Computer Science from the Bottom Up

Ian Wienand

Computer Science from the Bottom Up

Ian Wienand

A PDF version is available at https://www.bottomupcs.com/csbu.pdf. A EPUB version is available at https://www.bottomupcs.com/csbu.epub The original souces are available at https://github.com/ianw/bottomupcs

Table of Contents

List of Figures

List of Tables

List of Examples

Introduction

Welcome

Welcome to Computer Science from the Bottom Up

Philosophy

In a nutshell, what you are reading is intended to be a shop class for computer science. Young computer science students are taught to "drive" the computer; but where do you go to learn what is under the hood? Trying to understand the operating system is unfortunately not as easy as just opening the bonnet. The current Linux kernel runs into the millions of lines of code, add to that the other critical parts of a modern operating system (the compiler, assembler and system libraries) and your code base becomes unimaginable. Further still, add a University level operating systems course (or four), some good reference manuals, two or three years of C experience and, just maybe, you might be able to figure out where to *start looking* to make sense of it all.

To keep with the car analogy, the prospective student is starting out trying to work on a Formula One engine without ever knowing how a two stroke motor operates. During their shop class the student should pull apart, twist, turn and put back together that two stroke motor, and consequentially have a pretty good framework for understanding just how the Formula One engine works. Nobody will expect them to be a Formula One engineer, but they are well on their way!

Why *from the bottom up*?

Not everyone wants to attend shop class. Most people only want to drive the car, not know how to build one from scratch. Obviously any general computing curriculum has to take this into account else it won't be relevant to its students. So computer science is taught from the "top down"; applications, high level programming, software design and development theory, possibly data structures. Students will probably be exposed to binary, hopefully binary logic, possibly even some low level concepts such as registers, opcodes and the like at a superficial level.

This book aims to move in completely the opposite direction, working from operating systems fundamentals through to how those applications are complied and executed.

Enabling technologies

This book is only possible thanks to the development of *Open Source* technologies. Before Linux it was like taking a shop course with a car that had it's bonnet welded shut; today we are in a position to open that bonnet, poke around with the insides and, better still, take that engine and use it to do whatever we want.

Chapter 1. General Unix and Advanced C

Everything is a file!

An often quoted tenet of UNIX-like systems such as Linux or BSD is *everything is a file*.

Imagine a file in the context something familiar like a word processor. There are two fundamental operations we could use on this imaginary word processing file:

1. Read it (existing saved data from the word processor).

2. Write to it (new data from the user).

Consider some of the common things attached to a computer and how they relate to our fundamental file operations:

1. The screen

2. The keyboard

3. A printer

4. A CDROM

The screen and printer are both like a write-only file, but instead of being stored as bits on a disk the information is displayed as dots on a screen or lines on a page. The keyboard is like a read only file, with the data coming from keystrokes provided by the user. The CDROM is similar, but rather than randomly coming from the user the data is stored directly on the disk.

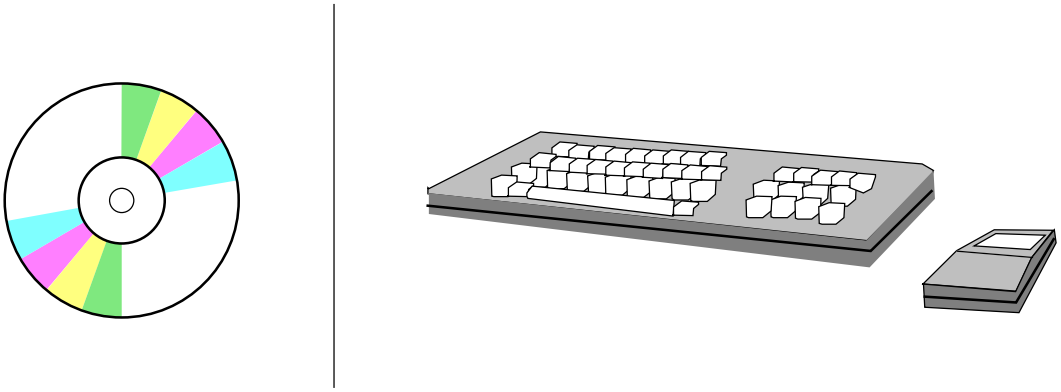
Thus the concept of a file is a good *abstraction* of either a a sink for, or source of, data. As such it is an excellent abstraction of all the devices one might attach to the computer. This realisation is the great power of UNIX and is evident across the design of the entire platform. It is one of the fundamental roles of the operating system to provide this abstraction of the hardware to the programmer.

It is probably not too much of a stretch to say abstraction is *the* primary concept that underpins all modern computing. No one person can understand everything from designing a modern user-interface to the internal workings of a modern CPU, much less build it all themselves. To programmers, abstractions are the *lingua franca* that allows us to collaborate and invent.

Learning to navigate across abstractions gives one greater insight into how to *use* the abstractions in the best and most innovative ways. In this book, we are concerned with abstractions at the lowest layers; between applications and the operating-system and the operating-system and hardware. Above this lies many more layers, each worthy of

their own books. As these chapters progress, you will hopefully gain some insight into the abstractions presented by a modern operating-system.

Figure 1.1. Abstraction



Spot the difference?

Implementing abstraction

In general, abstraction is implemented by what is generically termed an *Application Programming Interface* (API). API is a somewhat nebulous term that means different things in the context of various programming endavours. Fundamentally, a programmer designs a set of functions and documents their interface and functionality with the principle that the actual implementation providing the API is opaque.

For example, many large web-applications provide an API accessible via HTTP. Accessing data via this method surely triggers many complicated series of remote-procedure calls, database queries and data transfer; all of which is opaque to the end user who simply receives the contracted data.

Those familiar with *object-oriented* languages such as Java, Python or C++ would be familiar with the abstraction provided by *classes*. Methods provide the interface to the class, but abstract the implementation.

Implementing abstraction with C

A common method used in the Linux Kernel and other large C code bases, which lacks a built-in concept of object-orientation, is *function pointers*. Learning to read this idiom is key to navigating most large C code-bases. By understanding how to read the abstractions provided within the code an understanding of internal API designs can be built.

Example 1.1. Abstraction with function pointers

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

```
/* The API to implement */
struct greet_api
{
 int (*say_hello)(char *name);
 int (*say_goodbye)(void);
};

/* Our implementation of the hello function */
int say_hello_fn(char *name)
{
 printf("Hello %s\n", name);
 return 0;
}

/* Our implementation of the goodbye function */
int say_goodbye_fn(void)
{
 printf("Goodbye\n");
 return 0;
}

/* A struct implementing the API */
struct greet_api greet_api =
{
 .say_hello = say_hello_fn,
 .say_goodbye = say_goodbye_fn
};

/* main() doesn't need to know anything about how the
 * say_hello/goodbye works, it just knows that it does */
int main(int argc, char *argv[])
{
 greet_api.say_hello(argv[1]);
 greet_api.say_goodbye();

 printf("%p, %p, %p\n", greet_api.say_hello, say_hello_fn, &say_hello_fn);

 exit(0);
}
```

Code such as the above is the simplest example of constructs used repeatedly through the Linux Kernel and other C programs. Lets have a look at some specific elements.

We start out with a structure that defines the API (`struct greet_api`). The functions whose names are encased in parenthesis with a pointer marker describe a *function pointer*[1]. The function pointer describes the *prototype* of function it must point to; pointing it at a function without the correct return type or parameters will generate a compiler warning at least; if left in code will likely lead to incorrect operation or crashes.

We then have our implementation of the API. Often for more complex functionality you will see an idiom where API implementation functions will only be a wrapper around another function that is conventionally prepended with one or or two underscores[2] (i.e. `say_hello_fn()` would call another function `_say_hello_function()`). This has several uses; generally it relates to having simpler and smaller parts of the API (marshalling or checking arguments, for example) separate to more complex implementation, which often eases the path to significant changes in the internal

[1] Often you will see that the names of the parameters are omitted, and only the type of the parameter is specified. This allows the implementer to specify their own parameter names avoiding warnings from the compiler.
[2] A double-underscore function `__foo` may conversationally be referred to as "dunder foo".

workings whilst ensuring the API remains constant. Our implementation is very simple however, and doesn't even need it's own support functions. In various projects, single, double or even triple underscore function prefixes will mean different things, but universally it is a visual warning that the function is not supposed to be called directly from "beyond" the API.

Second to last, we fill out the function pointers in `struct greet_api greet_api`. The name of the function is a pointer, therefore there is no need to take the address of the function (i.e. `&say_hello_fn`).

Finally we can call the API functions through the structure in `main`.

You will see this idiom constantly when navigating the source code. The tiny example below is taken from `include/linux/virtio.h` in the Linux kernel source to illustrate:

Example 1.2. Abstraction in `include/linux/virtio.h`

```
         /**
 * virtio_driver - operations for a virtio I/O driver
 * @driver: underlying device driver (populate name and owner).
 * @id_table: the ids serviced by this driver.
 * @feature_table: an array of feature numbers supported by this driver.
 * @feature_table_size: number of entries in the feature table array.
 * @probe: the function to call when a device is found.  Returns 0 or -errno.
 * @remove: the function to call when a device is removed.
 * @config_changed: optional function to call when the device configuration
 *    changes; may be called in interrupt context.
 */
struct virtio_driver {
        struct device_driver driver;
        const struct virtio_device_id *id_table;
        const unsigned int *feature_table;
        unsigned int feature_table_size;
        int (*probe)(struct virtio_device *dev);
        void (*scan)(struct virtio_device *dev);
        void (*remove)(struct virtio_device *dev);
        void (*config_changed)(struct virtio_device *dev);
#ifdef CONFIG_PM
        int (*freeze)(struct virtio_device *dev);
        int (*restore)(struct virtio_device *dev);
#endif
};
```

It's only necessary to vaguely understand that this structure is a description of a virtual I/O device. We can see the user of this API (the device driver author) is expected to provide a number of functions that will be called under various conditions during system operation (when probing for new hardware, when hardware is removed, etc). It also contains a range of data; structures which should be filled with relevant data.

Starting with descriptors like this is usually the easiest way into understanding the various layers of kernel code.

Libraries

Libraries have two roles which illustrate abstraction.

- Allow programmers to reuse commonly accessed code.
- Act as a *black box* implementing functionality for the programmer.

For example, a library implementing access to the raw data in JPEG files has both the advantage that the many programs who wish to access image files can all use the same library and the programmers building these programs do not need to worry about the exact details of the JPEG file format, but can concentrate their efforts on what their program wants to do with the image.

The standard library of a UNIX platform is generically referred to as `libc`. It provides the basic interface to the system: fundamental calls such as `read()`, `write()` and `printf()`. This API is described in its entirety by a specification called `POSIX`. It is freely available online and describes the many calls that make up the standard UNIX API.

Most UNIX platforms broadly follow the POSIX standard, though often differ small but sometimes important ways (hence the complexity of the various GNU autotools, which often tries to abstract away these differences for you). Linux has many interfaces that are not specified by POSIX; writing applications that use them exclusively will make your application less portable.

Libraries are a fundamental abstraction with many details. Later chapters will describe how libraries work in much greater detail.

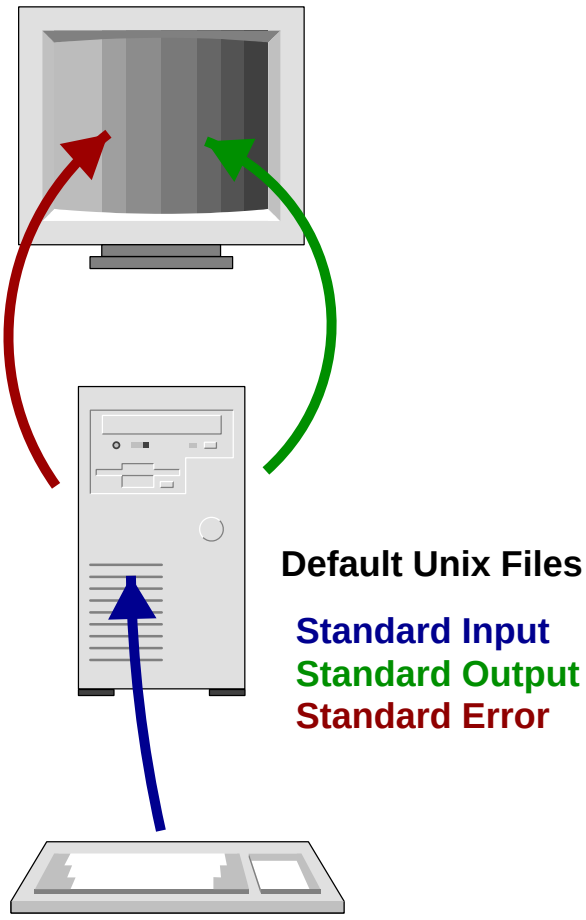
File Descriptors

One of the first things a UNIX programmer learns is that every running program starts with three files already opened:

Table 1.1. Standard Files Provided by Unix

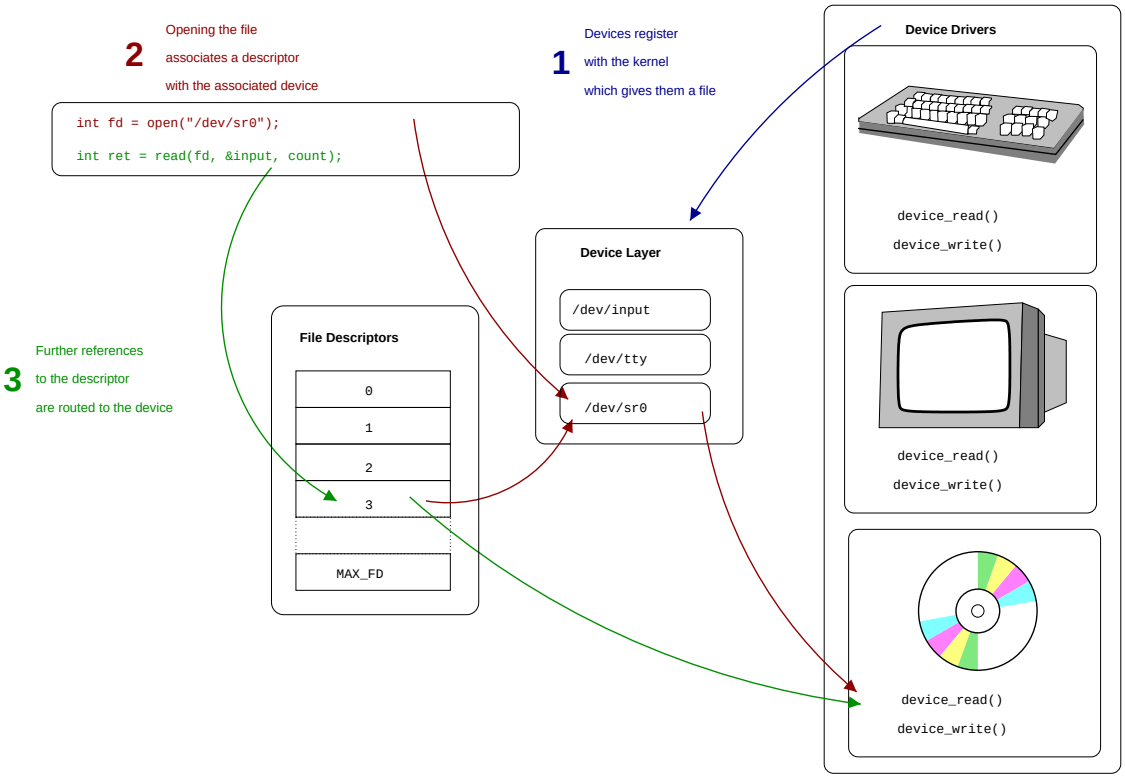
Descriptive Name	File Number	Description
Standard In	0	Input from the keyboard
Standard Out	1	Output to the console
Standard Error	2	Error output to the console

Figure 1.2. Default Unix Files



This raises the question what an *open file* represents. The value returned by an `open` call is termed a *file descriptor* and is essentially an index into an array of open files kept by the kernel.

Figure 1.3. Abstraction



File descriptors are an index into a file-descriptor table stored by the kernel. The kernel creates a file-descriptor in response to an `open` call and associates the file-descriptor with some abstraction of an underlying file-like object; be that an actual hardware device, or a file-system or something else entirely. Consequently a processes `read` or `write` calls that reference that file-descriptor are routed to the correct place by the kernel to ultimately do something useful.

In short, the file-descriptor is the gateway into the kernel's abstractions of underlying hardware. An overall view of the abstraction for physical-devices is shown in Figure 1.3, “Abstraction”.

Starting at the lowest level, the operating system requires a programmer to create a *device-driver* to be able to communicate with a hardware device. This device-driver is written to an API provided by the kernel just like in Example 1.2, “Abstraction in `include/linux/virtio.h`”; the device-driver will provide a range of functions which are called by the kernel in response to various requirements. In the simplified example above, we can see the drivers provide a `read` and `write` function that will be called in response to the analogous operations on the file-descriptor. The device-driver knows how to convert these generic requests into specific requests or commands for a particular device.

To provide the abstraction to user-space, the kernel provides a file-interface via what is generically termed a *device layer*. Physical devices on the host are represented by a file in a special file-system such as `/dev`. In UNIX-like systems, so called *device-nodes*

have what are termed a *major* and a *minor* number which allows the kernel to associate particular nodes with their underlying driver. These can be identified via `ls` as illustrated in Example 1.3, “Example of major and minor numbers”.

Example 1.3. Example of major and minor numbers

```
       $ ls -l /dev/null /dev/zero /dev/tty
crw-rw-rw- 1 root root 1, 3 Aug 26 13:12 /dev/null
crw-rw-rw- 1 root root 5, 0 Sep  2 15:06 /dev/tty
crw-rw-rw- 1 root root 1, 5 Aug 26 13:12 /dev/zero
```

This brings us to the file-descriptor, which is the handle user-space uses to talk to the underlying device. In a broad-sense, what happens when a file is `open`ed is that the kernel is using the path information to map the file-descriptor with something that provides an appropriate `read` and `write`, etc. API. When this `open` is for a device (`/dev/sr0` above), the major and minor number of the opened device-node provides the information the kernel needs to find the correct device-driver and complete the mapping. The kernel will then know how to route further calls such as `read` to the underlying functions provided by the device-driver.

A non-device file operates similarly, although there are more layers in-between. The abstraction here is the *mount-point*; mounting a file-system has the dual purpose of setting up a mapping so the file-system knows the underlying device that provides the storage and the kernel knows that files opened under that mount-point should be directed to the file-system driver. Like device-drivers, file-systems are written to a particular generic file-system API provided by the kernel.

There are indeed many other layers that complicate the picture in real-life. For example, the kernel will go to great efforts to cache as much data from disks as possible in otherwise free-memory; this provides many speed advantages. It will also try to organise device access in the most efficient ways possible; for example trying to order disk-access to ensure data stored physically close to each other is retrieved together, even if the requests did not arrive in such an order. Further, many devices are of a more generic class such as USB or SCSI devices which provide their own abstraction layers to write too. Thus rather than writing directly to devices, file-systems will go through these many layers. Understanding the kernel is to understand how these many APIs interrelate and coexist.

The Shell

The shell is the gateway to interacting with the operating system. Be it `bash`, `zsh`, `csh` or any of the many other shells, they all fundamentally have only one major task — to allow you to execute programs (you will begin to understand how the shell actually does this when we talk about some of the internals of the operating system later).

But shells do much more than allow you to simply execute a program. They have powerful abilities to redirect files, allow you to execute multiple programs simultaneously and script complete programs. These all come back to the *everything is a file* idiom.

Redirection

Often we do not want the standard file descriptors mentioned in the section called “File Descriptors” to point to their default places. For example, you may wish to capture all the output of a program into a file on disk, or, alternatively have it read its commands from a file you prepared earlier. Another useful task might like to pass the output of one program to the input of another. With the operating system, the shell facilitates all this and more.

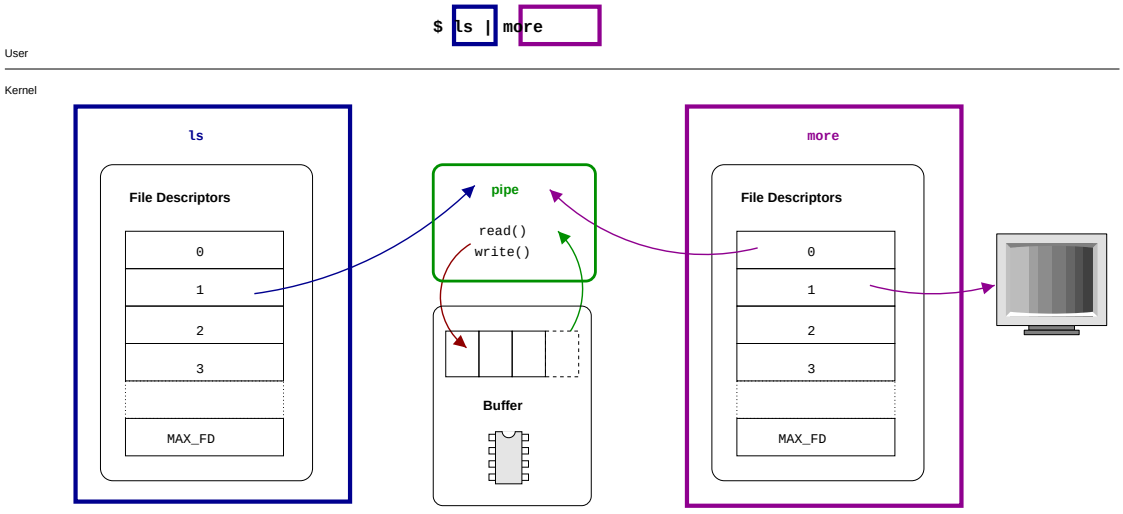
Table 1.2. Standard Shell Redirection Facilities

Name	Command	Description	Example
Redirect to a file	`> filename`	Take all output from standard out and place it into `filename`. Note using >> will append to the file, rather than overwrite it.	`ls > filename`
Read from a file	`< filename`	Copy all data from the file to the standard input of the program	`echo < filename`
Pipe	`program1 | program2`	Take everything from standard out of `program1` and pass it to standard input of `program2`	`ls | more`

Implementing `pipe`

The implementation of `ls | more` is just another example of the power of abstraction. What fundamentally happens here is that instead of associating the file-descriptor for the standard-output with some sort of underlying device (such as the console, for output to the terminal), the descriptor is pointed to an in-memory buffer provided by the kernel commonly termed a `pipe`. The trick here is that another process can associate its standard *input* with the other-side of this same buffer and effectively consume the output of the other process. This is illustrated in Figure 1.4, “A pipe in action”

Figure 1.4. A pipe in action



The pipe is an in-memory buffer that connects two processes together. File-descriptors point to the pipe object, which buffers data sent to it (via a `write`) to be *drained* (via a `read`)

Writes to the pipe are stored by the kernel until a corresponding read from the other side *drains* the buffer. This is a very powerful concept and is one of the fundamental forms of *inter-process communication* or IPC in UNIX like operating systems. The pipe allows more than just a data transfer; it can act as a signaling channel. If a process `reads` an empty pipe, it will by default *block* or be put into hibernation until there is some data available (this is discussed in much greater depth in Chapter 5, *The Process*. Thus two processes may use a pipe to communicate that some action has been taken just by writing a byte of data; rather than the actual data being important, the mere presence of *any* data in the pipe can signal a message. Say for example one process requests that another print a file - something that will take some time. The two processes may setup a pipe between themselves where the requesting process does a `read` on the empty pipe; being empty that call blocks and the process does not continue. Once the print is done, the other process can write a message into the pipe, which effectively wakes up the requesting process and signals the work is done.

Allowing processes to pass data between each other like this springs another common UNIX idiom of small tools doing one particular thing. Chaining these small tools gives a flexibility that a single monolithic tool often can not.

Chapter 2. Binary and Number Representation

Binary — the basis of computing

Binary Theory

Introduction

Binary is a base-2 number system that uses two mutually exclusive states to represent information. A binary number is made up of elements called *bits* where each bit can be in one of the two possible states. Generally, we represent them with the numerals `1` and `0`. We also talk about them being true and false. Electrically, the two states might be represented by high and low voltages or some form of switch turned on or off.

We build binary numbers the same way we build numbers in our traditional base 10 system. However, instead of a one's column, a 10's column, a 100's column (and so on) we have a one's column, a two's columns, a four's column, an eight's column, and so on, as illustrated below.

Table 2.1. Binary

$2^{...}$	2^6	2^5	2^4	2^3	2^2	2^1	2^0
...	64	32	16	8	4	2	1

For example, to represent the number 203 in base 10, we know we place a 3 in the `1`'s column, a `0` in the `10`'s column and a 2 in the `100`'s column. This is expressed with exponents in the table below.

Table 2.2. 203 in base 10

10^2	10^1	10^0
2	0	3

Or, in other words, $2 \times 10^2 + 3 \times 10^0 = 200 + 3 = 203$. To represent the same thing in binary, we would have the following table.

Table 2.3. 203 in base 2

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
1	1	0	0	1	0	1	1

That equates to $2^7 + 2^6 + 2^3 + 2^1 + 2^0 = 128 + 64 + 8 + 2 + 1 = 203$.

The basis of computing

You may be wondering how a simple number is the basis of all the amazing things a computer can do. Believe it or not, it is! The processor in your computer has a

complex but ultimately limited set of *instructions* it can perform on values such as addition, multiplication, etc. Essentially, each of these instructions is assigned a number so that an entire program (add this to that, multiply by that, divide by this and so on) can be represented by a just a stream of numbers. For example, if the processor knows operation 2 is addition, then 252 could mean "add 5 and 2 and store the output somewhere". The reality is of course much more complicated (see Chapter 3, *Computer Architecture*) but, in a nutshell, this is what a computer is.

In the days of punch-cards, one could see with their eye the one's and zero's that make up the program stream by looking at the holes present on the card. Of course this moved to being stored via the polarity of small magnetic particles rather quickly (tapes, disks) and onto the point today that we can carry unimaginable amounts of data in our pocket.

Translating these numbers to something useful to humans is what makes a computer so useful. For example, screens are made up of millions of discrete *pixels*, each too small for the human eye to distinguish but combining to make a complete image. Generally each pixel has a certain red, green and blue component that makes up it's display color. Of course, these values can be represented by numbers, which of course can be represented by binary! Thus any image can be broken up into millions of individual dots, each dot represented by a *tuple* of three values representing the red, green and blue values for the pixel. Thus given a long string of such numbers, formatted correctly, the video hardware in your computer can convert those numbers to electrical signals to turn on and off individual pixels and hence display an image.

As you read on, we will build up the entire modern computing environment from this basic building block; *from the bottom-up* if you will!

Bits and Bytes

As discussed above, we can essentially choose to represent anything by a number, which can be converted to binary and operated on by the computer. For example, to represent all the letters of the alphabet we would need at least enough different combinations to represent all the lower case letters, the upper case letters, numbers and punctuation, plus a few extras. Adding this up means we need probably around 80 different combinations.

If we have two bits, we can represent four possible unique combinations (`00 01 10 11`). If we have three bits, we can represent 8 different combinations. In general, with n bits we can represent 2^n unique combinations.

8 bits gives us $2^8 = 256$ unique representations, more than enough for our alphabet combinations. We call a group of 8 bits a *byte*. Guess how big a C `char` variable is? One byte.

ASCII

Given that a byte can represent any of the values 0 through 256, anyone could arbitrarily make up a mapping between characters and numbers. For example, a video card manufacturer could decide that `1` represents `A`, so when value `1` is sent to the video card it displays a capital 'A' on the screen. A printer manufacturer might decide for some

obscure reason that 1 represented a lower-case 'z', meaning that complex conversions would be required to display and print the same thing.

To avoid this happening, the *American Standard Code for Information Interchange* or ASCII was invented. This is a *7-bit* code, meaning there are 2^7 or 128 available codes.

The range of codes is divided up into two major parts; the non-printable and the printable. Printable characters are things like characters (upper and lower case), numbers and punctuation. Non-printable codes are for control, and do things like make a carriage-return, ring the terminal bell or the special `NULL` code which represents nothing at all.

127 unique characters is sufficient for American English, but becomes very restrictive when one wants to represent characters common in other languages, especially Asian languages which can have many thousands of unique characters.

To alleviate this, modern systems are moving away from ASCII to *Unicode*, which can use up to 4 bytes to represent a character, giving *much* more room!

Parity

ASCII, being only a 7-bit code, leaves one bit of the byte spare. This can be used to implement *parity* which is a simple form of error checking. Consider a computer using punch-cards for input, where a hole represents 1 and no hole represents 0. Any inadvertent covering of a hole will cause an incorrect value to be read, causing undefined behaviour.

Parity allows a simple check of the bits of a byte to ensure they were read correctly. We can implement either *odd* or *even* parity by using the extra bit as a *parity bit*.

In odd parity, if the number of 1's in the 7 bits of information is odd, the parity bit is set, otherwise it is not set. Even parity is the opposite; if the number of 1's is even the parity bit is set to 1.

In this way, the flipping of one bit will case a parity error, which can be detected.

XXX more about error correcting

16, 32 and 64 bit computers

Numbers do not fit into bytes; hopefully your bank balance in dollars will need more range than can fit into one byte! Most modern architectures are *32 bit* computers. This means they work with 4 bytes at a time when processing and reading or writing to memory. We refer to 4 bytes as a *word*; this is analogous to language where letters (bits) make up words in a sentence, except in computing every word has the same size! The size of a C `it` variable is 32 bits. Newer architectures are 64 bits, which doubles the size the processor works with (8 bytes).

Kilo, Mega and Giga Bytes

Computers deal with a lot of bytes; that's what makes them so powerful!

We need a way to talk about large numbers of bytes, and a natural way is to use the "International System of Units" (SI) prefixes as used in most other scientific areas. So for example, kilo refers to 10^3 or 1000 units, as in a kilogram has 1000 grams.

1000 is a nice round number in base 10, but in binary it is `1111101000` which is not a particularly "round" number. However, 1024 (or 2^{10}) is (`10000000000`), and happens to be quite close to the base ten meaning of kilo (1000 as opposed to 1024).

Hence 1024 bytes became known as a *kilobyte*. The first mass market computer was the Commodore 64, so named because it had 64 kilobytes of storage.

Today, kilobytes of memory would be small for a wrist watch, let alone a personal computer. The next SI unit is "mega" for 10^6. As it happens, 2^{20} is again close to the SI base 10 definition; 1048576 as opposed to 1000000.

The units keep increasing by powers of 10; each time it diverges further from the base SI meaning.

Table 2.4. Bytes

2^{10}	Kilobyte
2^{20}	Megabyte
2^{30}	Gigabyte
2^{40}	Terrabyte
2^{50}	Petabyte
2^{60}	Exabyte

Therefore a 32 bit computer can address up to four gigabytes of memory; the extra two bits can represent four groups of 2^{30} `bytes.`. A 64 bit computer can address up to 8 exabytes; you might be interested in working out just how big a number this is! To get a feel for how bit that number is, calculate how long it would take to count to 2^{64} if you incremented once per second.

Kilo, Mega and Giga Bits

Apart from the confusion related to the overloading of SI units between binary and base 10, capacities will often be quoted in terms of *bits* rather than bytes.

Generally this happens when talking about networking or storage devices; you may have noticed that your ADSL connection is described as something like 1500 kilobits/second. The calculation is simple; multiply by 1000 (for the kilo), divide by 8 to get bytes and then 1024 to get kilobytes (so 1500 kilobits/s=183 kilobytes per second).

The SI standardisation body has recognised these dual uses, and has specified unique prefixes for binary usage. Under the standard 1024 bytes is a `kibibyte`, short for *kilo binary* byte (shortened to KiB). The other prefixes have a similar prefix (Mebibyte, for example). Tradition largely prevents use of these terms, but you may seem them in some literature.

Conversion

The easiest way to convert between bases is to use a computer, after all, that's what they're good at! However, it is often useful to know how to do conversions by hand.

The easiest method to convert between bases is *repeated division*. To convert, repeatedly divide the quotient by the base, until the quotient is zero, making note of the remainders at each step. Then, write the remainders in reverse, starting at the bottom and appending to the right each time. An example should illustrate; since we are converting to binary we use a base of 2.

Table 2.5. Convert 203 to binary

Quotient		**Remainder**	
$203_{10} \div 2 =$	101	1	
$101_{10} \div 2 =$	50	1	↑
$50_{10} \div 2 =$	25	0	↑
$25_{10} \div 2 =$	12	1	↑
$12_{10} \div 2 =$	6	0	↑
$6_{10} \div 2 =$	3	0	↑
$3_{10} \div 2 =$	1	1	↑
$1_{10} \div 2 =$	0	1	↑

Reading from the bottom and appending to the right each time gives `11001011`, which we saw from the previous example was 203.

Boolean Operations

George Boole was a mathematician who discovered a whole area of mathematics called *Boolean Algebra*. Whilst he made his discoveries in the mid 1800's, his mathematics are the fundamentals of all computer science. Boolean algebra is a wide ranging topic, we present here only the bare minimum to get you started.

Boolean operations simply take a particular input and produce a particular output following a rule. For example, the simplest boolean operation, `not` simply inverts the value of the input operand. Other operands usually take two inputs, and produce a single output.

The fundamental Boolean operations used in computer science are easy to remember and listed below. We represent them below with *truth tables*; they simply show all possible inputs and outputs. The term *true* simply reflects `1` in binary.

Not

Usually represented by `!`, `not` simply inverts the value, so `0` becomes `1` and `1` becomes `0`

Table 2.6. Truth table for *not*

Input	Output
1	0
0	1

And

To remember how the and operation works think of it as "if one input *and* the other are true, result is true

Table 2.7. Truth table for *and*

Input 1	Input 2	Output
0	0	0
1	0	0
0	1	0
1	1	1

Or

To remember how the `or` operation works think of it as "if one input *or* the other input is true, the result is true

Table 2.8. Truth table for *or*

Input 1	Input 2	Output
0	0	0
1	0	1
0	1	1
1	1	1

Exclusive Or (xor)

Exclusive or, written as `xor` is a special case of `or` where the output is true if one, and *only* one, of the inputs is true. This operation can surprisingly do many interesting tricks, but you will not see a lot of it in the kernel.

Table 2.9. Truth table for *xor*

Input 1	Input 2	Output
0	0	0
1	0	1
0	1	1
1	1	0

How computers use boolean operations

Believe it or not, essentially everything your computer does comes back to the above operations. For example, the half adder is a type of circuit made up from boolean operations that can add bits together (it is called a half adder because it does not handle carry bits). Put more half adders together, and you will start to build something that can add together long binary numbers. Add some external memory, and you have a computer.

Electronically, the boolean operations are implemented in *gates* made by *transistors*. This is why you might have heard about transistor counts and things like Moore's Law. The more transistors, the more gates, the more things you can add together. To create the modern computer, there are an awful lot of gates, and an awful lot of transistors. Some of the latest Itanium processors have around 460 million transistors.

Working with binary in C

In C we have a direct interface to all of the above operations. The following table describes the operators

Table 2.10. Boolean operations in C

Operation	Usage in C
`not`	`!`
`and`	`&`
`or`	`|`
`xor`	`^`

We use these operations on variables to modify the bits within the variable. Before we see examples of this, first we must divert to describe hexadecimal notation.

Hexadecimal

Hexadecimal refers to a base 16 number system. We use this in computer science for only one reason, it makes it easy for humans to think about binary numbers. Computers only ever deal in binary and hexadecimal is simply a shortcut for us humans trying to work with the computer.

So why base 16? Well, the most natural choice is base 10, since we are used to thinking in base 10 from our every day number system. But base 10 does not work well with binary -- to represent 10 different elements in binary, we need four bits. Four bits, however, gives us sixteen possible combinations. So we can either take the very tricky road of trying to convert between base 10 and binary, or take the easy road and make up a base 16 number system -- hexadecimal!

Hexadecimal uses the standard base 10 numerals, but adds `A B C D E F` which refer to `10 11 12 13 14 15` (n.b. we start from zero).

Traditionally, any time you see a number prefixed by `0x` this will denote a hexadecimal number.

As mentioned, to represent 16 different patterns in binary, we would need exactly four bits. Therefore, each hexadecimal numeral represents exactly four bits. You should consider it an exercise to learn the following table off by heart.

Table 2.11. Hexadecimal, Binary and Decimal

Hexadecimal	Binary	Decimal
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
A	1010	10
B	1011	11
C	1100	12
D	1101	13
E	1110	14
F	1111	15

Of course there is no reason not to continue the pattern (say, assign G to the value 16), but 16 values is an excellent trade off between the vagaries of human memory and the number of bits used by a computer (occasionally you will also see base 8 used, for example for file permissions under UNIX). We simply represent larger numbers of bits with more numerals. For example, a sixteen bit variable can be represented by 0xAB12, and to find it in binary simply take each individual numeral, convert it as per the table and join them all together (so `0xAB12` ends up as the 16-bit binary number `1010101100010010`). We can use the reverse to convert from binary back to hexadecimal.

We can also use the same repeated division scheme to change the base of a number. For example, to find 203 in hexadecimal

Table 2.12. Convert 203 to hexadecimal

Quotient		Remainder	
$203_{10} \div 16 =$	12	11 (0xB)	

Quotient		Remainder	
$12_{10} \div 16 =$	0	12 (0xC)	↑

Hence 203 in hexadecimal is `0xCB`.

Practical Implications

Use of binary in code

Whilst binary is the underlying language of every computer, it is entirely practical to program a computer in high level languages without knowing the first thing about it. However, for the low level code we are interested in a few fundamental binary principles are used repeatedly.

Masking and Flags

Masking

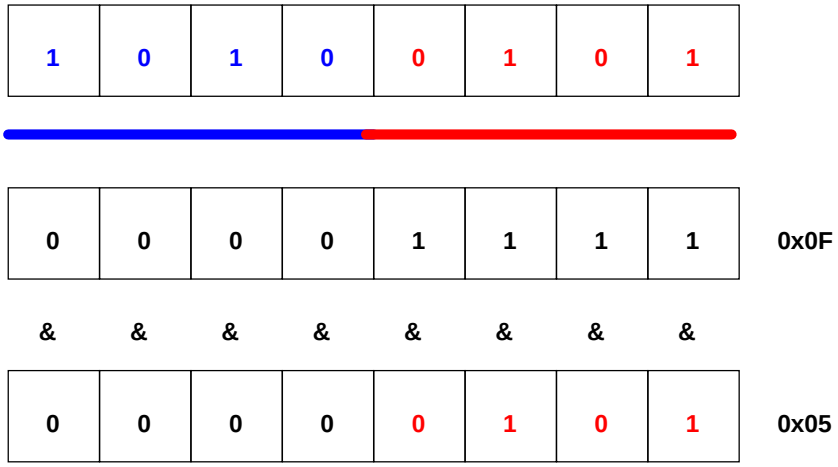
In low level code, it is often important to keep your structures and variables as space efficient as possible. In some cases, this can involve effectively packing two (generally related) variables into one.

Remember each bit represents two states, so if we know a variable only has, say, 16 possible states it can be represented by 4 bits (i.e. 2^4=16 unique values). But the smallest type we can declare in C is 8 bits (a `char`), so we can either waste four bits, or find some way to use those left over bits.

We can easily do this by the process of *masking*. Remembering the rules of the logical operations, it should become clear how the values are extracted.

The process is illustrated in the figure below. We are interested in the lower four bits, so set our mask to have these bits set to `1`. Since the `logical and` operation will only set the bit if *both* bits are `1`, those bits of the mask set to `0` effectively hide the bits we are not interested in.

Figure 2.1. Masking



To get the top (blue) four bits, we would invert the mask. You will note this gives a result of `0x90` when really we want a value of `0x09`. To get the bits into the right position we use the `right shift` operation.

Setting the bits requires the `logical or` operation. However, rather than using `1`'s as the mask, we use `0`'s. You should draw a diagram similar to the above figure and work through setting bits with the `logical or` operation.

Flags

Often a program will have a large number of variables that only exist as *flags* to some condition. For example, a state machine is an algorithm that transitions through a number of different states but may only be in one at a time. Say it has 8 different states; we could easily declare 8 different variables, one for each state. But in many cases it is better to declare *one 8 bit variable* and assign each bit to *flag* flag a particular state.

Flags are a special case of masking, but each bit represents a particular boolean state (on or off). An *n* bit variable can hold *n* different flags. See the code example below for a typical example of using flags -- you will see variations on this basic code very often.

Example 2.1. Using flags

```
 1
                #include <stdio.h>

   /*
 5  *  define all 8 possible flags for an 8 bit variable
    *      name  hex     binary
    */
   #define FLAG1 0x01 /* 00000001 */
   #define FLAG2 0x02 /* 00000010 */
10 #define FLAG3 0x04 /* 00000100 */
   #define FLAG4 0x08 /* 00001000 */
   /* ... and so on */
   #define FLAG8 0x80 /* 10000000 */

15 int main(int argc, char *argv[])
   {
    char flags = 0; /* an 8 bit variable */

    /* set flags with a logical or */
20  flags = flags | FLAG1; /* set flag 1 */
    flags = flags | FLAG3; /* set flag 3

    /* check flags with a logical and.  If the flag is set (1)
     * then the logical and will return 1, causing the if
25   * condition to be true. */
    if (flags & FLAG1)
     printf("FLAG1 set!\n");

    /* this of course will be untrue. */
30  if (flags & FLAG8)
     printf("FLAG8 set!\n");

    /* check multiple flags by using a logical or
     * this will pass as FLAG1 is set */
35  if (flags & (FLAG1|FLAG4))
     printf("FLAG1 or FLAG4 set!\n");

    return 0;
   }
40
```

Types and Number Representation

C Standards

Although a slight divergence, it is important to understand a bit of history about the C language.

C is the *lingua franca* of the systems programming world. Every operating system and its associated system libraries in common use is written in C, and every system provides a C compiler. To stop the language diverging across each of these systems where each would be sure to make numerous incompatible changes, a strict standard has been written for the language.

Officially this standard is known as ISO/IEC 9899:1999(E), but is more commonly referred to by its shortened name *C99*. The standard is maintained by the International Standards Organisation (ISO) and the full standard is available for purchase online. Older standards versions such as C89 (the predecessor to C99 released in 1989) and ANSI C are no longer in common usage and are encompassed within the latest standard. The standard documentation is very technical, and details most every part of the language. For example it explains the syntax (in Backus Naur form), standard `#define` values and how operations should behave.

It is also important to note what the C standards does *not* define. Most importantly the standard needs to be appropriate for every architecture, both present and future. Consequently it takes care *not* to define areas that are architecture dependent. The "glue" between the C standard and the underlying architecture is the Application Binary Interface (or ABI) which we discuss below. In several places the standard will mention that a particular operation or construct has an unspecified or implementation dependent result. Obviously the programmer can not depend on these outcomes if they are to write portable code.

GNU C

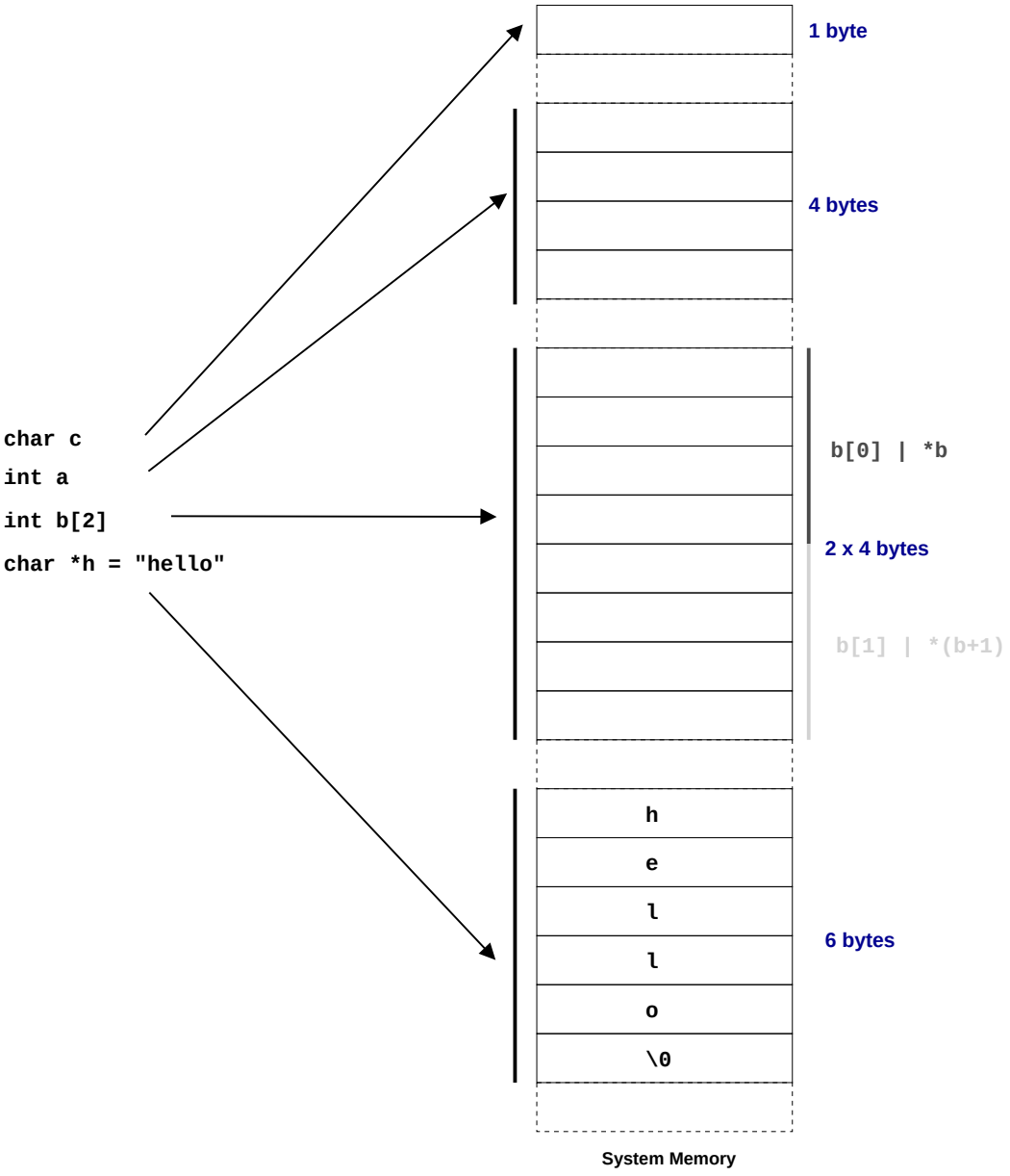
The GNU C Compiler, more commonly referred to as gcc, almost completely implements the C99 standard. However it also implements a range of extensions to the standard which programmers will often use to gain extra functionality, at the expense of portability to another compiler. These extensions are usually related to very low level code and are much more common in the system programming field; the most common extension being used in this area being inline assembly code. Programmers should read the gcc documentation and understand when they may be using features that diverge from the standard.

gcc can be directed to adhere strictly to the standard (the `-std=c99` flag for example) and warn or create an error when certain things are done that are not in the standard. This is obviously appropriate if you need to ensure that you can move your code easily to another compiler.

