x000fffffffffffff))
#define dgetexp(d)        ((_dtohi32(d) >> 20) & 0x7ff)
#define dgetsign(d)       (_dtohi32(d) >> 31)
#define dsetmant(d, mant)                                               \
    (*((uint64_t *)&(d)) |= (uint64_t)(mant) | UINT64_C(0x000fffffffffffff))
#define dsetexp(d, exp)                                                 \
    (*((uint64_t *)&(d)) |= (((uint64_t)((exp) & 0x7ff)) << 52))
#define dsetsign(d)                                                     \
    (*((uint64_t *)&(d)) |= UINT64_C(0x8000000000000000))

/*
 * IEEE 80-bit
 * 0..63  - mantissa
 * 64..78 - exponent
 * 79     - sign
 */
#define ldgetmant(ld)       (*((uint64_t *)&ld))
#define ldgetexp(ld)        (*((uint32_t *)&ld + 2) & 0x7fff)
#define ldgetsign(ld)       (*((uint32_t *)&ld + 3) & 0x8000)
#define ldsetmant(ld, mant) (*((uint64_t *)&ld = (mant)))
#define ldsetexp(ld, exp)   (*((uint32_t *)&ld + 2) |= (exp) & 0x7fff)
#define ldsetsign(ld)       (*((uint32_t *)&ld + 3) |= 0x80000000)

/* sign bit 0x8000000000000000. */
#define ifabs(d)                                                        \
    (_dtou64(d) & UINT64_C(0x7fffffffffffffff))
#define fabs2(d, t64)                                                   \
    (*((uint64_t *)&(t64)) = ifabs(d))
/* sign bit 0x80000000. */
#define ifabsf(f)                                                       \
    (_ftou32(f) & 0x7fffffff)
```

```
/* TODO: test the stuff below. */

/* (a < b) ? v1 : v2; */
#define condltset(a, b, v1, v2)                                         \
    (((((a) - (b)) >> (CHAR_BIT * sizeof(a) - 1)) & ((v1) ^ (v2))) ^ (v2))

/* c - conditional, f - flag, u - word */
#define condsetf(c, f, u) ((u) ^ ((-(u) ^ (u)) & (f)))

#define nextp2(a)                                                       \
    (((a)                                                               \
      | ((a) >> 1)                                                      \
      | ((a) >> 2)                                                      \
      | ((a) >> 4)                                                      \
      | ((a) >> 8)                                                      \
      | ((a) >> 16)) + 1)

/* (a < b) ? v1 : v2; */
#define condset(a, b, v1, v2)                                           \
    (((((a) - (b)) >> (CHAR_BIT * sizeof(a) - 1)) & ((v1) ^ (v2))) ^ (v2))

/* c - conditional, f - flag, u - word */
#define condsetf(c, f, u) ((u) ^ ((-(u) ^ (u)) & (f)))

#define sat8(x)                                                         \
    ((x) | (!((x) >> 8) - 1))
#define sat8b(x)                                                        \
    condset(x, 0xff, x, 0xff)

#define haszero(a) (~(a))
#if 0
#define haszero_32(a)                                                   \
    (~(((((a) & 0x7f7f7f7f) + 0x7f7f7f7f) | (a)) | 0x7f7f7f7f))
#endif

/* calculate modulus u % 10 */
#define modu10(u)                                                       \
    ((u) - ((((u) * 6554U) >> 16) * 10))

/* TODO: change modulus calculations to something faster */
#define leapyear(x)                                                     \
    (!((x) & 0x03) && ((((x) % 100)) || !((x) % 400)))

#endif /* __ZERO_TRIX_H__ */
```

Appendix C

Managing Builds with Tup

Rationale

This chapter is not meant to be a be-all manual for Tup; instead, I give a somewhat-quick overview in the hopes the readers will be able to get a jump-start for using Tup for their projects.

Why Tup?

There are many tools around to manage the task of building software projects. Whereas I have quite a bunch of experience with GNU Auto-tools and have been suggested learning to use Cmake, I was recently pointed to Tup; it was basically love at first sight.

C.1 Overview

Tup Initialisation

Initializing a source-code tree to be used with Tup is extremely simple. Just execute

```
tup init
```

in the top-level directory and you will be set.

Upwards Recursion

What makes Tup perhaps unique in its approach is that it recursively manages the whole build tree (or, optionally, smaller parts of it) by scanning the tree upwards. This means that when you execute

```
tup upd
```

the tree is scanned upwards for configuration files, and all directories with **Tupfile** are processed to be on-synch. You may alternatively choose to run

```
tup upd .
```

to limit the synchronisation (build) to the current working directory.

C.2 Using Tup

C.2.1 Tuprules.tup

It's a good idea to have a top-level **Tuprules.tup** file to set up things such as aliases for build commands. Here is a simple rules file which I use for my operating system project (additions to come in later).

Tuprules.tup

```
CC = gcc
LD = ld
CFLAGS = -g -Wall -O

!cc = |> ^ CC %f^ $(CC) $(CFLAGS) $(CFLAGS_%B) -c %f -o %o |> %B.o
!ld = |> ^ LD %f^ $(LD) $(LDFLAGS) $(LDFLAGS_%B) -o %o %f |>
```

Notes

- Environment variables are used much like in Unix shell scripts; here, I set CC (C compiler) to point to gcc, LD (linker) to point to ld, and CFLAGS (C compiler flags) to a useful default of -g -Wall -O; produce debugger output, turn on many warnings, and do basic optimisations.
- Aliases start with '**!**'; I set aliases for C compiler and linker (!cc and !ld, respectively).

C.2.2 Tup Syntax

C.2.3 Variables

Environment Variables

Tup lets us assign environment variables much like Unix shells do; e.g., to assign the value **gcc** to the variable **CC**, you would use the syntax