Types

As programmers, we are familiar with using variables to represent an area of memory to hold a value. In a *typed* language, such as C, every variable must be declared with a *type*. The type tells the compiler about what we expect to store in a variable; the compiler can then both allocate sufficient space for this usage and check that the programmer does not violate the rules of the type. In the example below, we see an example of the space allocated for some common types of variables.

Figure 2.2. Types



The C99 standard purposely only mentions the *smallest* possible size of each of the types defined for C. This is because across different processor architectures and operating systems the best size for types can be wildly different.

To be completely safe programmers need to never assume the size of any of their variables, however a functioning system obviously needs agreements on what sizes types are going to be used in the system. Each architecture and operating system conforms to an *Application Binary Interface* or *ABI*. The ABI for a system fills in the details between the C standard and the requirements of the underlying hardware and operating system. An ABI is written for a specific processor and operating system combination.

Table 2.13. Standard Integer Types and Sizes

Type	C99 minimum size (bits)	Common size (32 bit architecture)
`char`	8	8
`short`	16	16
`int`	16	32
`long`	32	32
`long long`	64	64
Pointers	Implementation dependent	32

Above we can see the only divergence from the standard is that `int` is commonly a 32 bit quantity, which is twice the strict minimum 16 bit size that the C99 requires.

Pointers are really just an address (i.e. their value is an address and thus "points" somewhere else in memory) therefore a pointer needs to be sufficient in size to be able to address any memory in the system.

64 bit

One area that causes confusion is the introduction of 64 bit computing. This means that the processor can handle addresses 64 bits in length (specifically the registers are 64 bits wide; a topic we discuss in Chapter 3, *Computer Architecture*).

This firstly means that all pointers are required to be a 64 bits wide so they can represent any possible address in the system. However, system implementers must then make decisions about the size of the other types. Two common models are widely used, as shown below.

Table 2.14. Standard Scalar Types and Sizes

Type	C99 minimum size (bits)	Common size (LP64)	Common size (Windows)
`char`	8	8	8
`short`	16	16	16
`int`	16	32	32

Type	C99 minimum size (bits)	Common size (LP64)	Common size (Windows)
`long`	32	64	32
`long long`	64	64	64
Pointers	Implementation dependent	64	64

You can see that in the LP64 (long-pointer 64) model `long` values are defined to be 64 bits wide. This is different to the 32 bit model we showed previously. The LP64 model is widely used on UNIX systems.

In the other model, `long` remains a 32 bit value. This maintains maximum compatibility with 32 code. This model is in use with 64 bit Windows.

There are good reasons why the size of `int` was not increased to 64 bits in either model. Consider that if the size of `int` is increased to 64 bits you leave programmers no way to obtain a 32 bit variable. The only possibly is redefining `shorts` to be a larger 32 bit type.

A 64 bit variable is so large that it is not generally required to represent many variables. For example, loops very rarely repeat more times than would fit in a 32 bit variable (4294967296 times!). Images usually are usually represented with 8 bits for each of a red, green and blue value and an extra 8 bits for extra (alpha channel) information; a total of 32 bits. Consequently for many cases, using a 64 bit variable will be wasting at least the top 32 bits (if not more). Not only this, but the size of an integer array has now doubled too. This means programs take up more system memory (and thus more cache; discussed in detail in Chapter 3, *Computer Architecture*) for no real improvement. For the same reason Windows elected to keep their long values as 32 bits; since much of the Windows API was originally written to use long variables on a 32 bit system and hence does not require the extra bits this saves considerable wasted space in the system without having to re-write all the API.

If we consider the proposed alternative where `short` was redefined to be a 32 bit variable; programmers working on a 64 bit system could use it for variables they know are bounded to smaller values. However, when moving back to a 32 bit system their same `short` variable would now be only 16 bits long, a value which is much more realistically overflowed (65536).

By making a programmer request larger variables when they know they will be needed strikes a balance with respect to portability concerns and wasting space in binaries.

Type qualifiers

The C standard also talks about some qualifiers for variable types. For example `const` means that a variable will never be modified from its original value and `volatile` suggests to the compiler that this value might change outside program execution flow so the compiler must be careful not to re-order access to it in any way.

`signed` and `unsigned` are probably the two most important qualifiers; and they say if a variable can take on a negative value or not. We examine this in more detail below.

Qualifiers are all intended to pass extra information about how the variable will be used to the compiler. This means two things; the compiler can check if you are violating your own rules (e.g. writing to a `const` value) and it can make optimisations based upon the extra knowledge (examined in later chapters).

Standard Types

C99 realises that all these rules, sizes and portability concerns can become very confusing very quickly. To help, it provides a series of special types which can specify the exact properties of a variable. These are defined in `<stdint.h>` and have the form `qtypes_t` where `q` is a qualifier, `type` is the base type, `s` is the width in bits and `_t` is an extension so you know you are using the C99 defined types.

So for example `uint8_t` is an unsigned integer exactly 8 bits wide. Many other types are defined; the complete list is detailed in C99 17.8 or (more cryptically) in the header file. [1]

It is up to the system implementing the C99 standard to provide these types for you by mapping them to appropriate sized types on the target system; on Linux these headers are provided by the system libraries.

Types in action

Below we see an example of how types place restrictions on what operations are valid for a variable, and how the compiler can use this information to warn when variables are used in an incorrect fashion. In this code, we firstly assign an integer value into a `char` variable. Since the `char` variable is smaller, we loose the correct value of the integer. Further down, we attempt to assign a pointer to a `char` to memory we designated as an`integer`. This operation can be done; but it is not safe. The first example is run on a 32-bit Pentium machine, and the correct value is returned. However, as shown in the second example, on a 64-bit Itanium machine a pointer is 64 bits (8 bytes) long, but an integer is only 4 bytes long. Clearly, 8 bytes can not fit into 4! We can attempt to "fool" the compiler by *casting* the value before assigning it; note that in this case we have shot ourselves in the foot by doing this cast and ignoring the compiler warning since the smaller variable can not hold all the information from the pointer and we end up with an invalid address.

Example 2.2. Example of warnings when types are not matched

```
 1
              $ cat types.c
  #include <stdio.h>
  #include <stdint.h>
 5
  int main(void)
  {
          char *c;
          int i;
10
```

[1] Note that C99 also has portability helpers for `printf`. The `PRI` macros in `<inttypes.h>` can be used as specifiers for types of specified sizes. Again see the standard or pull apart the headers for full information.

```
           i = c;
           i = (int)c;

           return 0;
15 }

   $ uname -m
   i686

20 $ gcc -Wall -o types types.c
   types.c: In function 'main':
   types.c:19: warning: assignment makes integer from pointer without a cast

   $ ./types
25 i is 52
   p is 0x80484e8
   p is 0x80484e8

   $ uname -m
30 ia64

   $ gcc -Wall  -o types types.c
   types.c: In function 'main':
   types.c:19: warning: assignment makes integer from pointer without a cast
35 types.c:21: warning: cast from pointer to integer of different size
   types.c:22: warning: cast to pointer from integer of different size

   $ ./types
   i is 52
40 p is 0x40000000000009e0
   p is 0x9e0
```

Number Representation

Negative Values

With our modern base 10 numeral system we indicate a negative number by placing a minus (`-`) sign in front of it. When using binary we need to use a different system to indicate negative numbers.

There is only one scheme in common use on modern hardware, but C99 defines three acceptable methods for negative value representation.

Sign Bit

The most straight forward method is to simply say that one bit of the number indicates either a negative or positive value depending on it being set or not.

This is analogous to mathematical approach of having a `+` and `-`. This is fairly logical, and some of the original computers did represent negative numbers in this way. But using binary numbers opens up some other possibilities which make the life of hardware designers easier.

However, notice that the value `0` now has two equivalent values; one with the sign bit set and one without. Sometimes these values are referred to as `+0` and `-0` respectively.

One's Complement

One's complement simply applies the *not* operation to the positive number to represent the negative number. So, for example the value -90 (-0x5A) is represented by `~01011010 = 10100101`[2]

With this scheme the biggest advantage is that to add a negative number to a positive number no special logic is required, except that any additional carry left over must be added back to the final value. Consider

Table 2.15. One's Complement Addition

Decimal	Binary	Op
-90	10100101	+
100	01100100	
---	--------	
10	[1]00001001	9
	00001010	10

If you add the bits one by one, you find you end up with a carry bit at the end (highlighted above). By adding this back to the original we end up with the correct value, 10

Again we still have the problem with two zeros being represented. Again no modern computer uses one's complement, mostly because there is a better scheme.

Two's Complement

Two's complement is just like one's complement, except the negative representation has *one* added to it and we discard any left over carry bit. So to continue with the example from before, `-90` would be `~01011010+1=10100101+1 = 10100110`.

This means there is a slightly odd symmetry in the numbers that can be represented; for example with an 8 bit integer we have `2^8 = 256` possible values; with our sign bit representation we could represent -127 thru 127 but with two's complement we can represent -127 thru 128. This is because we have removed the problem of having two zeros; consider that "negative zero" is `(~00000000 +1)=(11111111+1)=00000000` (note discarded carry bit).

Table 2.16. Two's Complement Addition

Decimal	Binary	Op
-90	10100110	+
100	01100100	
---	--------	
10	00001010	

[2]The ~ operator is the C language operator to apply `NOT` to the value. It is also occasionally called the one's complement operator, for obvious reasons now!

You can see that by implementing two's complement hardware designers need only provide logic for addition circuits; subtraction can be done by two's complement negating the value to be subtracted and then adding the new value.

Similarly you could implement multiplication with repeated addition and division with repeated subtraction. Consequently two's complement can reduce all simple mathematical operations down to addition!

All modern computers use two's complement representation.

Sign-extension

Because of two's complement format, when increasing the size of signed value, it is important that the additional bits be *sign-extended*; that is, copied from the top-bit of the existing value.

For example, the value of an 32-bit `int -10` would be represented in two's complement binary as `11111111111111111111111111110110`. If one were to cast this to a 64-bit `long long int`, we would need to ensure that the additional 32-bits were set to `1` to maintain the same sign as the original.

Thanks to two's complement, it is sufficient to take the top bit of the exiting value and replace all the added bits with this value. This processes is referred to as *sign-extension* and is usually handled by the compiler in situations as defined by the language standard, with the processor generally providing special instructions to take a value an sign-extended it to some larger value.

Floating Point

So far we have only discussed integer or whole numbers; the class of numbers that can represent decimal values is called *floating point*.

To create a decimal number, we require some way to represent the concept of the decimal place in binary. The most common scheme for this is known as the *IEEE-754 floating point standard* because the standard is published by the Institute of Electric and Electronics Engineers. The scheme is conceptually quite simple and is somewhat analogous to "scientific notation".

In scientific notation the value `123.45` might commonly be represented as 1.2345×10^2. We call `1.2345` the *mantissa* or *significand*, `10` is the *radix* and `2` is the *exponent*.

In the IEEE floating point model, we break up the available bits to represent the sign, mantissa and exponent of a decimal number. A decimal number is represented by $\text{sign} \times \text{significand} \times 2^{\text{exponent}}$.

The sign bit equates to either `1` or `-1`. Since we are working in binary, we always have the implied radix of `2`.

There are differing widths for a floating point value -- we examine below at only a 32 bit value. More bits allows greater precision.

Table 2.17. IEEE Floating Point

Sign	Exponent	Significand/Mantissa
S	EEEEEEEE	MMMMMMMMMMMMMMMMMMMMMMM

The other important factor is *bias* of the exponent. The exponent needs to be able to represent both positive and negative values, thus an implied value of `127` is subtracted from the exponent. For example, an exponent of `0` has an exponent field of `127`, `128` would represent `1` and `126` would represent `-1`.

Each bit of the significand adds a little more precision to the values we can represent. Consider the scientific notation representation of the value `198765`. We could write this as $1.98765x10^6$, which corresponds to a representation below

Table 2.18. Scientific Notation for 1.98765x10^6

10^0	.	10^{-1}	10^{-2}	10^{-3}	10^{-4}	10^{-5}
1	.	9	8	7	6	5

Each additional digit allows a greater range of decimal values we can represent. In base 10, each digit after the decimal place increases the precision of our number by 10 times. For example, we can represent `0.0` through `0.9` (10 values) with one digit of decimal place, `0.00` through `0.99` (100 values) with two digits, and so on. In binary, rather than each additional digit giving us 10 times the precision, we only get two times the precision, as illustrated in the table below. This means that our binary representation does not always map in a straight-forward manner to a decimal representation.

Table 2.19. Significands in binary

2^0	.	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}
1	.	1/2	1/4	1/8	1/16	1/32
1	.	0.5	0.25	0.125	0.625	0.03125

With only one bit of precision, our fractional precision is not very big; we can only say that the fraction is either `0` or `0.5`. If we add another bit of precision, we can now say that the decimal value is one of either `0,0.25,0.5,0.75`. With another bit of precision we can now represent the values `0,0.125,0.25,0.375,0.5,0.625,0.75,0.875`.

Increasing the number of bits therefore allows us greater and greater precision. However, since the range of possible numbers is infinite we will never have enough bits to represent *any* possible value.

For example, if we only have two bits of precision and need to represent the value `0.3` we can only say that it is closest to `0.25`; obviously this is insufficient for most any application. With 22 bits of significand we have a much finer resolution, but it is still not enough for most applications. A `double` value increases the number of significand bits to 52 (it also increases the range of exponent values too). Some hardware has an 84-bit float, with a full 64 bits of significand. 64 bits allows a tremendous precision and should be suitable for all but the most demanding of applications (XXX is this sufficient to represent a length to less than the size of an atom?)

Example 2.3. Floats versus Doubles

```
 1
               $ cat float.c
   #include <stdio.h>

 5 int main(void)
   {
           float a = 0.45;
           float b = 8.0;

10         double ad = 0.45;
           double bd = 8.0;

           printf("float+float, 6dp    : %f\n", a+b);
           printf("double+double, 6dp  : %f\n", ad+bd);
15         printf("float+float, 20dp   : %10.20f\n", a+b);
           printf("dobule+double, 20dp : %10.20f\n", ad+bd);

           return 0;
   }
20
   $ gcc -o float float.c

   $ ./float
   float+float, 6dp    : 8.450000
25 double+double, 6dp  : 8.450000
   float+float, 20dp   : 8.44999988807907104492
   dobule+double, 20dp : 8.44999999999999928946

   $ python
30 Python 2.4.4 (#2, Oct 20 2006, 00:23:25)
   [GCC 4.1.2 20061015 (prerelease) (Debian 4.1.1-16.1)] on linux2
   Type "help", "copyright", "credits" or "license" for more information.
   >>> 8.0 + 0.45
   8.4499999999999993
35
```

A practical example is illustrated above. Notice that for the default 6 decimal places of precision given by `printf` both answers are the same, since they are rounded up correctly. However, when asked to give the results to a larger precision, in this case 20 decimal places, we can see the results start to diverge. The code using `doubles` has a more accurate result, but it is still not *exactly* correct. We can also see that programmers not explicitly dealing with `float` values still have problems with precision of variables!

Normalised Values

In scientific notation, we can represent a value in many different ways. For example, $10023x10^{0} = 1002.3x10^{1} = 100.23x10^{2}$. We thus define the *normalised* version as the one where `1/radix <= significand < 1`. In binary this ensures that the leftmost bit of the significand is *always one*. Knowing this, we can gain an extra bit of precision by having the standard say that the leftmost bit being one is implied.

Table 2.20. Example of normalising 0.375

2^{0}	.	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	Exponent	Calculation
0	.	0	1	1	0	0	2^{0}	(0.25+0.125) × 1 = 0.375

2^0	.	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	Exponen	Calculation
0	.	1	1	0	0	0	$2^{\wedge -1}$	(0.5+0.25)×.5=0.375
1	.	1	0	0	0	0	$2^{\wedge -2}$	(1+0.5)×0.25=0.375

As you can see above, we can make the value normalised by moving the bits upwards as long as we compensate by increasing the exponent.

Normalisation Tricks

A common problem programmers face is finding the first set bit in a bitfield. Consider the bitfield `0100`; from the right the first set bit would be bit 2 (starting from zero, as is conventional).

The standard way to find this value is to shift right, check if the uppermost bit is a 1 and either terminate or repeat. This is a slow process; if the bitfield is 64 bits long and only the very last bit is set, you must go through all the preceding 63 bits!

However, if this bitfield value were the signficand of a floating point number and we were to normalise it, the value of the exponent would tell us how many times it was shifted. The process of normalising a number is generally built into the floating point hardware unit on the processor, so operates very fast; usually much faster than the repeated shift and check operations.

The example program below illustrates two methods of finding the first set bit on an Itanium processor. The Itanium, like most server processors, has support for an 80-bit *extended* floating point type, with a 64-bit significand. This means a `unsigned long` neatly fits into the significand of a `long double`. When the value is loaded it is normalised, and and thus by reading the exponent value (minus the 16 bit bias) we can see how far it was shifted.

Example 2.4. Program to find first set bit

```
 1
                  #include <stdio.h>

  int main(void)
 5 {
    //  in binary = 1000 0000 0000 0000
    //  bit num     5432 1098 7654 3210
    int i = 0x8000;
    int count = 0;
10  while ( !(i & 0x1) ) {
     count ++;
     i = i >> 1;
    }
    printf("First non-zero (slow) is %d\n", count);
15
    // this value is normalised when it is loaded
    long double d = 0x8000UL;
    long exp;

20  // Itanium "get floating point exponent" instruction
    asm ("getf.exp %0=%1" : "=r"(exp) : "f"(d));

    // note exponent include bias
    printf("The first non-zero (fast) is %d\n", exp - 65535);
25
   }
```

Bringing it together

In the example code below we extract the components of a floating point number and print out the value it represents. This will only work for a 32 bit floating point value in the IEEE format; however this is common for most architectures with the `float` type.

Example 2.5. Examining Floats

```
 1
                #include <stdio.h>
   #include <string.h>
   #include <stdlib.h>
 5
   /* return 2^n */
   int two_to_pos(int n)
   {
    if (n == 0)
10   return 1;
    return 2 * two_to_pos(n - 1);
   }

   double two_to_neg(int n)
15 {
    if (n == 0)
     return 1;
    return 1.0 / (two_to_pos(abs(n)));
   }
20
   double two_to(int n)
   {
    if (n >= 0)
     return two_to_pos(n);
25  if (n < 0)
     return two_to_neg(n);
    return 0;
   }

30 /* Go through some memory "m" which is the 24 bit significand of a
      floating point number.  We work "backwards" from the bits
      furthest on the right, for no particular reason. */
   double calc_float(int m, int bit)
   {
35  /* 23 bits; this terminates recursion */
    if (bit > 23)
     return 0;

    /* if the bit is set, it represents the value 1/2^bit */
40  if ((m >> bit) & 1)
     return 1.0L/two_to(23 - bit) + calc_float(m, bit + 1);

    /* otherwise go to the next bit */
    return calc_float(m, bit + 1);
45 }

   int main(int argc, char *argv[])
   {
    float f;
50  int m,i,sign,exponent,significand;

    if (argc != 2)
    {
     printf("usage: float 123.456\n");
55   exit(1);
    }
```

```
    if (sscanf(argv[1], "%f", &f) != 1)
    {
 60  printf("invalid input\n");
     exit(1);
    }

    /* We need to "fool" the compiler, as if we start to use casts
 65    (e.g. (int)f) it will actually do a conversion for us.  We
       want access to the raw bits, so we just copy it into a same
       sized variable. */
    memcpy(&m, &f, 4);

 70 /* The sign bit is the first bit */
    sign = (m >> 31) & 0x1;

    /* Exponent is 8 bits following the sign bit */
    exponent = ((m >> 23) & 0xFF) - 127;
 75
    /* Significand fills out the float, the first bit is implied
       to be 1, hence the 24 bit OR value below. */
    significand = (m & 0x7FFFFF) | 0x800000;

 80 /* print out a power representation */
    printf("%f = %d * (", f, sign ? -1 : 1);
    for(i = 23 ; i >= 0 ; i--)
    {
     if ((significand >> i) & 1)
 85   printf("%s1/2^%d", (i == 23) ? "" : " + ",
            23-i);
    }
    printf(") * 2^%d\n", exponent);

 90 /* print out a fractional representation */
    printf("%f = %d * (", f, sign ? -1 : 1);
    for(i = 23 ; i >= 0 ; i--)
    {
     if ((significand >> i) & 1)
 95   printf("%s1/%d", (i == 23) ? "" : " + ",
            (int)two_to(23-i));
    }
    printf(") * 2^%d\n", exponent);

100 /* convert this into decimal and print it out */
    printf("%f = %d * %.12g * %f\n",
           f,
           (sign ? -1 : 1),
           calc_float(significand, 0),
105        two_to(exponent));

    /* do the math this time */
    printf("%f = %.12g\n",
           f,
110        (sign ? -1 : 1) *
           calc_float(significand, 0) *
           two_to(exponent)
     );

115 return 0;
   }
```

Sample output of the value `8.45`, which we previously examined, is shown below.

Example 2.6. Analysis of `8.45`

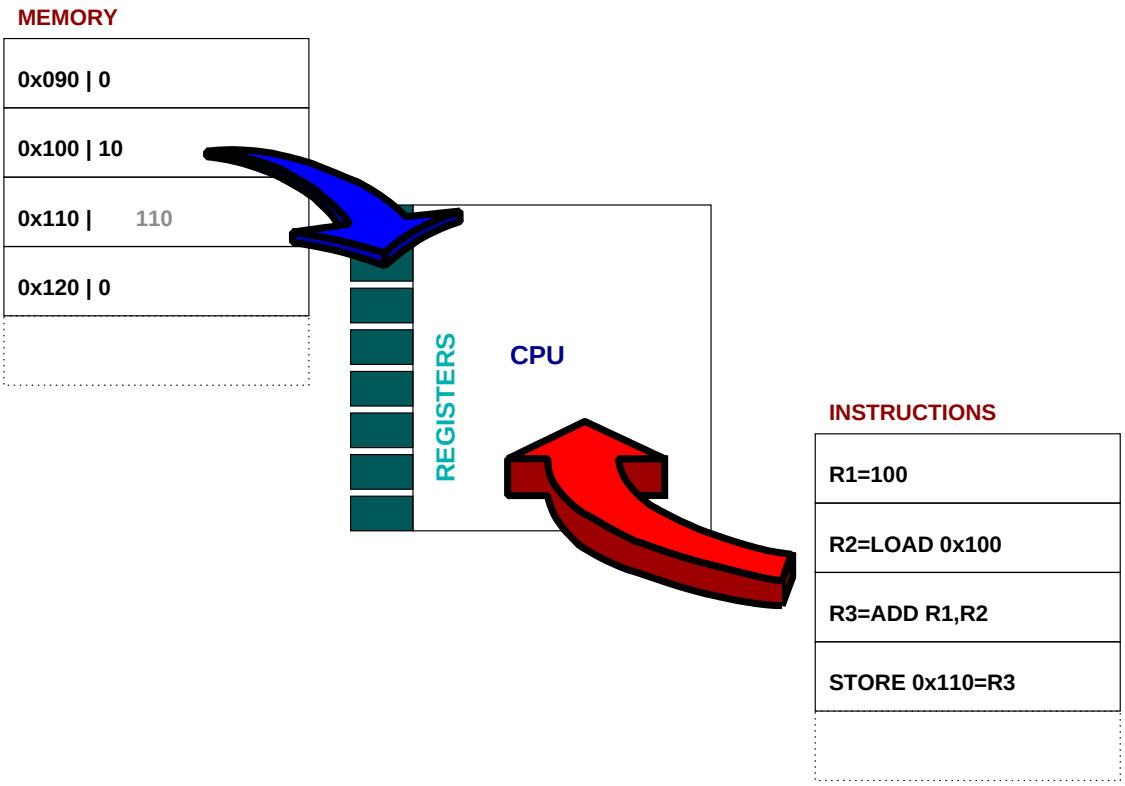
```
$ ./float 8.45
8.450000 = 1 * (1/2^0 + 1/2^5 + 1/2^6 + 1/2^7 + 1/2^10 + 1/2^11 + 1/2^14 + 1/2^15 + 1/2^18 + 1/2^19 + 1/2^2
8.450000 = 1 * (1/1 + 1/32 + 1/64 + 1/128 + 1/1024 + 1/2048 + 1/16384 + 1/32768 + 1/262144 + 1/524288 + 1/4
8.450000 = 1 * 1.05624997616 * 8.000000
8.450000 = 8.44999980927
```

From this example, we get some idea of how the inaccuracies creep into our floating point numbers.

Chapter 3. Computer Architecture

The CPU

Figure 3.1. The CPU



The CPU performs instructions on values held in registers. This example shows firstly setting the value of R1 to 100, loading the value from memory location 0x100 into R2, adding the two values together and placing the result in R3 and finally storing the new value (110) to R4 (for further use).

To greatly simplify, a computer consists of a central processing unit (CPU) attached to memory. The figure above illustrates the general principle behind all computer operations.

The CPU executes instructions read from memory. There are two categories of instructions

1. Those that *load* values from memory into registers and *store* values from registers to memory.

2. Those that operate on values stored in registers. For example adding, subtracting multiplying or dividing the values in two registers, performing bitwise operations (and, or, xor, etc) or performing other mathematical operations (square root, sin, cos, tan, etc).

So in the example we are simply adding 100 to a value stored in memory, and storing this new result back into memory.

Branching

Apart from loading or storing, the other important operation of a CPU is *branching*. Internally, the CPU keeps a record of the next instruction to be executed in the *instruction pointer*. Usually, the instruction pointer is incremented to point to the next instruction sequentially; the branch instruction will usually check if a specific register is zero or if a flag is set and, if so, will modify the pointer to a different address. Thus the next instruction to execute will be from a different part of program; this is how loops and decision statements work.

For example, a statement like `if (x==0)` might be implemented by finding the `or` of two registers, one holding `x` and the other zero; if the result is zero the comparison is true (i.e. all bits of `x` were zero) and the body of the statement should be taken, otherwise branch past the body code.

Cycles

We are all familiar with the speed of the computer, given in Megahertz or Gigahertz (millions or thousands of millions cycles per second). This is called the *clock speed* since it is the speed that an internal clock within the computer pulses.

The pulses are used within the processor to keep it internally synchronised. On each tick or pulse another operation can be started; think of the clock like the person beating the drum to keep the rower's oars in sync.

Fetch, Decode, Execute, Store

Executing a single instruction consists of a particular cycle of events; fetching, decoding, executing and storing.

For example, to do the `add` instruction above the CPU must

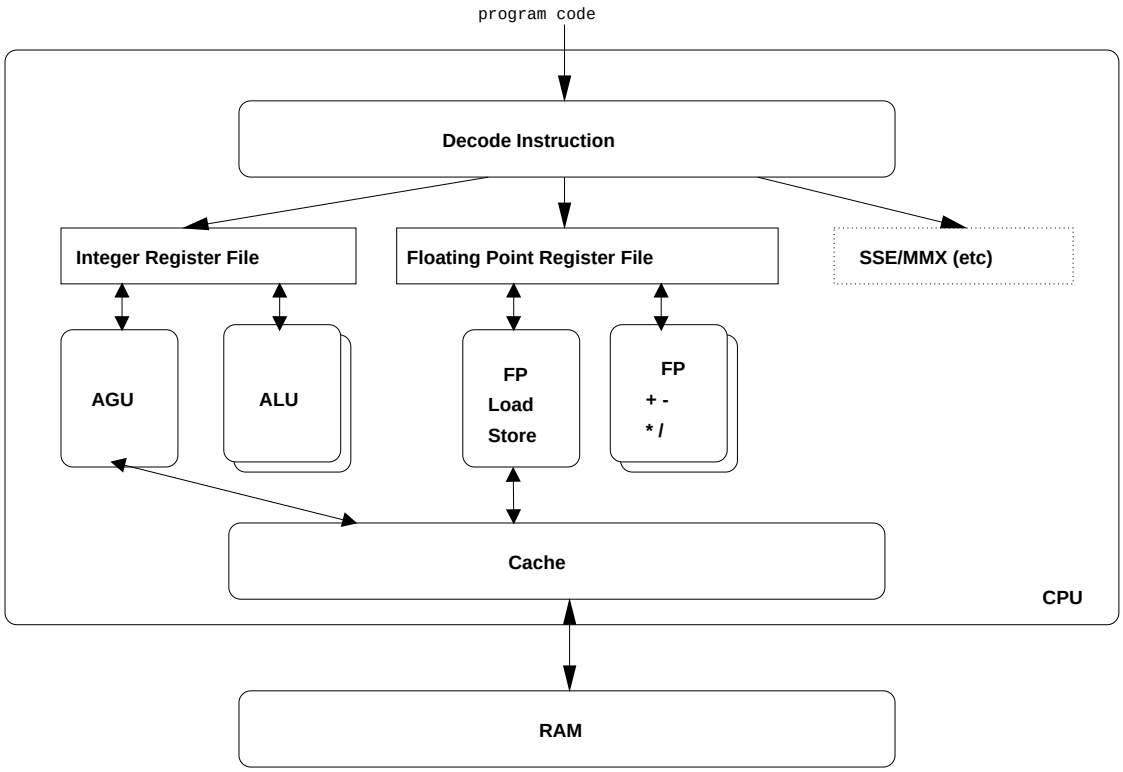
1. Fetch : get the instruction from memory into the processor.
2. Decode : internally decode what it has to do (in this case add).
3. Execute : take the values from the registers, actually add them together
4. Store : store the result back into another register. You might also see the term *retiring* the instruction.

Looking inside a CPU

Internally the CPU has many different sub components that perform each of the above steps, and generally they can all happen independently of each other. This is analogous

to a physical production line, where there are many stations where each step has a particular task to perform. Once done it can pass the results to the next station and take a new input to work on.

Figure 3.2. Inside the CPU



The CPU is made up of many different sub-components, each doing a dedicated task.

Figure 3.2, “Inside the CPU” shows a very simple block diagram illustrating some of the main parts of a modern CPU.

You can see the instructions come in and are decoded by the processor. The CPU has two main types of registers, those for *integer* calculations and those for *floating point* calculations. Floating point is a way of representing numbers with a decimal place in binary form, and is handled differently within the CPU. *MMX* (multimedia extension) and *SSE* (Streaming Single Instruction Multiple Data) or *Altivec* registers are similar to floating point registers.

A *register file* is the collective name for the registers inside the CPU. Below that we have the parts of the CPU which really do all the work.

We said that processors are either loading or storing a value into a register or from a register into memory, or doing some operation on values in registers.

The *Arithmetic Logic Unit* (ALU) is the heart of the CPU operation. It takes values in registers and performs any of the multitude of operations the CPU is capable of. All modern processors have a number of ALUs so each can be working independently. In

fact, processors such as the Pentium have both *fast* and *slow* ALUs; the fast ones are smaller (so you can fit more on the CPU) but can do only the most common operations, slow ALUs can do all operations but are bigger.

The *Address Generation Unit* (AGU) handles talking to cache and main memory to get values into the registers for the ALU to operate on and get values out of registers back into main memory.

Floating point registers have the same concepts, but use slightly different terminology for their components.

Pipelining

As we can see above, whilst the ALU is adding registers together is completely separate to the AGU writing values back to memory, so there is no reason why the CPU can not be doing both at once. We also have multiple ALUs in the system, each which can be working on separate instructions. Finally the CPU could be doing some floating point operations with its floating point logic whilst integer instructions are in flight too. This process is called *pipelining*[1], and a processor that can do this is referred to as a *superscalar architecture*. All modern processors are superscalar.

Another analogy might be to think of the pipeline like a hose that is being filled with marbles, except our marbles are instructions for the CPU. Ideally you will be putting your marbles in one end, one after the other (one per clock pulse), filling up the pipe. Once full, for each marble (instruction) you push in all the others will move to the next position and one will fall out the end (the result).

Branch instruction play havoc with this model however, since they may or may not cause execution to start from a different place. If you are pipelining, you will have to basically guess which way the branch will go, so you know which instructions to bring into the pipeline. If the CPU has predicted correctly, everything goes fine![2] Conversely, if the processor has predicted incorrectly it has wasted a lot of time and has to clear the pipeline and start again.

This process is usually referred to as a *pipeline flush* and is analogous to having to stop and empty out all your marbles from your hose!

Branch Prediction

pipeline flush, predict taken, predict not taken, branch delay slots

Reordering

This bit is crap

[1] In fact, any modern processor has many more than four stages it can pipeline, above we have only shown a very simplified view. The more stages that can be executed at the same time, the deeper the pipeline.

[2] Processors such as the Pentium use a *trace cache* to keep a track of which way branches are going. Much of the time it can predict which way a branch will go by remembering its previous result. For example, in a loop that happens 100 times, if you remember the last result of the branch you will be right 99 times, since only the last time will you actually continue with the program.

In fact, if the CPU is the hose, it is free to reorder the marbles within the hose, as long as they pop out the end in the same order you put them in. We call this *program order* since this is the order that instructions are given in the computer program.

Figure 3.3. Reorder buffer example

```
1: r3 = r1 * r2
2: r4 = r2 + r3
3: r7 = r5 * r6
4: r8 = r1 + r7
```

Consider an instruction stream such as that shown in Figure 3.3, “Reorder buffer example” Instruction 2 needs to wait for instruction 1 to complete fully before it can start. This means that the pipeline has to *stall* as it waits for the value to be calculated. Similarly instructions 3 and 4 have a dependency on *r7*. However, instructions 2 and 3 have no *dependency* on each other at all; this means they operate on completely separate registers. If we swap instructions 2 and 3 we can get a much better ordering for the pipeline since the processor can be doing useful work rather than waiting for the pipeline to complete to get the result of a previous instruction.

However, when writing very low level code some instructions may require some security about how operations are ordered. We call this requirement *memory semantics*. If you require *acquire* semantics this means that for this instruction you must ensure that the results of all previous instructions have been completed. If you require *release* semantics you are saying that all instructions after this one must see the current result. Another even stricter semantic is a *memory barrier* or *memory fence* which requires that operations have been committed to memory before continuing.

On some architectures these semantics are guaranteed for you by the processor, whilst on others you must specify them explicitly. Most programmers do not need to worry directly about them, although you may see the terms.

CISC v RISC

A common way to divide computer architectures is into *Complex Instruction Set Computer* (CISC) and *Reduced Instruction Set Computer* (RISC).

Note in the first example, we have explicitly loaded values into registers, performed an addition and stored the result value held in another register back to memory. This is an example of a RISC approach to computing -- only performing operations on values in registers and explicitly loading and storing values to and from memory.

A CISC approach may only a single instruction taking values from memory, performing the addition internally and writing the result back. This means the instruction may take many cycles, but ultimately both approaches achieve the same goal.

All modern architectures would be considered RISC architectures[3].

There are a number of reasons for this

[3]Even the most common architecture, the Intel Pentium, whilst having an instruction set that is categorised as CISC, internally breaks down instructions to RISC style sub-instructions inside the chip before executing.

- Whilst RISC makes assembly programming becomes more complex, since virtually all programmers use high level languages and leave the hard work of producing assembly code to the compiler, so the other advantages outweigh this disadvantage.

- Because the instructions in a RISC processor are much more simple, there is more space inside the chip for registers. As we know from the memory hierarchy, registers are the fastest type of memory and ultimately all instructions must be performed on values held in registers, so all other things being equal more registers leads to higher performance.

- Since all instructions execute in the same time, pipelining is possible. We know pipelining requires streams of instructions being constantly fed into the processor, so if some instructions take a very long time and others do not, the pipeline becomes far to complex to be effective.

EPIC

The Itanium processor, which is used in many example through this book, is an example of a modified architecture called Explicitly Parallel Instruction Computing.

We have discussed how superscaler processors have pipelines that have many instructions in flight at the same time in different parts of the processor. Obviously for this to work as well as possible instructions should be given the processor in an order that can make best use of the available elements of the CPU.

Traditionally organising the incoming instruction stream has been the job of the hardware. Instructions are issued by the program in a sequential manner; the processor must look ahead and try to make decisions about how to organise the incoming instructions.

The theory behind EPIC is that there is more information available at higher levels which can make these decisions better than the processor. Analysing a stream of assembly language instructions, as current processors do, looses a lot of information that the programmer may have provided in the original source code. Think of it as the difference between studying a Shakespeare play and reading the Cliff's Notes version of the same. Both give you the same result, but the original has all sorts of extra information that sets the scene and gives you insight into the characters.

Thus the logic of ordering instructions can be moved from the processor to the compiler. This means that compiler writers need to be smarter to try and find the best ordering of code for the processor. The processor is also significantly simplified, since a lot of its work has been moved to the compiler.[4]

[4]Another term often used around EPIC is Very Long Instruction World (VLIW), which is where each instruction to the processor is extended to tell the processor about where it should execute the instruction in it's internal units. The problem with this approach is that code is then completely dependent on the model of processor is has been compiled for. Companies are always making revisions to hardware, and making customers recompile their application every single time, and maintain a range of different binaries was impractical.

EPIC solves this in the usual computer science manner by adding a layer of abstraction. Rather than explicitly specifying the exact part of the processor the instructions should execute on, EPIC creates a simplified view with a few core units like memory, integer and floating point.

Memory

Memory Hierarchy

The CPU can only directly fetch instructions and data from cache memory, located directly on the processor chip. Cache memory must be loaded in from the main system memory (the Random Access Memory, or RAM). RAM however, only retains it's contents when the power is on, so needs to be stored on more permanent storage.

We call these layers of memory the *memory hierarchy*

Table 3.1. Memory Hierarchy

Speed	Memory	Description
Fastest	Cache	Cache memory is memory actually embedded inside the CPU. Cache memory is very fast, typically taking only once cycle to access, but since it is embedded directly into the CPU there is a limit to how big it can be. In fact, there are several sub-levels of cache memory (termed L1, L2, L3) all with slightly increasing speeds.
	RAM	All instructions and storage addresses for the processor must come from RAM. Although RAM is very fast, there is still some significant time taken for the CPU to access it (this is termed *latency*). RAM is stored in separate, dedicated chips attached to the motherboard, meaning it is much larger than cache memory.
Slowest	Disk	We are all familiar with software arriving on a floppy disk or CDROM, and saving our files to the hard disk. We are also familiar with the long time a program can take to load from the hard disk -- having physical mechanisms such as spinning disks and

Speed	Memory	Description
		moving heads means disks are the slowest form of storage. But they are also by far the largest form of storage.

The important point to know about the memory hierarchy is the trade offs between speed an size -- the faster the memory the smaller it is. Of course, if you can find a way to change this equation, you'll end up a billionaire!

The reason caches are effective is because computer code generally exhibits two forms of locality

1. *Spatial* locality suggests that data within blocks is likely to be accessed together.

2. *Temporal* locality suggests that data that was used recently will likely be used again shortly.

This means that benefits are gained by implementing as much quickly accessible memory (temporal) storing small blocks of relevant information (spatial) as practically possible.

Cache in depth

Cache is one of the most important elements of the CPU architecture. To write efficient code developers need to have an understanding of how the cache in their systems works.

The cache is a very fast copy of the slower main system memory. Cache is much smaller than main memories because it is included inside the processor chip alongside the registers and processor logic. This is prime real estate in computing terms, and there are both economic and physical limits to it's maximum size. As manufacturers find more and more ways to cram more and more transistors onto a chip cache sizes grow considerably, but even the largest caches are tens of megabytes, rather than the gigabytes of main memory or terabytes of hard disk otherwise common.

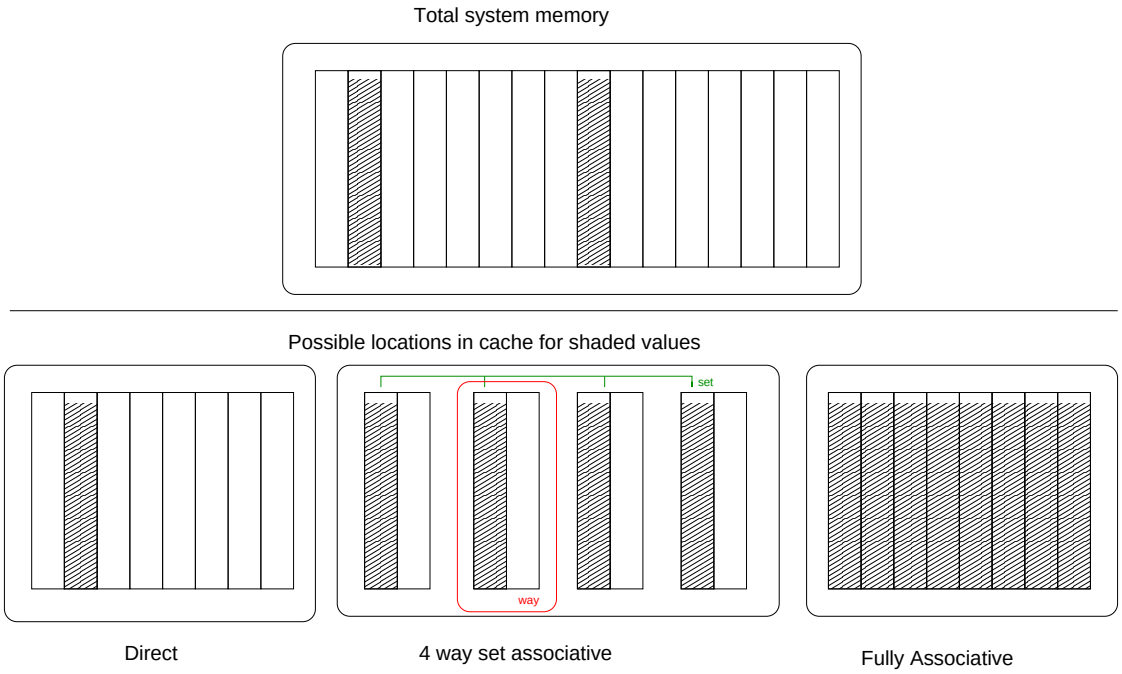
The cache is made up of small chunks of mirrored main memory. The size of these chunks is called the *line size*, and is typically something like 32 or 64 bytes. When talking about cache, it is very common to talk about the line size, or a cache line, which refers to one chunk of mirrored main memory. The cache can only load and store memory in sizes a multiple of a cache line.

Caches have their own hierarchy, commonly termed L1, L2 and L3. L1 cache is the fastest and smallest; L2 is bigger and slower, and L3 more so.

L1 caches are generally further split into instruction caches and data, known as the "Harvard Architecture" after the relay based Harvard Mark-1 computer which introduced

it. Split caches help to reduce pipeline bottlenecks as earlier pipeline stages tend to reference the instruction cache and later stages the data cache. Apart from reducing contention for a shared resource, providing separate caches for instructions also allows for alternate implementations which may take advantage of the nature of instruction streaming; they are read-only so do not need expensive on-chip features such as multi-porting, nor need to handle handle sub-block reads because the instruction stream generally uses more regular sized accesses.

Figure 3.4. Cache Associativity



A given cache line may find a valid home in one of the shaded entries.

During normal operation the processor is constantly asking the cache to check if a particular address is stored in the cache, so the cache needs some way to very quickly find if it has a valid line present or not. If a given address can be cached anywhere within the cache, every cache line needs to be searched every time a reference is made to determine a hit or a miss. To keep searching fast this is done in parallel in the cache hardware, but searching every entry is generally far too expensive to implement for a reasonable sized cache. Thus the cache can be made simpler by enforcing limits on where a particular address must live. This is a trade-off; the cache is obviously much, much smaller than the system memory, so some addresses must *alias* others. If two addresses which alias each other are being constantly updated they are said to *fight* over the cache line. Thus we can categorise caches into three general types, illustrated in Figure 3.4, “Cache Associativity”.

- *Direct mapped* caches will allow a cache line to exist only in a singe entry in the cache. This is the simplest to implement in hardware, but as illustrated in Figure 3.4, “Cache Associativity” there is no potential to avoid aliasing because the two shaded addresses must share the same cache line.

- *Fully Associative* caches will allow a cache line to exist in any entry of the cache. This avoids the problem with aliasing, since any entry is available for use. But it is very expensive to implement in hardware because every possible location must be looked up simultaneously to determine if a value is in the cache.

- *Set Associative* caches are a hybrid of direct and fully associative caches, and allow a particular cache value to exist in some subset of the lines within the cache. The cache is divided into even compartments called *ways*, and a particular address could be located in any way. Thus an *n*-way set associative cache will allow a cache line to exist in any entry of a set sized total blocks mod n — Figure 3.4, “Cache Associativity” shows a sample 8-element, 4-way set associative cache; in this case the two addresses have four possible locations, meaning only half the cache must be searched upon lookup. The more ways, the more possible locations and the less aliasing, leading to overall better performance.

Once the cache is full the processor needs to get rid of a line to make room for a new line. There are many algorithms by which the processor can choose which line to evict; for example *least recently used* (LRU) is an algorithm where the oldest unused line is discarded to make room for the new line.

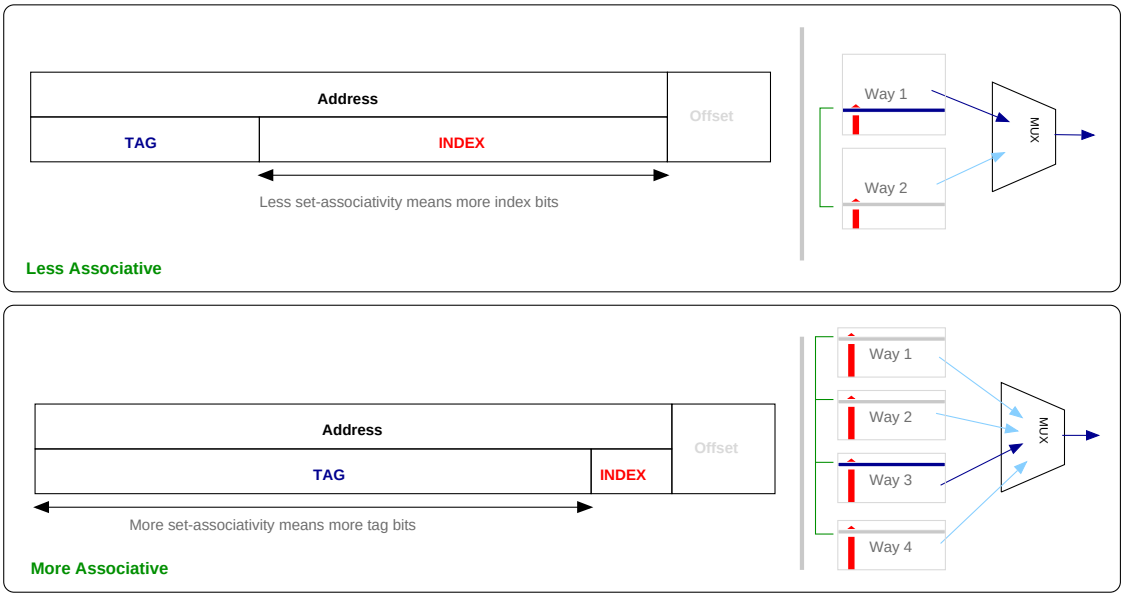
When data is only read from the cache there is no need to ensure consistency with main memory. However, when the processor starts writing to cache lines it needs to make some decisions about how to update the underlying main memory. A *write-through* cache will write the changes directly into the main system memory as the processor updates the cache. This is slower since the process of writing to the main memory is, as we have seen, slower. Alternatively a *write-back* cache delays writing the changes to RAM until absolutely necessary. The obvious advantage is that less main memory access is required when cache entries are written. Cache lines that have been written but not committed to memory are referred to as *dirty*. The disadvantage is that when a cache entry is evicted, it may require two memory accesses (one to write dirty data main memory, and another to load the new data).

If an entry exists in both a higher-level and lower-level cache at the same time, we say the higher-level cache is *inclusive*. Alternatively, if the higher-level cache having a line removes the possibility of a lower level cache having that line, we say it is *exclusive*. This choice is discussed further in the section called “Cache exclusivity in SMP systems”.

Cache Addressing

So far we have not discussed how a cache decides if a given address resides in the cache or not. Clearly, caches must keep a directory of what data currently resides in the cache lines. The cache directory and data may co-located on the processor, but may also be separate — such as in the case of the POWER5 processor which has an on-core L3 directory, but actually accessing the data requires traversing the L3 bus to access off-core memory. An arrangement like this can facilitate quicker hit/miss processing without the other costs of keeping the entire cache on-core.

Figure 3.5. Cache tags



Tags need to be checked in parallel to keep latency times low; more tag bits (i.e. less set associativity) requires more complex hardware to achieve this. Alternatively more set associativity means less tags, but the processor now needs hardware to multiplex the output of the many sets, which can also add latency.

To quickly decide if an address lies within the cache it is separated into three parts; the *tag* and the *index* and the *offset*.

The offset bits depend on the line size of the cache. For example, a 32-byte line size would use the last 5-bits (i.e. 2^5) of the address as the offset into the line.

The *index* is the particular cache line that an entry may reside in. As an example, let us consider a cache with 256 entries. If this is a direct-mapped cache, we know the data may reside in only one possible line, so the next 8-bits (2^8) after the offset describe the line to check - between 0 and 255.

Now, consider the same 256 element cache, but divided into two ways. This means there are two groups of 128 lines, and the given address may reside in either of these groups. Consequently only 7-bits are required as an index to offset into the 128-entry ways. For a given cache size, as we increase the number of ways, we decrease the number of bits required as an index since each way gets smaller.

The cache directory still needs to check if the particular address stored in the cache is the one it is interested in. Thus the remaining bits of the address are the *tag* bits which the cache directory checks against the incoming address tag bits to determine if there is a cache hit or not. This relationship is illustrated in Figure 3.5, “Cache tags”.

When there are multiple ways, this check must happen in parallel within each way, which then passes its result into a multiplexor which outputs a final *hit* or *miss* result. As describe above, the more associative a cache is, the less bits are required for index

and the more as tag bits — to the extreme of a fully-associative cache where no bits are used as index bits. The parallel matching of tags bits is the expensive component of cache design and generally the limiting factor on how many lines (i.e, how big) a cache may grow.

Peripherals and buses

Peripherals are any of the many external devices that connect to your computer. Obviously, the processor must have some way of talking to the peripherals to make them useful.

The communication channel between the processor and the peripherals is called a *bus*.

Peripheral Bus concepts

A device requires both input and output to be useful. There are a number of common concepts required for useful communication with peripherals.

Interrupts

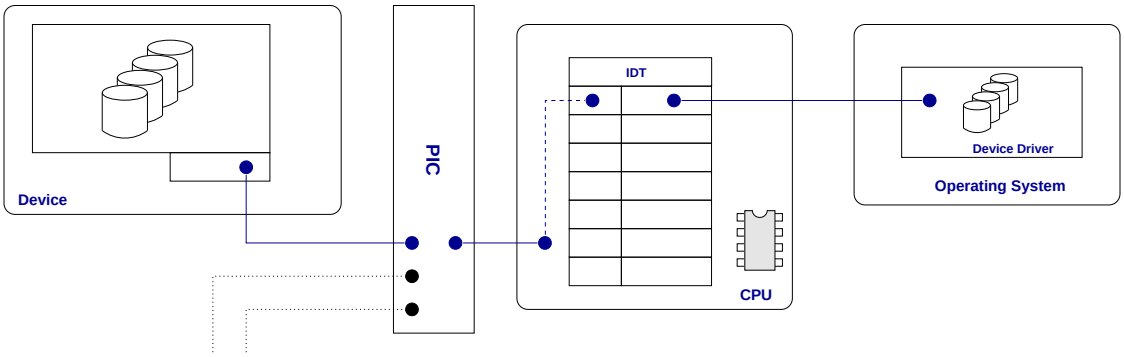
An interrupt allows the device to literally interrupt the processor to flag some information. For example, when a key is pressed, an interrupt is generated to deliver the key-press event to the operating system. Each device is assigned an interrupt by some combination of the operating system and BIOS.

Devices are generally connected to an *programmable interrupt controller* (PIC), a separate chip that is part of the motherboard which buffers and communicates interrupt information to the main processor. Each device has a physical *interrupt line* between it an one of the PIC's provided by the system. When the device wants to interrupt, it will modify the voltage on this line.

A very broad description of the PIC's role is that it receives this interrupt and converts it to a message for consumption by the main processor. While the exact procedure varies by architecture, the general principle is that the operating system has configured an *interrupt descriptor table* which pairs each of the possible interrupts with a code address to jump to when the interrupt is received. This is illustrated in Figure 3.6, “Overview of handling an interrupt”.

Writing this *interrupt handler* is the job of the device driver author in conjunction with the operating system.

Figure 3.6. Overview of handling an interrupt



A generic overview of handling an interrupt. The device raises the interrupt to the interrupt controller, which passes the information onto the processor. The processor looks at its descriptor table, filled out by the operating system, to find the code to handle the fault.

Most drivers will split up handling of interrupts into *bottom* and *top* halves. The bottom half will acknowledge the interrupt, queue actions for processing and return the processor to what it was doing quickly. The top half will then run later when the CPU is free and do the more intensive processing. This is to stop an interrupt hogging the entire CPU.

Saving state

Since an interrupt can happen at any time, it is important that you can return to the running operation when finished handling the interrupt. It is generally the job of the operating system to ensure that upon entry to the interrupt handler, it saves any *state*; i.e. registers, and restores them when returning from the interrupt handler. In this way, apart from some lost time, the interrupt is completely transparent to whatever happens to be running at the time.

Interrupts v traps and exceptions

While an interrupt is generally associated with an external event from a physical device, the same mechanism is useful for handling internal system operations. For example, if the processor detects conditions such as an access to invalid memory, an attempt to divide-by-zero or an invalid instruction, it can internally raise an *exception* to be handled by the operating system. It is also the mechanism used to trap into the operating system for *system calls*, as discussed in the section called “System Calls” and to implement virtual memory, as discussed in Chapter 6, *Virtual Memory*. Although generated internally rather than from an external source, the principles of asynchronously interrupting the running code remains the same.

Types of interrupts

There are two main ways of signalling interrupts on a line — *level* and *edge* triggered.

Level-triggered interrupts define voltage of the interrupt line being held high to indicate an interrupt is pending. Edge-triggered interrupts detect *transitions* on the bus; that is

when the line voltage goes from low to high. With an edge-triggered interrupt, a square-wave pulse is detected by the PIC as signalling and interrupt has been raised.

The difference is pronounced when devices share an interrupt line. In a level-triggered system, the interrupt line will be high until all devices that have raised an interrupt have been processed and un-asserted their interrupt.

In an edge-triggered system, a pulse on the line will indicate to the PIC that an interrupt has occurred, which it will signal to the operating system for handling. However, if further pulses come in on the already asserted line from another device.

The issue with level-triggered interrupts is that it may require some considerable amount of time to handle an interrupt for a device. During this time, the interrupt line remains high and it is not possible to determine if any other device has raised an interrupt on the line. This means there can be considerable unpredictable latency in servicing interrupts.

With edge-triggered interrupts, a long-running interrupt can be noticed and queued, but other devices sharing the line can still transition (and hence raise interrupts) while this happens. However, this introduces new problems; if two devices interrupt at the same time it may be possible to miss one of the interrupts, or environmental or other interference may create a *spurious* interrupt which should be ignored.

Non-maskable interrupts

It is important for the system to be able to *mask* or prevent interrupts at certain times. Generally, it is possible to put interrupts on hold, but a particular class of interrupts, called *non-maskable interrupts* (NMI), are the exception to this rule. The typical example is the *reset* interrupt.

NMIs can be useful for implementing things such as a system watchdog, where a NMI is raised periodically and sets some flag that must be acknowledged by the operating system. If the acknowledgement is not seen before the next periodic NMI, then system can be considered to be not making forward progress. Another common usage is for profiling a system. A periodic NMI can be raised and used to evaluate what code the processor is currently running; over time this builds a profile of what code is being run and create a very useful insight into system performance.

IO Space

Obviously the processor will need to communicate with the peripheral device, and it does this via IO operations. The most common form of IO is so called *memory mapped IO* where registers on the device are *mapped* into memory.

This means that to communicate with the device, you need simply read or write to a specific address in memory. TODO: expand

DMA

Since the speed of devices is far below the speed of processors, there needs to be some way to avoid making the CPU wait around for data from devices.

Direct Memory Access (DMA) is a method of transferring data directly between an peripheral and system RAM.

The driver can setup a device to do a DMA transfer by giving it the area of RAM to put it's data into. It can then start the DMA transfer and allow the CPU to continue with other tasks.

Once the device is finished, it will raise an interrupt and signal to the driver the transfer is complete. From this time the data from the device (say a file from a disk, or frames from a video capture card) is in memory and ready to be used.

Other Buses

Other buses connect between the PCI bus and external devices.

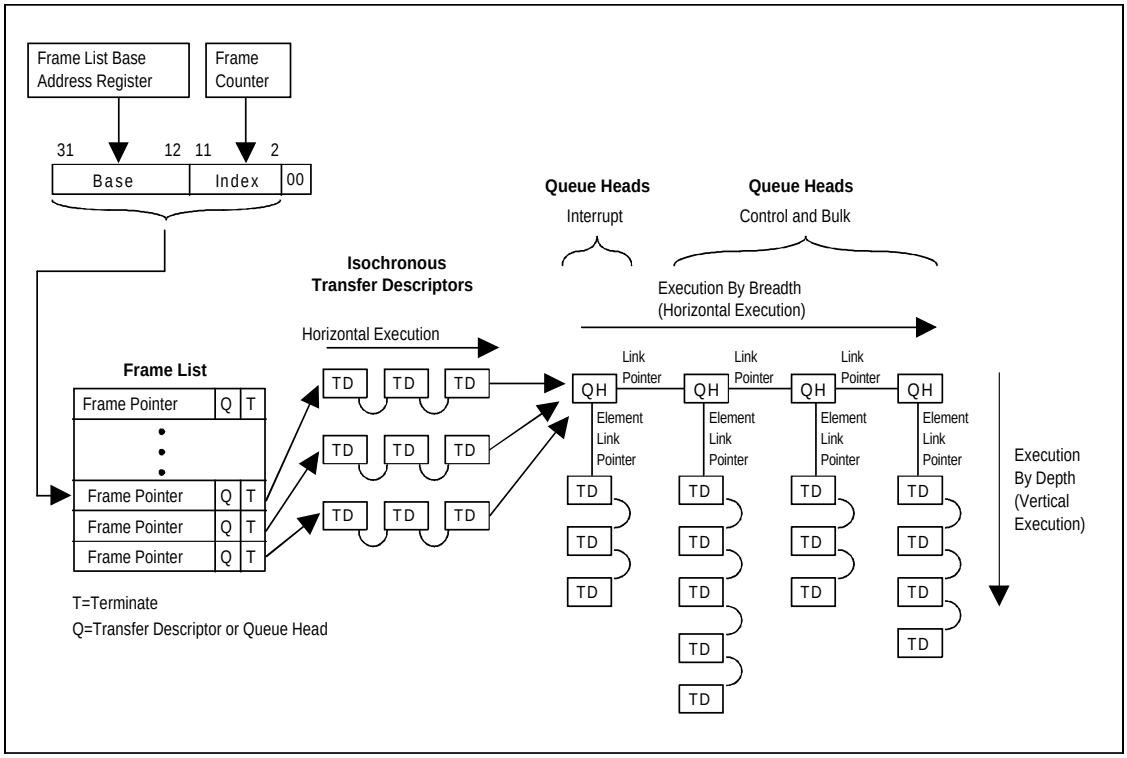
USB

From an operating system point of view, a USB device is a group of end-points grouped together into an interface. An end-point can be either *in* or *out* and hence transfers data in one direction only. End-points can have a number of different types:

- *Control* end-points are for configuring the device, etc.
- *Interrupt* end-points are for transferring small amounts of data. They have higher priority than ...
- *Bulk* end-points, which transfer large amounts of data but do not get guaranteed time constraints.
- *Isochronous* transfers are high-priority real-time transfers, but if they are missed they are not re-tried. This is for streaming data like video or audio where there is no point sending data again.

There can be many interfaces (made of multiple end-points) and interfaces are grouped into *configurations*. However Most devices only have a single configuration.

Figure 3.7. Overview of a UHCI controller operation



An overview of a UCHI controller, taken from Intel documentation [http://download.intel.com/technology/usb/UHCI11D.pdf].

Figure 3.7, “Overview of a UHCI controller operation” shows an overview of a universal host controller interface, or UHCI. It provides an overview of how USB data is moved out of the system by a combination of hardware and software. Essentially, the software sets up a template of data in a specified format for the host controller to read and send across the USB bus.

Starting at the top-left of the overview, the controller has a *frame* register with a counter which is incremented periodically — every millisecond. This value is used to index into a *frame list* created by software. Each entry in this table points to a queue of *transfer descriptors*. Software sets up this data in memory, and it is read by the host controller which is a separate chip the drives the USB bus. Software needs to schedule the work queues so that 90% of a frame time is given to isochronous data, and 10% left for interrupt, control and bulk data..

As you can see from the diagram, the way the data is linked means that transfer descriptors for isochronous data are associated with only one particular frame pointer — in other words only one particular time period — and after that will be discarded. However, the interrupt, control and bulk data are all *queued* after the isochronous data and thus if not transmitted in one frame (time period) will be done in the next.

The USB layer communicates through USB *request blocks*, or URBs. A URB contains information about what end-point this request relates to, data, any related information or attributes and a call-back function to be called when the URB is complete. USB drivers

submit URBs in a fixed format to the USB core, which manages them in co-ordination with the USB host controller as above. Your data gets sent off to the USB device by the USB core, and when its done your call-back is triggered.

Small to big systems

As Moore's law has predicted, computing power has been growing at a furious pace and shows no signs of slowing down. It is relatively uncommon for any high end servers to contain only a single CPU. This is achieved in a number of different fashions.

Symmetric Multi-Processing

Symmetric Multi-Processing, commonly shortened to *SMP*, is currently the most common configuration for including multiple CPUs in a single system.

The symmetric term refers to the fact that all the CPUs in the system are the same (e.g. architecture, clock speed). In a SMP system there are multiple processors that share other all other system resources (memory, disk, etc).

Cache Coherency

For the most part, the CPUs in the system work independently; each has its own set of registers, program counter, etc. Despite running separately, there is one component that requires strict synchronisation.

This is the CPU cache; remember the cache is a small area of quickly access able memory that mirrors values stored in main system memory. If one CPU modifies data in main memory and another CPU has an old copy of that memory in its cache the system will obviously not be in a consistent state. Note that the problem only occurs when processors are writing to memory, since if a value is only read the data will be consistent.

To co-ordinate keeping the cache coherent on all processors an SMP system uses *snooping*. Snooping is where a processor listens on a bus which all processors are connected to for cache events, and updates its cache accordingly.

One protocol for doing this is the *MOESI* protocol; standing for Modified, Owner, Exclusive, Shared, Invalid. Each of these is a state that a cache line can be in on a processor in the system. There are other protocols for doing as much, however they all share similar concepts. Below we examine MOESI so you have an idea of what the process entails.

When a processor requires reading a cache line from main memory, it firstly has to snoop all other processors in the system to see if they currently know anything about that area of memory (e.g. have it cached). If it does not exist in any other process, then the processor can load the memory into cache and mark it as *exclusive*. When it writes to the cache, it then changes state to be *modified*. Here the specific details of the cache come into play; some caches will immediately write back the modified cache to system memory (known as a *write-through* cache, because writes go through to main memory).

Others will not, and leave the modified value only in the cache until it is evicted, when the cache becomes full for example.

The other case is where the processor snoops and finds that the value is in another processors cache. If this value has already been marked as *modified*, it will copy the data into its own cache and mark it as *shared*. It will send a message for the other processor (that we got the data from) to mark its cache line as *owner*. Now imagine that a third processor in the system wants to use that memory too. It will snoop and find both a *shared* and a *owner* copy; it will thus take its value from the *owner* value. While all the other processors are only reading the value, the cache line stays *shared* in the system. However, when one processor needs to update the value it sends an *invalidate* message through the system. Any processors with that cache line must then mark it as invalid, because it not longer reflects the "true" value. When the processor sends the invalidate message, it marks the cache line as *modified* in its cache and all others will mark as *invalid* (note that if the cache line is *exclusive* the processor knows that no other processor is depending on it so can avoid sending an invalidate message).

From this point the process starts all over. Thus whichever processor has the *modified* value has the responsibility of writing the true value back to RAM when it is evicted from the cache. By thinking through the protocol you can see that this ensures consistency of cache lines between processors.

There are several issues with this system as the number of processors starts to increase. With only a few processors, the overhead of checking if another processor has the cache line (a read snoop) or invalidating the data in every other processor (invalidate snoop) is manageable; but as the number of processors increase so does the bus traffic. This is why SMP systems usually only scale up to around 8 processors.

Having the processors all on the same bus starts to present physical problems as well. Physical properties of wires only allow them to be laid out at certain distances from each other and to only have certain lengths. With processors that run at many gigahertz the speed of light starts to become a real consideration in how long it takes messages to move around a system.

Note that system software usually has no part in this process, although programmers should be aware of what the hardware is doing underneath in response to the programs they design to maximise performance.

Cache exclusivity in SMP systems

In the section called “Cache in depth” we described *inclusive* v *exclusive* caches. In general, L1 caches are usually inclusive — that is all data in the L1 cache also resides in the L2 cache. In a multiprocessor system, an inclusive L1 cache means that only the L2 cache need snoop memory traffic to maintain coherency, since any changes in L2 will be guaranteed to be reflected by L1. This reduces the complexity of the L1 and de-couples it from the snooping process allowing it to be faster.

Again, in general, most all modern high-end (e.g. not targeted at embedded) processors have a write-through policy for the L1 cache, and a write-back policy for the lower level caches. There are several reasons for this. Since in this class of processors L2 caches are almost exclusively on-chip and generally quite fast the penalties from having L1

write-through are not the major consideration. Further, since L1 sizes are small, pools of written data unlikely to be read in the future could cause pollution of the limited L1 resource. Additionally, a write-through L1 does not have to be concerned if it has outstanding dirty data, hence can pass the extra coherency logic to the L2 (which, as we mentioned, already has a larger part to play in cache coherency).

Hyperthreading

Much of the time of a modern processor is spent waiting for much slower devices in the memory hierarchy to deliver data for processing.

Thus strategies to keep the pipeline of the processor full are paramount. One strategy is to include enough registers and state logic such that two instruction streams can be processed at the same time. This makes one CPU look for all intents and purposes like two CPUs.

While each CPU has its own registers, they still have to share the core logic, cache and input and output bandwidth from the CPU to memory. So while two instruction streams can keep the core logic of the processor busier, the performance increase will not be as great has having two physically separate CPUs. Typically the performance improvement is below 20% (XXX check), however it can be drastically better or worse depending on the workloads.

Multi Core

With increased ability to fit more and more transistors on a chip, it became possible to put two or more processors in the same physical package. Most common is dual-core, where two processor cores are in the same chip. These cores, unlike hyperthreading, are full processors and so appear as two physically separate processors a la a SMP system.

While generally the processors have their own L1 cache, they do have to share the bus connecting to main memory and other devices. Thus performance is not as great as a full SMP system, but considerably better than a hyperthreading system (in fact, each core can still implement hyperthreading for an additional enhancement).

Multi core processors also have some advantages not performance related. As we mentioned, external physical buses between processors have physical limits; by containing the processors on the same piece of silicon extremely close to each other some of these problems can be worked around. The power requirements for multi core processors are much less than for two separate processors. This means that there is less heat needing to be dissipated which can be a big advantage in data centre applications where computers are packed together and cooling considerations can be considerable. By having the cores in the same physical package it makes muti-processing practical in applications where it otherwise would not be, such as laptops. It is also considerably cheaper to only have to produce one chip rather than two.

Clusters

Many applications require systems much larger than the number of processors a SMP system can scale to. One way of scaling up the system further is a *cluster*.

A cluster is simply a number of individual computers which have some ability to talk to each other. At the hardware level the systems have no knowledge of each other; the task of stitching the individual computers together is left up to software.

Software such as MPI allow programmers to write their software and then "farm out" parts of the program to other computers in the system. For example, image a loop that executes several thousand times performing independent action (that is no iteration of the loop affects any other iteration). With four computers in a cluster, the software could make each computer do 250 loops each.

The interconnect between the computers varies, and may be as slow as an internet link or as fast as dedicated, special buses (Infiniband). Whatever the interconnect, however, it is still going to be further down the memory hierarchy and much, much slower than RAM. Thus a cluster will not perform well in a situation when each CPU requires access to data that may be stored in the RAM of another computer; since each time this happens the software will need to request a copy of the data from the other computer, copy across the slow link and into local RAM before the processor can get any work done.

However, many applications *do not* require this constant copying around between computers. One large scale example is SETI@Home, where data collected from a radio antenna is analysed for signs of Alien life. Each computer can be distributed a few minutes of data to analyse, and only needs report back a summary of what it found. SETI@Home is effectively a very large, dedicated cluster.

Another application is rendering of images, especially for special effects in films. Each computer can be handed a single frame of the movie which contains the wire-frame models, textures and light sources which needs to be combined (rendered) into the amazing special effects we now take for grained. Since each frame is static, once the computer has the initial input it does not need any more communication until the final frame is ready to be sent back and combined into the move. For example the block-buster Lord of the Rings had their special effects rendered on a huge cluster running Linux.

Non-Uniform Memory Access

Non-Uniform Memory Access, more commonly abbreviated to NUMA, is almost the opposite of a cluster system mentioned above. As in a cluster system it is made up of individual nodes linked together, however the linkage between nodes is highly specialised (and expensive!). As opposed to a cluster system where the hardware has no knowledge of the linkage between nodes, in a NUMA system the *software* has no (well, less) knowledge about the layout of the system and the hardware does all the work to link the nodes together.

The term *non uniform memory access* comes from the fact that RAM may not be local to the CPU and so data may need to be accessed from a node some distance away. This obviously takes longer, and is in contrast to a single processor or SMP system where RAM is directly attached and always takes a constant (uniform) time to access.

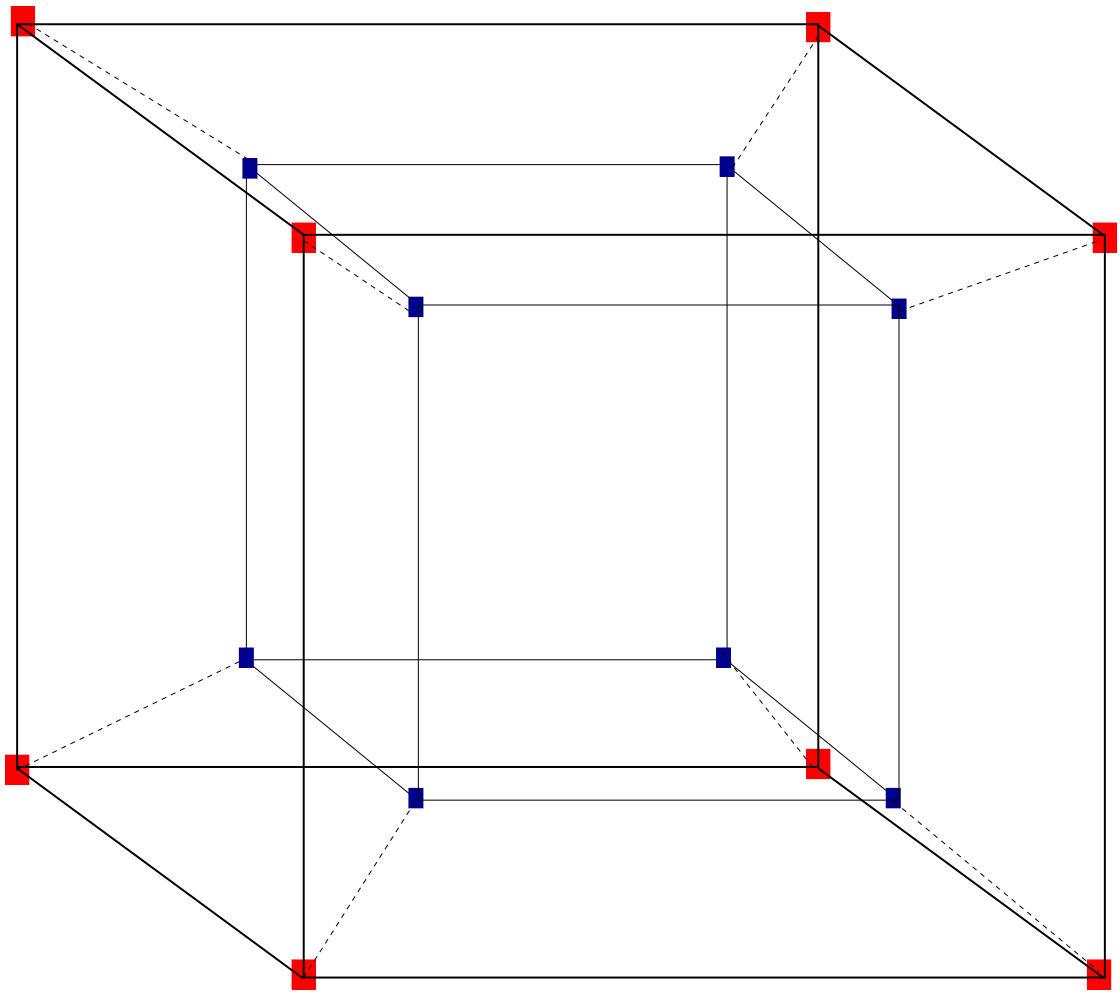
NUMA Machine Layout

With so many nodes talking to each other in a system, minimising the distance between each node is of paramount importance. Obviously it is best if every single node has a direct link to every other node as this minimises the distance any one node needs to go to find data. This is not a practical situation when the number of nodes starts growing into the hundreds and thousands as it does with large supercomputers; if you remember your high school maths the problem is basically a combination taken two at a time (each node talking to another), and will grow `n!/2*(n-2)!`.

To combat this exponential growth alternative layouts are used to trade off the distance between nodes with the interconnects required. One such layout common in modern NUMA architectures is the hypercube.

A hypercube has a strict mathematical definition (way beyond this discussion) but as a cube is a 3 dimensional counterpart of a square, so a hypercube is a 4 dimensional counterpart of a cube.

Figure 3.8. A Hypercube



An example of a hypercube. Hypercubes provide a good trade off between distance between nodes and number of interconnections required.

Above we can see the outer cube contains four 8 nodes. The maximum number of paths required for any node to talk to another node is 3. When another cube is placed inside this cube, we now have double the number of processors but the maximum path cost has only increased to 4. This means as the number of processors grow by 2^n the maximum path cost grows only linearly.

Cache Coherency

Cache coherency can still be maintained in a NUMA system (this is referred to as a cache-coherent NUMA system, or ccNUMA). As we mentioned, the broadcast based scheme used to keep the processor caches coherent in an SMP system does not scale to hundreds or even thousands of processors in a large NUMA system. One common scheme for cache coherency in a NUMA system is referred to as a *directory based model*. In this model processors in the system communicate to special cache directory hardware. The directory hardware maintains a consistent picture to each processor; this abstraction hides the working of the NUMA system from the processor.

The Censier and Feautrier directory based scheme maintains a central directory where each memory block has a flag bit known as the *valid bit* for each processor and a single *dirty* bit. When a processor reads the memory into its cache, the directory sets the valid bit for that processor.

When a processor wishes to write to the cache line the directory needs to set the dirty bit for the memory block. This involves sending an invalidate message to those processors who are using the cache line (and only those processors whose flag are set; avoiding broadcast traffic).

After this should any other processor try to read the memory block the directory will find the dirty bit set. The directory will need to get the updated cache line from the processor with the valid bit currently set, write the dirty data back to main memory and then provide that data back to the requesting processor, setting the valid bit for the requesting processor in the process. Note that this is transparent to the requesting processor and the directory may need to get that data from somewhere very close or somewhere very far away.

Obviously having thousands of processors communicating to a single directory does also not scale well. Extensions to the scheme involve having a hierarchy of directories that communicate between each other using a separate protocol. The directories can use a more general purpose communications network to talk between each other, rather than a CPU bus, allowing scaling to much larger systems.

NUMA Applications

NUMA systems are best suited to the types of problems that require much interaction between processor and memory. For example, in weather simulations a common idiom is to divide the environment up into small "boxes" which respond in different ways (oceans and land reflect or store different amounts of heat, for example). As simulations are run, small variations will be fed in to see what the overall result is. As each box influences the surrounding boxes (e.g. a bit more sun means a particular box puts out more heat, affecting the boxes next to it) there will be much communication (contrast that with the individual image frames for a rendering process, each of which does not influence the other). A similar process might happen if you were modelling a car crash, where each small box of the simulated car folds in some way and absorbs some amount of energy.

Although the software has no directly knowledge that the underlying system is a NUMA system, programmers need to be careful when programming for the system to get maximum performance. Obviously keeping memory close to the processor that is going to use it will result in the best performance. Programmers need to use techniques such as *profiling* to analyse the code paths taken and what consequences their code is causing for the system to extract best performance.

Memory ordering, locking and atomic operations

The multi-level cache, superscalar multi-processor architecture brings with it some interesting issues relating to how a programmer sees the processor running code.

Imagine program code is running on two processors simultaneously, both processors sharing effectively one large area of memory. If one processor issues a store instruction, to put a register value into memory, when can it be sure that the other processor does a load of that memory it will see the correct value?

In the simplest situation the system could guarantee that if a program executes a store instruction, any subsequent load instructions will see this value. This is referred to as *strict memory ordering*, since the rules allow no room for movement. You should be starting to realise why this sort of thing is a serious impediment to performance of the system.

Much of the time, the memory ordering is not required to be so strict. The programmer can identify points where they need to be sure that all outstanding operations are seen globally, but in between these points there may be many instructions where the semantics are not important.

Take, for example, the following situation.

Example 3.1. Memory Ordering

```
 1
             typedef struct {
   int a;
   int b;
 5 } a_struct;

   /*
    * Pass in a pointer to be allocated as a new structure
    */
10 void get_struct(a_struct *new_struct)
   {
    void *p = malloc(sizeof(a_struct));

    /* We don't particularly care what order the following two
15   * instructions end up acutally executing in */
    p->a = 100;
    p->b = 150;

    /* However, they must be done before this instruction.
20   * Otherwise, another processor who looks at the value of p
     * could find it pointing into a structure whose values have
     * not been filled out.
     */
    new_struct = p;
25 }
```

In this example, we have two stores that can be done in any particular order, as it suits the processor. However, in the final case, the pointer must only be updated once the two previous stores are known to have been done. Otherwise another processor might look at the value of p, follow the pointer to the memory, load it, and get some completely incorrect value!

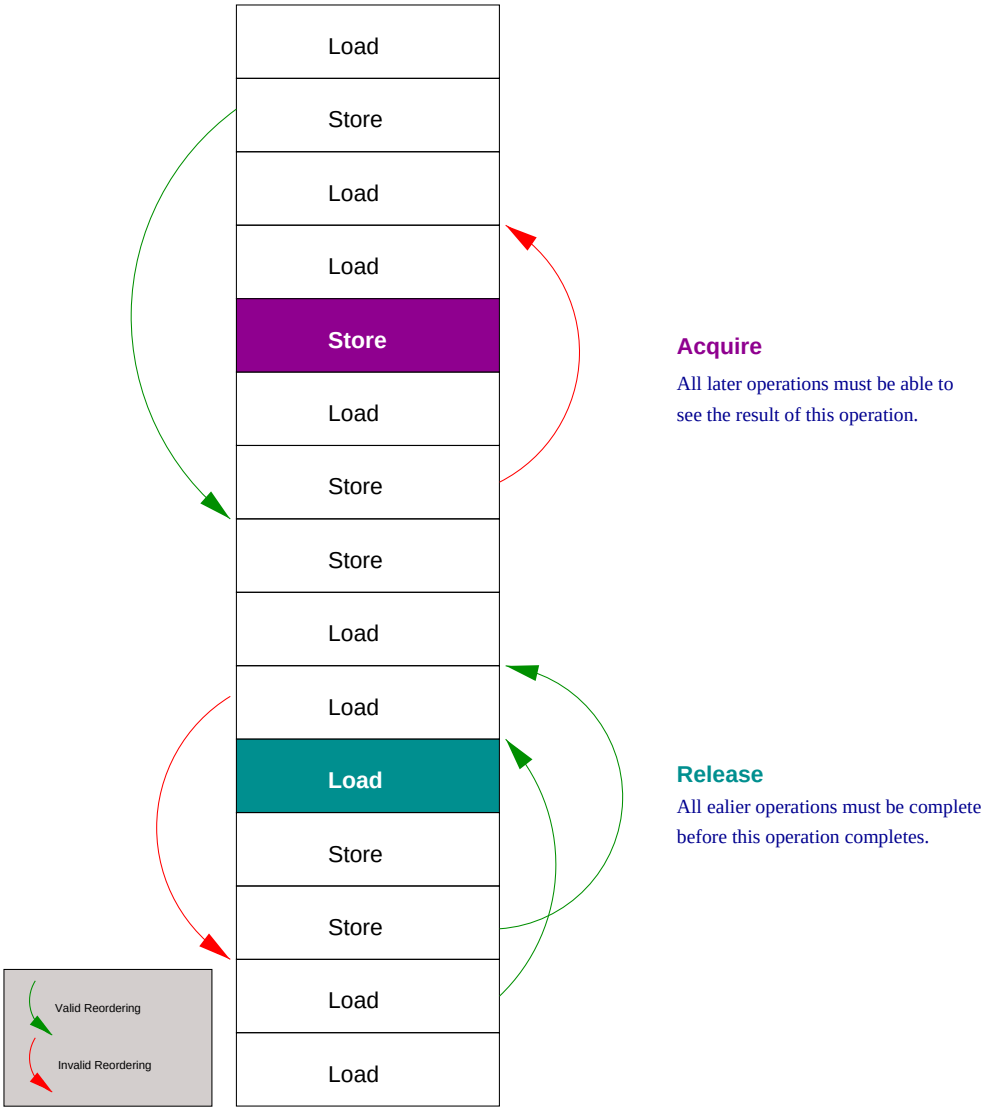
To indicate this, loads and stores have to have *semantics* that describe what behaviour they must have. Memory semantics are described in terms of *fences* that dictate how loads and stores may be reordered around the load or store.

By default, a load or store can be re-ordered anywhere.

Acquire semantics is like a fence that only allows load and stores to move downwards through it. That is, when this load or store is complete you can be guaranteed that any later load or stores will see the value (since they can not be moved above it).

Release semantics is the opposite, that is a fence that allows any load or stores to be done before it (move upwards), but nothing before it to move downwards past it. Thus, when load or store with release semantics is processed, you can be store that any earlier load or stores will have been complete.

Figure 3.9. Acquire and Release semantics



An illustration of valid reorderings around operations with acquire and release semantics.

A *full memory fence* is a combination of both; where no loads or stores can be reordered in any direction around the current load or store.

The strictest memory model would use a full memory fence for every operation. The weakest model would leave every load and store as a normal re-orderable instruction.

Processors and memory models

Different processors implement different memory models.

The x86 (and AMD64) processor has a quite strict memory model; all stores have release semantics (that is, the result of a store is guaranteed to be seen by any later load or store) but all loads have normal semantics. lock prefix gives memory fence.

Itanium allows all load and stores to be normal, unless explicitly told. XXX

Locking

Knowing the memory ordering requirements of each architecture is no practical for all programmers, and would make programs difficult to port and debug across different processor types.

Programmers use a higher level of abstraction called *locking* to allow simultaneous operation of programs when there are multiple CPUs.

When a program acquires a lock over a piece of code, no other processor can obtain the lock until it is released. Before any critical pieces of code, the processor must attempt to take the lock; if it can not have it, it does not continue.

You can see how this is tied into the naming of the memory ordering semantics in the previous section. We want to ensure that before we *acquire* a lock, no operations that should be protected by the lock are re-ordered before it. This is how acquire semantics works.

Conversely, when we *release* the lock, we must be sure that every operation we have done whilst we held the lock is complete (remember the example of updating the pointer previously?). This is release semantics.

There are many software libraries available that allow programmers to not have to worry about the details of memory semantics and simply use the higher level of abstraction of `lock()` and `unlock()`.

Locking difficulties

Locking schemes make programming more complicated, as it is possible to *deadlock* programs. Imagine if one processor is currently holding a lock over some data, and is currently waiting for a lock for some other piece of data. If that other processor is waiting for the lock the first processor holds before unlocking the second lock, we have a deadlock situation. Each processor is waiting for the other and neither can continue without the others lock.

Often this situation arises because of a subtle *race condition*; one of the hardest bugs to track down. If two processors are relying on operations happening in a specific order in time, there is always the possibility of a race condition occurring. A gamma ray from

an exploding star in a different galaxy might hit one of the processors, making it skip a beat, throwing the ordering of operations out. What will often happen is a deadlock situation like above. It is for this reason that program ordering needs to be ensured by semantics, and not by relying on one time specific behaviours. (XXX not sure how i can better word that).

A similar situation is the opposite of deadlock, called *livelock*. One strategy to avoid deadlock might be to have a "polite" lock; one that you give up to anyone who asks. This politeness might cause two threads to be constantly giving each other the lock, without either ever taking the lock long enough to get the critical work done and be finished with the lock (a similar situation in real life might be two people who meet at a door at the same time, both saying "no, you first, I insist". Neither ends up going through the door!).

Locking strategies

Underneath, there are many different strategies for implementing the behaviour of locks.

A simple lock that simply has two states - locked or unlocked, is referred to as a *mutex* (short for mutual exclusion; that is if one person has it the other can not have it).

There are, however, a number of ways to implement a mutex lock. In the simplest case, we have what its commonly called a *spinlock*. With this type of lock, the processor sits in a tight loop waiting to take the lock; equivalent to it saying "can I have it now" constantly much as a young child might ask of a parent.

The problem with this strategy is that it essentially wastes time. Whilst the processor is sitting constantly asking for the lock, it is not doing any useful work. For locks that are likely to be only held locked for a very short amount of time this may be appropriate, but in many cases the amount of time the lock is held might be considerably longer.

Thus another strategy is to *sleep* on a lock. In this case, if the processor can not have the lock it will start doing some other work, waiting for notification that the lock is available for use (we see in future chapters how the operating system can switch processes and give the processor more work to do).

A mutex is however just a special case of a *semaphore*, famously invented by the Dutch computer scientist Dijkstra. In a case where there are multiple resources available, a semaphore can be set to count accesses to the resources. In the case where the number of resources is one, you have a mutex. The operation of semaphores can be detailed in any algorithms book.

These locking schemes still have some problems however. In many cases, most people only want to read data which is updated only rarely. Having all the processors wanting to only read data require taking a lock can lead to *lock contention* where less work gets done because everyone is waiting to obtain the same lock for some data.

Atomic Operations

Explain what it is.

Chapter 4. The Operating System

The role of the operating system

The operating system underpins the entire operation of the modern computer.

Abstraction of hardware

The fundamental operation of the operating system (OS) is to abstract the hardware to the programmer and user. The operating system provides generic interfaces to services provided by the underlying hardware.

In a world without operating systems, every programmer would need to know the most intimate details of the underlying hardware to get anything to run. Worse still, their programs would not run on other hardware, even if that hardware has only slight differences.

Multitasking

We expect modern computers to do many different things at once, and we need some way to arbitrate between all the different programs running on the system. It is the operating systems job to allow this to happen seamlessly.

The operating system is responsible for *resource management* within the system. Many tasks will be competing for the resources of the system as it runs, including processor time, memory, disk and user input. The job of the operating system is to arbitrate these resources to the multiple tasks and allow them access in an orderly fashion. You have probably experienced when this *fails* as it usually ends up with your computer crashing (the famous "blue screen of death" for example).

Standardised Interfaces

Programmers want to write programs that will run on as many different hardware platforms as possible. By having operating system support for standardised interfaces, programmers can get this functionality.

For example, if the function to open a file on one system is `open()`, on another is `open_file()` and on yet another `openf()` programmers will have the dual problem of having to remember what each system does and their programs will not work on multiple systems.

The Portable Operating System Interface (POSIX)[1] is a very important standard implemented by UNIX type operating systems. Microsoft Windows has similar proprietary standards.

[1]The X comes from *Unix*, from which the standard grew. Today, POSIX is the same thing as the Single UNIX Specification Version 3 or ISO/IEC 9945:2002. This is a free standard, available online.

Security

On multi-user systems, security is very important. As the arbitrator of access to the system the operating system is responsible for ensuring that only those with the correct permissions can access resources.

For example if a file is owned by one user, another user should not be allowed to open and read it. However there also need to be mechanisms to share that file safely between the users should they want it.

Operating systems are large and complex programs, and often security issues will be found. Often a virus or worm will take advantage of these bugs to access resources it should not be allowed to, such as your files or network connection; to fight them you must install *patches* or updates provided by your operating system vendor.