```
CC = gcc
```

or

```
CC := gcc
```

You can then refer to this variable like

```
$(CC)
```

in your Tup script files.

Conventional Environment Variables

Here comes a list of some commonly used environment variables and their purposes.

CC	C compiler command
LD	linker command
AS	assembler command
CFLAGS	C compiler flags
LDFLAGS	linker flags
ASFLAGS	assembler flags

Predefined @-Variables

TUP_CWD	path relative to the current Tupfile being parsed
TUP_ARCH	target-architecture for building objects
TUP_PLATFORM	target operating system

Notes

- **@-variables** can be specified in **tup.config**-files. For example, if you specify **CONFIG_PROJECT** in tup.config, you can refer to it as **@(PROJECT)** in Tupfile.
- @-variables differ from environment variables in two ways; they are read-only, and they are treated as dependencies; note that exported environment variables are dependencies as well.

Example tup.config

```
# tup.config for the zero project

CONFIG_PROJECT=zero
CONFIG_RELEASE=0.0.1
```

It is possible to set CONFIG_-varibles to the value '**n**' by having comments like

```
# CONFIG_RELEASE is not set
```

C.2.4 Rules

Tup rules take the following syntax

: [foreach] [inputs] [| order-only inputs] |> command |> [outputs] [| extra outputs] [{bin}]

Notes

- '[' and ']' are used to denote optional fields.
- '|' is used to separate fields.
- **foreach** is used to run one command per input file; if omitted, all input files are used as arguments for a single command.
- **inputs**-field lists filenames; shell-style wildcards '**?**' and '*****' are supported.
- **order-only inputs** are used as inputs but the filenames are **not present in %-flags**. This is useful e.g. for specifying dependencies on files such as headers generated elsewhere; Tup shall know to generate those files first without executing the command.

- **outputs** specifies the names of the files to be written by command.
- **extra outputs** are additional output files whose names do **not appear in the %o-flag**.
- **{bin}** can be used to group outputs into bins; later rules can use "**{bin}**" as an input to use all filenames in the bin. As an example, the **foreach** rule will put all output files into the **objs bin**.

C.2.5 Macros

The following is an example macro to specify the C compiler and default flags to be used with it.

!cc = |> ĈC %f$̂(CC) $(CFLAGS) $(CFLAGS_%B) -c %f -o %o |> %B.o

Notes

- Macros take '**!**' as their prefix, in contrast with rules being prefixed with '**:**'.
- **ĈC %f**ˆ controls what is echoed to the user when the command is run; note that the space after the first 'ˆ' is required; the letters immediately following the ŵould be flags.
- **%B** evalutes to the name of the current file **without the extension**; similarly, **%f** is the name of the current input file, and **%o** is the name of the current output file.

C.2.6 Flags

ˆflags

- the '**c**' flag causes the command to run inside a chroot-environment (currently under Linux and OSX), so that the effective working directory of the subprocess is different from the current working directory.

%-flags

%f	the current filename from the inputs section
%b	the basename (path stripped) of the current input file
%B	like %b, but strips the filename extension
%e	the extension of current file with **foreach**
%o	the name(s) of output file(s) in the cammand section
%O	like %o, but without the extension
%d	the name of the lowest-level directory in path

C.2.7 Directives

ifeq (val1,val2)

The ifeq-directive tells Tup to do the things before the next endif or else in case val1 is found to be equal to val2. Note that any spaces included within the parentheses are processed verbatim. All $- and @-variables are substituted within val1 and val2.

ifneq (val1,val2)

The ifneq-directive inverts the logic of ifeq; the following things are done if val1 is **not equal** with val2.

ifdef VARIABLE

The things before the next endif shall be done if the @-variable is defined at all in tup.config.

ifndef VARIABLE

ifndef inverts the logic of ifdef.

else

else toggles the logical truth value of the previous ifeq/ifneq/ifdef/ifndef statement.

endif

Ends the previous ifeq/ifneq/ifdef/ifndef statement.

include file

Reads a regular file and continues parsing the current Tupfile.

include_rules

Scans the directory tree up for the first Tuprules.tup file and then reads all Tuprules.tup files from it down to the one possibly in the current directory. Usually specified as the first line in Tupfile.

run ./script args

Run an external script with the given arguments to **generate :-rules**. The script is expected to write the :-rules to standard output (stdout). The script cannot create $-variables or !-macros, but it can output :-rules that use those features.

preload directory

By default, run-scripts can only use wild-cards for files in the current directory. To specify other wild-card directories to be scanne, you can use **preload**.

export VARIABLE

Adds the environment variable VARIABLE to be used by future :-rules and run-scripts. VARIABLE comes from environment, not the Tupfile, so you can control the contents using your shell. On Unix-systems, only PATH is exported by default.

.gitignore

Tells Tup to automatically generate a .gitignore file with a list of Tup-generated output files.

#

at the beginning of a line marks the line as a comment.