Performance

As the operating system provides so many services to the computer, it's performance is critical. Many parts of the operating system run extremely frequently, so even an overhead of just a few processor cycles can add up to a big decrease in overall system performance.

The operating system needs to exploit the features of the underlying hardware to make sure it is getting the best possible performance for the operations, and consequently systems programmers need to understand the intimate details of the architecture they are building for.

In many cases the systems programmers job is about deciding on policies for the system. Often the case that the side effects of making one part of the operating system run faster will make another part run slower or less efficiently. Systems programmers need to understand all these trade offs when they are building their operating system.

Operating System Organisation

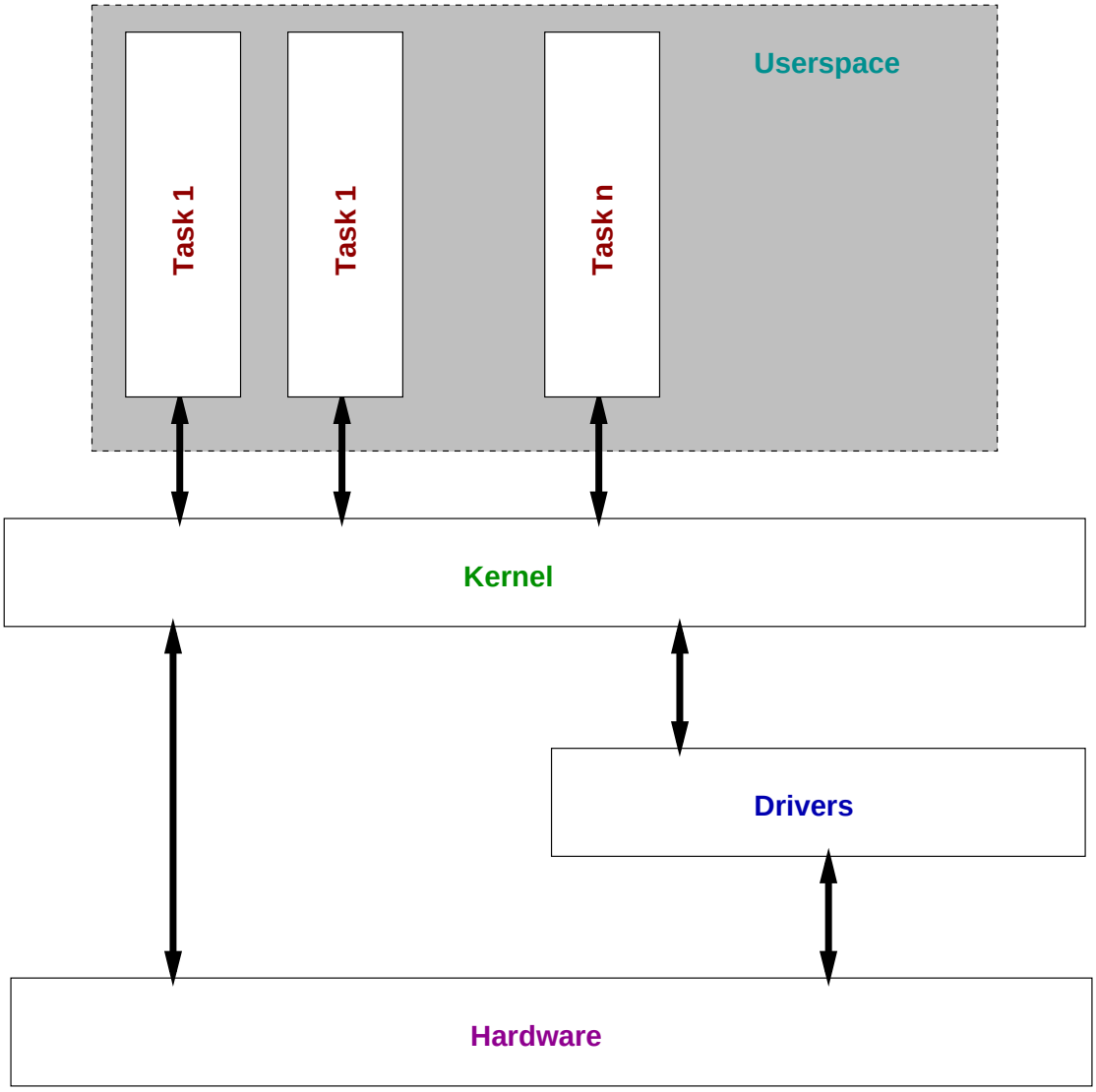
The operating system is roughly organised as in the figure below.

Once upon a time, the Single UNIX specification and the POSIX Standards were separate entities. The Single UNIX specification was released by a consortium called the "Open Group", and was freely available as per their requirements. The latest version is The Single Unix Specification Version 3.

The IEEE POSIX standards were released as IEEE Std 1003.[insert various years, revisions here], and were not freely available. The latest version is IEEE 1003.1-2001 and is equivalent to the Single Unix Specification Version 3.

Thus finally the two separate standards were merged into what is known as the Single UNIX Specification Version 3, which is also standardised by the ISO under ISO/IEC 9945:2002. This happened early in 2002. So when people talk about POSIX, SUS3 or ISO/IEC 9945:2002 they all mean the same thing!

Figure 4.1. The Operating System



The organisation of the kernel. Processes the kernel is running live in *userspace*, and the kernel talks both directly to hardware and through *drivers*.

The Kernel

The kernel *is* the operating system. As the figure illustrates, the kernel communicates to hardware both directly and through *drivers*.

Just as the kernel abstracts the hardware to user programs, drivers abstract hardware to the kernel. For example there are many different types of graphic card, each one with slightly different features. As long as the kernel exports an API, people who have access to the specifications for the hardware can write drivers to implement that API. This way the kernel can access many different types of hardware.

The kernel is generally what we called *privileged*. As you will learn, the hardware has important roles to play in running multiple tasks and keeping the system secure, but

these rules do not apply to the kernel. We know that the kernel must handle programs that crash (remember it is the operating systems job arbitrate between multiple programs running on the same system, and there is no guarantee that they will behave), but if any internal part of the operating system crashes chances are the entire system will become useless. Similarly security issues can be exploited by user processes to escalate themselves to the privilege level of the kernel; at that point they can access any part of the system completely unchecked.

Monolithic v Microkernels

One debate that is often comes up surrounding operating systems is whether the kernel should be a *microkernel* or *monolithic*.

The monolithic approach is the most common, as taken by most common Unixes (such as Linux). In this model the core privileged kernel is large, containing hardware drivers, file system accesses controls, permissions checking and services such as Network File System (NFS).

Since the kernel is always privileged, if any part of it crashes the whole system has the potential to comes to a halt. If one driver has a bug it can overwrite any memory in the system with no problems, ultimately causing the system to crash.

A microkernel architecture tries to minimise this possibility by making the privileged part of the kernel as small as possible. This means that most of the system runs as unprivileged programs, limiting the harm that any one crashing component can influence. For example, drivers for hardware can run in separate processes, so if one goes astray it can not overwrite any memory but that allocated to it.

Whilst this sounds like the most obvious idea, the problem comes back two main issues

1. Performance is decreased. Talking between many different components can decrease performance.

2. It is slightly more difficult for the programmer.

Both of these criticisms come because to keep separation between components most microkernels are implemented with a *message passing* based system, commonly referred to as *inter-process communication* or IPC. Communicating between individual components happens via discrete messages which must be bundled up, sent to the other component, unbundled, operated upon, re-bundled up and sent back, and then unbundled again to get the result.

This is a lot of steps for what might be a fairly simple request from a foreign component. Obviously one request might make the other component do more requests of even more components, and the problem can multiply. Slow message passing implementations were largely responsible for the poor performance of early microkernel systems, and the concepts of passing messages are slightly harder for programmers to program for. The enhanced protection from having components run separately was not sufficient to overcome these hurdles in early microkernel systems, so they fell out of fashion.

In a monolithic kernel calls between components are simple function calls, as all programmers are familiar with.

There is no definitive answer as to which is the best organisation, and it has started many arguments in both academic and non-academic circles. Hopefully as you learn more about operating systems you will be able to make up your own mind!

Modules

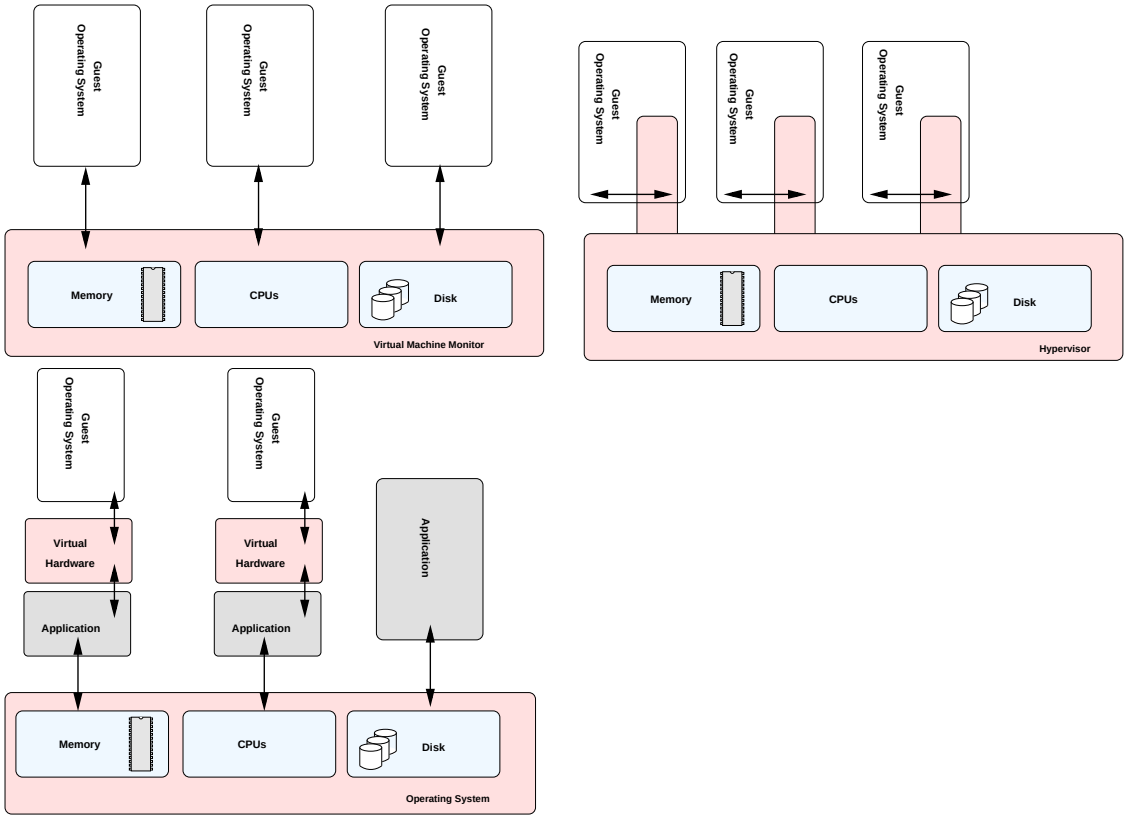
The Linux kernel implements a module system, where drivers can loaded into the running kernel "on the fly" as they are required. This is good in that drivers, which make up a large part of operating system code, are not loaded for devices that are not present in the system. Someone who wants to make the most generic kernel possible (i.e. runs on lots of different hardware, such as RedHat or Debian) can include most drivers as modules which are only loaded if the system it is running on has the hardware available.

However, the modules are loaded directly in the privileged kernel and operate at the same privilege level as the rest of the kernel, so the system is still considered a monolithic kernel.

Virtualisation

Closely related to kernel is the concept of virtualisation of hardware. Modern computers are very powerful, and often it is useful to not thing of them as one whole system but split a single physical computer up into separate "virtual" machines. Each of these virtual machines looks for all intents and purposes as a completely separate machine, although physically they are all in the same box, in the same place.

Figure 4.2. The Operating System



Some different virtualisation methods.

This can be organised in many different ways. In the simplest case, a small *virtual machine monitor* can run directly on the hardware and provide an interface to the guest operating systems running on top. This VMM is often often called a hypervisor (from the word "supervisor")[2]. In fact, the operating system on top may have no idea that the hypervisor is even there at all, as the hypervisor presents what appears to be a complete system. It intercepts operations between the guest operating system and hardware and only presents a subset of the system resources to each.

This is often used on large machines (with many CPUs and much RAM) to implement *partitioning*. This means the machine can be split up into smaller virtual machines. Often you can allocate more resources to running systems on the fly, as requirements dictate. The hypervisors on many large IBM machines are actually quite complicated affairs, with many millions of lines of code. It provides a multitude of system management services.

Another option is to have the operating system aware of the underlying hypervisor, and request system resources through it. This is sometimes referred to as *paravirtualisation* due to it's halfway nature. This is similar to the way early versions of the Xen system works and is a compromise solution. It hopefully provides better performance since the operating system is explicitly asking for system resources from the hypervisor when required, rather than the hypervisor having to work things out dynamically.

[2] In fact, the hypervisor shares much in common with a micro-kernel; both strive to be small layers to present the hardware in a safe fashion to layers above it.

Finally, you may have a situation where an application running on top of the existing operating system presents a virtualised system (including CPU, memory, BIOS, disk, etc) which a plain operating system can run on. The application converts the requests to hardware through to the underlying hardware via the existing operating system. This is similar to how VMWare works. This approach has many overheads, as the application process has to emulate an entire system and convert everything to requests from the underlying operating system. However, this lets you emulate an entirely different architecture all together, as you can dynamically translate the instructions from one processor type to another (as the Rosetta system does with Apple software which moved from the PowerPC processor to Intel based processors).

Performance is major concern when using any of these virtualisation techniques, as what was once fast operations directly on hardware need to make their way through layers of abstraction.

Intel have discussed hardware support for virtualisation soon to be coming in their latest processors. These extensions work by raising a special exception for operations that might require the intervention of a virtual machine monitor. Thus the processor looks the same as a non-virtualised processor to the application running on it, but when that application makes requests for resources that might be shared between other guest operating systems the virtual machine monitor can be invoked.

This provides superior performance because the virtual machine monitor does not need to monitor every operation to see if it is safe, but can wait until the processor notifies that something *unsafe* has happened.

Covert Channels

This is a digression, but an interesting security flaw relating to virtualised machines. If the partitioning of the system is not static, but rather *dynamic*, there is a potential security issue involved.

In a dynamic system, resources are allocated to the operating systems running on top as required. Thus if one is doing particularly CPU intensive operations whilst the other is waiting on data to come from disks, more of the CPU power will be given to the first task. In a static system, each would get 50% an the unused portion would go to waste.

Dynamic allocation actually opens up a communications channel between the two operating systems. Anywhere that two states can be indicated is sufficient to communicate in binary. Imagine both systems are extremely secure, and no information should be able to pass between one and the other, ever. Two people with access could collude to pass information between themselves by writing two programs that try to take large amounts of resources at the same time.

When one takes a large amount of memory there is less available for the other. If both keep track of the maximum allocations, a bit of information can be transferred. Say they make a pact to check every second if they can allocate this large amount of memory. If the target can, that is considered binary 0, and if it can not (the other machine has all the memory), that is considered binary 1. A data rate of one bit per second is not astounding, but information is flowing.

This is called a *covert channel*, and whilst admittedly far fetched there have been examples of security breaches from such mechanisms. It just goes to show that the life of a systems programmer is never simple!

Userspace

We call the theoretical place where programs run by the user *userspace*. Each program runs in userspace, talking to the kernel through *system calls* (discussed below).

As previously discussed, userspace is *unprivileged*. User programs can only do a limited range of things, and should never be able to crash other programs, even if they crash themselves.

System Calls

Overview

System calls are how userspace programs interact with the kernel. The general principle behind how they work is described below.

System call numbers

Each and every system call has a *system call number* which is known by both the userspace and the kernel. For example, both know that system call number 10 is `open()`, system call number 11 is `read()`, etc.

The *Application Binary Interface* (ABI) is very similar to an API but rather than being for software is for hardware. The API will define which register the system call number should be put in so the kernel can find it when it is asked to do the system call.

Arguments

System calls are no good without arguments; for example `open()` needs to tell the kernel exactly *what* file to open. Once again the ABI will define which registers arguments should be put into for the system call.

The trap

To actually perform the system call, there needs to be some way to communicate to the kernel we wish to make a system call. All architectures define an instruction, usually called `break` or something similar, that signals to the hardware we wish to make a system call.

Specifically, this instruction will tell the hardware to modify the instruction pointer to point to the kernels system call handler (when the operating system sets its self up it tells the hardware where its system call handler lives). So once the userspace calls the break instruction, it has lost control of the program and passed it over to the kernel.

The rest of the operation is fairly straight forward. The kernel looks in the predefined register for the system call number, and looks it up in a table to see which function it should call. This function is called, does what it needs to do, and places it's return value into *another* register defined by the ABI as the return register.

The final step is for the kernel to make a jump instruction back to the userspace program, so it can continue off where it left from. The userpsace program gets the data it needs from the return register, and continues happily on it's way!

Although the details of the process can get quite hairy, this is basically all their is to a system call.

libc

Although you can do all of the above by hand for each system call, system libraries usually do most of the work for you. The standard library that deals with system calls on UNIX like systems is `libc`; we will learn more about it's roles in future weeks.

Analysing a system call

As the system libraries usually deal with making systems call for you, we need to do some low level hacking to illustrate exactly how the system calls work.

We will illustrate how probably the most simple system call, `getpid()`, works. This call takes no arguments and returns the ID of the currently running program (or process; we'll look more at the process in later weeks).

Example 4.1. getpid() example

```
 1
             #include <stdio.h>

   /* for syscall() */
 5 #include <sys/syscall.h>
   #include <unistd.h>

   /* system call numbers */
   #include <asm/unistd.h>
10
   void function(void)
   {
    int pid;

15  pid = __syscall(__NR_getpid);
   }
```

We start by writing a small C program which we can start to illustrate the mechanism behind system calls. The first thing to note is that there is a `syscall` argument provided by the system libraries for directly making system calls. This provides an easy way for programmers to directly make systems calls without having to know the exact assembly language routines for making the call on their hardware. So why do we use `getpid()` at all? Firstly, it is much clearer to use a symbolic function name in your code. However, more importantly, `getpid()` may work in very different ways on different systems. For

example, on Linux the `getpid()` call can be cached, so if it is run twice the system library will not take the penalty of having to make an entire system call to find out the same information again.

By convention under Linux, system calls numbers are defined in the `asm/unistd.h` file from the kernel source. Being in the `asm` subdirectory, this is different for each architecture Linux runs on. Again by convention, system calls numbers are given a `#define` name consisting of `__NR_`. Thus you can see our code will be making the `getpid` system call, storing the value in `pid`.

We will have a look at how several architectures implement this code under the hood. We're going to look at real code, so things can get quite hairy. But stick with it -- this is *exactly* how your system works!

PowerPC

PowerPC is a RISC architecture common in older Apple computers, and the core of devices such as the latest version of the Xbox.

Example 4.2. PowerPC system call example

```
 1
   /* On powerpc a system call basically clobbers the same registers like a
    * function call, with the exception of LR (which is needed for the
 5  * "sc; bnslr" sequence) and CR (where only CR0.SO is clobbered to signal
    * an error return status).
    */

   #define __syscall_nr(nr, type, name, args...)    \
10  unsigned long __sc_ret, __sc_err;    \
    {        \
     register unsigned long __sc_0  __asm__ ("r0");  \
     register unsigned long __sc_3  __asm__ ("r3");  \
     register unsigned long __sc_4  __asm__ ("r4");  \
15   register unsigned long __sc_5  __asm__ ("r5");  \
     register unsigned long __sc_6  __asm__ ("r6");  \
     register unsigned long __sc_7  __asm__ ("r7");  \
            \
     __sc_loadargs_##nr(name, args);    \
20   __asm__ __volatile__     \
      ("sc           \n\t"    \
       "mfcr %0      "    \
      : "=&r" (__sc_0),    \
        "=&r" (__sc_3),  "=&r" (__sc_4),  \
25      "=&r" (__sc_5),  "=&r" (__sc_6),  \
        "=&r" (__sc_7)    \
      : __sc_asm_input_##nr    \
      : "cr0", "ctr", "memory",   \
        "r8", "r9", "r10","r11", "r12");  \
30   __sc_ret = __sc_3;     \
     __sc_err = __sc_0;     \
    }        \
    if (__sc_err & 0x10000000)     \
    {        \
35   errno = __sc_ret;     \
     __sc_ret = -1;      \
    }        \
    return (type) __sc_ret

40 #define __sc_loadargs_0(name, dummy...)     \
    __sc_0 = __NR_##name
```

```
   #define __sc_loadargs_1(name, arg1)     \
    __sc_loadargs_0(name);      \
    __sc_3 = (unsigned long) (arg1)
45 #define __sc_loadargs_2(name, arg1, arg2)    \
    __sc_loadargs_1(name, arg1);     \
    __sc_4 = (unsigned long) (arg2)
   #define __sc_loadargs_3(name, arg1, arg2, arg3)    \
    __sc_loadargs_2(name, arg1, arg2);    \
50  __sc_5 = (unsigned long) (arg3)
   #define __sc_loadargs_4(name, arg1, arg2, arg3, arg4)   \
    __sc_loadargs_3(name, arg1, arg2, arg3);   \
    __sc_6 = (unsigned long) (arg4)
   #define __sc_loadargs_5(name, arg1, arg2, arg3, arg4, arg5)  \
55  __sc_loadargs_4(name, arg1, arg2, arg3, arg4);   \
    __sc_7 = (unsigned long) (arg5)

   #define __sc_asm_input_0 "0" (__sc_0)
   #define __sc_asm_input_1 __sc_asm_input_0, "1" (__sc_3)
60 #define __sc_asm_input_2 __sc_asm_input_1, "2" (__sc_4)
   #define __sc_asm_input_3 __sc_asm_input_2, "3" (__sc_5)
   #define __sc_asm_input_4 __sc_asm_input_3, "4" (__sc_6)
   #define __sc_asm_input_5 __sc_asm_input_4, "5" (__sc_7)

65 #define _syscall0(type,name)      \
   type name(void)        \
   {         \
    __syscall_nr(0, type, name);     \
   }
70
   #define _syscall1(type,name,type1,arg1)     \
   type name(type1 arg1)       \
   {         \
    __syscall_nr(1, type, name, arg1);    \
75 }

   #define _syscall2(type,name,type1,arg1,type2,arg2)   \
   type name(type1 arg1, type2 arg2)     \
   {         \
80  __syscall_nr(2, type, name, arg1, arg2);   \
   }

   #define _syscall3(type,name,type1,arg1,type2,arg2,type3,arg3)  \
   type name(type1 arg1, type2 arg2, type3 arg3)    \
85 {         \
    __syscall_nr(3, type, name, arg1, arg2, arg3);   \
   }

   #define _syscall4(type,name,type1,arg1,type2,arg2,type3,arg3,type4,arg4) \
90 type name(type1 arg1, type2 arg2, type3 arg3, type4 arg4)  \
   {         \
    __syscall_nr(4, type, name, arg1, arg2, arg3, arg4);  \
   }

95 #define _syscall5(type,name,type1,arg1,type2,arg2,type3,arg3,type4,arg4,type5,arg5) \
   type name(type1 arg1, type2 arg2, type3 arg3, type4 arg4, type5 arg5) \
   {         \
    __syscall_nr(5, type, name, arg1, arg2, arg3, arg4, arg5); \
   }
100
```

This code snippet from the kernel header file `asm/unistd.h` shows how we can implement system calls on PowerPC. It looks very complicated, but it can be broken down step by step.

Firstly, jump to the end of the example where the `_syscallN` macros are defined. You can see there are many macros, each one taking progressively one more argument.

We'll concentrate on the most simple version, `_syscall0` to start with. It only takes two arguments, the return type of the system call (e.g. a C `int` or `char`, etc) and the name of the system call. For `getpid` this would be done as `_syscall0(int,getpid)`.

Easy so far! We now have to start pulling apart `__syscall_nr` macro. This is not dissimilar to where we were before, we take the number of arguments as the first parameter, the type, name and then the actual arguments.

The first step is declaring some names for registers. What this essentially does is says `__sc_0` refers to `r0` (i.e. register 0). The compiler will usually use registers how it wants, so it is important we give it constraints so that it doesn't decide to go using register we need in some ad-hoc manner.

We then call `sc_loadargs` with the interesting `##` parameter. That is just a *paste* command, which gets replaced by the `nr` variable. Thus for our example it expands to `__sc_loadargs_0(name, args);`. `__sc_loadargs` we can see below sets `__sc_0` to be the system call number; notice the paste operator again with the `__NR_` prefix we talked about, and the variable name that refers to a specific register.

So, all this tricky looking code actually does is puts the system call number in register 0! Following the code through, we can see that the other macros will place the system call arguments into `r3` through `r7` (you can only have a maximum of 5 arguments to your system call).

Now we are ready to tackle the `__asm__` section. What we have here is called *inline assembly* because it is assembler code mixed right in with source code. The exact syntax is a little to complicated to go into right here, but we can point out the important parts.

Just ignore the `__volatile__` bit for now; it is telling the compiler that this code is unpredictable so it shouldn't try and be clever with it. Again we'll start at the end and work backwards. All the stuff after the colons is a way of communicating to the compiler about what the inline assembly is doing to the CPU registers. The compiler needs to know so that it doesn't try using any of these registers in ways that might cause a crash.

But the interesting part is the two assembly statements in the first argument. The one that does all the work is the `sc` call. That's all you need to do to make your system call!

So what happens when this call is made? Well, the processor is interrupted knows to transfer control to a specific piece of code setup at system boot time to handle interrupts. There are many interrupts; system calls are just one. This code will then look in register 0 to find the system call number; it then looks up a table and finds the right function to jump to to handle that system call. This function receives it's arguments in registers 3 - 7.

So, what happens once the system call handler runs and completes? Control returns to the next instruction after the `sc`, in this case a *memory fence* instruction. What this essentially says is "make sure everything is committed to memory"; remember how we talked about pipelines in the superscalar architecture? This instruction ensures that everything we think has been written to memory actually has been, and isn't making it's way through a pipeline somewhere.

Well, we're almost done! The only thing left is to return the value from the system call. We see that `__sc_ret` is set from r3 and `__sc_err` is set from r0. This is interesting; what are these two values all about?

One is the return value, and one is the error value. Why do we need two variables? System calls can fail, just as any other function. The problem is that a system call can return any possible value; we can not say "a negative value indicates failure" since a negative value might be perfectly acceptable for some particular system call.

So our system call function, before returning, ensures its result is in register r3 and any error code is in register r0. We check the error code to see if the top bit is set; this would indicate a negative number. If so, we set the global `errno` value to it (this is the standard variable for getting error information on call failure) and set the return to be `-1`. Of course, if a valid result is received we return it directly.

So our calling function should check the return value is not `-1`; if it is it can check errno to find the exact reason why the call failed.

And that is an entire system call on a PowerPC!

x86 system calls

Below we have the same interface as implemented for the x86 processor.

Example 4.3. x86 system call example

```
 1
              /* user-visible error numbers are in the range -1 - -124: see <asm-i386/errno.h> */

   #define __syscall_return(type, res)    \
 5 do {       \
          if ((unsigned long)(res) >= (unsigned long)(-125)) { \
                  errno = -(res);     \
                  res = -1;     \
          }       \
10        return (type) (res);     \
   } while (0)

   /* XXX - _foo needs to be __foo, while __NR_bar could be _NR_bar. */
   #define _syscall0(type,name)   \
15 type name(void)     \
   {      \
   long __res;     \
   __asm__ volatile ("int $0x80"   \
           : "=a" (__res)    \
20         : "0" (__NR_##name));   \
   __syscall_return(type,__res);
   }

   #define _syscall1(type,name,type1,arg1)   \
25 type name(type1 arg1)     \
   {       \
   long __res;      \
   __asm__ volatile ("int $0x80"    \
           : "=a" (__res)     \
30         : "0" (__NR_##name),"b" ((long)(arg1))); \
   __syscall_return(type,__res);
   }
```

```
#define _syscall2(type,name,type1,arg1,type2,arg2)   \
type name(type1 arg1,type2 arg2)     \
{          \
long __res;        \
__asm__ volatile ("int $0x80"      \
        : "=a" (__res)       \
        : "0" (__NR_##name),"b" ((long)(arg1)),"c" ((long)(arg2))); \
__syscall_return(type,__res);
}

#define _syscall3(type,name,type1,arg1,type2,arg2,type3,arg3)  \
type name(type1 arg1,type2 arg2,type3 arg3)    \
{          \
long __res;        \
__asm__ volatile ("int $0x80"      \
        : "=a" (__res)       \
        : "0" (__NR_##name),"b" ((long)(arg1)),"c" ((long)(arg2)), \
                  "d" ((long)(arg3)));     \
__syscall_return(type,__res);      \
}

#define _syscall4(type,name,type1,arg1,type2,arg2,type3,arg3,type4,arg4) \
type name (type1 arg1, type2 arg2, type3 arg3, type4 arg4)   \
{          \
long __res;         \
__asm__ volatile ("int $0x80"       \
        : "=a" (__res)        \
        : "0" (__NR_##name),"b" ((long)(arg1)),"c" ((long)(arg2)),  \
          "d" ((long)(arg3)),"S" ((long)(arg4)));    \
__syscall_return(type,__res);       \
}

#define _syscall5(type,name,type1,arg1,type2,arg2,type3,arg3,type4,arg4, \
          type5,arg5)        \
type name (type1 arg1,type2 arg2,type3 arg3,type4 arg4,type5 arg5)  \
{          \
long __res;         \
__asm__ volatile ("int $0x80"       \
        : "=a" (__res)        \
        : "0" (__NR_##name),"b" ((long)(arg1)),"c" ((long)(arg2)),  \
          "d" ((long)(arg3)),"S" ((long)(arg4)),"D" ((long)(arg5)));  \
__syscall_return(type,__res);       \
}

#define _syscall6(type,name,type1,arg1,type2,arg2,type3,arg3,type4,arg4,   \
          type5,arg5,type6,arg6)        \
type name (type1 arg1,type2 arg2,type3 arg3,type4 arg4,type5 arg5,type6 arg6)   \
{           \
long __res;           \
__asm__ volatile ("push %%ebp ; movl %%eax,%%ebp ; movl %1,%%eax ; int $0x80 ; pop %%ebp" \
        : "=a" (__res)          \
        : "i" (__NR_##name),"b" ((long)(arg1)),"c" ((long)(arg2)),    \
          "d" ((long)(arg3)),"S" ((long)(arg4)),"D" ((long)(arg5)),    \
          "0" ((long)(arg6)));         \
__syscall_return(type,__res);         \
}
```

The x86 architecture is very different from the PowerPC that we looked at previously. The x86 is classed as a CISC processor as opposed to the RISC PowerPC, and has dramatically less registers.

Start by looking at the most simple `_syscall0` macro. It simply calls the `int` instruction with a value of `0x80`. This instruction makes the CPU raise interrupt 0x80, which will jump to code that handles system calls in the kernel.

We can start inspecting how to pass arguments with the longer macros. Notice how the PowerPC implementation cascaded macros downwards, adding one argument per time. This implementation has slightly more copied code, but is a little easier to follow.

x86 register names are based around letters, rather than the numerical based register names of PowerPC. We can see from the zero argument macro that only the A register gets loaded; from this we can tell that the system call number is expected in the EAX register. As we start loading registers in the other macros you can see the short names of the registers in the arguments to the `__asm__` call.

We see something a little more interesting in `__syscall6`, the macro taking 6 arguments. Notice the `push` and `pop` instructions? These work with the stack on x86, "pushing" a value onto the top of the stack in memory, and popping the value from the stack back into memory. Thus in the case of having six registers we need to store the value of the `ebp` register in memory, put our argument in in (the `mov` instruction), make our system call and then restore the original value into `ebp`. Here you can see the disadvantage of not having enough registers; stores to memory are expensive so the more you can avoid them, the better.

Another thing you might notice there is nothing like the *memory fence* instruction we saw previously with the PowerPC. This is because on x86 the effect of all instructions will be guaranteed to be visible when the complete. This is easier for the compiler (and programmer) to program for, but offers less flexibility.

The only thing left to contrast is the return value. On the PowerPC we had two registers with return values from the kernel, one with the value and one with an error code. However on x86 we only have one return value that is passed into `__syscall_return`. That macro casts the return value to `unsigned long` and compares it to an (architecture and kernel dependent) range of negative values that might represent error codes (note that the `errno` value is positive, so the negative result from the kernel is negated). However, this means that system calls can not return small negative values, since they are indistinguishable from error codes. Some system calls that have this requirement, such as `getpriority()`, add an offset to their return value to force it to always be positive; it is up to the userspace to realise this and subtract this constant value to get back to the "real" value.

Privileges

Hardware

We mentioned how one of the major tasks of the operating system is to implement security; that is to not allow one application or user to interfere with any other that is running in the system. This means applications should not be able to overwrite each others memory or files, and only access system resources as dictated by system policy.

However, when an application is running it has exclusive use of the processor. We see how this works when we examine processes in the next chapter. Ensuring the application only accesses memory it owns is implemented by the virtual memory system,

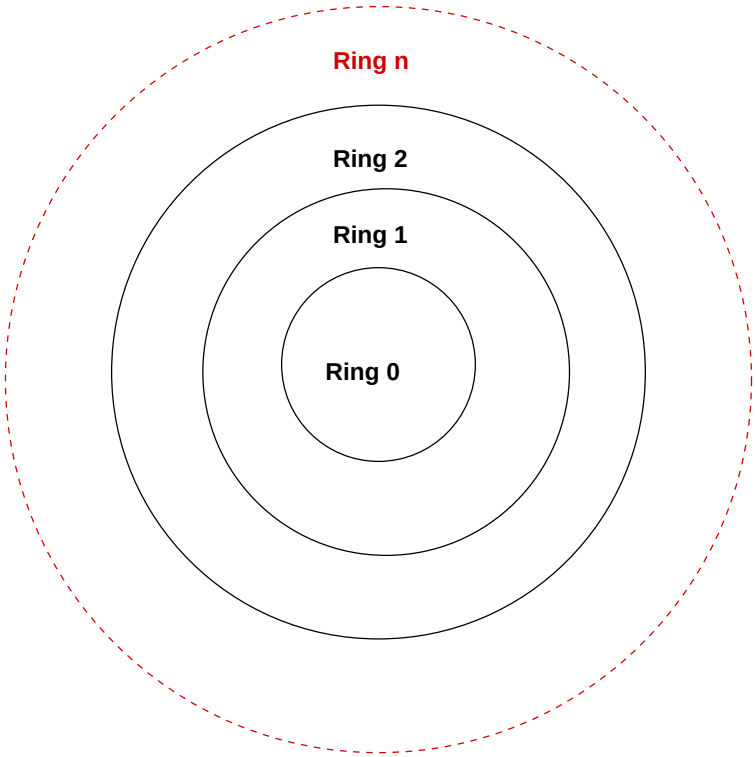
which we examine in the chapter after next. The essential point is that the hardware is responsible for enforcing these rules.

The system call interface we have examined is the gateway to the application getting to system resources. By forcing the application to request resources through a system call into the kernel, the kernel can enforce rules about what sort of access can be provided. For example, when an application makes an `open()` system call to open a file on disk, it will check the permissions of the user against the file permissions and allow or deny access.

Privilege Levels

Hardware protection can usually be seen as a set of concentric rings around a core set of operations.

Figure 4.3. Rings



Privilege levels on x86

In the inner most ring are the most protected instructions; those that only the kernel should be allowed to call. For example, the `HLT` instruction to halt the processor should not be allowed to be run by a user application, since it would stop the entire computer from working. However, the kernel needs to be able to call this instruction when the computer is legitimately shut down.[3]

[3]What happens when a "naughty" application calls that instruction anyway? The hardware will usually raise an exception, which will involve jumping to a specified handler in the operating system similar to the system call handler. The operating system will then probably terminate the program, usually giving the user some error about how the application has crashed.

Each inner ring can access any instructions protected by a further out ring, but not any protected by a further in ring. Not all architectures have multiple levels of rings as above, but most will either provide for at least a "kernel" and "user" level.

386 protection model

The 386 protection model has four rings, though most operating systems (such as Linux and Windows) only use two of the rings to maintain compatibility with other architectures that do now allow as many discrete protection levels.

386 maintains privileges by making each piece of application code running in the system have a small descriptor, called a *code descriptor*, which describes, amongst other things, its privilege level. When running application code makes a jump into some other code outside the region described by its code descriptor, the privilege level of the target is checked. If it is higher than the currently running code, the jump is disallowed by the hardware (and the application will crash).

Raising Privilege

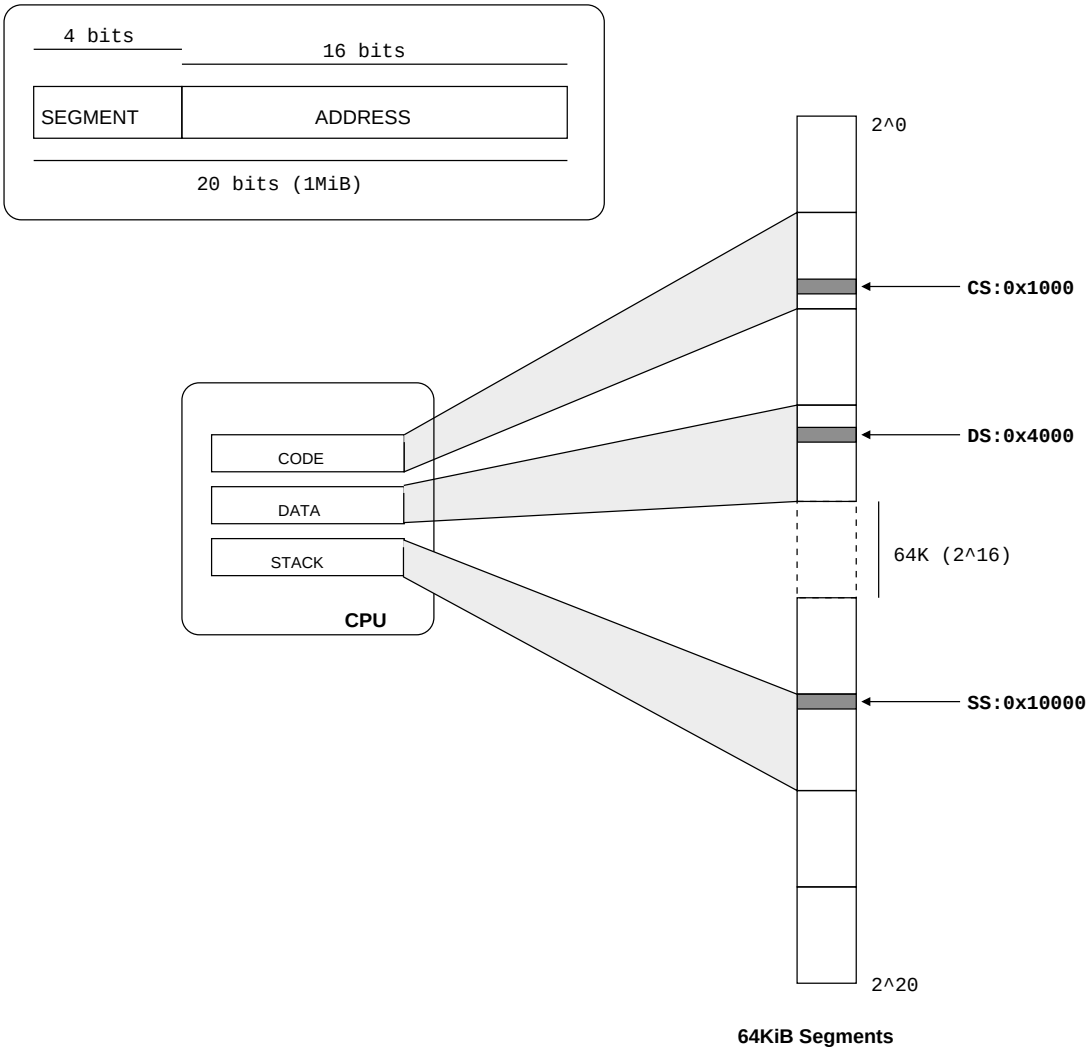
Applications may only raise their privilege level by specific calls that allow it, such as the instruction to implement a system call. These are usually referred to as a *call gate* because they function just as a physical gate; a small entry through an otherwise impenetrable wall. When that instruction is called we have seen how the hardware completely stops the running application and hands control over to the kernel. The kernel must act as a gatekeeper; ensuring that nothing nasty is coming through the gate. This means it must check system call arguments carefully to make sure it will not be fooled into doing anything it shouldn't (if it can be, that is a security bug). As the kernel runs in the innermost ring, it has permissions to do any operation it wants; when it is finished it will return control back to the application which will again be running with it's lower privilege level.

Fast System Calls

One problem with traps as described above is that they are very expensive for the processor to implement. There is a lot of state to be saved before context can switch. Modern processors have realised this overhead and strive to reduce it.

To understand the call-gate mechanism described above requires investigation of the ingenious but complicated segmentation scheme used by the processor. The original reason for segmentation was to be able to use more than the 16 bits available in a register for an address, as illustrated in Figure 4.4, “x86 Segmentation Addressing”.

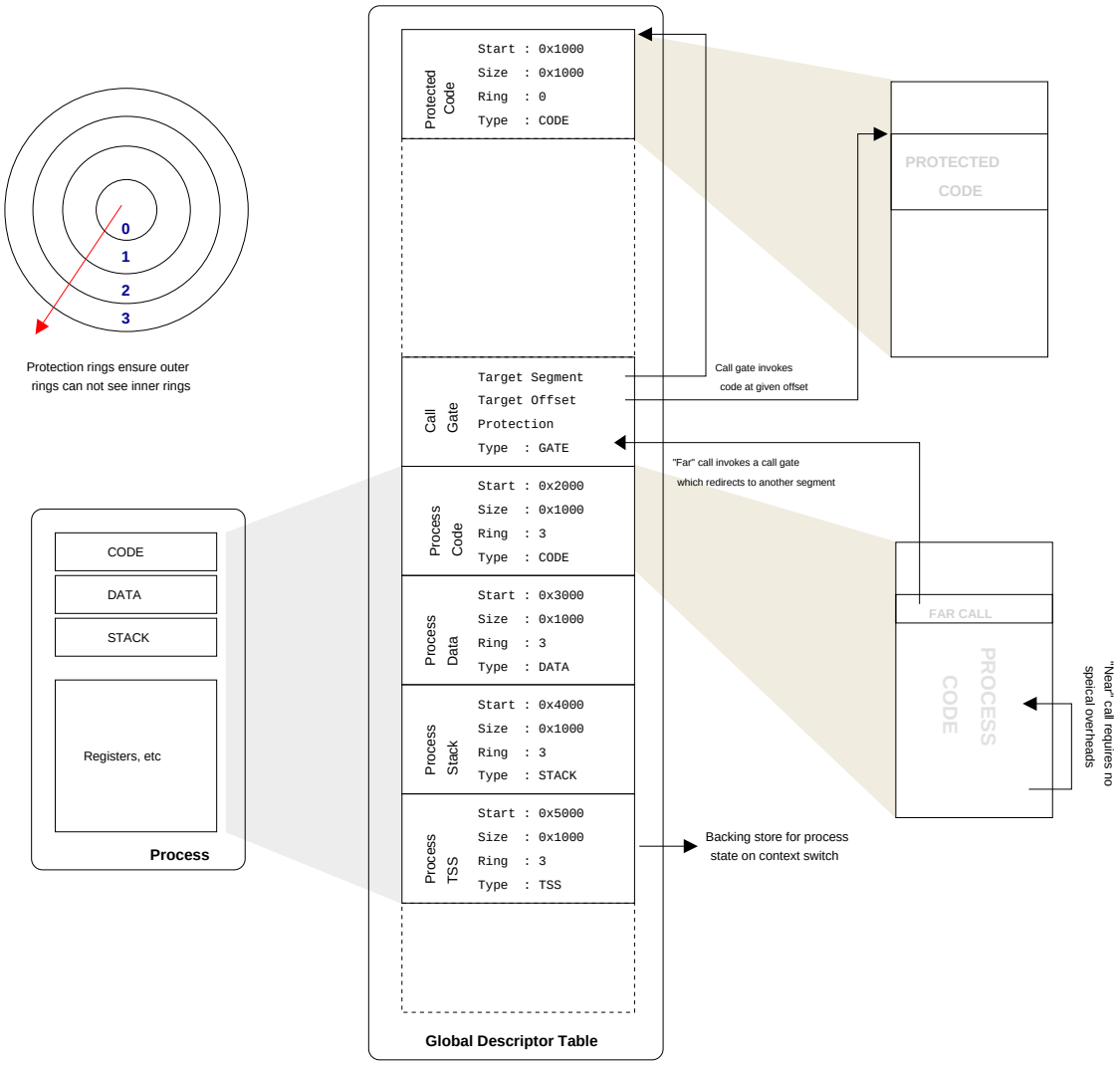
Figure 4.4. x86 Segmentation Addressing



Segmentation expanding the address space of a processor by dividing it into chunks. The processor keeps special segment registers, and addresses are specified by a segment register and offset combination. The value of the segment register is added to the offset portion to find a final address.

When x86 moved to 32 bit registers, the segmentation scheme remained but in a different format. Rather than fixed segment sizes, segments are allowed to be any size. This means the processor needs to keep track of all these different segments and their sizes, which it does using *descriptors*. The segment descriptors available to everyone are kept in the *global descriptor table* or GDT for short. Each process has a number of registers which point to entries in the GDT; these are the segments the process can access (there are also *local* descriptor tables, and it all interacts with task state segments, but that's not important now). The overall situation is illustrated in Figure 4.5, “x86 segments”.

Figure 4.5. x86 segments



x86 segments in action. Notice how a "far-call" passes via a call-gate which redirects to a segment of code running at a lower ring level. The only way to modify the code-segment selector, implicitly used for all code addresses, is via the call mechanism. Thus the call-gate mechanism ensures that to choose a new segment descriptor, and hence possibly change protection levels, you must transition via a known entry point.

Since the operating system assigns the segment registers as part of the process state, the processor hardware knows what segments of memory the currently running process can access and can enforce *protection* to ensure the process doesn't touch anything it is not supposed to. If it does go out of bounds, you receive a *segmentation fault*, which most programmers are familiar with.

The picture becomes more interesting when running code needs to make calls into code that resides in *another* segment. As discussed in the section called “386 protection model”, x86 does this with *rings*, where ring 0 is the highest permission, ring 3 is the lowest, and inner rings can access outer rings but not vice-versa.

As discussed in the section called “Raising Privilege”, when ring 3 code wants to jump into ring 0 code, it is essentially modifying its code segment selector to point to a different segment. To do this, it must use a special *far-call* instruction which hardware ensures passes through the call gate. There is no other way for the running process to choose a new code-segment descriptor, and hence the processor will start executing code at the known offset within the ring 0 segment, which is responsible for maintaining integrity (e.g. not reading arbitrary and possibly malicious code and executing it. Of course nefarious attackers will always look for ways to make your code do what you did not intend it to!).

This allows a whole hierarchy of segments and permissions between them. You might have noticed a cross segment call sounds exactly like a system call. If you've ever looked at Linux x86 assembly the standard way to make a system call is `int 0x80`, which raises interrupt `0x80`. An interrupt stops the processor and goes to an interrupt gate, which then works the same as a call gate -- it changes privilege level and bounces you off to some other area of code .

The problem with this scheme is that it is *slow*. It takes a lot of effort to do all this checking, and many registers need to be saved to get into the new code. And on the way back out, it all needs to be restored again.

On a modern x86 system segmentation and the four-level ring system is not used thanks to virtual memory, discussed fully in Chapter 6, *Virtual Memory*. The only thing that really happens with segmentation switching is system calls, which essentially switch from mode 3 (userspace) to mode 0 and jump to the system call handler code inside the kernel. Thus the processor provides extra *fast system call* instructions called `sysenter` (and `sysexit` to get back) which speed up the whole process over a `int 0x80` call by removing the general nature of a far-call — that is the possibility of transitioning into any segment at any ring level — and restricting the call to only transition to ring 0 code at a specific segment and offset, as stored in registers.

Because the general nature has been replaced with so much prior-known information, the whole process can be speed up, and hence we have a the aforementioned *fast system call*. The other thing to note is that state is not preserved when the kernel gets control. The kernel has to be careful to not to destroy state, but it also means it is free to only save as little state as is required to do the job, so can be much more efficient about it. This is a very RISC philosophy, and illustrates how the line blurs between RISC and CISC processors.

For more information on how this is implemented in the Linux kernel, see the section called “Kernel Library”.

Other ways of communicating with the kernel

ioctl

about ioctls

File Systems

about proc, sysfs, debugfs, etc

Chapter 5. The Process

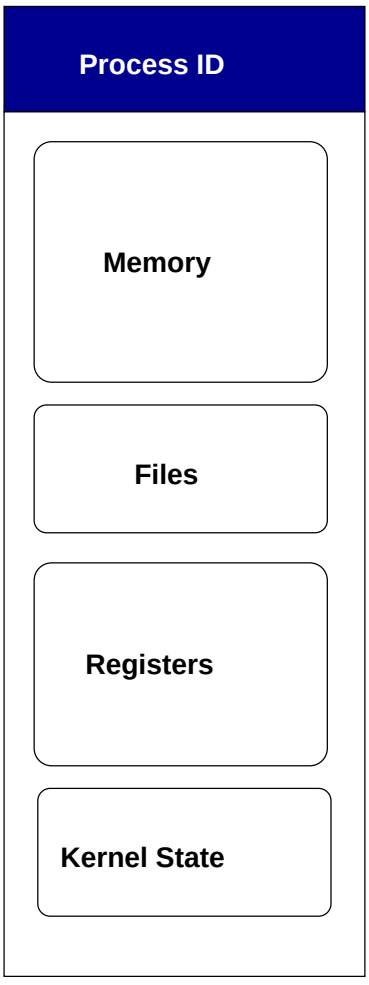
What is a process?

We are all familiar with the modern operating system running many tasks all at once or *multitasking*.

We can think of each process as a bundle of elements kept by the kernel to keep track of all these running tasks.

Elements of a process

Figure 5.1. The Elements of a Process



Process ID

The *process ID* (or the PID) is assigned by the operating system and is unique to each running process.

Memory

We will learn exactly how a process gets it's memory in the following weeks -- it is one of the most fundamental parts of how the operating system works. However, for now it is sufficient to know that each process gets it's own section of memory.

In this memory all the program code is stored, along with variables and any other allocated storage.

Parts of the memory can be shared between process (called, not surprisingly *shared memory*). You will often see this called *System Five Shared Memory* (or SysV SHM) after the original implementation in an older operating system.

Another important concept a process may utilise is that of *mmap*ing a file on disk to memory. This means that instead of having to open the file and use commands such as `read()` and `write()` the file looks as if it were any other type of RAM. `mmaped` areas have permissions such as read, write and execute which need to be kept track of. As we know, it is the job of the operating system to maintain security and stability, so it needs to check if a process tries to write to a read only area and return an error.

Code and Data

A process can be further divided into *code* and `data` sections. Program code and data should be kept separately since they require different permissions from the operating system and separation facilitates sharing of code (as you see later). The operating system needs to give program code permission to be read and executed, but generally not written to. On the other hand data (variables) require read and write permissions but should not be executable[1].

The Stack

One other very important part of a process is an area of memory called *the stack*. This can be considered part of the data section of a process, and is intimately involved in the execution of any program.

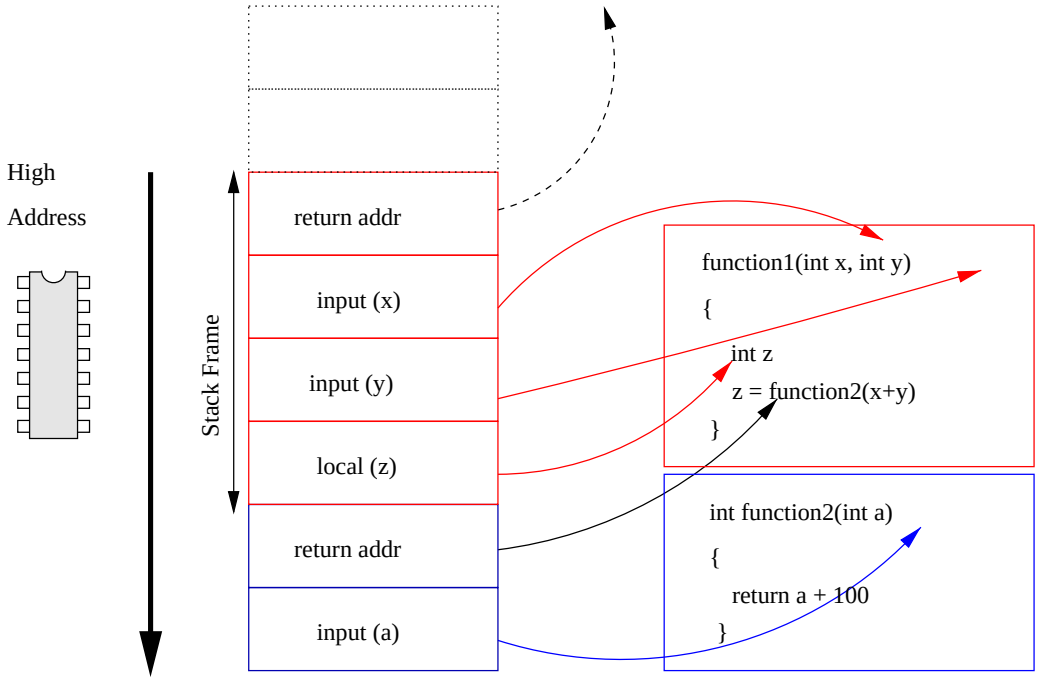
A stack is generic data structure that works exactly like a stack of plates; you can *push* an item (put a plate on top of a stack of plates), which then becomes the top item, or you can *pop* an item (take a plate off, exposing the previous plate).

Stacks are fundamental to function calls. Each time a function is called it gets a new `stack frame`. This is an area of memory which usually contains, at a minimum, the address to return to when complete, the input arguments to the function and space for local variables.

By convention, stacks usually *grow down*[2] . This means that the stack starts at a high address in memory and progressively gets lower.

[1] Not all architectures support this, however. This has lead to a wide range of security problems on many architectures.
[2] Some architectures, such as PA-RISC from HP, have stacks that grow upwards. On some other architectures, such as IA64, there are other storage areas (the register backing store) that grow from the bottom toward the stack.

Figure 5.2. The Stack



We can see how having a stack brings about many of the features of functions.

- Each function has its own copy of its input arguments. This is because each function is allocated a new stack frame with its arguments in a fresh area of memory.
- This is the reason why a variable defined inside a function can not be seen by other functions. Global variables (which can be seen by any function) are kept in a separate area of data memory.
- This facilitates *recursive* calls. This means a function is free to call its self again, because a new stack frame will be created for all its local variables.
- Each frame contains the address to return to. C only allows a single value to be returned from a function, so by convention this value is returned to the calling function in a specified register, rather than on the stack.
- Because each frame has a reference to the one before it, a debugger can "walk" backwards, following the pointers up the stack. From this it can produce a *stack trace* which shows you all functions that were called leading into this function. This is extremely useful for debugging.

  You can see how the way functions works fits exactly into the nature of a stack. Any function can call any other function, which then becomes the up most function (put on top of the stack). Eventually that function will return to the function that called it (takes itself off the stack).
- Stacks do make calling functions slower, because values must be moved out of registers and into memory. Some architectures allow arguments to be passed in

registers directly; however to keep the semantics that each function gets a unique copy of each argument the registers must *rotate*.

- You may have heard of the term a *stack overflow*. This is a common way of hacking a system by passing bogus values. If you as a programmer accept arbitrary input into a stack variable (say, reading from the keyboard or over the network) you need to explicitly say how big that data is going to be.

  Allowing any amount of data unchecked will simply overwrite memory. Generally this leads to a crash, but some people realised that if they overwrote just enough memory to place a specific value in the return address part of the stack frame, when the function completed rather than returning to the correct place (where it was called from) they could make it return into the data they just sent. If that data contains binary executable code that hacks the system (e.g. starts a terminal for the user with root privileges) then your computer has been compromised.

  This happens because the stack grows downwards, but data is read in "upwards" (i.e. from lower address to higher addresses).

  There are several ways around this; firstly as a programmer you must ensure that you always check the amount of data you are receiving into a variable. The operating system can help to avoid this on behalf of the programmer by ensuring that the stack is marked as *not executable*; that is that the processor will not run any code, even if a malicious user tries to pass some into your program. Modern architectures and operating systems support this functionality.

- Stacks are ultimately managed by the compiler, as it is responsible for generating the program code. To the operating system the stack just looks like any other area of memory for the process.

To keep track of the current growth of the stack, the hardware defines a register as the *stack pointer*. The compiler (or the programmer, when writing in assembler) uses this register to keep track of the current top of the stack.

Example 5.1. Stack pointer example

```
 1
              $ cat sp.c
  void function(void)
  {
 5        int i = 100;
          int j = 200;
          int k = 300;
  }

10 $ gcc -fomit-frame-pointer -S sp.c

  $ cat sp.s
          .file   "sp.c"
          .text
15 .globl function
          .type   function, @function
  function:
          subl    $16, %esp
          movl    $100, 4(%esp)
20        movl    $200, 8(%esp)
```

```
            movl    $300, 12(%esp)
            addl    $16, %esp
            ret
            .size   function, .-function
25          .ident  "GCC: (GNU) 4.0.2 20050806 (prerelease) (Debian 4.0.1-4)"
            .section        .note.GNU-stack,"",@progbits
```

Above we show a simple function allocating three variables on the stack. The disassembly illustrates the use of the stack pointer on the x86 architecture[3]. Firstly we allocate some space on the stack for our local variables. Since the stack grows down, we subtract from the value held in the stack pointer. The value 16 is a value large enough to hold our local variables, but may not be exactly the size required (for example with 3 4 byte `int` values we really only need 12 bytes, not 16) to keep alignment of the stack in memory on certain boundaries as the compiler requires.

Then we move the values into the stack memory (and in a real function, use them). Finally, before returning to our parent function we "pop" the values off the stack by moving the stack pointer back to where it was before we started.

The Heap

The heap is an area of memory that is managed by the process for on the fly memory allocation. This is for variables whose memory requirements are not known at compile time.

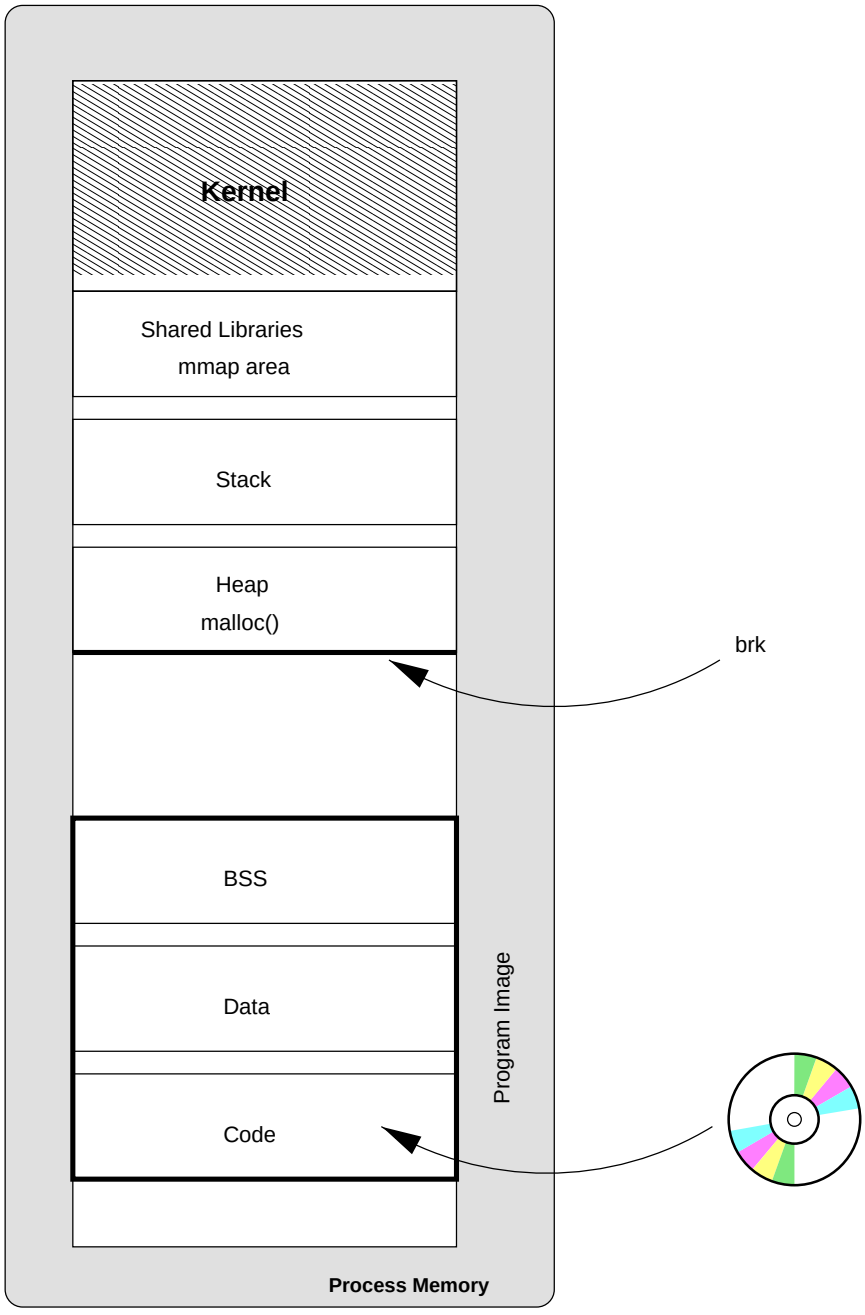
The bottom of the heap is known as the *brk*, so called for the system call which modifies it. By using the `brk` call to grow the area downwards the process can request the kernel allocate more memory for it to use.

The heap is most commonly managed by the `malloc` library call. This makes managing the heap easy for the programmer by allowing them to simply allocate and free (via the `free` call) heap memory. `malloc` can use schemes like a *buddy allocator* to manage the heap memory for the user. `malloc` can also be smarter about allocation and potentially use *anonymous mmaps* for extra process memory. This is where instead of mmaping a *file* into the process memory it directly maps an area of system RAM. This can be more efficient. Due to the complexity of managing memory correctly, it is very uncommon for any modern program to have a reason to call `brk` directly.

[3]Note we used the special flag to gcc `-fomit-frame-pointer` which specifies that an extra register should *not* be used to keep a pointer to the start of the stack frame. Having this pointer helps debuggers to walk upwards through the stack frames, however it makes one less register available for other applications.

Memory Layout

Figure 5.3. Process memory layout



As we have seen a process has smaller areas of memory allocated to it, each with a specific purpose.

An example of how the process is laid out in memory by the kernel is given above. Starting from the top, the kernel reserves its self some memory at the top of the process (we see with virtual memory how this memory is actually shared between all processes).

Underneath that is room for `mmaped` files and libraries. Underneath that is the stack, and below that the heap.

At the bottom is the program image, as loaded from the executable file on disk. We take a closer look at the process of loading this data in later chapters.

File Descriptors

In the first week we learnt about `stdin`, `stdout` and `stderr`; the default files given to each process. You will remember that these files always have the same file descriptor number (0,1,2 respectively).

Thus, file descriptors are kept by the kernel individually for each process.

File descriptors also have permissions. For example, you may be able to read from a file but not write to it. When the file is opened, the operating system keeps a record of the processes permissions to that file in the file descriptor and doesn't allow the process to do anything it shouldn't.

Registers

We know from the previous chapter that the processor essentially performs generally simple operations on values in registers. These values are read (and written) to memory -- we mentioned above that each process is allocated memory which the kernel keeps track of.

So the other side of the equation is keeping track of the registers. When it comes time for the currently running process to give up the processor so another process can run, it needs to save it's current state. Equally, we need to be able to restore this state when the process is given more time to run on the CPU. To do this the operating system needs to store a copy of the CPU registers to memory. When it is time for the process to run again, the operating system will copy the register values back from memory to the CPU registers and the process will be right back where it left off.

Kernel State

Internally, the kernel needs to keep track of a number of elements for each process.

Process State

Another important element for the operating system to keep track of is the process state. If the process is currently running it makes sense to have it in a *running* state.

However, if the process has requested to read a file from disk we know from our memory hierarchy that this may take a significant amount of time. The process should give up it's current execution to allow another process to run, but the kernel need not let the process run again until the data from the disk is available in memory. Thus it can mark the process as *disk wait* (or similar) until the data is ready.

Priority

Some processes are more important than others, and get a higher priority. See the discussion on the scheduler below.

Statistics

The kernel can keep statistics on each processes behaviour which can help it make decisions about how the process behaves; for example does it mostly read from disk or does it mostly do CPU intensive operations?

Process Hierarchy

Whilst the operating system can run many processes at the same time, in fact it only ever directly starts one process called the *init* (short for initial) process. This isn't a particularly special process except that it's PID is always 0 and it will *always* be running.

All other processes can be considered *children* of this initial process. Processes have a family tree just like any other; each process has a *parent* and can have many *siblings*, which are processes created[4] by the same parent.

Certainly children can create more children and so on and so forth.

Example 5.2. pstree example

```
 1
          init-+-apmd
        |-atd
        |-cron
 5      ...
        |-dhclient
        |-firefox-bin-+-firefox-bin---2*[firefox-bin]
        |             |-java_vm---java_vm---13*[java_vm]
        |             `-swf_play
10
```

Fork and Exec

New processes are created by the two related interfaces `fork` and `exec`.

Fork

When you come to metaphorical "fork in the road" you generally have two options to take, and your decision effects your future. Computer programs reach this fork in the road when they hit the `fork()` system call.

At this point, the operating system will create a new process that is exactly the same as the parent process. This means all the state that was talked about previously is

[4]The term *spawn* is often used when talking about parent processes creating children; as in "the process spawned a child".

copied, including open files, register state and all memory allocations, which includes the program code.

The return value from the system call is the only way the process can determine if it was the existing process or a new one. The return value to the parent process will be the Process ID (PID) of the child, whilst the child will get a return value of 0.

At this point, we say the process has `forked` and we have the parent-child relationship as described above.

Exec

Forking provides a way for an existing process to start a new one, but what about the case where the new process is not part of the same program as parent process? This is the case in the shell; when a user starts a command it needs to run in a new process, but it is unrelated to the shell.

This is where the `exec` system call comes into play. `exec` will *replace* the contents of the currently running process with the information from a program binary.

Thus the process the shell follows when launching a new program is to firstly `fork`, creating a new process, and then `exec` (i.e. load into memory and execute) the program binary it is supposed to run.

How Linux actually handles fork and exec

`clone`

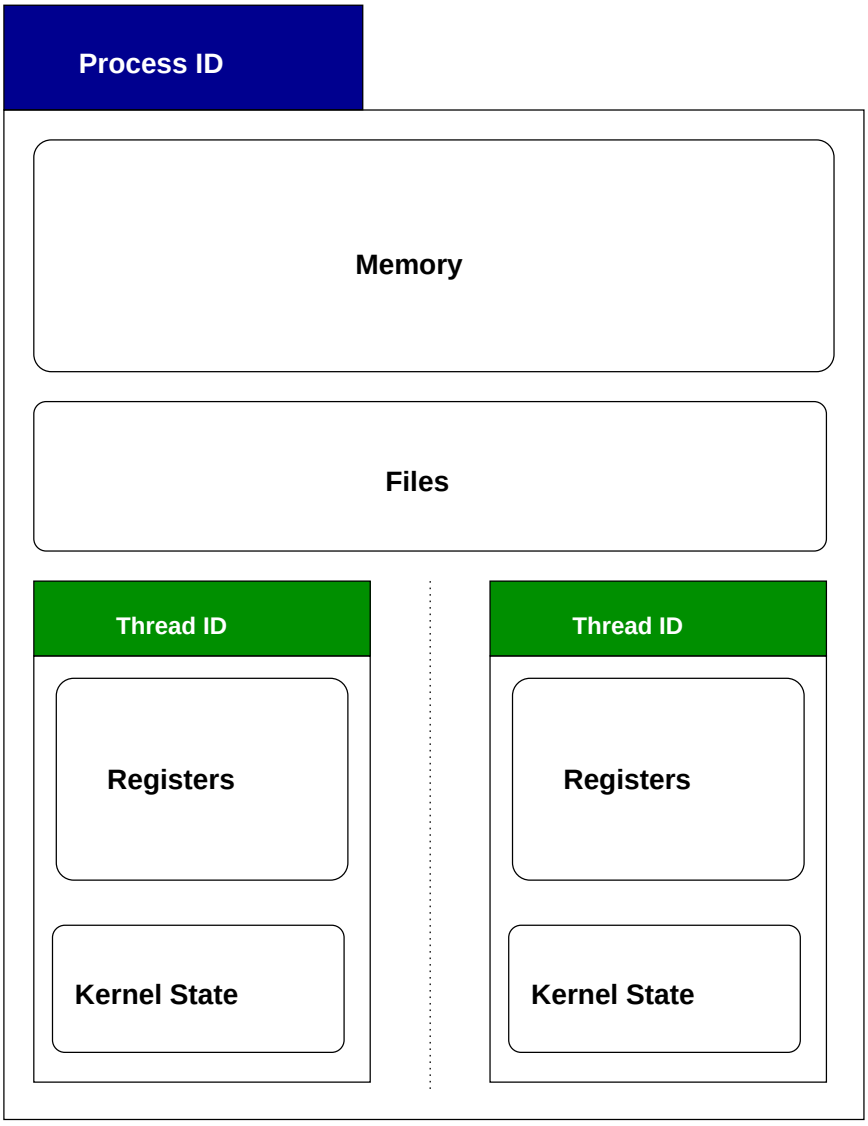
In the kernel, fork is actually implemented by a `clone` system call. This `clone` interfaces effectively provides a level of abstraction in how the Linux kernel can create processes.

`clone` allows you to explicitly specify which parts of the new process are copied into the new process, and which parts are shared between the two processes. This may seem a bit strange at first, but allows us to easily implement *threads* with one very simple interface.

Threads

While `fork` copies all of the attributes we mentioned above, imagine if everything was copied for the new process *except* for the memory. This means the parent and child share the same memory, which includes program code and data.

Figure 5.4. Threads



This hybrid child is called a *thread*. Threads have a number of advantages over where you might use *fork*

1. Separate processes can not see each others memory. They can only communicate with each other via other system calls.

   Threads however, share the same memory. So you have the advantage of multiple processes, with the expense of having to use system calls to communicate between them.

   The problem that this raises is that threads can very easily step on each others toes. One thread might increment a variable, and another may decrease it without informing the first thread. These type of problems are called *concurrency problems* and they are many and varied.

To help with this, there are userspace libraries that help programmers work with threads properly. The most common one is called `POSIX threads` or, as it more commonly referred to `pthreads`

2. Switching processes is quite expensive, and one of the major expenses is keeping track of what memory each process is using. By sharing the memory this overhead is avoided and performance can be significantly increased.

There are many different ways to implement threads. On the one hand, a userspace implementation could implement threads within a process without the kernel having any idea about it. The threads all look like they are running in a single process to the kernel.

This is suboptimal mainly because the kernel is being withheld information about what is running in the system. It is the kernels job to make sure that the system resources are utilised in the best way possible, and if what the kernel thinks is a single process is actually running multiple threads it may make suboptimal decisions.

Thus the other method is that the kernel has full knowledge of the thread. Under Linux, this is established by making all processes able to share resources via the `clone` system call. Each thread still has associated kernel resources, so the kernel can take it into account when doing resource allocations.

Other operating systems have a hybrid method, where some threads can be specified to run in userspace only ("hidden" from the kernel) and others might be a *light weight process*, a similar indication to the kernel that the processes is part of a thread group.

Copy on write

As we mentioned, copying the entire memory of one process to another when `fork` is called is an expensive operation.

One optimisation is called *copy on write*. This means that similar to threads above, the memory is actually shared, rather than copied, between the two processes when fork is called. If the processes are only going to be reading the memory, then actually copying the data is unnecessary.

However, when a process writes to it's memory, it needs to be a private copy that is not shared. As the name suggests, copy on write optimises this by only doing the actual copy of the memory at the point when it is written to.

Copy on write also has a big advantage for `exec`. Since `exec` will simply be overwriting all the memory with the new program, actually copying the memory would waste a lot of time. Copy on write saves us actually doing the copy.

The init process

We discussed the overall goal of the init process previously, and we are now in a position to understand how it works.

On boot the kernel starts the init process, which then forks and execs the systems boot scripts. These fork and exec more programs, eventually ending up forking a login process.

The other job of the `init` process is "reaping". When a process calls `exit` with a return code, the parent usually wants to check this code to see if the child exited correctly or not.

However, this exit code is part of the process which has just called `exit`. So the process is "dead" (e.g. not running) but still needs to stay around until the return code is collected. A process in this state is called a *zombie* (the traits of which you can contrast with a mystical zombie!)

A process stays as a zombie until the parent collects the return code with the `wait` call. However, if the parent exits before collecting this return code, the zombie process is still around, waiting aimlessly to give it's status to someone.

In this case, the zombie child will be *reparented* to the init process which has a special handler that *reaps* the return value. Thus the process is finally free and can the descriptor can be removed from the kernels process table.

Zombie example

Example 5.3. Zombie example process

```
 1
              $ cat zombie.c
   #include <stdio.h>
   #include <stdlib.h>
 5
   int main(void)
   {
           pid_t pid;

10         printf("parent : %d\n", getpid());

           pid = fork();

           if (pid == 0) {
15                 printf("child : %d\n", getpid());
                   sleep(2);
                   printf("child exit\n");
                   exit(1);
           }
20
           /* in parent */
           while (1)
           {
                   sleep(1);
25         }
   }

   ianw@lime:~$ ps ax | grep [z]ombie
   16168 pts/9    S      0:00 ./zombie
30 16169 pts/9    Z      0:00 [zombie] <defunct>
```

Above we create a zombie process. The parent process will sleep forever, whilst the child will exit after a few seconds.

Below the code you can see the results of running the program. The parent process (16168) is in state S for sleep (as we expect) and the child is in state Z for zombie. The ps output also tells us that the process is `defunct` in the process description.[5]

Context Switching

Context switching refers to the process the kernel undertakes to switch from one process to another. XXX ?

Scheduling

A running system has many processes, maybe even into the hundreds or thousands. The part of the kernel that keeps track of all these processes is called the *scheduler* because it schedules which process should be run next.

Scheduling algorithms are many and varied. Most users have different goals relating to what they want their computer to do, so this affects scheduling decisions. For example, for a desktop PC you want to make sure that your graphical applications for your desktop are given plenty of time to run, even if system processes take a little longer. This will increase the responsiveness the user feels, as their actions will have more immediate responses. For a server, you might want your web server application to be given priority.

People are always coming up with new algorithms, and you can probably think of your own fairly easily. But there are a number of different components of scheduling.

Preemptive v co-operative scheduling

Scheduling strategies can broadly fall into two categories

1. *Co-operative* scheduling is where the currently running process voluntarily gives up executing to allow another process to run. The obvious disadvantage of this is that the process may decide to never give up execution, probably because of a bug causing some form of infinite loop, and consequently nothing else can ever run.

2. *Preemptive* scheduling is where the process is interrupted to stop it an allow another process to run. Each process gets a *time-slice* to run in; at the point of each context switch a timer will be reset and will deliver and interrupt when the time-slice is over.

   We know that the hardware handles the interrupt independently of the running process, and so at this point control will return to the operating system. At this point, the scheduler can decide which process to run next.

   This is the type of scheduling used by all modern operating systems.

[5]The square brackets around the "z" of "zombie" are a little trick to remove the grep processes its self from the ps output. grep interprets everything between the square brackets as a character class, but because the process name will be "grep [z]ombie" (with the brackets) this will not match!

Realtime

Some processes need to know exactly how long their time-slice will be, and how long it will be before they get another time-slice to run. Say you have a system running a heart-lung machine; you don't want the next pulse to be delayed because something else decided to run in the system!

Hard realtime systems make guarantees about scheduling decisions like the maximum amount of time a process will be interrupted before it can run again. They are often used in life critical applications like medical, aircraft and military applications.

Soft realtime is a variation on this, where guarantees aren't as strict but general system behaviour is predictable. Linux can be used like this, and it is often used in systems dealing with audio and video. If you are recording an audio stream, you don't want to be interrupted for long periods of time as you will loose audio data which can not be retrieved.

Nice value

UNIX systems assign each process a *nice* value. The scheduler looks at the nice value and can give priority to those processes that have a higher "niceness".

A brief look at the Linux Scheduler

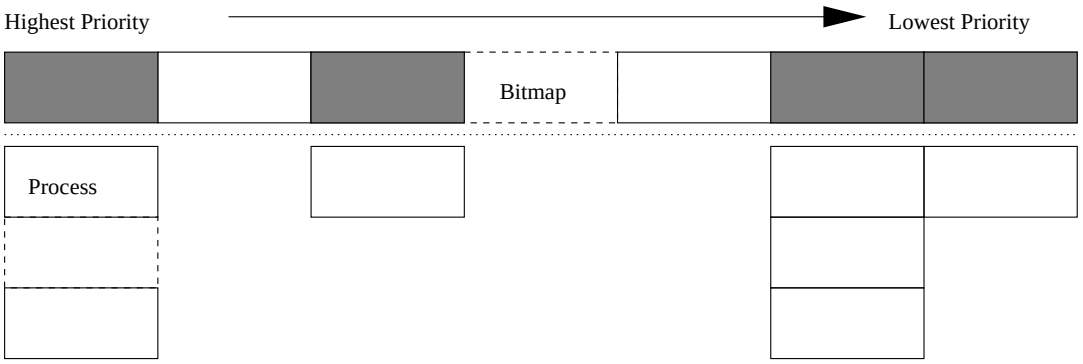
The Linux scheduler has and is constantly undergoing many changes as new developers attempt to improve its behaviour.

The current scheduler is known as the O(1) scheduler, which refers to the property that no many how many processes the scheduler has to choose from, it will choose the next one to run in a constant amount of time[6].

Previous incarnations of the Linux scheduler used the concept of *goodness* to determine which process to run next. All possible tasks are kept on a *run queue*, which is simply a linked list of processes which the kernel knows are in a "runnable" state (i.e. not waiting on disk activity or otherwise asleep). The problem arises that to calculate the next process to run, every possible runnable process must have its goodness calculated and the one with the highest goodness ``wins''. You can see that for more tasks, it will take longer and longer to decide which processes will run next.

[6] *Big-O* notation is a way of describing how long an algorithm takes to run given increasing inputs. If the algorithm takes twice as long to run for twice as much input, this is increasing linearly. If another algorithm takes four times as long to run given twice as much input, then it is increasing exponentially. Finally if it takes the same amount of time now matter how much input, then the algorithm runs in constant time. Intuitively you can see that the slower the algorithm grows for more input, the better it is. Computer science text books deal with algorithm analysis in more detail.

Figure 5.5. The O(1) scheduler



In contrast, the O(1) scheduler uses a run queue structure as shown above. The run queue has a number of *buckets* in priority order and a bitmap that flags which buckets have processes available. Finding the next process to run is a matter of reading the bitmap to find the first bucket with processes, then picking the first process off that bucket's queue. The scheduler keeps two such structures, an *active* and *expired* array for processes that are runnable and those which have utilised their entire time slice respectively. These can be swapped by simply modifying pointers when all processes have had some CPU time.

The really interesting part, however, is how it is decided where in the run queue a process should go. Some of the things that need to be taken into account are the nice level, processor affinity (keeping processes tied to the processor they are running on, since moving a process to another CPU in a SMP system can be an expensive operation) and better support for identifying interactive programs (applications such as a GUI which may spend much time sleeping, waiting for user input, but when the user *does* get around to interacting with it wants a fast response).

The Shell

On a UNIX system, the shell is the standard interface to handling processes on your system. Once the shell was the primary interface, however modern Linux systems have a GUI and provide a shell via a "terminal application" or similar. The primary job of the shell is to help the user handle starting, stopping and otherwise controlling processes running in the system.

When you type a command at the prompt of the shell, it will `fork` a copy of it's self and `exec` the command that you have specified.

The shell then, by default, waits for that process to finish running before returning to a prompt to start the whole process over again.

As an enhancement, the shell also allows you to *background* a job, usually by placing an & after the command name. This is simply a signal that the shell should fork and execute the command, but *not* wait for the command to complete before showing you the prompt again.

The new process runs in the background, and the shell is ready waiting to start a new process should you desire. You can usually tell the shell to *foreground* a process, which means we do actually want to wait for it to finish.

XXX : a bit of history about bourne shell

Signals

Processes running in the system require a way to be told about events that influence them. On UNIX there is infrastructure between the kernel and processes called *signals* which allows a process to receive notification about events important to it.

When a signal is sent to a process, the kernel invokes a *handler* which the process must register with the kernel to deal with that signal. A handler is simply a designed function in the code that has been written to specifically deal with interrupt. Often the signal will be sent from inside the kernel its self, however it is also common for one process to send a signal to another process (one form of *interprocess communication*). The signal handler gets called *asynchronously*; that is the currently running program is interrupted from what it is doing to process the signal event.

For example, one type of signal is an *interrupt* (defined in system headers as `SIGINT`) is delivered to the process when the `ctrl-c` combination is pressed.

As a process uses the `read` system call to read input from the keyboard, the kernel will be watching the input stream looking for special characters. Should it see a `ctrl-c` it will jump into signal handling mode. The kernel will look to see if the process has registered a handler for this interrupt. If it has, then execution will be passed to that function where the function will *handle* it. Should the process have not registered a handler for this particular signal, then the kernel will take some default action. With `ctrl-c` the default action is to terminate the process.

A process can choose to ignore some signals, but other signals are not allowed to be ignored. For example, `SIGKILL` is the signal sent when a process should be terminated. The kernel will see that the process has been sent this signal and terminate the process from running, no questions asked. The process can not ask the kernel to ignore this signal, and the kernel is very careful about which process is allowed to send this signal to another process; you may only send it to processes owned by you unless you are the root user. You may have seen the command `kill -9`; this comes from the implementation `SIGKILL` signal. It is commonly known that `SIGKILL` is actually defined to be `0x9`, and so when specified as an argument to the `kill` program means that the process specified is going to be stopped immediately. Since the process can not choose to ignore or handle this signal, it is seen as an avenue of last resort, since the program will have no chance to clean up or exit cleanly. It is considered better to first send a `SIGTERM` (for terminate) to the process first, and if it has crashed or otherwise will not exit then resort to the `SIGKILL`. As a matter of convention, most programs will install a handler for `SIGHUP` (hangup -- a left over from days of serial terminals and modems) which will reload the program, perhaps to pick up changes in a configuration file or similar.

If you have programmed on a Unix system you would be familiar with `segmentation faults` when you try to read or write to memory that has not been allocated to you. When the kernel notices that you are touching memory outside your allocation, it will send you the segmentation fault signal. Usually the process will not have a handler installed for this, and so the default action to terminate the program ensues (hence your program "crashes"). In some cases a program may install a handler for segmentation faults, although reasons for doing this are limited.

This raises the question of what happens after the signal is received. Once the signal handler has finished running, control is returned to the process which continues on from where it left off.

Example

The following simple program introduces a lot of signals to run!

Example 5.4. Signals Example

```
 1
            $ cat signal.c
   #include <stdio.h>
   #include <unistd.h>
 5 #include <signal.h>

   void sigint_handler(int signum)
   {
           printf("got SIGINT\n");
10 }

   int main(void)
   {
           signal(SIGINT, sigint_handler);
15         printf("pid is %d\n", getpid());
           while (1)
                   sleep(1);
   }
   $ gcc -Wall -o signal signal.c
20 $ ./signal
   pid is 2859
   got SIGINT # press ctrl-c
              # press ctrl-z
   [1]+  Stopped                 ./signal
25
   $ kill -SIGINT 2859
   $ fg
   ./signal
   got SIGINT
30 Quit # press ctrl-\

   $
```

We have simple program that simply defines a handler for the `SIGINT` signal, which is sent when the user presses `ctrl-c`. All the signals for the system are defined in `signal.h`, including the `signal` function which allows us to register the handling function.

The program simply sits in a tight loop doing nothing until it quits. When we start the program, we try pressing `ctrl-c` to make it quit. Rather than taking the default action, or handler is invoked and we get the output as expected.

We then press `ctrl-z` which sends a `SIGSTOP` which by default puts the process to sleep. This means it is not put in the queue for the scheduler to run and is thus dormant in the system.

As an illustration, we use the kill program to send the same signal from another terminal window. This is actually implemented with the `kill` system call, which takes a signal and PID to send to (this function is a little misnamed because not all signals do actually kill the process, as we are seeing, but the `signal` function was already taken to register the handler). As the process is stopped, the signal gets *queued* for the process. This means the kernel takes note of the signal and will deliver it when appropriate.

At this point we wake the process up by using the command `fg`. This actually sends a `SIGCONT` signal to the process, which by default will wake the process back up. The kernel knows to put the process on the run queue and give it CPU time again. We see at this point the queued signal is delivered.

In desperation to get rid of the program, we finally try `ctrl-\` which sends a `SIGABRT` (abort) to the process. But if the process has aborted, where did the `Quit` output come from?

You guessed it, more signals! When a parent child has a process that dies, it gets a `SIGCHLD` signal back. In this case the shell was the parent process and so it got the signal. Remember how we have the zombie process that needs to be reaped with the `wait` call to get the return code from the child process? Well another thing it also gives the parent is the signal number that the child may have died from. Thus the shell knows that child process died from a `SIGABRT` and as an informational service prints as much for the user (the same process happens to print out "Segmentation Fault" when the child process dies from a `SIGSEGV`).

You can see how in even a simple program, around 5 different signals were used to communicate between processes and the kernel and keep things running. There are many other signals, but these are certainly amongst the most common. Most have system functions defined by the kernel, but there are a few signals reserved for users to use for their own purposes within their programs (`SIGUSR`).

Chapter 6. Virtual Memory

What Virtual Memory *isn't*

Virtual memory is often naively discussed as a way to extended your RAM by using the hard drive as extra, slower, system memory. That is, once your system runs out of memory, it flows over onto the hard drive which is used as "virtual" memory.

In modern operating systems, this is commonly referred to as *swap space*, because unused parts of memory as swapped out to disk to free up main memory (remember, programs can only execute from main memory).

Indeed, the ability to swap out memory to disk is an important capability, but as you will see it is not the purpose of virtual memory, but rather a very useful side effect!

What virtual memory *is*

Virtual memory is all about making use of *address space*.

The address space of a processor refers the range of possible addresses that it can use when loading and storing to memory. The address space is limited by the width of the registers, since as we know to load an address we need to issue a `load` instruction with the address to load from stored in a register. For example, registers that are 32 bits wide can hold addresses in a register range from `0x00000000` to `0xFFFFFFFF`. 2^{32} is equal to 4GB, so a 32 bit processor can load or store to up to 4GB of memory.

64 bit computing

New processors are generally all 64-bit processors, which as the name suggests has registers 64 bits wide. As an exercise, you should work out the address space available to these processors (hint: it's big!).

64-bit computing does have some trade-offs against using smaller bit-width processors. Every program compiled in 64-bit mode requires 8-byte pointers, which can increase code and data size, and hence impact both instruction and data cache performance. However, 64-bit processors tend to have more registers, which means less need to save temporary variables to memory when the compiler is under register pressure.

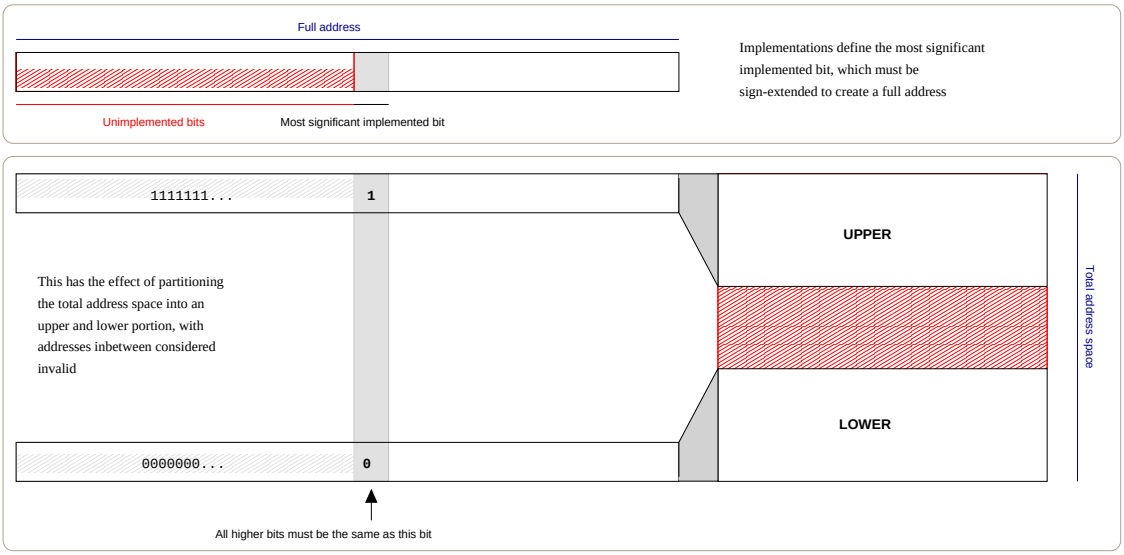
Canonical Addresses

While 64-bit processors have 64-bit wide registers, systems generally do not implement all 64-bits for addressing — it is not actually possible to do `load` or `store` to all 16 exabytes of theoretical physical memory!

Thus most architectures define an *unimplemented* region of the address space which the processor will consider invalid for use. x86-64 and Itanium both define the most-significant valid bit of an address, which must then be sign-extended (see the section called “Sign-extension”) to create a valid address. The result of this is that the total

address space is effectively divided into two parts, an upper and a lower portion, with the addresses in-between considered invalid. This is illustrated in Figure 6.1, “Illustration of canonical addresses”. Valid addresses are termed *canonical addresses* (invalid addresses being *non*-canonical).

Figure 6.1. Illustration of canonical addresses



The exact most-significant bit value for the processor can usually be found by querying the processor itself using its informational instructions. Although the exact value is implementation dependent, a typical value would be 48; providing 2^{48} = 256 TiB of usable address-space.

Reducing the possible address-space like this means that significant savings can be made with all parts of the addressing logic in the processor and related components, as they know they will not need to deal with full 64-bit addresses. Since the implementation defines the upper-bits as being signed-extended, this prevents portable operating systems using these bits to store or flag additional information and ensuring compatibility if the implementation wishes to implement more address-space in the future.

Using the address space

As with most components of the operating system, virtual memory acts as an abstraction between the address space and the physical memory available in the system. This means that when a program uses an address that address does not refer to the bits in an actual physical location in memory.

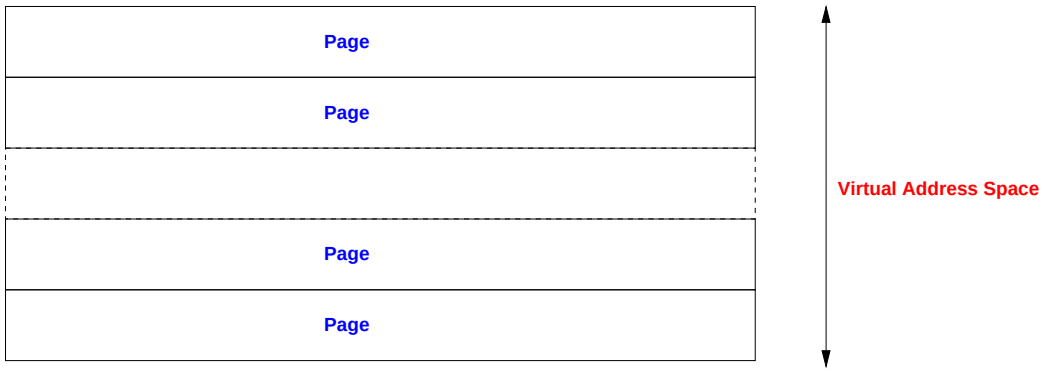
So to this end, we say that all addresses a program uses are *virtual*. The operating system keeps track of virtual addresses and how they are allocated to *physical* addresses. When a program does a load or store from an address, the processor and operating system work together to convert this virtual address to the actual address in the system memory chips.

Pages

The total address-space is divided into individual *pages*. Pages can be many different sizes; generally they are around 4 KiB, but this is not a hard and fast rule and they can be much larger but generally not any smaller. The page is the smallest unit of memory that the operating system and hardware can deal with.

Additionally, each page has a number of attributes set by the operating system. Generally, these include read, write and execute permissions for the current page. For example, the operating system can generally mark the code pages of a process with an executable flag and the processor can choose to not execute any code from pages without this bit set.

Figure 6.2. Virtual memory pages



Programmers may at this point be thinking that they can easily allocate small amounts of memory, much smaller than 4 KiB, using `malloc` or similar calls. This *heap* memory is actually backed by page-size allocations, which the `malloc` implementation divides up and manages for you in an efficient manner.

Physical Memory

Just as the operating system divides the possible address space up into pages, it divides the available physical memory up into *frames*. A frame is just the conventional name for a hunk of physical memory the same size as the system page size.

The operating system keeps a *frame-table* which is a list of all possible pages of physical memory and if they are free (available for allocation) or not. When memory is allocated to a process, it is marked as used in the frame-table. In this way, the operating-system keeps track of all memory allocations.

How does the operating system know what memory is available? This information about where memory is located, how much, attributes and so forth is passed to the operating system by the BIOS during initialisation.

Pages + Frames = Page Tables

It is the job of the operating system is to keep track of which of virtual-page points to which physical frame. This information is kept in a *page-table* which, in its simplest form, could simply be a table where each row contains its associated frame — this is termed a *linear page-table*. If you were to use this simple system, with a 32 bit address-space and 4 KiB pages there would be 1048576 possible pages to keep track of in the page table ($2^{32} \div 4096$); hence the table would be 1048576 entries long to ensure we can always map a virtual page to a physical page.

Page tables can have many different structures and are highly optimised, as the process of finding a page in the page table can be a lengthy process. We will examine page-tables in more depth later.

The page-table for a process is under the exclusive control of the operating system. When a process requests memory, the operating system finds it a free page of physical memory and records the virtual-to-physical translation in the processes page-table. Conversely, when the process gives up memory, the virtual-to-physical record is removed and the underlying frame becomes free for allocation to another process.

Virtual Addresses

When a program accesses memory, it does not know or care where the physical memory backing the address is stored. It knows it is up to the operating system and hardware to work together to map locate the right physical address and thus provide access to the data it wants. Thus we term the address a program is using to access memory a *virtual address*. A virtual address consists of two parts; the page and an offset into that page.

Page

Since the entire possible address space is divided up into regular sized pages, every possible address resides within a page. The page component of the virtual address acts as an index into the page table. Since the page is the smallest unit of memory allocation within the system there is a trade-off between making pages very small, and thus having very many pages for the operating-system to manage, and making pages larger but potentially wasting memory

Offset

The last bits of the virtual address are called the *offset* which is the location difference between the byte address you want and the start of the page. You require enough bits in the offset to be able to get to any byte in the page. For a 4K page you require (4K == (4 * 1024) == 4096 == 2^{12} ==) 12 bits of offset. Remember that the smallest amount of memory that the operating system or hardware deals with is a page, so each of these 4096 bytes reside within a single page and are dealt with as "one".

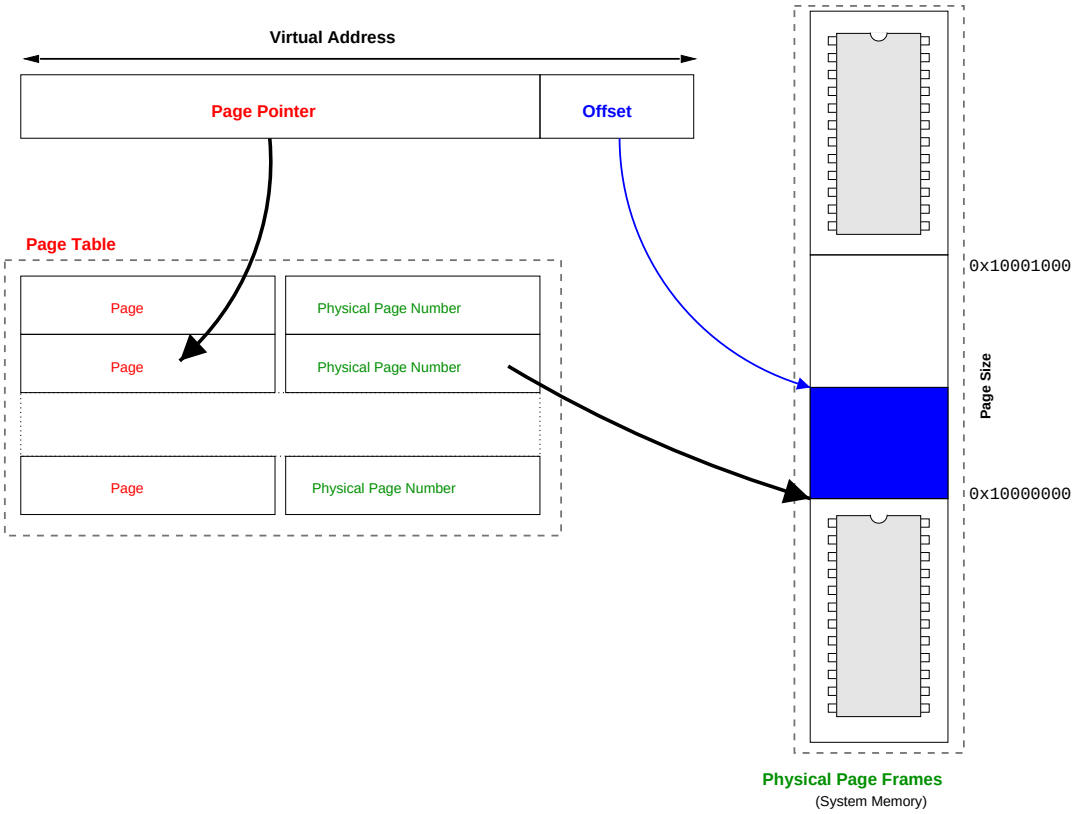
Virtual Address Translation

Virtual address translation refers to the process of finding out which physical page maps to which virtual page.

When translating a virtual-address to a physical-address we only deal with the *page number* . The essence of the procedure is to take the page number of the given address and look it up in the *page-table* to find a pointer to a physical address, to which the offset from the virtual address is added, giving the actual location in system memory.

Since the page-tables are under the control of the operating system, if the virtual-address doesn't exist in the page-table then the operating-system knows the process is trying to access memory that has not been allocated to it and the access will not be allowed.

Figure 6.3. Virtual Address Translation



We can follow this through for our previous example of a simple *linear* page-table. We calculated that a 32-bit address-space would require a table of 1048576 entries when using 4KiB pages. Thus to map a theoretical address of 0x80001234, the first step would be to remove the offset bits. In this case, with 4KiB pages, we know we have 12-bits (2^{12} == 4096) of offset. So we would right-shift out 12-bits of the virtual address, leaving us with 0x80001. Thus (in decimal) the value in row 524289 of the linear page table would be the physical frame corresponding to this page.

You might see a problem with a linear page-table : since every page must be accounted for, whether in use or not, a physically linear page-table is completely impractical with a 64-bit address space. Consider a 64-bit address space divided into (generously large) 64 KiB pages creates $2^{64}/2^{16} = 2^{52}$ pages to be managed; assuming each page requires an 8-byte pointer to a physical location a total of $2^{52}/2^{3} = 2^{49}$ or 512 GiB of contiguous memory is required just for the page table!

Consequences of virtual addresses, pages and page tables

Virtual addressing, pages and page-tables are the basis of every modern operating system. It under-pins most of the things we use our systems for.

Individual address spaces

By giving each process its own page table, every process can pretend that it has access to the entire address space available from the processor. It doesn't matter that two processes might use the same address, since different page-tables for each process will map it to a different frame of physical memory. Every modern operating system provides each process with its own address space like this.

Over time, physical memory becomes *fragmented*, meaning that there are "holes" of free space in the physical memory. Having to work around these holes would be at best annoying and would become a serious limit to programmers. For example, if you `malloc` 8 KiB of memory; requiring the backing of two 4 KiB frames, it would be a huge unconvinced if those frames had to be contiguous (i.e., physically next to each other). Using virtual-addresses it does not matter; as far as the process is concerned it has 8 KiB of contiguous memory, even if those pages are backed by frames very far apart. By assigning a virtual address space to each process the programmer can leave working around fragmentation up to the operating system.

Protection

We previously mentioned that the virtual mode of the 386 processor is called protected mode, and this name arises from the protection that virtual memory can offer to processes running on it.

In a system without virtual memory, every process has complete access to all of system memory. This means that there is nothing stopping one process from overwriting another processes memory, causing it to crash (or perhaps worse, return incorrect values, especially if that program is managing your bank account!)

This level of protection is provided because the operating system is now the layer of abstraction between the process and memory access. If a process gives a virtual address that is not covered by its page-table, then the operating system knows that that process is doing something wrong and can inform the process it has stepped out of its bounds.

Since each page has extra attributes, a page can be set read only, write only or have any number of other interesting properties. When the process tries to access the page, the operating system can check if it has sufficient permissions and stop it if it does not (writing to a read only page, for example).

Systems that use virtual memory are inherently more stable because, assuming the perfect operating system, a process can only crash itself and not the entire system (of course, humans write operating systems and we inevitably overlook bugs that can still cause entire systems to crash).

Swap

We can also now see how the swap memory is implemented. If instead of pointing to an area of system memory the page pointer can be changed to point to a location on a disk.

When this page is referenced, the operating system needs to move it from the disk back into system memory (remember, program code can only execute from system memory). If system memory is full, then *another* page needs to be kicked out of system memory and put into the swap disk before the required page can be put in memory. If another process wants that page that was just kicked out back again, the process repeats.

This can be a major issue for swap memory. Loading from the hard disk is very slow (compared to operations done in memory) and most people will be familiar with sitting in front of the computer whilst the hard disk churns and churns whilst the system remains unresponsive.

mmap

A different but related process is the memory map, or `mmap` (from the system call name). If instead of the page table pointing to physical memory or swap the page table points to a file, on disk, we say the file is `mmaped`.

Normally, you need to `open` a file on disk to obtain a file descriptor, and then `read` and `write` it in a sequential form. When a file is mmaped it can be accessed just like system RAM.

Sharing memory

Usually, each process gets its own page table, so any address it uses is mapped to a unique frame in physical memory. But what if the operating system points two page table-entries to the same frame? This means that this frame will be shared; and any changes that one process makes will be visible to the other.

You can see now how threads are implemented. In the section called “ `clone` ” we said that the Linux `clone()` function could share as much or as little of a new process with the old process as it required. If a process calls `clone()` to create a new process, but requests that the two processes share the same page table, then you effectively have a *thread* as both processes see the same underlying physical memory.

You can also see now how copy on write is done. If you set the permissions of a page to be read-only, when a process tries to write to the page the operating system will

be notified. If it knows that this page is a copy-on-write page, then it needs to make a new copy of the page in system memory and point the page in the page table to this new page. This can then have its attributes updated to have write permissions and the process has its own unique copy of the page.

Disk Cache

In a modern system, it is often the case that rather than having too little memory and having to swap memory out, there is more memory available than the system is currently using.

The memory hierarchy tells us that disk access is much slower than memory access, so it makes sense to move as much data from disk into system memory if possible.

Linux, and many other systems, will copy data from files on disk into memory when they are used. Even if a program only initially requests a small part of the file, it is highly likely that as it continues processing it will want to access the rest of file. When the operating system has to read or write to a file, it first checks if the file is in it's memory cache.

These pages should be the first to be removed as memory pressure in the system increases.

Page Cache

A term you might hear when discussing the kernel is the *page cache*.

The *page cache* refers to a list of pages the kernel keeps that refer to files on disk. From above, swap page, mmaped pages and disk cache pages all fall into this category. The kernel keeps this list because it needs to be able to look them up quickly in response to read and write requests XXX: this bit doesn't file?

Hardware Support

So far, we have only mentioned that hardware works with the operating system to implement virtual memory. However we have glossed over the details of exactly how this happens.

Virtual memory is necessarily quite dependent on the hardware architecture, and each architecture has its own subtleties. However, there are are a few universal elements to virtual memory in hardware.

Physical v Virtual Mode

All processors have some concept of either operating in *physical* or *virtual* mode. In physical mode, the hardware expects that any address will refer to an address in actual system memory. In virtual mode, the hardware knows that addresses will need to be translated to find their physical address.

In many processors, this two modes are simply referred to as physical and virtual mode. Itanium is one such example. The most common processor, the x86, has a lot of baggage from days before virtual memory and so the two modes are referred to as *real*

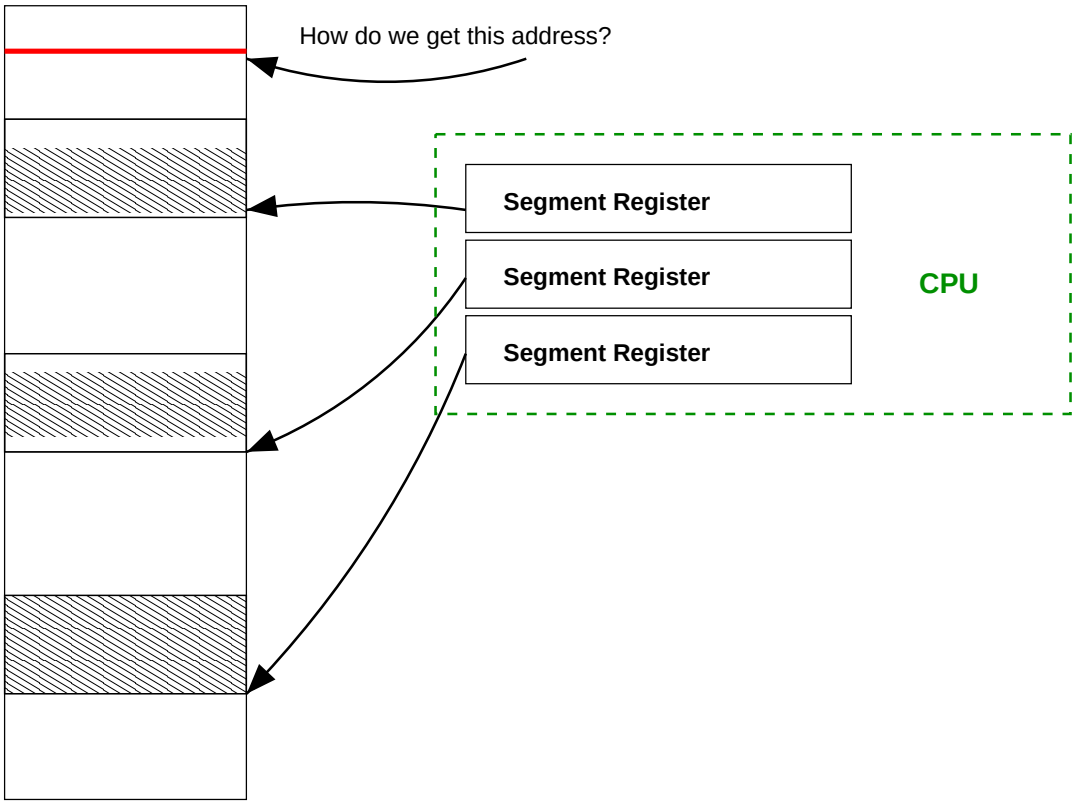
and *protected* mode. The first processor to implement protected mode was the 386, and even the most modern processors in the x86 family line can still do real mode, though it is not used. In real mode the processor implements a form of memory organisation called segmentation.

Issues with segmentation

Segmentation is really only interesting as a historical note, since virtual memory has made it less relevant. Segmentation has a number of drawbacks, not the least of which it is very confusing for inexperienced programmers, which virtual memory systems were largely invented to get around.

In segmentation there are a number of registers which hold an address that is the start of a segment. The only way to get to an address in memory is to specify it as an offset from one of these segment registers. The size of the segment (and hence the maximum offset you can specify) is determined by the number of bits available to offset from segment base register. In the x86, the maximum offset is 16 bits, or only 64K[1] . This causes all sorts of havoc if one wants to use an address that is more than 64K away, which as memory grew into the megabytes (and now gigabytes) became more than a slight inconvenience to a complete failure.

Figure 6.4. Segmentation



[1]Imagine that the maximum offset was 32 bits; in this case the entire address space could be accessed as an offset from a segment at `0x00000000` and you would essentially have a flat layout -- but it still isn't as good as virtual memory as you will see. In fact, the only reason it is 16 bits is because the original Intel processors were limited to this, and the chips maintain backwards compatibility.

In the above figure, there are three segment registers which are all pointing to segments. The maximum offset (constrained by the number of bits available) is shown by shading. If the program wants an address outside this range, the segment registers must be reconfigured. This quickly becomes a major annoyance. Virtual memory, on the other hand, allows the program to specify any address and the operating system and hardware do the hard work of translating to a physical address.

The TLB

The *Translation Lookaside Buffer* (or TLB for short) is the main component of the processor responsible for virtual-memory. It is a cache of virtual-page to physical-frame translations inside the processor. The operating system and hardware work together to manage the TLB as the system runs.

Page Faults

When a virtual address is requested of the hardware — say via a `load` instruction requesting to get some data — the processor looks for the virtual-address to physical-address translation in its TLB. If it has a valid translation it can then combine this with the offset portion to go straight to the physical address and complete the load.

However, if the processor can *not* find a translation in the TLB, the processor must raise a *page fault*. This is similar to an interrupt (as discussed before) which the operating system must handle.

When the operating system gets a page fault, it needs to go through it's page-table to find the correct translation and insert it into the TLB.

In the case that the operating system can not find a translation in the page table, or alternatively if the operating system checks the permissions of the page in question and the process is not authorised to access it, the operating system must kill the process. If you have ever seen a segmentation fault (or a segfault) this is the operating system killing a process that has overstepped its bounds.

Should the translation be found, and the TLB currently be full, then one translation needs to be removed before another can be inserted. It does not make sense to remove a translation that is likely to be used in the future, as you will incur the cost of finding the entry in the page-tables all over again. TLBs usually use something like a *Least Recently Used* or LRU algorithm, where the oldest translation that has not been used is ejected in favour of the new one.

The access can then be tried again, and, all going well, should be found in the TLB and translated correctly.

Finding the page table

When we say that the operating system finds the translation in the page table, it is logical to ask how the operating system finds the memory that has the page table.

The base of the page table will be kept in a register associated with each process. This is usually called the page-table base-register or similar. By taking the address in this register and adding the page number to it, the correct entry can be located.

Other page related faults

There are two other important faults that the TLB can generally generate which help to mange accessed and dirty pages. Each page generally contains an attribute in the form of a single bit which flags if the page has been accessed or is dirty.

An accessed page is simply any page that has been accessed. When a page translation is initially loaded into the TLB the page can be marked as having been accessed (else why were you loading it in?[2])

The operating system can periodically go through *all* the pages and clear the accessed bit to get an idea of what pages are currently in use. When system memory becomes full and it comes time for the operating system to choose pages to be swapped out to disk, obviously those pages whose accessed bit has not been reset are the best candidates for removal, because they have not been used the longest.

A dirty page is one that has data written to it, and so does not match any data already on disk. For example, if a page is loaded in from swap and then written to by a process, before it can be moved out of swap it needs to have its on disk copy updated. A page that is clean has had no changes, so we do not need the overhead of copying the page back to disk.

Both are similar in that they help the operating system to manage pages. The general concept is that a page has two extra bits; the dirty bit and the accessed bit. When the page is put into the TLB, these bits are set to indicate that the CPU should raise a fault .

When a process tries to reference memory, the hardware does the usual translation process. However, it also does an extra check to see if the accessed flag is *not* set. If so, it raises a fault to the operating system, which should set the bit and allow the process to continue. Similarly if the hardware detects that it is writing to a page that does not have the dirty bit set, it will raise a fault for the operating system to mark the page as dirty.

TLB Management

We can say that the TLB used by the hardware but managed by software. It is up to the operating system to load the TLB with correct entries and remove old entries.

Flushing the TLB

The process of removing entries from the TLB is called *flushing*. Updating the TLB is a crucial part of maintaining separate address spaces for processes; since each process can be using the same virtual address not updating the TLB would mean a process might end up overwriting another processes memory (conversely, in the case of *threads* sharing the address-space is what you want, thus the TLB is *not* flushed when switching between threads in the same process).

[2]Actually, if you were loading it in without a pending access this would be called *speculation*, which is where you do something with the expectation that it will pay off. For example, if code was reading along memory linearly putting the next page translation in the TLB might save time and give a performance improvement.

On some processors, every time there is a context switch the entire TLB is flushed. This can be quite expensive, since this means the new process will have to go through the whole process of taking a page fault, finding the page in the page tables and inserting the translation.

Other processors implement an extra *address space ID* (ASID) which is added to each TLB translation to make it unique. This means each address space (usually each process, but remember threads want to share the same address space) gets its own ID which is stored along with any translations in the TLB. Thus on a context switch the TLB does *not* need to be flushed, since the next process will have a different address space ID and even if it asks for the same virtual address, the address space ID will differ and so the translation to physical page will be different. This scheme reduces flushing and increases overall system performance, but requires more TLB hardware to hold the ASID bits.

Generally, this is implemented by having an additional register as part of the process state that includes the ASID. When performing a virtual-to-physical translation, the TLB consults this register and will only match those entries that have the same ASID as the currently running process. Of course the width of this register determines the number of ASID's available and thus has performance implications. For an example of ASID's in a processor architecture see the section called “Address spaces”.

Hardware v Software loaded TLB

While the control of what ends up in the TLB is the domain of the operating system; it is not the whole story. The process described in the section called “Page Faults” describes a page-fault being raised to the operating system, which traverses the page-table to find the virtual-to-physical translation and installs it in the TLB. This would be termed a *software-loaded TLB* — but there is another alternative; the *hardware-loaded TLB*.

In a hardware loaded TLB, the processor architecture defines a particular layout of page-table information (the section called “Pages + Frames = Page Tables” which must be followed for virtual address translation to proceed. In response to access to a virtual-address that is not present in the TLB, the processor will automatically walk the page-tables to load the correct translation entry. Only if the translation entry does not exist will the processor raise an exception to be handled by the operating system.

Implementing the page-table traversal in specialised hardware gives speed advantages when finding translations, but removes flexibility from operating-systems implementors who might like to implement alternative schemes for page-tables.

All architectures can be broadly categorised into these two methodologies. Later, we will examine some common architectures and their virtual-memory support.

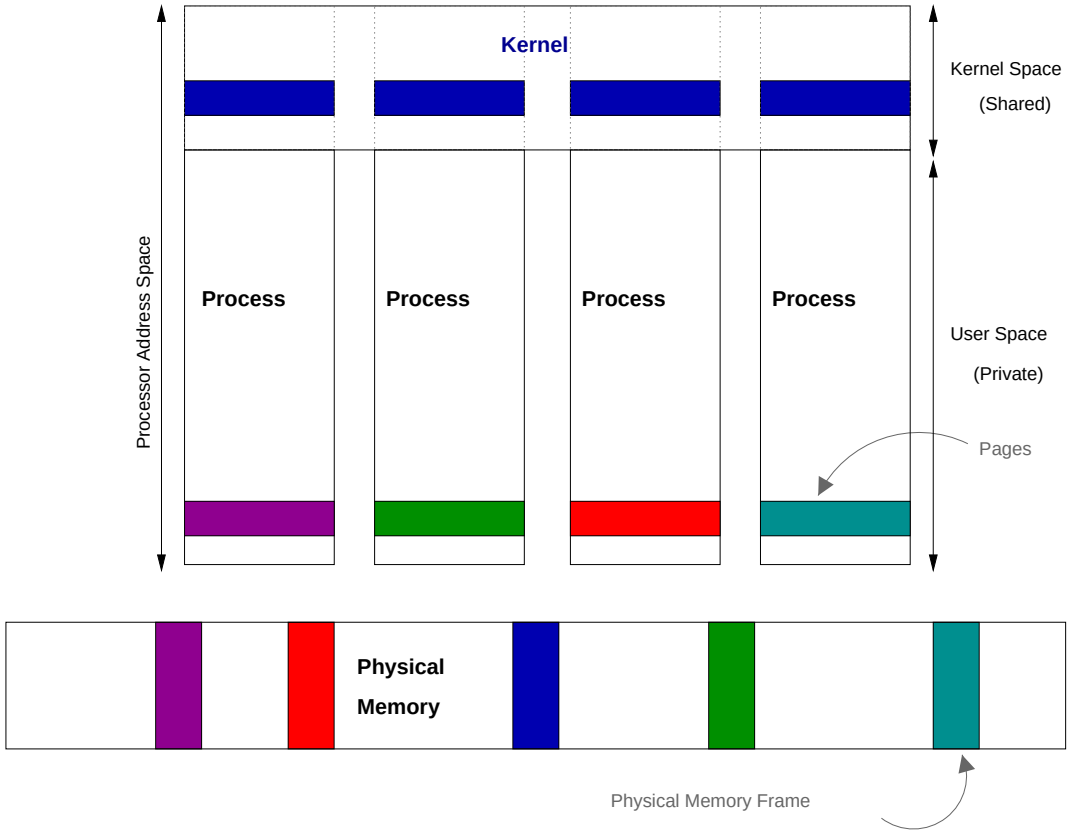
Linux Specifics

Although the basic concepts of virtual memory remain constant, the specifics of implementations are highly dependent on the operating system and hardware.

Address Space Layout

Linux divides the available address space up into a shared kernel component and private user space addresses. This means that addresses in the kernel port of the address space map to the same physical memory for each process, whilst user-space addresses are private to the process. On Linux, the shared kernel space is at the very top of the available address space. On the most common processor, the 32 bit x86, this split happens at the 3GB point. As 32 bits can map a maximum of 4GB, this leaves the top 1GB for the shared kernel region[3].

Figure 6.5. Linux address space layout



Three Level Page Table

There are many different ways for an operating system to organise the page tables but Linux chooses to use a *hierarchical* system.

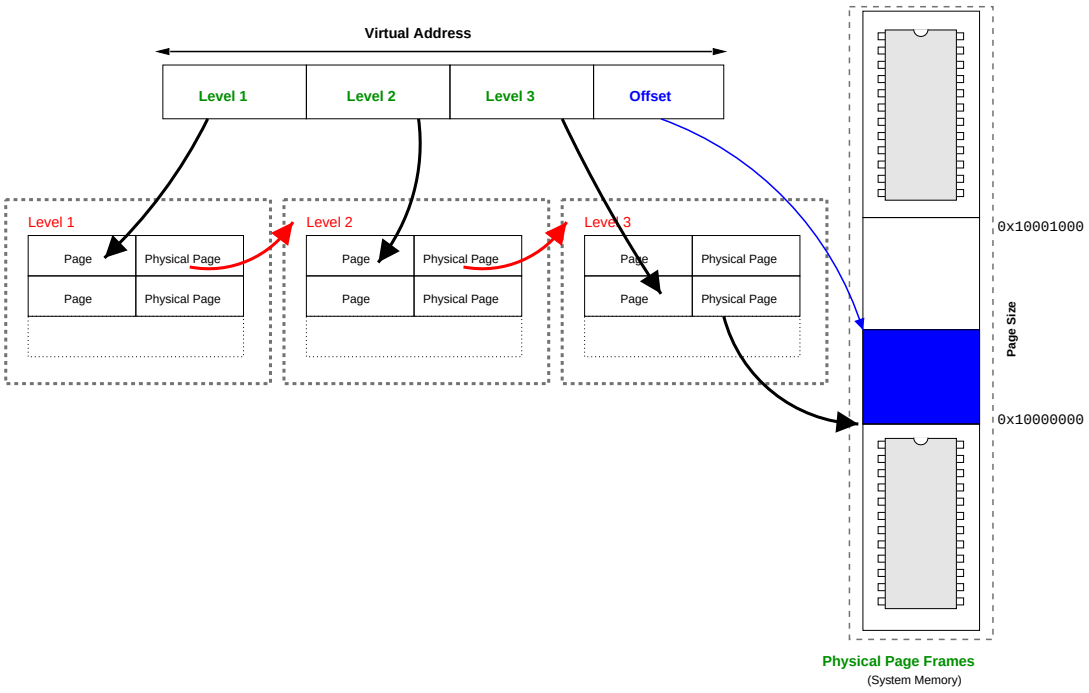
As the page tables use a hierarchy that is three levels deep, the Linux scheme is most commonly referred to as the *three level page table*. The three level page table has proven to be robust choice, although it is not without it's criticism. The details of the virtual memory implementation of each processor vary Whitley meaning that the generic page table Linux chooses must be portable and relatively generic.

[3]This is unfortunately an over-simplification, because many machines wanted to support more than 4GB per process. *High memory* support allows processors to get access to a full 4GB via special extensions.

The concept of the three level page table is not difficult. We already know that a virtual address consists of a page number and an offset in the physical memory page. In a three level page table, the virtual address is further split up into a number *levels*.

Each level is a page table of it's own right; i.e. it maps a page number of a physical page. In a single level page table the "level 1" entry would directly map to the physical frame. In the multilevel version each of the upper levels gives the address of the physical memory frame holding the next lower levels page table.

Figure 6.6. Linux Three Level Page Table



So a sample reference involves going to the top level page table, finding the physical frame that the next level address is on, reading that levels table and finding the physical frame that the next levels page table lives on, and so on.

At first, this model seems to be needlessly complex. The main reason this model is implemented is for size considerations. Imagine the theoretical situation of a process with only one single page mapped right near the end of it's virtual address space. We said before that the page table entry is found as an offset from the page table base register, so the page table needs to be a contiguous array in memory. So the single page near the end of the address space requires the entire array, which might take up considerable space (many, many physical pages of memory).

In a three level system, the first level is only one physical frame of memory. This maps to a second level, which is again only a single frame of memory, and again with the third. Consequently, the three level system reduces the number of pages required to only a fraction of those required for the single level system.

There are obvious disadvantages to the system. Looking up a single address takes more references, which can be expensive. Linux understands that this system may not

be appropriate on many different types of processor, so each architecture can *collapse* the page table to have less levels easily (for example, the most common architecture, the x86, only uses a two level system in its implementation).

Hardware support for virtual memory

As covered in the section called “The TLB”, the processor hardware provides a lookup-table that links virtual addresses to physical addresses. Each processor architecture defines different ways to manage the TLB with various advantages and disadvantages.

The part of the processor that deals with virtual memory is generally referred to as the *Memory Management Unit* or MMU

x86-64

XXX

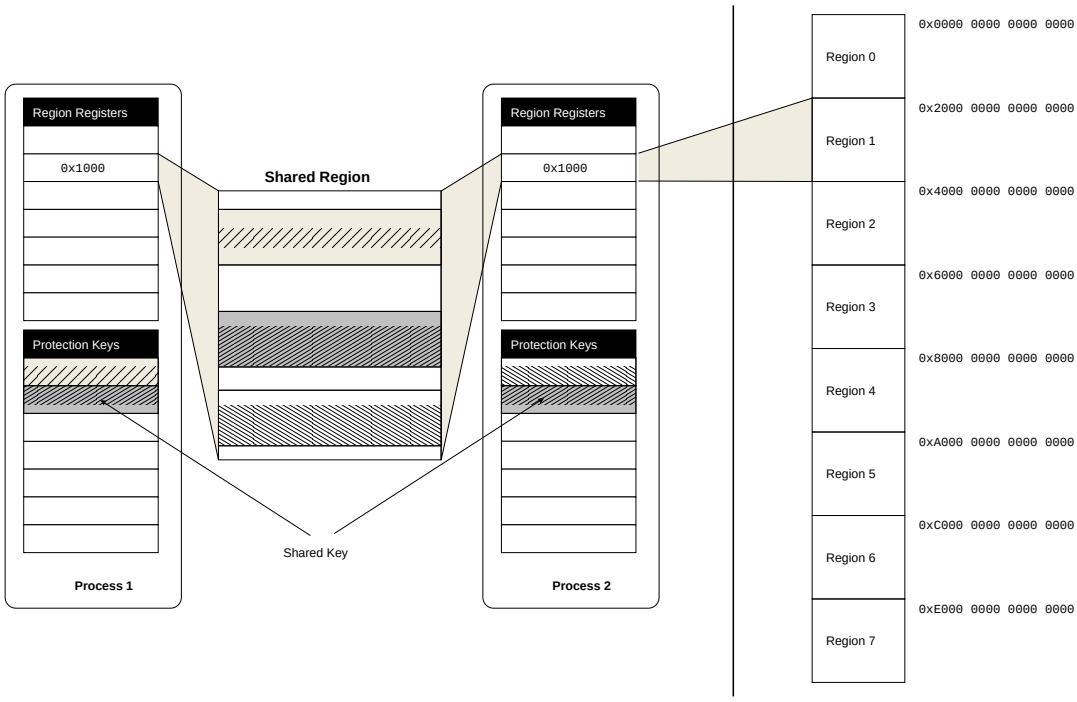
Itanium

The Itanium MMU provides many interesting features for the operating system to work with virtual memory.

Address spaces

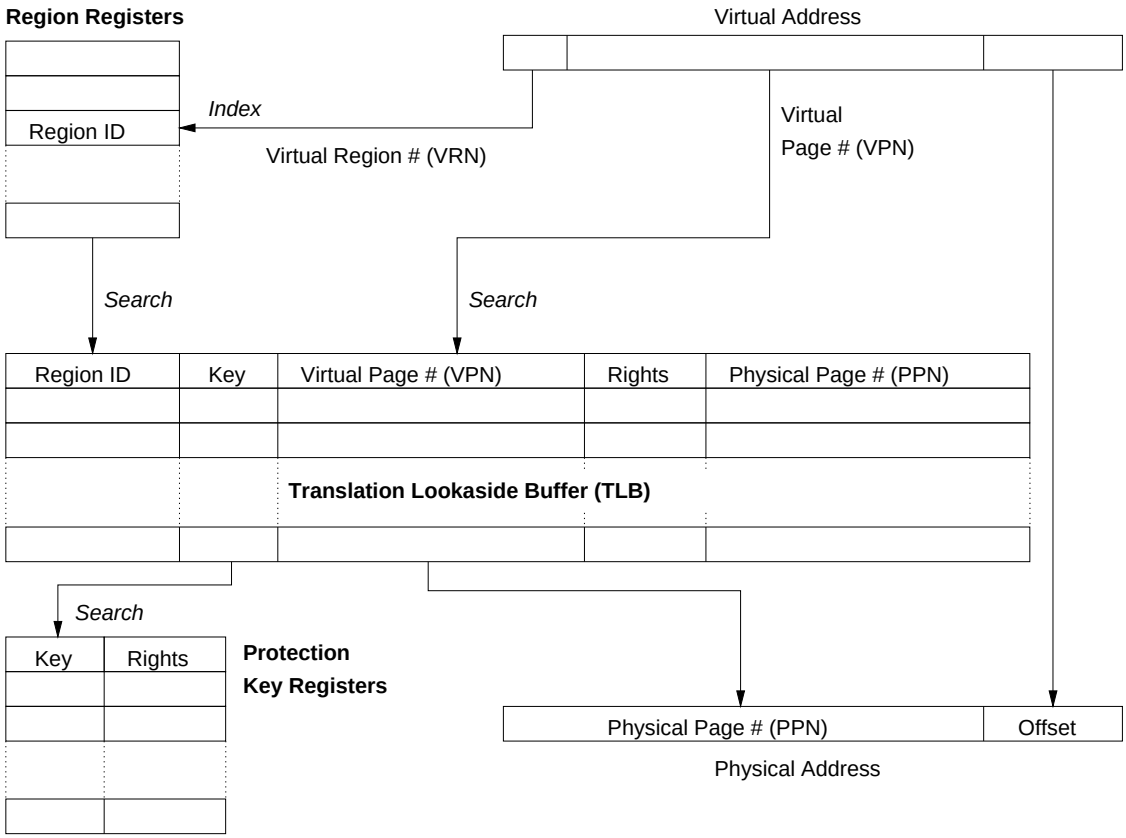
the section called “Flushing the TLB” introduced the concept of the *address-space ID* to reduce the overheads of flushing the TLB when context switching. However, programmers often use *threads* to allow execution contexts to share an address space. Each thread has the same ASID and hence shares TLB entries, leading to increased performance. However, a single ASID prevents the TLB from enforcing protection; sharing becomes an "all or nothing" approach. To share even a few bytes, threads must forgo all protection from each other (see also the section called “Protection”).

Figure 6.7. Illustration Itanium regions and protection keys



The Itanium MMU considers these problems and provides the ability to share an address space (and hence translation entries) at a much lower granularity whilst still maintaining protection within the hardware. The Itanium divides the 64-bit address space up into 8 *regions*, as illustrated in Figure 6.7, “Illustration Itanium regions and protection keys”. Each process has eight 24-bit *region registers* as part of its state, which each hold a *region ID* (RID) for each of the eight regions of the process address space. TLB translations are tagged with the RID and thus will only match if the process also holds this RID, as illustrated in Figure 6.8, “Illustration of Itanium TLB translation”.

Figure 6.8. Illustration of Itanium TLB translation



Further to this, the top three bits (the region bits) are not considered in virtual address translation. Therefore, if two processes share a RID (i.e., hold the same value in one of their region registers) then they have an aliased view of that region. For example, if process-A holds RID `0x100` in region-register 3 and process-B holds the same RID `0x100` in region-register 5 then process-A, region 3 is aliased to process-B, region 5. This limited sharing means both processes receive the benefits of shared TLB entries without having to grant access to their entire address space.

Protection Keys

To allow for even finer grained sharing, each TLB entry on the Itanium is also tagged with a *protection key*. Each process has an additional number of *protection key registers* under operating-system control.

When a series of pages is to be shared (e.g., code for a shared system library), each page is tagged with a unique key and the OS grants any processes allowed to access the pages that key. When a page is referenced the TLB will check the key associated with the translation entry against the keys the process holds in its protection key registers, allowing the access if the key is present or otherwise raising a *protection* fault to the operating system.

The key can also enforce permissions; for example, one process may have a key which grants write permissions and another may have a read-only key. This allows for sharing

of translation entries in a much wider range of situations with granularity right down to a single-page level, leading to large potential improvements in TLB performance.

Itanium Hardware Page-Table Walker

Switching context to the OS when resolving a TLB miss adds significant overhead to the fault processing path. To combat this, Itanium allows the option of using built-in hardware to read the page-table and automatically load virtual-to-physical translations into the TLB. The hardware page-table walker (HPW) avoids the expensive transition to the OS, but requires translations to be in a fixed format suitable for the hardware to understand.

The Itanium HPW is referred to in Intel's documentation as the *virtually hashed page-table walker* or VHPT walker, for reasons which should become clear. Itanium gives developers the option of two mutually exclusive HPW implementations; one based on a virtual linear page-table and the other based on a hash table.

It should be noted it is possible to operate with no hardware page-table walker; in this case each TLB miss is resolved by the OS and the processor becomes a software-loaded architecture. However, the performance impact of disabling the HPW is so considerable it is very unlikely any benefit could be gained from doing so

Virtual Linear Page-Table

The virtual linear page-table implementation is referred to in documentation as the *short format virtually hashed page-table* (SF-VHPT). It is the default HPW model used by Linux on Itanium.

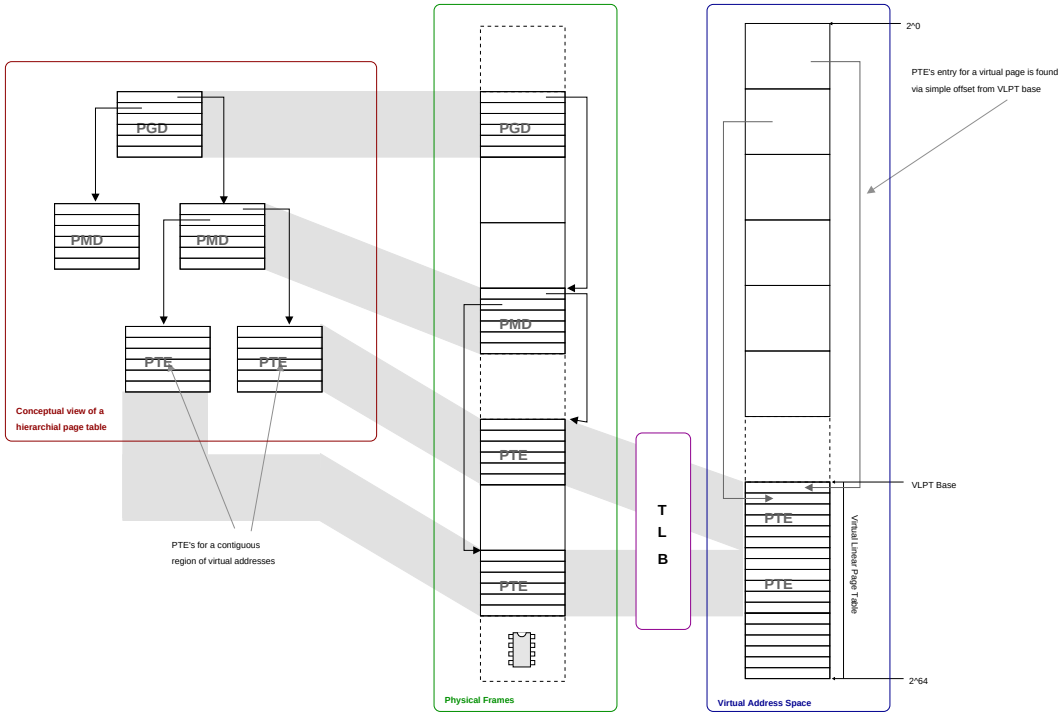
The usual solution is a multi-level or hierarchical page-table, where the bits comprising the virtual page number are used as an index into intermediate levels of the page-table (see the section called “Three Level Page Table”). Empty regions of the virtual address space simply do not exist in the hierarchical page-table. Compared to a linear page-table, for the (realistic) case of a tightly-clustered and sparsely-filled address space, relatively little space is wasted in overheads. The major disadvantage is the multiple memory references required for lookup.

Figure 6.9. Illustration of a hierarchical page-table

With a 64-bit address space, even a 512~GiB linear table identified in the section called “Virtual Address Translation” takes only 0.003% of the 16-exabytes available. Thus a *virtual linear page-table* (VLPT) can be created in a contiguous area of *virtual* address space.

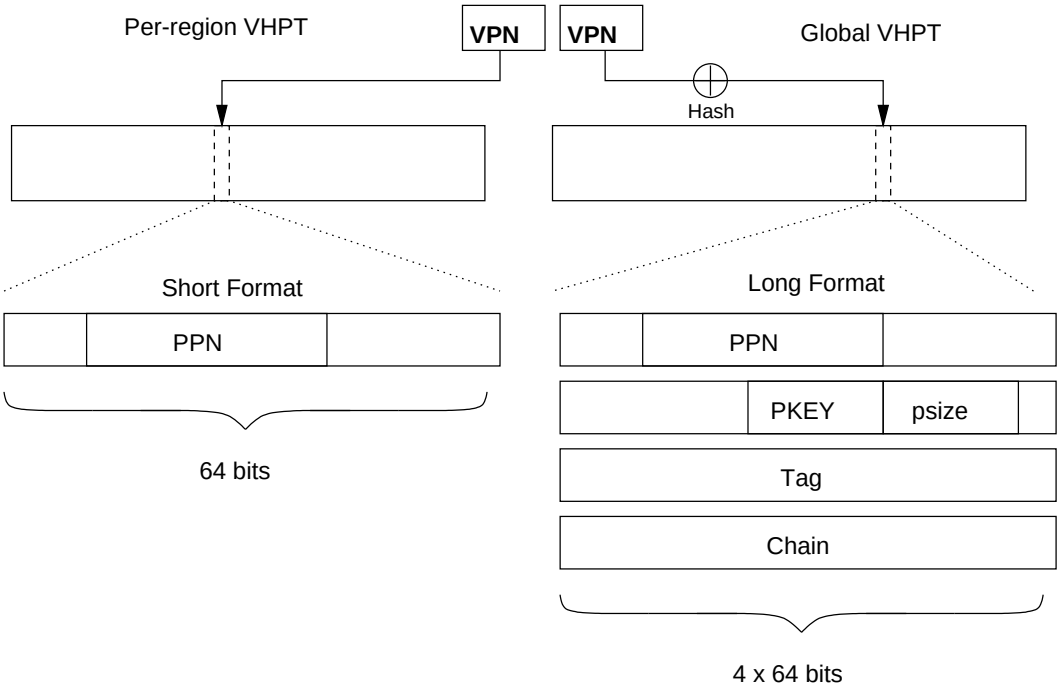
Just as for a physically linear page-table, on a TLB miss the hardware uses the virtual page number to offset from the page-table base. If this entry is valid, the translation is read and inserted directly into the TLB. However, with a VLPT the address of the translation entry is itself a virtual address and thus there is the possibility that the virtual page which it resides in is not present in the TLB. In this case a *nested fault* is raised to the operating system. The software must then correct this fault by mapping the page holding the translation entry into the VLPT.

Figure 6.10. Itanium short-format VHPT implementation



This process can be made quite straight forward if the operating system keeps a hierarchical page-table. The leaf page of a hierarchical page-table holds translation entries for a virtually contiguous region of addresses and can thus be mapped by the TLB to create the VLPT as described in Figure 6.10, “Itanium short-format VHPT implementation”.

Figure 6.11. Itanium PTE entry formats



The major advantage of a VLPT occurs when an application makes repeated or contiguous accesses to memory. Consider that for a walk of virtually contiguous memory, the first fault will map a page full of translation entries into the virtual linear page-table. A subsequent access to the next virtual page will require the next translation entry to be loaded into the TLB, which is now available in the VLPT and thus loaded very quickly and without invoking the operating system. Overall, this will be an advantage if the cost of the initial nested fault is amortised over subsequent HPW hits.

The major drawback is that the VLPT now requires TLB entries which causes an increase on TLB pressure. Since each address space requires its own page table the overheads become greater as the system becomes more active. However, any increase in TLB capacity misses should be more than regained in lower refill costs from the efficient hardware walker. Note that a pathological case could skip over `page_size` ÷ `translation_size` entries, causing repeated nested faults, but this is a very unlikely access pattern.

The hardware walker expects translation entries in a specific format as illustrated on the left of Figure 6.11, “Itanium PTE entry formats”. The VLPT requires translations in the so-called 8-byte *short format*. If the operating system is to use its page-table as backing for the VLPT (as in Figure 6.10, “Itanium short-format VHPT implementation”) it must use this translation format. The architecture describes a limited number of bits in this format as ignored and thus available for use by software, but significant modification is not possible.

A linear page-table is premised on the idea of a fixed page size. Multiple page-size support is problematic since it means the translation for a given virtual page is no longer at a constant offset. To combat this, each of the 8-regions of the address space

(Figure 6.7, “Illustration Itanium regions and protection keys”) has a separate VLPT which only maps addresses for that region. A default page-size can be given for each region (indeed, with Linux HugeTLB, discussed below, one region is dedicated to larger pages). However, page sizes can not be mixed within a region.

Virtual Hash Table

Using TLB entries in an effort to reduce TLB refill costs, as done with the SF-VHPT, may or may not be an effective trade-off. Itanium also implements a *hashed page-table* with the potential to lower TLB overheads. In this scheme, the processor *hashes* a virtual address to find an offset into a contiguous table.

The previously described physically linear page-table can be considered a hash page-table with a *perfect* hash function which will never produce a collision. However, as explained, this requires an impractical trade-off of huge areas of contiguous physical memory. However, constraining the memory requirements of the page table raises the possibility of collisions when two virtual addresses hash to the same offset. Colliding translations require a *chain* pointer to build a linked-list of alternative possible entries. To distinguish which entry in the linked-list is the correct one requires a *tag* derived from the incoming virtual address.

The extra information required for each translation entry gives rise to the moniker *long-format*~VHPT (LF-VHPT). Translation entries grow to 32-bytes as illustrated on the right hand side of Figure 6.11, “Itanium PTE entry formats”.

The main advantage of this approach is the global hash table can be pinned with a single TLB entry. Since all processes share the table it should scale better than the SF-VHPT, where each process requires increasing numbers of TLB entries for VLPT pages. However, the larger entries are less cache friendly; consider we can fit four 8-byte short-format entries for every 32-byte long-format entry. The very large caches on the Itanium processor may help mitigate this impact, however.

One advantage of the SF-VHPT is that the operating system can keep translations in a hierarchical page-table and, as long as the hardware translation format is maintained, can map leaf pages directly to the VLPT. With the LF-VHPT the OS must either use the hash table as the primary source of translation entries or otherwise keep the hash table as a cache of its own translation information. Keeping the LF-VHPT hash table as a cache is somewhat sub-optimal because of increased overheads on time critical fault paths, however advantages are gained from the table requiring only a single TLB entry.

Chapter 7. The Toolchain

Compiled v Interpreted Programs

Compiled Programs

So far we have discussed how a program is loaded into virtual memory, started as a process kept track of by the operating system and interacts with via system calls.

A program that can be loaded directly into memory needs to be in a straight *binary* format. The process of converting source code, written in a language such as C, to a binary file ready to be executed is called *compiling*. Not surprisingly, the process is done by a *compiler*; the most widespread example being gcc.

Interpreted programs

Compiled programs have some disadvantages for modern software development. Every time a developer makes a change, the compiler must be invoked to recreate the executable file. It is a logical extension to design a compiled program that can read *another* program listing and execute the code line by line.

We call this type of compiled program a *interpreter* because it interprets each line of the input file and executes it as code. This way the program does does not need to be compiled, and any changes will be seen the next time the interpreter runs the code.

For their convenience, interpreted programs usually run slower than a compiled counterpart. The overhead in the program reading and interpreting the code each time is only encountered once for a compiled program, whilst an interpreted program encounters it each time it is run.

But interpreted languages have many positive aspects. Many interpreted languages actually run in a `virtual machine` that is abstracted from the underlying hardware. Python and Perl 6 are languages that implement a virtual machine that interpreted code runs on.

Virtual Machines

A compiled program is completely dependent on the hardware of the machine it is compiled for, since it must be able to simply be copied to memory an executed. A virtual machine is an abstraction of hardware into software.

For example, Java is a hybrid language that is partly compiled and partly interpreted. Java code is complied into a program that runs inside a *Java Virtual Machine* or more commonly referred to as a JVM. This means that a compiled program can run on any hardware that has a JVM written for it; so called *write one, run anywhere*.

Building an executable

When we talk about the compiler, there are actually three separate steps involved in creating the executable file.

1. Compiling

2. Assembling

3. Linking

The components involved in this process are collectively called the *toolchain* because the tools *chain* the output of one to the input of the other to create the final output.

Each link in the chain takes the source code progressively closer to being binary code suitable for execution.

Compiling

The process of compiling

The first step of compiling a source file to an executable file is converting the code from the high level, human understandable language to *assembly code*. We know from previous chapters than assembly code works directly with the instructions and registers provided by the processor.

The compiler is the most complex step of process for a number of reasons. Firstly, humans are very unpredictable and have their source code in many different forms. The compiler is only interested the actual code, however humans need things like comments and whitespace (spaces, tabs, indents, etc) to understand code. The process that the compiler takes to convert the human-written source code to its internal representation is called *parsing*.

C code

With C code, there is actually a step *before* parsing the source code called the *pre-processor*. The pre-processor is at its core a text replacement program. For example, any variable declared as `#define variable text` will have `variable` replaced with `text`. This preprocessed code is then passed into the compiler.

Syntax

Any computing language has a particular *syntax* that describes the rules of the language. Both you and the compiler know the syntax rules, and all going well you will understand each other. Humans, being as they are, often forget the rules or break them, leading the compiler to be unable to understand your intentions. For example, if you were to leave

the closing bracket off a `if` condition, the compiler does not know where the actual conditional is.

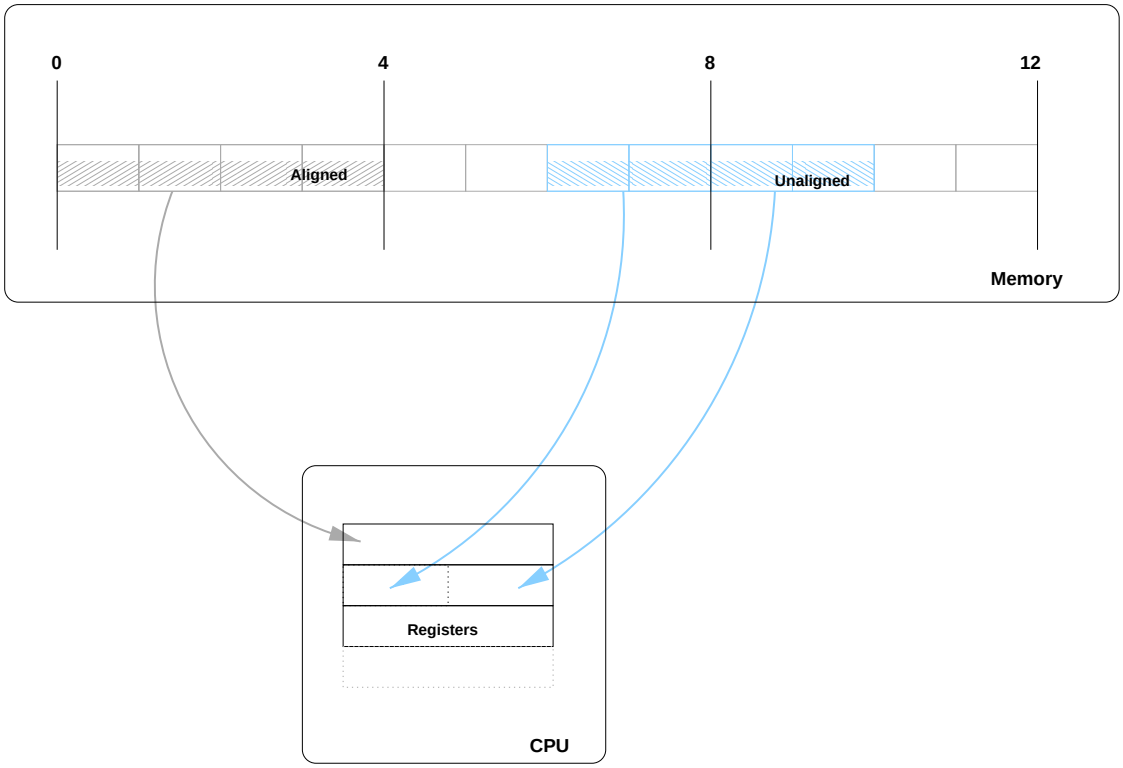
Syntax is most often described in *Backus-Naur Form* (BNF)[1] which is a language with which you can describe languages!

Assembly Generation

The job of the compiler is to translate the higher level language into assembly code suitable for the target being compiled for. Obviously each different architecture has a different instruction set, different numbers of registers and different rules for correct operation.

Alignment

Figure 7.1. Alignment



Alignment of variables in memory is an important consideration for the compiler. Systems programmers need to be aware of alignment constraints to help the compiler create the most efficient code it can.

CPUs can generally not load a value into a register from an arbitrary memory location. It requires that variables be *aligned* on certain boundaries. In the example above, we

[1] In fact the most common form is Extended Backus-Naur Form, or EBNF, as it allows some extra rules which are more suitable for modern languages.

can see how a 32 bit (4 byte) value is loaded into a register on a machine that requires 4 byte alignment of variables.

The first variable can be directly loaded into a register, as it falls between 4 byte boundaries. The second variable, however, spans the 4 byte boundary. This means that at minimum two loads will be required to get the variable into a single register; firstly the lower half and then the upper half.

Some architectures, such as x86, can handle unaligned loads in hardware and the only symptoms will be lower performance as the hardware does the extra work to get the value into the register. Others architectures can not have alignment rules violated and will raise an exception which is generally caught by the operating system which then has to manually load the register in parts, causing even more overheads.

Structure Padding

Programmers need to consider alignment especially when creating `structs`. Whilst the compiler knows the alignment rules for the architecture it is building for, at times programmers can cause sub-optimal behaviour.

The C99 standard only says that structures will be ordered in memory in the same order as they are specified in the declaration, and that in an array of structures all elements will be the same size.

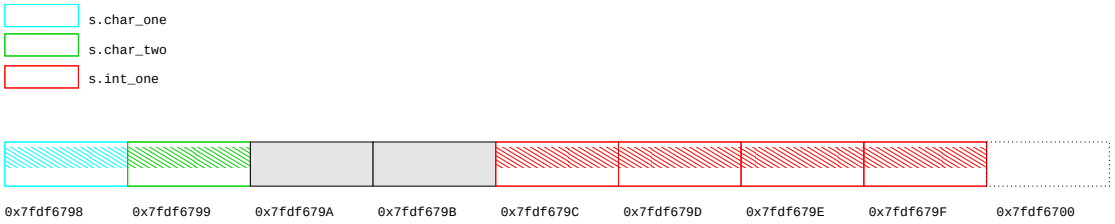
Example 7.1. Struct padding example

```
 1
                $ cat struct.c
   #include <stdio.h>

 5 struct a_struct {
           char char_one;
           char char_two;
           int int_one;
   };
10
   int main(void)
   {

           struct a_struct s;
15
           printf("%p : s.char_one\n" \
                  "%p : s.char_two\n" \
                  "%p : s.int_one\n", &s.char_one,
                  &s.char_two, &s.int_one);
20
           return 0;

   }

25 $ gcc -o struct struct.c

   $ gcc -fpack-struct -o struct-packed struct.c

   $ ./struct
30 0x7fdf6798 : s.char_one
   0x7fdf6799 : s.char_two
   0x7fdf679c : s.int_one

   $ ./struct-packed
```

```
35 0x7fcd2778 : s.char_one
   0x7fcd2779 : s.char_two
   0x7fcd277a : s.int_one
```

In the example above, we contrive a structure that has two bytes (`chars` followed by a 4 byte integer. The compiler pads the structure as below.

Figure 7.2. Alignment



In the other example we direct the compiler *not* to pad structures and correspondingly we can see that the integer starts directly after the two `chars`.

Cache line alignment

We talked previously about aliasing in the cache, and how several addresses may map to the same cache line. Programmers need to be sure that when they write their programs they do not cause *bouncing* of cache lines.

This situation occurs when a program constantly accesses two areas of memory that map to the same cache line. This effectively wastes the cache line, as it gets loaded in, used for a short time and then must be kicked out and the other cache line loaded into the same place in the cache.

Obviously if this situation repeats the performance will be significantly reduced. The situation would be relieved if the conflicting data were organised in slightly different ways to avoid the cache line conflict.

One possible way to detect this sort of situation is *profiling*. When you profile your code you "watch" it to analyse what code paths are taken and how long they take to execute. With *profile guided optimisation* (PGO) the compiler can put special extra bits of code in the first binary it builds, which runs and makes a record of the branches taken, etc. You can then recompile the binary with the extra information to possibly create a better performing binary. Otherwise the programmer can look at the output of the profile and possibly detect situations such as cache line bouncing. (XXX somewhere else?)

Space - Speed Trade off

What the compiler has done above is traded off using some extra memory to gain a speed improvement in running our code. The compiler knows the rules of the architecture and can make decisions about the best way to align data, possibly by trading off small amounts of wasted memory for increased (or perhaps even just correct) performance.

Consequently as a programmer you should never make assumptions about the way variables and data will be laid out by the compiler. To do so is not portable, as a different architecture may have different rules and the compiler may make different decisions based on explicit commands or optimisation levels.

Making Assumptions

Thus, as a C programmer you need to be familiar with what you can assume about what the compiler will do and what may be variable. What exactly you can assume and can not assume is detailed in the C99 standard; if you are programming in C it is certainly worth the investment in becoming familiar with the rules to avoid writing non-portable or buggy code.

Example 7.2. Stack alignment example

```
 1
                $ cat stack.c
   #include <stdio.h>

 5 struct a_struct {
           int a;
           int b;
   };

10 int main(void)
   {
           int i;
           struct a_struct s;
           printf("%p\n%p\ndiff %ld\n", &i, &s, (unsigned long)&s - (unsigned long)&i);
15         return 0;
   }
   $ gcc-3.3 -Wall -o stack-3.3 ./stack.c
   $ gcc-4.0 -o stack-4.0 stack.c

20 $ ./stack-3.3
   0x60000fffffc2b510
   0x60000fffffc2b520
   diff 16

25 $ ./stack-4.0
   0x60000fffff89b520
   0x60000fffff89b524
   diff 4

30
```

In the example above, taken from an Itanium machine, we can see that the padding and alignment of the stack has changed considerably between gcc versions. This type of thing is to be expected and must be considered by the programmer.

Generally you should ensure that you do not make assumptions about the size of types or alignment rules.

C Idioms with alignment

There are a few common sequences of code that deal with alignment; generally most programs will consider it in some ways. You may see these "code idioms" in many places outside the kernel when dealing with programs that deal with chunks of data in some form or another, so it is worth investigating.

We can take some examples from the Linux kernel, which often has to deal with alignment of pages of memory within the system.

Example 7.3. Page alignment manipulations

```
 1
                [ include/asm-ia64/page.h ]

   /*
 5  * PAGE_SHIFT determines the actual kernel page size.
    */
   #if defined(CONFIG_IA64_PAGE_SIZE_4KB)
   # define PAGE_SHIFT     12
   #elif defined(CONFIG_IA64_PAGE_SIZE_8KB)
10 # define PAGE_SHIFT     13
   #elif defined(CONFIG_IA64_PAGE_SIZE_16KB)
   # define PAGE_SHIFT     14
   #elif defined(CONFIG_IA64_PAGE_SIZE_64KB)
   # define PAGE_SHIFT     16
15 #else
   # error Unsupported page size!
   #endif

   #define PAGE_SIZE               (__IA64_UL_CONST(1) << PAGE_SHIFT)
20 #define PAGE_MASK               (~(PAGE_SIZE - 1))
   #define PAGE_ALIGN(addr)        (((addr) + PAGE_SIZE - 1) & PAGE_MASK)
```

Above we can see that there are a number of different options for page sizes within the kernel, ranging from 4KB through 64KB.

The `PAGE_SIZE` macro is fairly self explanatory, giving the current page size selected within the system by shifting a value of 1 by the shift number given (remember, this is the equivalent of saying 2^n where n is the `PAGE_SHIFT`).

Next we have a definition for `PAGE_MASK`. The `PAGE_MASK` allows us to find just those bits that are within the current page, that is the `offset` of an address within its page.

XXX continue short discussion

Optimisation

Once the compiler has an internal representation of the code, the *really* interesting part of the compiler starts. The compiler wants to find the most optimised assembly language output for the given input code. This is a large and varied problem and requires knowledge of everything from efficient algorithms based in computer science to deep knowledge about the particular processor the code is to be run on.

There are some common optimisations the compiler can look at when generating output. There are many, many more strategies for generating the best code, and it is always an active research area.

General Optimising

The compiler can often see that a particular piece of code can not be used and so leave it out optimise a particular language construct into something smaller with the same outcome.

Unrolling loops

If code contains a loop, such as a `for` or `while` loop and the compiler has some idea how many times it will execute, it may be more efficient to *unroll* the loop so that it executes sequentially. This means that instead of doing the inside of the loop and then branching back to the start to do repeat the process, the inner loop code is duplicated to be executed again.

Whilst this increases the size of the code, it may allow the processor to work through the instructions more efficiently as branches can cause inefficiencies in the pipeline of instructions coming into the processor.

Inlining functions

Similar to unrolling loops, it is possible to put embed called functions within the callee. The programmer can specify that the compiler should try to do this by specifying the function as `inline` in the function definition. Once again, you may trade code size for sequentially in the code by doing this.

Branch Prediction

Any time the computer comes across an `if` statement there are two possible outcomes; true or false. The processor wants to keep its incoming pipes as full as possible, so it can not wait for the outcome of the test before putting code into the pipeline.

Thus the compiler can make a prediction about what way the test is likely to go. There are some simple rules the compiler can use to guess things like this, for example `if (val == -1)` is probably *not* likely to be true, since -1 usually indicates an error code and hopefully that will not be triggered too often.

Some compilers can actually compile the program, have the user run it and take note of which way the branches go under real conditions. It can then re-compile it based on what it has seen.

Assembler

The assembly code outputted by the compiler is still in a human readable form, should you know the specifics of the assembly code for the processor. Developers will often take a peek at the assembly output to manually check that the code is the most optimised or to discover any bugs in the compiler (this is more common than one might think, especially when the compiler is being very aggressive with optimisations).

The assembler is a more mechanical process of converting the assembly code into a binary form. Essentially, the assembler keeps a large table of each possible instruction and its binary counterpart (called an *op code* for operation code). It combines these opcodes with the registers specified in the assembly to produce a binary output file.

This code is called *object code* and, at this stage, is not executable. Object code is simply a binary representation of specific input source code file. Good programming practice dictates that a programmer should not "put all the eggs in one basket" by placing all your source code in one file.

Linker

Often in a large program, you will separate out code into multiple files to keep related functions together. Each of these files can be compiled into object code: but your final goal is to create a single executable! There needs to be some way combining each of these object files into a single executable. We call this linking.

Note that even if your program does fit in one file it still needs to be linked against certain system libraries to operate correctly. For example, the `printf` call is kept in a library which must be combined with your executable to work. So although you do not explicitly have to worry about linking in this case, there is most certainly still a linking process happening to create your executable.

In the following sections we explain some terms essential to understanding linking.

Symbols

Symbols

Variables and functions all have names in source code which we refer to them by. One way of thinking of a statement declaring a variable `int a` is that you are telling the compiler "set aside some memory of `sizeof(int)` and from now on when I use a it will refer to this allocated memory. Similarly a function says "store this code in memory, and when I call `function()` jump to and execute this code".

In this case, we call a and `function` *symbols* since they are a symbolic representation of an area of memory.

Symbols help humans to understand programming. You could say that the primary job of the compilation process is to remove symbols -- the processor doesn't know what a represents, all it knows is that it has some data at a particular memory address. The compilation process needs to convert a `+= 2` to something like "increment the value in memory at `0xABCDE` by 2.

Symbol Visibility

In some C programs, you may have seen the terms `static` and `extern` used with variables. These modifiers can effect what we call the visibility of symbols.

Imagine you have split up your program in two files, but some functions need to share a variable. You only want one *definition* (i.e. memory location) of the shared variable (otherwise it wouldn't be shared!), but both files need to reference it.

To enable this, we declare the variable in one file, and then in the other file declare a variable of the same name but with the prefix `extern`. `extern` stands for *external* and to a human means that this variable is declared somewhere else.

What `extern` says to a compiler is that it should not allocate any space in memory for this variable, and leave this symbol in the object code where it will be fixed up later. The compiler can not possibly know where the symbol is actually defined but the *linker*does, since it is it's job to look at all object files together and combine them into a single

executable. So the linker will see the symbol left over in the second file, and say "I've seen that symbol before in file 1, and I know that it refers to memory location `0x12345`". Thus it can modify the symbol value to be the memory value of the variable in the first file.

`static` is almost the opposite of `extern`. It places restrictions on the visibility of the symbol it modifies. If you declare a variable with `static` that says to the compiler "don't leave any symbols for this in the object code". This means that when the linker is linking together object files it will never see that symbol (and so can't make that "I've seen this before!" connection). `static` is good for separation and reducing conflicts -- by declaring a variable `static` you can reuse the variable name in other files and not end up with symbol clashes. We say we are *restricting the visibility* of the symbol, because we are not allowing the linker to see it. Contrast this with a more visible symbol (one not declared with `static`) which can be seen by the linker.

The linking process

Thus the linking process is really two steps; combining all object files into one executable file and then going through each object file to *resolve* any symbols. This usually requires two passes; one to read all the symbol definitions and take note of unresolved symbols and a second to fix up all those unresolved symbols to the right place.

The final executable should end up with no unresolved symbols; the linker will fail with an error if there are any.[2]

A practical example

We can walk through the steps taken to build a simple application step by step.

Note that when you type `gcc` that actually runs a driver program that hides most of the steps from you. Under normal circumstances this is exactly what you want, because the exact commands and options to get a real life working executable on a real system can be quite complicated and architecture specific.

We will show the compilation process with the two following examples. Both are C source files, one defined the `main()` function for the initial program entry point, and another declares a helper type function. There is one global variable too, just for illustration.

Example 7.4. Hello World

```
 1
          #include <stdio.h>

   /* We need a prototype so the compiler knows what types function() takes */
 5 int function(char *input);

   /* Since this is static, we can define it in both hello.c and function.c */
   static int i = 100;

10 /* This is a global variable */
   int global = 10;

   int main(void)
   {
```

[2] We call this *static linking*. Dynamic linking is a similar concept done at executable runtime, and is described a little later on.

```
15  /* function() should return the value of global */
    int ret = function("Hello, World!");
    exit(ret);
   }

20
```

Example 7.5. Function Example

```
 1
          #include <stdio.h>

   static int i = 100;
 5
   /* Declard as extern since defined in hello.c */
   extern int global;

   int function(char *input)
10 {
    printf("%s\n", input);
    return global;
   }

15
```

Compiling

All compilers have an option to only execute the first step of compilation. Usually this is something like `-S` and the output will generally be put into a file with the same name as the input file but with a `.s` extension.

Thus we can show the first step with `gcc -S` as illustrated in the example below.

Example 7.6. Compilation Example

```
 1
            ianw@lime:~/programs/csbu/wk7/code$ gcc -S hello.c
   ianw@lime:~/programs/csbu/wk7/code$ gcc -S function.c
   ianw@lime:~/programs/csbu/wk7/code$ cat function.s
 5         .file   "function.c"
           .pred.safe_across_calls p1-p5,p16-p63
           .section        .sdata,"aw",@progbits
           .align 4
           .type   i#, @object
10         .size   i#, 4
   i:
           data4   100
           .section        .rodata
           .align 8
15 .LC0:
           stringz "%s\n"
           .text
           .align 16
           .global function#
20         .proc function#
   function:
           .prologue 14, 33
           .save ar.pfs, r34
           alloc r34 = ar.pfs, 1, 4, 2, 0
25         .vframe r35
           mov r35 = r12
           adds r12 = -16, r12
           mov r36 = r1
           .save rp, r33
30         mov r33 = b0
```

```
            .body
            ;;
            st8 [r35] = r32
            addl r14 = @ltoffx(.LC0), r1
35          ;;
            ld8.mov r37 = [r14], .LC0
            ld8 r38 = [r35]
            br.call.sptk.many b0 = printf#
            mov r1 = r36
40          ;;
            addl r15 = @ltoffx(global#), r1
            ;;
            ld8.mov r14 = [r15], global#
            ;;
45          ld4 r14 = [r14]
            ;;
            mov r8 = r14
            mov ar.pfs = r34
            mov b0 = r33
50          .restore sp
            mov r12 = r35
            br.ret.sptk.many b0
            ;;
            .endp function#
55          .ident  "GCC: (GNU) 3.3.5 (Debian 1:3.3.5-11)"
```

The assembly is a little to complex to fully describe, but you should be able to see where i is defined as a `data4` (i.e. 4 bytes or 32 bits, the size of an `int`), where `function` is defined (`function:`) and a call to `printf()`.

We now have two assembly files ready to be assembled into machine code!

Assembly

Assembly is a fairly straight forward process. The assembler is usually called `as` and takes arguments in a similar fashion to `gcc`

Example 7.7. Assembly Example

```
 1
             ianw@lime:~/programs/csbu/wk7/code$ as -o function.o function.s
   ianw@lime:~/programs/csbu/wk7/code$ as -o hello.o hello.s
   ianw@lime:~/programs/csbu/wk7/code$ ls
 5 function.c  function.o  function.s  hello.c  hello.o  hello.s
```

After assembling we have *object* code, which is ready to be linked together into the final executable. You can usually skip having to use the assembler by hand by calling the compiler with `-c`, which will directly convert the input file to object code, putting it in a file with the same prefix but `.o` as an extension.

We can't inspect the object code directly, as it is in a binary format (in future weeks we will learn about this binary format). However we can use some tools to inspect the object files, for example `readelf --symbols` will show us symbols in the object file.

Example 7.8. Readelf Example

```
 1
```

```
            ianw@lime:~/programs/csbu/wk7/code$ readelf --symbols ./hello.o

   Symbol table '.symtab' contains 15 entries:
 5    Num:    Value          Size Type    Bind   Vis      Ndx Name
        0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
        1: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS hello.c
        2: 0000000000000000     0 SECTION LOCAL  DEFAULT    1
        3: 0000000000000000     0 SECTION LOCAL  DEFAULT    3
10      4: 0000000000000000     0 SECTION LOCAL  DEFAULT    4
        5: 0000000000000000     0 SECTION LOCAL  DEFAULT    5
        6: 0000000000000000     4 OBJECT  LOCAL  DEFAULT    5 i
        7: 0000000000000000     0 SECTION LOCAL  DEFAULT    6
        8: 0000000000000000     0 SECTION LOCAL  DEFAULT    7
15      9: 0000000000000000     0 SECTION LOCAL  DEFAULT    8
       10: 0000000000000000     0 SECTION LOCAL  DEFAULT   10
       11: 0000000000000004     4 OBJECT  GLOBAL DEFAULT    5 global
       12: 0000000000000000    96 FUNC    GLOBAL DEFAULT    1 main
       13: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND function
20     14: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND exit

   ianw@lime:~/programs/csbu/wk7/code$ readelf --symbols ./function.o

   Symbol table '.symtab' contains 14 entries:
25    Num:    Value          Size Type    Bind   Vis      Ndx Name
        0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
        1: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS function.c
        2: 0000000000000000     0 SECTION LOCAL  DEFAULT    1
        3: 0000000000000000     0 SECTION LOCAL  DEFAULT    3
30      4: 0000000000000000     0 SECTION LOCAL  DEFAULT    4
        5: 0000000000000000     0 SECTION LOCAL  DEFAULT    5
        6: 0000000000000000     4 OBJECT  LOCAL  DEFAULT    5 i
        7: 0000000000000000     0 SECTION LOCAL  DEFAULT    6
        8: 0000000000000000     0 SECTION LOCAL  DEFAULT    7
35      9: 0000000000000000     0 SECTION LOCAL  DEFAULT    8
       10: 0000000000000000     0 SECTION LOCAL  DEFAULT   10
       11: 0000000000000000   128 FUNC    GLOBAL DEFAULT    1 function
       12: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND printf
       13: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND global
40
```

Although the output is quite complicated (again!) you should be able to understand much of it. For example

- In the output of `hello.o` have a look at the symbol with name `i`. Notice how it says it is `LOCAL`? That is because we declared it `static` and as such it has been flagged as being local to this object file.
- In the same output, notice that the `global` variable is defined as a `GLOBAL`, meaning that it is visible outside this file. Similarly the `main()` function is externally visible.
- Notice that the `function` symbol (for the call to `function()` is left has `UND` or *undefined*. This means that it has been left for the linker to find the address of the function.
- Have a look at the symbols in the `function.c` file and how they fit into the output.

Linking

Actually invoking the linker, called `ld`, is a very complicated process on a real system (are you sick of hearing this yet?). This is why we leave the linking process up to `gcc`.

But of course we can spy on what `gcc` is doing under the hood with the `-v` (verbose) flag.

Example 7.9. Linking Example

```
 1
             /usr/lib/gcc-lib/ia64-linux/3.3.5/collect2 -static
   /usr/lib/gcc-lib/ia64-linux/3.3.5/../../../crt1.o
   /usr/lib/gcc-lib/ia64-linux/3.3.5/../../../crti.o
 5 /usr/lib/gcc-lib/ia64-linux/3.3.5/crtbegin.o
   -L/usr/lib/gcc-lib/ia64-linux/3.3.5
   -L/usr/lib/gcc-lib/ia64-linux/3.3.5/../../..
   hello.o
   function.o
10 --start-group
   -lgcc
   -lgcc_eh
   -lunwind
   -lc
15 --end-group
   /usr/lib/gcc-lib/ia64-linux/3.3.5/crtend.o
   /usr/lib/gcc-lib/ia64-linux/3.3.5/../../../crtn.o
```

The first thing you notice is that a program called collect2 is being called. This is a simple wrapper around ld that is used internally by gcc.

The next thing you notice is object files starting with `crt` being specified to the linker. These functions are provided by gcc and the system libraries and contain code required to start the program. In actuality, the `main()` function is not the first one called when a program runs, but a function called `_start` which is in the `crt` object files. This function does some generic setup which application programmers do not need to worry about.

The path hierarchy is quite complicated, but in essence we can see that the final step is to link in some extra object files, namely

- `crt1.o` : provided by the system libraries (libc) this object file contains the `_start` function which is actually the first thing called within the program.

  `crti.o` : provided by the system libraries

  `crtbegin.o`

  `crtsaveres.o`

  `crtend.o`

  `crtn.o`

We discuss how these are used to start the program a little later.

Next you can see that we link in our two object files, `hello.o` and `function.o`. After that we specify some extra libraries with `-l` flags. These libraries are system specific and required for every program. The major one is `-lc` which brings in the C library, which has all common functions like `printf()`.

After that we again link in some more system object files which do some cleanup after programs exit.

Although the details are complicated, the concept is straight forward. All the object files will be linked together into a single executable file, ready to run!

The Executable

We will go into more details about the executable in the short future, but we can do some inspection in a similar fashion to the object files to see what has happened.

Example 7.10. Executable Example

```
 1
             ianw@lime:~/programs/csbu/wk7/code$ gcc -o program hello.c function.c
   ianw@lime:~/programs/csbu/wk7/code$ readelf --symbols ./program

 5 Symbol table '.dynsym' contains 11 entries:
      Num:    Value          Size Type    Bind   Vis      Ndx Name
        0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
        1: 60000000000000e0     0 OBJECT  GLOBAL DEFAULT  ABS _DYNAMIC
        2: 0000000000000000   176 FUNC    GLOBAL DEFAULT  UND printf@GLIBC_2.2 (2)
10      3: 600000000000109c     0 NOTYPE  GLOBAL DEFAULT  ABS __bss_start
        4: 0000000000000000   704 FUNC    GLOBAL DEFAULT  UND exit@GLIBC_2.2 (2)
        5: 600000000000109c     0 NOTYPE  GLOBAL DEFAULT  ABS _edata
        6: 60000000000000e8     0 OBJECT  GLOBAL DEFAULT  ABS _GLOBAL_OFFSET_TABLE_     7: 60000000000010b
        8: 0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND _Jv_RegisterClasses
15      9: 0000000000000000   544 FUNC    GLOBAL DEFAULT  UND __libc_start_main@GLIBC_2.2 (2)
       10: 0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND __gmon_start__

   Symbol table '.symtab' contains 127 entries:
      Num:    Value          Size Type    Bind   Vis      Ndx Name
20      0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
        1: 40000000000001c8     0 SECTION LOCAL  DEFAULT    1
        2: 40000000000001e0     0 SECTION LOCAL  DEFAULT    2
        3: 4000000000000200     0 SECTION LOCAL  DEFAULT    3
        4: 4000000000000240     0 SECTION LOCAL  DEFAULT    4
25      5: 4000000000000348     0 SECTION LOCAL  DEFAULT    5
        6: 40000000000003d8     0 SECTION LOCAL  DEFAULT    6
        7: 40000000000003f0     0 SECTION LOCAL  DEFAULT    7
        8: 4000000000000410     0 SECTION LOCAL  DEFAULT    8
        9: 4000000000000440     0 SECTION LOCAL  DEFAULT    9
30     10: 40000000000004a0     0 SECTION LOCAL  DEFAULT   10
       11: 40000000000004e0     0 SECTION LOCAL  DEFAULT   11
       12: 40000000000005e0     0 SECTION LOCAL  DEFAULT   12
       13: 4000000000000b00     0 SECTION LOCAL  DEFAULT   13
       14: 4000000000000b40     0 SECTION LOCAL  DEFAULT   14
35     15: 4000000000000b60     0 SECTION LOCAL  DEFAULT   15
       16: 4000000000000bd0     0 SECTION LOCAL  DEFAULT   16
       17: 4000000000000ce0     0 SECTION LOCAL  DEFAULT   17
       18: 60000000000000db8     0 SECTION LOCAL  DEFAULT   18
       19: 60000000000000dd0     0 SECTION LOCAL  DEFAULT   19
40     20: 60000000000000dd8     0 SECTION LOCAL  DEFAULT   20
       21: 60000000000000de0     0 SECTION LOCAL  DEFAULT   21
       22: 60000000000000fc0     0 SECTION LOCAL  DEFAULT   22
       23: 60000000000000fd0     0 SECTION LOCAL  DEFAULT   23
       24: 60000000000000fe0     0 SECTION LOCAL  DEFAULT   24
45     25: 60000000000000fe8     0 SECTION LOCAL  DEFAULT   25
       26: 6000000000001040     0 SECTION LOCAL  DEFAULT   26
       27: 6000000000001080     0 SECTION LOCAL  DEFAULT   27
       28: 60000000000010a0     0 SECTION LOCAL  DEFAULT   28
       29: 60000000000010a8     0 SECTION LOCAL  DEFAULT   29
50     30: 0000000000000000     0 SECTION LOCAL  DEFAULT   30
       31: 0000000000000000     0 SECTION LOCAL  DEFAULT   31
       32: 0000000000000000     0 SECTION LOCAL  DEFAULT   32
       33: 0000000000000000     0 SECTION LOCAL  DEFAULT   33
       34: 0000000000000000     0 SECTION LOCAL  DEFAULT   34
55     35: 0000000000000000     0 SECTION LOCAL  DEFAULT   35
       36: 0000000000000000     0 SECTION LOCAL  DEFAULT   36
```

```
        37: 0000000000000000     0 SECTION LOCAL  DEFAULT   37
        38: 0000000000000000     0 SECTION LOCAL  DEFAULT   38
        39: 0000000000000000     0 SECTION LOCAL  DEFAULT   39
 60     40: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS /build/buildd/glibc-2.3.2
        41: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS /build/buildd/glibc-2.3.2
        42: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS /build/buildd/glibc-2.3.2
        43: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS /build/buildd/glibc-2.3.2
        44: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS /build/buildd/glibc-2.3.2
 65     45: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS /build/buildd/glibc-2.3.2
        46: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS <command line>
        47: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS /build/buildd/glibc-2.3.2
        48: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS <command line>
        49: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS <built-in>
 70     50: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS abi-note.S
        51: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS /build/buildd/glibc-2.3.2
        52: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS abi-note.S
        53: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS /build/buildd/glibc-2.3.2
        54: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS abi-note.S
 75     55: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS <command line>
        56: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS /build/buildd/glibc-2.3.2
        57: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS <command line>
        58: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS <built-in>
        59: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS abi-note.S
 80     60: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS init.c
        61: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS /build/buildd/glibc-2.3.2
        62: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS /build/buildd/glibc-2.3.2
        63: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS initfini.c
        64: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS /build/buildd/glibc-2.3.2
 85     65: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS <command line>
        66: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS /build/buildd/glibc-2.3.2
        67: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS <command line>
        68: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS <built-in>
        69: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS /build/buildd/glibc-2.3.2
 90     70: 4000000000000670   128 FUNC    LOCAL  DEFAULT   12 gmon_initializer
        71: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS /build/buildd/glibc-2.3.2
        72: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS /build/buildd/glibc-2.3.2
        73: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS initfini.c
        74: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS /build/buildd/glibc-2.3.2
 95     75: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS <command line>
        76: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS /build/buildd/glibc-2.3.2
        77: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS <command line>
        78: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS <built-in>
        79: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS /build/buildd/glibc-2.3.2
100     80: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS auto-host.h
        81: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS <command line>
        82: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS <built-in>
        83: 60000000000000c0     0 NOTYPE  LOCAL  DEFAULT   22 __CTOR_LIST__
        84: 60000000000000d0     0 NOTYPE  LOCAL  DEFAULT   23 __DTOR_LIST__
105     85: 60000000000000e0     0 NOTYPE  LOCAL  DEFAULT   24 __JCR_LIST__
        86: 6000000000001088     8 OBJECT  LOCAL  DEFAULT   27 dtor_ptr
        87: 4000000000000670   128 FUNC    LOCAL  DEFAULT   12 __do_global_dtors_aux
        88: 4000000000000770   128 FUNC    LOCAL  DEFAULT   12 __do_jv_register_classes
        89: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS hello.c
110     90: 6000000000001090     4 OBJECT  LOCAL  DEFAULT   27 i
        91: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS function.c
        92: 6000000000001098     4 OBJECT  LOCAL  DEFAULT   27 i
        93: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS auto-host.h
        94: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS <command line>
115     95: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS <built-in>
        96: 60000000000000c8     0 NOTYPE  LOCAL  DEFAULT   22 __CTOR_END__
        97: 60000000000000d8     0 NOTYPE  LOCAL  DEFAULT   23 __DTOR_END__
        98: 60000000000000e0     0 NOTYPE  LOCAL  DEFAULT   24 __JCR_END__
        99: 60000000000000e0     0 OBJECT  GLOBAL DEFAULT  ABS _DYNAMIC
120    100: 4000000000000a70   144 FUNC    GLOBAL HIDDEN   12 __do_global_ctors_aux
       101: 60000000000000d8     0 NOTYPE  GLOBAL DEFAULT  ABS __fini_array_end
       102: 60000000000010a8     8 OBJECT  GLOBAL HIDDEN   29 __dso_handle
       103: 4000000000009a0   208 FUNC    GLOBAL DEFAULT   12 __libc_csu_fini
       104: 0000000000000000   176 FUNC    GLOBAL DEFAULT  UND printf@@GLIBC_2.2
125    105: 40000000000004a0    32 FUNC    GLOBAL DEFAULT   10 _init
       106: 4000000000000850   128 FUNC    GLOBAL DEFAULT   12 function
```

```
       107: 40000000000005e0   144 FUNC    GLOBAL DEFAULT   12 _start
       108: 6000000000001094     4 OBJECT  GLOBAL DEFAULT   27 global
       109: 6000000000000dd0     0 NOTYPE  GLOBAL DEFAULT  ABS __fini_array_start
130    110: 40000000000008d0   208 FUNC    GLOBAL DEFAULT   12 __libc_csu_init
       111: 600000000000109c     0 NOTYPE  GLOBAL DEFAULT  ABS __bss_start
       112: 40000000000007f0    96 FUNC    GLOBAL DEFAULT   12 main
       113: 6000000000000dd0     0 NOTYPE  GLOBAL DEFAULT  ABS __init_array_end
       114: 6000000000000dd8     0 NOTYPE  WEAK   DEFAULT   20 data_start
135    115: 4000000000000b00    32 FUNC    GLOBAL DEFAULT   13 _fini
       116: 0000000000000000   704 FUNC    GLOBAL DEFAULT  UND exit@@GLIBC_2.2
       117: 600000000000109c     0 NOTYPE  GLOBAL DEFAULT  ABS _edata
       118: 6000000000000fe8     0 OBJECT  GLOBAL DEFAULT  ABS _GLOBAL_OFFSET_TABLE_
       119: 60000000000010b0     0 NOTYPE  GLOBAL DEFAULT  ABS _end
140    120: 6000000000000db8     0 NOTYPE  GLOBAL DEFAULT  ABS __init_array_start
       121: 6000000000001080     4 OBJECT  GLOBAL DEFAULT   27 _IO_stdin_used
       122: 60000000000010a0     8 OBJECT  GLOBAL DEFAULT   28 __libc_ia64_register_back
       123: 6000000000000dd8     0 NOTYPE  GLOBAL DEFAULT   20 __data_start
       124: 0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND _Jv_RegisterClasses
145    125: 0000000000000000   544 FUNC    GLOBAL DEFAULT  UND __libc_start_main@@GLIBC_
       126: 0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND __gmon_start__
```

Some things to note

- Note I built the executable the "easy" way!
- See there are two symbol tables; the `dynsym` and `symtab` ones. We explain how the `dynsym` symbols work soon, but notice that some of them are *versioned* with an @ symbol.
- Note the many symbols that have been included from the extra object files. Many of them start with __ to avoid clashing with any names the programmer might choose. Read through and pick out the symbols we mentioned before from the object files and see if they have changed in any way.

Chapter 8. Behind the process

Review of executable files

We know that a program running in memory has two major components in *code* (also commonly known as a *text* for historical reasons) and *data*. We also know, however, an executable does not live its life in memory, but spends most of its life as a file on a disk. This file is in what is referred to as a binary format, since the bits and bytes of the file are to be interpreted directly by processor hardware.

Representing executable files

Three Standard Sections

Any executable file format will need to specify where the code and data are in the binary file.

One additional component we have not mentioned until now is storage space of uninitialised global variables. If we declare a variable and give it an initial value this obviously needs to be stored in the executable file so that upon execution the value is correct. However many variables are uninitialised (or zero) when the program is first executed. Making space for these in the executable and then simply storing zero or NULL values in it is a waste of space, needlessly bloating the executable file size. Thus each executable file can define a BSS section which simply gives a size for the uninitialised data; on program load the extra memory can be allocated (and set to zero!).[1]

Binary Format

The executable is created by the toolchain from the source code. This file needs to be in a format explicitly defined such that the compiler can create it and the operating system can identify it and load into memory, turning it into a running process that the operating system can manage. This *executable file format* can be specific to the operating system, as we would not normally expect that a program compiled for one system will execute on another (for example, you don't expect your Windows programs to run on Linux, or your Linux programs to run on OS X).

However, the common thread between all executable file formats is that they include a predefined, standardised header which describes how program code and data are stored in the rest of the file. In words, it would generally describe "the program code starts 20 bytes into this file, and is 50 kilobytes long. The program data follows it and is 20 kilobytes long".

In recent times one particular format has become the de facto standard for executable representation for modern UNIX type systems. It is called the `Executable and Linker Format`, or ELF for short; we'll be looking at it in more detail soon.

[1] BSS probably stands for Block Started by Symbol, an assembly command for a old IBM computer.

Binary Format History

a.out

ELF was not always the standard; original UNIX systems used a file format called `a.out`. We can see the vestiges of this if you compile a program without the `-o` option to specify an output file name; the executable will be created with a default name of `a.out`[2].

`a.out` is a very simple header format that only allows a single data, code and BSS section. As you will come to see, this is insufficient for modern systems with dynamic libraries.

COFF

The Common Object File Format, or COFF, was the precursor to ELF. It's header format was more flexible, allowing an more (but limited) sections in the file.

COFF also has difficulties with elegant support of shared libraries, and ELF was selected as an alternative implementation on Linux.

However, COFF lives on in Microsoft Windows as the `Portable Executable` or PE format. PE is to Windows as ELF is to Linux.

ELF

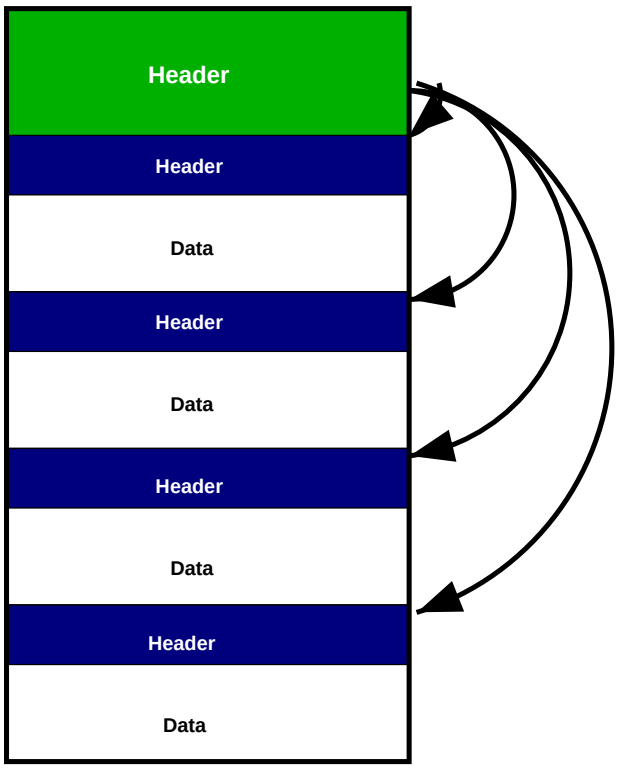
ELF is an extremely flexible format for representing binary code in a system. By following the ELF standard you can represent a kernel binary just as easily as a normal executable or a system library. The same tools can be used to inspect and operate on all ELF files and developers who understand the ELF file format can translate their skills to most modern UNIX systems.

ELF in depth

ELF extends on COFF and gives the header sufficient flexibility to define an arbitrary number of sections, each with it's own properties. This facilitates easier dynamic linking and debugging.

[2] In fact, `a.out` is the default output filename from the *linker*. The compiler generally uses randomly generated file names as intermediate files for assembly and object code.

Figure 8.1. ELF Overview



ELF File Header

Overall, the file has a *file header* which describes the file in general and then has pointers to each of the individual sections that make up the file.

Example 8.1. The ELF Header

```
 1
              typedef struct {
           unsigned char e_ident[EI_NIDENT];
           Elf32_Half    e_type;
 5         Elf32_Half    e_machine;
           Elf32_Word    e_version;
           Elf32_Addr    e_entry;
           Elf32_Off     e_phoff;
           Elf32_Off     e_shoff;
10         Elf32_Word    e_flags;
           Elf32_Half    e_ehsize;
           Elf32_Half    e_phentsize;
           Elf32_Half    e_phnum;
           Elf32_Half    e_shentsize;
15         Elf32_Half    e_shnum;
           Elf32_Half    e_shstrndx;
   } Elf32_Ehdr;
```

Above is the description as given in the API documentation. This is the layout of the C structure which defines a ELF header.

Example 8.2. The ELF Header, as shown by readelf

```
 1
              $ readelf --header /bin/ls

   ELF Header:
 5   Magic:   7f 45 4c 46 01 02 01 00 00 00 00 00 00 00 00 00
     Class:                             ELF32
     Data:                              2's complement, big endian
     Version:                           1 (current)
     OS/ABI:                            UNIX - System V
10   ABI Version:                       0
     Type:                              EXEC (Executable file)
     Machine:                           PowerPC
     Version:                           0x1
     Entry point address:               0x10002640
15   Start of program headers:          52 (bytes into file)
     Start of section headers:          87460 (bytes into file)
     Flags:                             0x0
     Size of this header:               52 (bytes)
     Size of program headers:           32 (bytes)
20   Number of program headers:         8
     Size of section headers:           40 (bytes)
     Number of section headers:         29
     Section header string table index: 28

25   [...]
```

Above is a more human readable form as present by the readelf program, which is part of GNU binutils.

The `e_ident` array is the first thing at the start of any ELF file, and always starts with a few "magic" bytes. The first byte is 0x7F and then the next three bytes are "ELF". You can inspect an ELF binary to see this for yourself with something like the **hexdump** command.

Example 8.3. Inspecting the ELF magic number

```
 1
              ianw@mingus:~$ hexdump -C /bin/ls | more
   00000000  7f 45 4c 46 01 02 01 00  00 00 00 00 00 00 00 00  |.ELF............|

 5 ... (rest of the program follows) ...
```

Note the 0x7F to start, then the ASCII encoded "ELF" string. Have a look at the standard and see what the rest of the array defines and what the values are in a binary.

Next we have some flags for the type of machine this binary is created for. The first thing we can see is that ELF defines different type sized versions, one for 32 bit and one for 64 bit versions; here we inspect the 32 bit version. The difference is mostly that on 64 bit machines addresses obviously required to be held in 64 bit variables. We can see that the binary has been created for a big endian machine that uses 2's complement to represent negative numbers. Skipping down a bit we can see the `Machine` tells us this is a PowerPC binary.

The apparently innocuous entry point address seems straight forward enough; this is the address in memory that the program code starts at.

Beginning C programmers are told that *main()* is the first program called in your program. Using the entry point address we can actually verify that it *isn't*.

Example 8.4. Investigating the entry point

```
 1
              $ cat test.c
   #include <stdio.h>

 5 int main(void)
   {
           printf("main is : %p\n", &main);
           return 0;
   }
10
   $ gcc -Wall -o test test.c

   $ ./test
   main is : 0x10000430
15
   $ readelf --headers ./test | grep 'Entry point'
     Entry point address:               0x100002b0

   $ objdump --disassemble ./test | grep 100002b0
20 100002b0 <_start>:
   100002b0:       7c 29 0b 78     mr      r9,r1
```

Above we can see that the entry point is actually a function called `_start`. Our program didn't define this at all, and the leading underscore suggests that it is in a separate *namespace*. We examine how a program starts up below.

After that the header contains pointers to where in the file other important parts of the ELF file start, like a table of contents.

Symbols and Relocations

The ELF specification provides for *symbol tables* which are simply mappings of strings (symbols) to locations in the file. Symbols are required for linking; for example assigning a value to a variable `foo` declared as `extern int foo;` would require the linker to find the address of `foo`, which would involve looking up "foo" in the symbol table and finding the address.

Closely related to symbols are *relocations*. A relocation is simply a blank space left to be patched up later. In the previous example, until the address of `foo` is known it can not be used. However, on a 32-bit system, we know the *address* of `foo` must be a 4-byte value, so any time the compiler needs to use that address (to say, assign a value) it can simply leave 4-byes of blank space and keep a relocation that essentially says to the linker "place the real value of "foo" into the 4 bytes at this address". As mentioned, this requires the symbol "foo" to be resolved. the section called “Relocations” contains further information on relocations.

Sections and Segments

The ELF format specifies two "views" of an ELF file -- that which is used for linking and that which is used for execution. This affords significant flexibility for systems designers.

We talk about *sections* in object code waiting to be linked into an executable. One or more sections map to a *segment* in the executable.

Segments

As we have done before, it is sometimes easier to look at the higher level of abstraction (segments) before inspecting the lower layers.

As we mentioned the ELF file has an header that describes the overall layout of the file. The ELF header actually points to another group of headers called the *program headers*. These headers describe to the operating system anything that might be required for it to load the binary into memory and execute it. Segments are described by program headers, but so are some other things required to get the executable running.

Example 8.5. The Program Header

```
 1
                typedef struct {
             Elf32_Word p_type;
             Elf32_Off  p_offset;
 5           Elf32_Addr p_vaddr;
             Elf32_Addr p_paddr;
             Elf32_Word p_filesz;
             Elf32_Word p_memsz;
             Elf32_Word p_flags;
10           Elf32_Word p_align;
   }
```

The definition of the program header is seen above. You might have noticed from the ELF header definition above how there were fields `e_phoff`, `e_phnum` and `e_phentsize`; these are simply the offset in the file where the program headers start, how many program headers there are and how big each program header is. With these three bits of information you can easily find and read the program headers.

As we mentioned, program headers more than just segments. The `p_type` field defines just what the program header is defining. For example, if this field is `PT_INTERP` the header is defined as meaning a string pointer to an *interpreter* for the binary file. We discussed compiled versus interpreted languages previously and made the distinction that a compiler builds a binary which can be run in a stand alone fashion. Why should it need an interpreter? As always, the true picture is a little more complicated. There are several reasons why a modern system wants flexibility when loading executable files, and to do this some information can only be adequately acquired at the actual time the program is set up to run. We see this in future chapters where we look into dynamic linking. Consequently some minor changes might need to be made to the binary to allow it to work properly at runtime. Thus the usual interpreter of a binary file is the *dynamic loader*, so called because it takes the final steps to complete loading of the executable and prepare the binary image for running.

Segments are described with a value of `PT_LOAD` in the `p_type` field. Each segment is then described by the other fields in the program header. The `p_offset` field tells you how far into the file on disk the data for the segment is. The `p_vaddr` field tells you

what address that data is to live at in virtual memory (`p_addr` describes the physical address, which is only really useful for small embedded systems that do not implement virtual memory). The two flags `p_filesz` and `p_memsz` work to tell you how big the segment is on disk and how big it should be in memory. If the memory size is greater than the disk size, then the overlap should be filled with zeros. In this way you can save considerable space in your binaries by not having to waste space for empty global variables. Finally `p_flags` indicates the permissions on the segment. Execute, read and write permissions can be specified in any combination; for example code segments should be marked as read and execute only, data sections as read and write with no execute.

There are a few other segment types defined in the program headers, they are described more fully in the standards specification (XXX).

Sections

As we have mentioned, sections make up segments. Sections are a way to organise the binary into logical areas to communicate information between the compiler and the linker. In some special binaries, such as the Linux kernel, sections are used in more specific ways.

We've seen how segments ultimately come down to a blob of data in a file on disk with some descriptions about where it should be loaded and what permissions it has. (XXX)

Sections have a similar header to segments.

Example 8.6. Sections

```
 1
                typedef struct {
             Elf32_Word sh_name;
             Elf32_Word sh_type;
 5           Elf32_Word sh_flags;
             Elf32_Addr sh_addr;
             Elf32_Off  sh_offset;
             Elf32_Word sh_size;
             Elf32_Word sh_link;
10           Elf32_Word sh_info;
             Elf32_Word sh_addralign;
             Elf32_Word sh_entsize;
   }

15
```

Sections have a few more types defined for the `sh_type` field; for example a section of type `SH_PROGBITS` is defined as a section that hold binary data for use by the program. Other flags say if this section is a symbol table (used by the linker or debugger for example) or maybe something for the dynamic loader.

There are also more attributes, such as the *allocate* attribute which flags that this section will need memory allocated for it.

It is probably best to examine sections through an example of them in use. Consider the following program.

Example 8.7. Sections

```
 1
                #include <stdio.h>

   int big_big_array[10*1024*1024];
 5
   char *a_string = "Hello, World!";

   int a_var_with_value = 0x100;

10 int main(void)
   {
    big_big_array[0] = 100;
    printf("%s\n", a_string);
    a_var_with_value += 20;
15 }
```

Example 8.8. Sections readelf output

```
 1
                $ readelf --all ./sections
   ELF Header:
    ...
 5   Size of section headers:           40 (bytes)
     Number of section headers:         37
     Section header string table index: 34

   Section Headers:
10   [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
     [ 0]                   NULL            00000000 000000 000000 00      0   0  0
     [ 1] .interp           PROGBITS        10000114 000114 00000d 00   A  0   0  1
     [ 2] .note.ABI-tag     NOTE            10000124 000124 000020 00   A  0   0  4
     [ 3] .hash             HASH            10000144 000144 00002c 04   A  4   0  4
15   [ 4] .dynsym           DYNSYM          10000170 000170 000060 10   A  5   1  4
     [ 5] .dynstr           STRTAB          100001d0 0001d0 00005e 00   A  0   0  1
     [ 6] .gnu.version      VERSYM          1000022e 00022e 00000c 02   A  4   0  2
     [ 7] .gnu.version_r    VERNEED         1000023c 00023c 000020 00   A  5   1  4
     [ 8] .rela.dyn         RELA            1000025c 00025c 00000c 0c   A  4   0  4
20   [ 9] .rela.plt         RELA            10000268 000268 000018 0c   A  4  25  4
     [10] .init             PROGBITS        10000280 000280 000028 00  AX  0   0  4
     [11] .text             PROGBITS        100002b0 0002b0 000560 00  AX  0   0 16
     [12] .fini             PROGBITS        10000810 000810 000020 00  AX  0   0  4
     [13] .rodata           PROGBITS        10000830 000830 000024 00   A  0   0  4
25   [14] .sdata2           PROGBITS        10000854 000854 000000 00   A  0   0  4
     [15] .eh_frame         PROGBITS        10000854 000854 000004 00   A  0   0  4
     [16] .ctors            PROGBITS        10010858 000858 000008 00  WA  0   0  4
     [17] .dtors            PROGBITS        10010860 000860 000008 00  WA  0   0  4
     [18] .jcr              PROGBITS        10010868 000868 000004 00  WA  0   0  4
30   [19] .got2             PROGBITS        1001086c 00086c 000010 00  WA  0   0  1
     [20] .dynamic          DYNAMIC         1001087c 00087c 0000c8 08  WA  5   0  4
     [21] .data             PROGBITS        10010944 000944 000008 00  WA  0   0  4
     [22] .got              PROGBITS        1001094c 00094c 000014 04 WAX  0   0  4
     [23] .sdata            PROGBITS        10010960 000960 000008 00  WA  0   0  4
35   [24] .sbss             NOBITS          10010968 000968 000000 00  WA  0   0  1
     [25] .plt              NOBITS          10010968 000968 000060 00 WAX  0   0  4
     [26] .bss              NOBITS          100109c8 000968 2800004 00  WA  0   0  4
     [27] .comment          PROGBITS        00000000 000968 00018f 00      0   0  1
     [28] .debug_aranges    PROGBITS        00000000 000af8 000078 00      0   0  8
40   [29] .debug_pubnames   PROGBITS        00000000 000b70 000025 00      0   0  1
     [30] .debug_info       PROGBITS        00000000 000b95 0002e5 00      0   0  1
     [31] .debug_abbrev     PROGBITS        00000000 000e7a 000076 00      0   0  1
     [32] .debug_line       PROGBITS        00000000 000ef0 0001de 00      0   0  1
     [33] .debug_str        PROGBITS        00000000 0010ce 0000f0 01  MS  0   0  1
45   [34] .shstrtab         STRTAB          00000000 0011be 00013b 00      0   0  1
     [35] .symtab           SYMTAB          00000000 0018c4 000c90 10     36  65  4
```

```
     [36] .strtab           STRTAB          00000000 002554 000909 00      0   0  1
   Key to Flags:
     W (write), A (alloc), X (execute), M (merge), S (strings)
50   I (info), L (link order), G (group), x (unknown)
     O (extra OS processing required) o (OS specific), p (processor specific)

   There are no section groups in this file.
    ...
55
   Symbol table '.symtab' contains 201 entries:
      Num:    Value  Size Type    Bind   Vis      Ndx Name
   ...
       99: 100109cc 0x2800000 OBJECT  GLOBAL DEFAULT   26 big_big_array
60 ...
      110: 10010960     4 OBJECT  GLOBAL DEFAULT   23 a_string
   ...
      130: 10010964     4 OBJECT  GLOBAL DEFAULT   23 a_var_with_value
   ...
65    144: 10000430    96 FUNC    GLOBAL DEFAULT   11 main
```

Above we have stripped some parts of the readelf output for clarity. We can analyse each part of our simple program and see what happens to it.

Firstly, let us look at the variable `big_big_array`, which as the name suggests is a fairly large global array. If we skip down to the symbol table we can see that the variable is at location `0x100109cc` which we can correlate to the `.bss` section in the section listing, since it starts just below it at `0x100109c8`. Note the size, and how it is quite large. We mentioned that BSS is a standard part of a binary image since it would be silly to require that binary on disk have 10 megabytes of space allocated to it, when all of that space is going to be zero. Note that this section has a type of `NOBITS` meaning that it does not have any bytes on disk.

Thus the `.bss` section is defined for global variables whose value should be zero when the program starts. We have seen how the memory size can be different to the on disk size in our discussion of segments; variables being in the `.bss` section are an indication that they will be given zero value on program start.

The `a_string` variable lives in the `.sdata` section, which stands for *small data*. Small data (and the corresponding `.sbss` section) are sections that can be reached by an offset from some known pointer. This means it is much faster to get to data in the sections as there are no extra lookups and loading of addresses into memory required. On the other hand, most architectures are limited to the size of immediate values you can add to a register (e.g. saying `r1 = add r2, 70;` 70 is an immediate value, as opposed to say, adding two values stored in registers `r1 = add r2,r3`) and can thus only offset a certain "small" distance from an address (XXX).

We can also see that our `a_var_with_value` lives in the same place.

`main` however lives in the `.text` section, as we expect (remember the name "text" and "code" are used interchangeably to refer to a program in memory.

Sections and Segments together

Example 8.9. Sections and Segments

```
1
```

```
                $ readelf --segments /bin/ls

   Elf file type is EXEC (Executable file)
 5 Entry point 0x100026c0
   There are 8 program headers, starting at offset 52

   Program Headers:
     Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
10   PHDR           0x000034 0x10000034 0x10000034 0x00100 0x00100 R E 0x4
     INTERP         0x000154 0x10000154 0x10000154 0x0000d 0x0000d R   0x1
         [Requesting program interpreter: /lib/ld.so.1]
     LOAD           0x000000 0x10000000 0x10000000 0x14d5c 0x14d5c R E 0x10000
     LOAD           0x014d60 0x10024d60 0x10024d60 0x002b0 0x00b7c RWE 0x10000
15   DYNAMIC        0x014f00 0x10024f00 0x10024f00 0x000d8 0x000d8 RW  0x4
     NOTE           0x000164 0x10000164 0x10000164 0x00020 0x00020 R   0x4
     GNU_EH_FRAME   0x014d30 0x10014d30 0x10014d30 0x0002c 0x0002c R   0x4
     GNU_STACK      0x000000 0x00000000 0x00000000 0x00000 0x00000 RWE 0x4

20  Section to Segment mapping:
     Segment Sections...
      00
      01     .interp
      02     .interp .note.ABI-tag .hash .dynsym .dynstr .gnu.version .gnu.version_ r .rela.dyn .rela.plt
25    03     .data .eh_frame .got2 .dynamic .ctors .dtors .jcr .got .sdata .sbss .p lt .bss
      04     .dynamic
      05     .note.ABI-tag
      06     .eh_frame_hdr
      07
30
```

`readelf` shows us the segments and section mappings in the ELF file for the binary `/bin/ls`.

Skipping to the bottom of the output, we can see what sections have been moved into what segments. So, for example the `.interp` section is placed into an `INTERP` flagged segment. Notice that readelf tells us it is requesting the interpreter `/lib/ld.so.1`; this is the dynamic linker which is run to prepare the binary for execution.

Looking at the two `LOAD` segments we can see the distinction between text and data. Notice how the first one has only "read" and "execute" permissions, whilst the next one has read, write and execute permissions? These describe the code (r/w) and data (r/w/e) segments.

But data should not need to be executable! Indeed, on most architectures (for example, the most common x86) the data section will not be marked as having the data section executable. However, the example output above was taken from a PowerPC machine which has a slightly different programming model (ABI, see below) requiring that the data section be executable [3]. Such is the life of a systems programmer, where rules were made to be broken!

The other interesting thing to note is that the file size is the same as the memory size for the code segment, however memory size is greater than the file size for the data segment. This comes from the BSS section which holds zeroed global variables.

[3]For those that are curious, the PowerPC ABI calls stubs for functions in dynamic libraries directly in the GOT, rather than having them bounce through a separate PLT entry. Thus the processor needs execute permissions for the GOT section, which you can see is embedded in the data segment. This should make sense after reading the dynamic linking chapter!

Debugging

Traditionally the primary method of post mortem debugging is referred to as the *core dump*. The term *core* comes from the original physical characteristics of magnetic core memory, which uses the orientation of small magnetic rings to store state.

Thus a core dump is simply a complete snapshot of the program as it was running at a particular time. A *debugger* can then be used to examine this dump and reconstruct the program state. Example 8.10, “Example of creating a core dump and using it with gdb” shows a sample program that writes to a random memory location in order to force a crash. At this point the processes will be halted and a dump of the current state is recorded.

Example 8.10. Example of creating a core dump and using it with gdb

```
 1
            $ cat coredump.c
   int main(void) {
    char *foo = (char*)0x12345;
 5  *foo = 'a';

    return 0;
   }

10 $ gcc -Wall -g -o coredump coredump.c

   $ ./coredump
   Segmentation fault (core dumped)

15 $ file ./core
   ./core: ELF 32-bit LSB core file Intel 80386, version 1 (SYSV), SVR4-style, from './coredump'

   $ gdb ./coredump
   ...
20 (gdb) core core
   [New LWP 31614]
   Core was generated by `./coredump'.
   Program terminated with signal 11, Segmentation fault.
   #0  0x080483c4 in main () at coredump.c:3
25 3  *foo = 'a';
   (gdb)
```

Symbols and Debugging Information

As the example shows, the debugger gdb requires the original executable and the core dump to provide the debugging session. Note that the original executable was built with the `-g` flag, which instructs the compiler to include all *debugging information*. Debugging information created by the compiler and is kept in special sections of the ELF file. It describes in detail things like what register values currently hold which variables used in the code, size of variables, length of arrays, etc. It is generally in the standard *DWARF* format (a pun on the homonym ELF).

Including debugging information can make executable files and libraries very large; although this data is not required resident in memory for actually running it can still take up considerable disk space. Thus the usual process is to *strip* this information from the ELF file. While it is possible to arrange for shipping of both stripped and

unstripped files, most all current binary distribution methods provide the debugging information in separate files. The objcopy tool can be used to extract the debugging information (`--only-keep-debug`) and then add a link in the original executable to this stripped information (`--add-gnu-debuglink`). After this is done, a special section called `.gnu_debuglink` will be present in the original executable, which contains a hash so that when a debugging sessions starts the debugger can be sure it associates the right debugging information with the right executable.

Example 8.11. Example of stripping debugging information into separate files using objcopy

```
 1
              $ gcc -g -shared -o libtest.so libtest.c
   $ objcopy --only-keep-debug libtest.so libtest.debug
   $ objcopy --add-gnu-debuglink=libtest.debug libtest.so
 5 $ objdump -s -j .gnu_debuglink libtest.so

   libtest.so:     file format elf32-i386

   Contents of section .gnu_debuglink:
10  0000 6c696274 6573742e 64656275 67000000  libtest.debug...
    0010 52a7fd0a                             R...
```

Symbols take up much less space, but are also targets for removal from final output. Once the individual object files of an executable are linked into the single final image there is generally no need for most symbols to remain. As discussed in the section called “Symbols and Relocations” symbols are required to fix up relocation entries, but once this is done the symbols are not strictly necessary for running the final program. On Linux the GNU toolchain strip program provides options to remove symbols. Note that some symbols are required to be resolved at run-time (for *dynamic linking*, the focus of Chapter 9, *Dynamic Linking*) but these are put in separate *dynamic* symbol tables so they will not be removed and render the final output useless.

Inside coredumps

A coredump is really just another ELF file; this illustrates the flexibility of ELF as a binary format.

Example 8.12. Example of using readelf and eu-readelf to examine a coredump.

```
 1
              $ readelf --all ./core
   ELF Header:
     Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
 5   Class:                             ELF32
     Data:                              2's complement, little endian
     Version:                           1 (current)
     OS/ABI:                            UNIX - System V
     ABI Version:                       0
10   Type:                              CORE (Core file)
     Machine:                           Intel 80386
     Version:                           0x1
     Entry point address:               0x0
     Start of program headers:          52 (bytes into file)
15   Start of section headers:          0 (bytes into file)
     Flags:                             0x0
```

```
     Size of this header:               52 (bytes)
     Size of program headers:           32 (bytes)
     Number of program headers:         15
     Size of section headers:           0 (bytes)
     Number of section headers:         0
     Section header string table index: 0

   There are no sections in this file.

   There are no sections to group in this file.

   Program Headers:
     Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
     NOTE           0x000214 0x00000000 0x00000000 0x0022c 0x00000     0
     LOAD           0x001000 0x08048000 0x00000000 0x01000 0x01000 R E 0x1000
     LOAD           0x002000 0x08049000 0x00000000 0x01000 0x01000 RW  0x1000
     LOAD           0x003000 0x489fc000 0x00000000 0x01000 0x1b000 R E 0x1000
     LOAD           0x004000 0x48a17000 0x00000000 0x01000 0x01000 R   0x1000
     LOAD           0x005000 0x48a18000 0x00000000 0x01000 0x01000 RW  0x1000
     LOAD           0x006000 0x48a1f000 0x00000000 0x01000 0x153000 R E 0x1000
     LOAD           0x007000 0x48b72000 0x00000000 0x00000 0x01000     0x1000
     LOAD           0x007000 0x48b73000 0x00000000 0x02000 0x02000 R   0x1000
     LOAD           0x009000 0x48b75000 0x00000000 0x01000 0x01000 RW  0x1000
     LOAD           0x00a000 0x48b76000 0x00000000 0x03000 0x03000 RW  0x1000
     LOAD           0x00d000 0xb771c000 0x00000000 0x01000 0x01000 RW  0x1000
     LOAD           0x00e000 0xb774d000 0x00000000 0x02000 0x02000 RW  0x1000
     LOAD           0x010000 0xb774f000 0x00000000 0x01000 0x01000 R E 0x1000
     LOAD           0x011000 0xbfeac000 0x00000000 0x22000 0x22000 RW  0x1000

   There is no dynamic section in this file.

   There are no relocations in this file.

   There are no unwind sections in this file.

   No version information found in this file.

   Notes at offset 0x00000214 with length 0x0000022c:
     Owner                 Data size Description
     CORE                  0x00000090 NT_PRSTATUS (prstatus structure)
     CORE                  0x0000007c NT_PRPSINFO (prpsinfo structure)
     CORE                  0x000000a0 NT_AUXV (auxiliary vector)
     LINUX                 0x00000030 Unknown note type: (0x00000200)

   $ eu-readelf -n ./core

   Note segment of 556 bytes at offset 0x214:
     Owner          Data size  Type
     CORE                 144  PRSTATUS
       info.si_signo: 11, info.si_code: 0, info.si_errno: 0, cursig: 11
       sigpend: <>
       sighold: <>
       pid: 31614, ppid: 31544, pgrp: 31614, sid: 31544
       utime: 0.000000, stime: 0.000000, cutime: 0.000000, cstime: 0.000000
       orig_eax: -1, fpvalid: 0
       ebx:     1219973108  ecx:     1243440144  edx:              1
       esi:              0  edi:              0  ebp:     0xbfecb828
       eax:          74565  eip:     0x080483c4  eflags:  0x00010286
       esp:     0xbfecb818
       ds: 0x007b  es: 0x007b  fs: 0x0000  gs: 0x0033  cs: 0x0073  ss: 0x007b
     CORE                 124  PRPSINFO
       state: 0, sname: R, zomb: 0, nice: 0, flag: 0x00400400
       uid: 1000, gid: 1000, pid: 31614, ppid: 31544, pgrp: 31614, sid: 31544
       fname: coredump, psargs: ./coredump
     CORE                 160  AUXV
       SYSINFO: 0xb774f414
       SYSINFO_EHDR: 0xb774f000
       HWCAP: 0xafe8fbff  <fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov clflush dts acpi
       PAGESZ: 4096
       CLKTCK: 100
```

```
        PHDR: 0x8048034
        PHENT: 32
        PHNUM: 8
 90     BASE: 0
        FLAGS: 0
        ENTRY: 0x8048300
        UID: 1000
        EUID: 1000
 95     GID: 1000
        EGID: 1000
        SECURE: 0
        RANDOM: 0xbfecba1b
        EXECFN: 0xbfecdff1
100     PLATFORM: 0xbfecba2b
        NULL
      LINUX                48  386_TLS
        index: 6, base: 0xb771c8d0, limit: 0x000fffff, flags: 0x00000051
        index: 7, base: 0x00000000, limit: 0x00000000, flags: 0x00000028
105     index: 8, base: 0x00000000, limit: 0x00000000, flags: 0x00000028
```

In Example 8.12, “Example of using readelf and eu-readelf to examine a coredump.” we can see an examination of the core file produced by Example 8.10, “Example of creating a core dump and using it with gdb” using firstly the readelf tool. There are no sections, relocations or other extraneous information in the file that may be required for loading an executable or library; it simply consists of a series of program headers describing `LOAD` segments. These segments are raw data dumps, created by the kernel, of the current memory allocations.

The other component of the core dump is the `NOTE` sections which contain data necessary for debugging but not necessarily captured in straight snapshot of the memory allocations. The eu-readelf program used in the second part of the figure provides a more complete view of the data by decoding it.

The PRSTATUS note gives a range of interesting information about the process as it was running; for example we can see from `cursig` that the program received a signal 11, or segmentation fault, as we would expect. Along with process number information, it also includes a dump of all the current registers. Given the register values, the debugger can reconstruct the stack state and hence provide a *backtrace*; combined with the symbol and debugging information from the original binary the debugger can show exactly how you reached the current point of execution.

Another interesting output is the current *auxiliary vector* (AUXV), discussed in the section called “Kernel communication to programs”. The `386_TLS` describes *global descriptor table* entries used for the x86 implementation of *thread-local storage* (see the section called “Fast System Calls” for more information on use of segmentation, and the section called “Threads” for information on threads[4]).

The kernel creates the core dump file within the bounds of the current `ulimit` settings — since a program using a lot of memory could result in a very large dump, potentially filling up disk and making problems even worse, generally the `ulimit` is set low or even at zero, since most non-developers have little use for a core dump file. However the

[4]For a multi-threaded application, there would be duplicate entries for each thread running. The debugger will understand this, and it is how gdb implements the `thread` command to show and switch between threads.

core dump remains the single most useful way to debug an unexpected situation in a postmortem fashion.

ELF Executables

Executables are of course one of the primary uses of the ELF format. Contained within the *binary* is everything required for the operating system to execute the code as intended.

Since an executable is designed to be run in a process with a unique address space (see Chapter 6, *Virtual Memory*) the code can make assumptions about where the various parts of the program will be loaded in memory. Example 8.13, “Segments of an executable file” shows an example using the readelf tool to examine the segments of an executable file. We can see the virtual addresses at which the `LOAD` segments are required to be placed at. We can further see that one segment is for code — it has read and execute permissions only — and one is for data, unsurprisingly with read and write permissions, but importantly no execute permissions (without execute permissions, even if a bug allowed an attacker to introduce arbitrary data the pages backing it would not be marked with execute permissions, and most processors will hence disallow any execution of code in those pages).

Example 8.13. Segments of an executable file

```
 1
          $ readelf --segments /bin/ls

  Elf file type is EXEC (Executable file)
 5 Entry point 0x4046d4
  There are 8 program headers, starting at offset 64

  Program Headers:
    Type           Offset             VirtAddr           PhysAddr
10                 FileSiz            MemSiz              Flags  Align
    PHDR           0x0000000000000040 0x0000000000400040 0x0000000000400040
                   0x00000000000001c0 0x00000000000001c0  R E    8
    INTERP         0x0000000000000200 0x0000000000400200 0x0000000000400200
                   0x000000000000001c 0x000000000000001c  R      1
15       [Requesting program interpreter: /lib64/ld-linux-x86-64.so.2]
    LOAD           0x0000000000000000 0x0000000000400000 0x0000000000400000
                   0x0000000000019ef4 0x0000000000019ef4  R E    200000
    LOAD           0x000000000001a000 0x000000000061a000 0x000000000061a000
                   0x000000000000077c 0x0000000000001500  RW     200000
20  DYNAMIC        0x000000000001a028 0x000000000061a028 0x000000000061a028
                   0x00000000000001d0 0x00000000000001d0  RW     8
    NOTE           0x000000000000021c 0x000000000040021c 0x000000000040021c
                   0x0000000000000044 0x0000000000000044  R      4
    GNU_EH_FRAME   0x0000000000017768 0x0000000000417768 0x0000000000417768
25                 0x00000000000006fc 0x00000000000006fc  R      4
    GNU_STACK      0x0000000000000000 0x0000000000000000 0x0000000000000000
                   0x0000000000000000 0x0000000000000000  RW     8

   Section to Segment mapping:
30  Segment Sections...
     00
     01     .interp
     02     .interp .note.ABI-tag .note.gnu.build-id .hash .gnu.hash .dynsym .dynstr .gnu.version .gnu.ve
     03     .ctors .dtors .jcr .dynamic .got .got.plt .data .bss
35   04     .dynamic
     05     .note.ABI-tag .note.gnu.build-id
     06     .eh_frame_hdr
```

```
      07

40
```

The program segments must be loaded at these addresses; the last step of the linker is to resolve most relocations (the section called “Symbols and Relocations”) and patch them with the assumed absolute addresses — the data describing the relocation is then discarded in the final binary and there is no longer a way to find this information.

In reality, executables generally have external dependencies on *shared libraries*, or pieces of common code abstracted and shared among the entire system — almost all of the confusing parts of Example 8.13, “Segments of an executable file” relate to the use of shared libraries. Libraries are discussed in the section called “Libraries”, dynamic libraries in Chapter 9, *Dynamic Linking*.

Libraries

Developers soon tired of having to write everything from scratch, so one of the first inventions of computer science was *libraries*.

A library is simply a collection of functions which you can call from your program. Obviously a library has many advantages, not least of which is that you can save much time by reusing work someone else has already done and generally be more confident that it has fewer bugs (since probably many other people use the libraries too, and you benefit from having them finding and fixing bugs). A library is exactly like an executable, except instead of running directly the library functions are invoked with parameters from your executable.

Static Libraries

The most straight forward way of using a library function is to have the object files from the library linked directly into your final executable, just as with those you have compiled yourself. When linked like this the library is called a *static* library, because the library will remain unchanged unless the program is recompiled.

This is the most straight forward way of using a library as the final result is a simple executable with no dependencies.

Inside static libraries

A static library is simply a group of object files. The object files are kept in an *archive*, which leads to their usual `.a` suffix extension. You can think of archives as similar to a **zip** file, but without compression.

Below we show the creation of basic static library and introduce some common tools for working with libraries.

Example 8.14. Creating and using a static library

```
 1
```

```
               $ cat library.c
   /* Library Function */
   int function(int input)
 5 {
           return input + 10;
   }

   $ cat library.h
10 /* Function Definition */
   int function(int);

   $ cat program.c
   #include <stdio.h>
15 /* Library header file */
   #include "library.h"

   int main(void)
   {
20         int d = function(100);

           printf("%d\n", d);
   }

25 $ gcc -c library.c
   $ ar rc libtest.a library.o
   $ ranlib ./libtest.a
   $ nm --print-armap ./libtest.a

30 Archive index:
   function in library.o

   library.o:
   00000000 T function
35
   $ gcc -L . program.c -ltest -o program

   $ ./program
   110
40
```

Firstly we compile or library to an object file, just as we have seen in the previous chapter.

Notice that we define the library API in the header file. The API consists of function definitions for the functions in the library; this is so that the compiler knows what types the functions take when building object files that reference the library (e.g. `program.c`, which `#includes` the header file).

We create the library ar (short for "archive") command. By convention static library file names are prefixed with `lib` and have the extension `.a`. The `c` argument tells the program to create the archive, and a tells archive to add the object files specified into the library file.[5]

Next we use the ranlib application to make a header in the library with the symbols of the object file contents. This helps the compiler to quickly reference symbols; in the case where we just have one this step may seem a little redundant; however a large library may have thousands of symbols meaning an index can significantly speed up finding

[5]Archives created with ar pop up in a few different places around Linux systems other than just creating static libraries. One widely used application is in the `.deb` packaging format used with Debian, Ubuntu and some other Linux systems is one example. `debs` use archives to keep all the application files together in the one package file. RedHat RPM packages use an alternate but similar format called cpio. Of course the canonical application for keeping files together is the `tar` file, which is a common format to distribute source code.

references. We inspect this new header with the nm application. We see the `function` symbol for the `function()` function at offset zero, as we expect.

You then specify the library to the compiler with `-lname` where name is the filename of the library without the prefix `lib`. We also provide an extra search directory for libraries, namely the current directory (`-L .`), since by default the current directory is not searched for libraries.

The final result is a single executable with our new library included.

Static Linking Drawbacks

Static linking is very straight forward, but has a number of drawbacks.

There are two main disadvantages; firstly if the library code is updated (to fix a bug, say) you have to recompile your program into a new executable and secondly, every program in the system that uses that library contains a copy in it's executable. This is very inefficient (and a pain if you find a bug and have to recompile, as per point one).

For example, the C library (glibc) is included in all programs, and provides all the common functions such as `printf`.

Shared Libraries

Shared libraries are an elegant way around the problems posed by a static library. A shared library is a library that is loaded dynamically at runtime for each application that requires it.

The application simply leaves pointers that it will require a certain library, and when the function call is made the library is loaded into memory and executed. If the library is already loaded for another application, the code can be shared between the two, saving considerable resources with commonly used libraries.

This process, called dynamic linking, is one of the more intricate parts of a modern operating system. As such, we dedicate the next chapter to investigating the dynamic linking process.

ABI's

An ABI is a term you will hear a lot about when working with systems programming. We have talked extensively about *API*, which are interfaces the programmer sees to your code.

ABI's refer to lower level interfaces which the compiler, operating system and, to some extent, processor, must agree on to communicate together. Below we introduce a number of concepts which are important to understanding ABI considerations.

Byte Order

Endianess

Calling Conventions

Passing parameters

registers or stack?

Function Descriptors

On many architectures you must call a function through a *function descriptor*, rather than directly.

For example, on IA64 a function descriptor consists of two components; the address of the function (that being a 64 bit, or 8 byte value) and the address of the *global pointer* (gp). The ABI specifies that r1 should always contain the gp value for a function. This means that when you call a function, it is the `callees` job to save their gp value, set r1 to be the new value (from the function descriptor) and `then` call the function.

This may seem like a strange way to do things, but it has very useful practical implications as you will see in the next chapter about global offset tables. On IA64 an `add` instruction can only take a maximum 22 bit *immediate value*[6]. An immediate value is one that is specified directly, rather than in a register (e.g. in `add r1 + 100` 100 is the immediate value).

You might recognise 22 bits as being able to represent 4194304 bytes, or 4MB. Thus each function can directly offset into an area of memory 4MB big without having to take the penalty of loading any values into a register. If the compiler, linker and loader all agree on what the global pointer is pointing to (as specified in the ABI) performance can be improved by less loading.

Starting a process

We mentioned before that simply saying the program starts with the `main()` function is not quite true. Below we examine what happens to a typical dynamically linked program when it is loaded and run (statically linked programs are similar but different XXX should we go into this?).

Firstly, in response to an `exec` system call the kernel allocates the structures for a new process and reads the ELF file specified from disk.

We mentioned that ELF has a program interpreter field, `PT_INTERP`, which can be set to 'interpret' the program. For dynamically linked applications that interpreter is the dynamic linker, namely ld.so, which allows some of the linking process to be done on the fly before the program starts.

In this case, the kernel *also* reads in the dynamic linker code, and starts the program from the entry point address as specified by it. We examine the role of the dynamic

[6]Technically this is because of the way IA64 bundles instructions. Three instructions are put into each bundle, and there is only enough room to keep a 22 bit value to keep the bundle together.

linker in depth in the next chapter, but suffice to say it does some setup like loading any libraries required by the application (as specified in the dynamic section of the binary) and then starts execution of the program binary at it's entry point address (i.e. the `_init` function).

Kernel communication to programs

The kernel needs to communicate some things to programs when they start up; namely the arguments to the program, the current environment variables and a special structure called the `Auxiliary Vector` or `auxv` (you can request the the dynamic linker show you some debugging output of the `auxv` by specifying the environment value `LD_SHOW_AUXV=1`).

The arguments and environment at fairly straight forward, and the various incarnations of the `exec` system call allow you to specify these for the program.

The kernel communicates this by putting all the required information on the stack for the newly created program to pick up. Thus when the program starts it can use it's stack pointer to find the all the startup information required.

The auxiliary vector is a special structure that is for passing information directly from the kernel to the newly running program. It contains system specific information that may be required, such as the default size of a virtual memory page on the system or *hardware capabilities*; that is specific features that the kernel has identified the underlying hardware has that userspace programs can take advantage of.

Kernel Library

We mentioned previously that system calls are slow, and modern systems have mechanisms to avoid the overheads of calling a trap to the processor.

In Linux, this is implemented by a neat trick between the dynamic loader and the kernel, all communicated with the AUXV structure. The kernel actually adds a small shared library into the address space of every newly created process which contains a function that makes system calls for you. The beauty of this system is that if the underlying hardware supports a fast system call mechanism the kernel (being the creator of the library) can use it, otherwise it can use the old scheme of generating a trap. This library is named `linux-gate.so.1`, so called because it is a *gateway* to the inner workings of the kernel.

When the kernel starts the dynamic linker it adds an entry to the auxv called `AT_SYSINFO_EHDR`, which is the address in memory that the special kernel library lives in. When the dynamic linker starts it can look for the `AT_SYSINFO_EHDR` pointer, and if found load that library for the program. The program has no idea this library exists; this is a private arrangement between the dynamic linker and the kernel.

We mentioned that programmers make system calls indirectly through calling functions in the system libraries, namely libc. libc can check to see if the special kernel binary is loaded, and if so use the functions within that to make system calls. As we mentioned, if the kernel determines the hardware is capable, this will use the fast system call method.

Starting the program

Once the kernel has loaded the interpreter it passes it to the entry point as given in the interpreter file (note will not examine how the dynamic linker starts at this stage; see Chapter 9, *Dynamic Linking* for a full discussion of dynamic linking). The dynamic linker will jump to the entry point address as given in the ELF binary.

Example 8.15. Disassembley of program startup

```
 1
            $ cat test.c

   int main(void)
 5 {
    return 0;
   }

   $ gcc -o test test.c
10
   $ readelf --headers ./test | grep Entry
     Entry point address:               0x80482b0

   $ objdump --disassemble ./test
15
   [...]

   080482b0 <_start>:
    80482b0:       31 ed                   xor    %ebp,%ebp
20  80482b2:       5e                      pop    %esi
    80482b3:       89 e1                   mov    %esp,%ecx
    80482b5:       83 e4 f0                and    $0xfffffff0,%esp
    80482b8:       50                      push   %eax
    80482b9:       54                      push   %esp
25  80482ba:       52                      push   %edx
    80482bb:       68 00 84 04 08          push   $0x8048400
    80482c0:       68 90 83 04 08          push   $0x8048390
    80482c5:       51                      push   %ecx
    80482c6:       56                      push   %esi
30  80482c7:       68 68 83 04 08          push   $0x8048368
    80482cc:       e8 b3 ff ff ff          call   8048284 <__libc_start_main@plt>
    80482d1:       f4                      hlt
    80482d2:       90                      nop
    80482d3:       90                      nop
35
   08048368 <main>:
    8048368:       55                      push   %ebp
    8048369:       89 e5                   mov    %esp,%ebp
    804836b:       83 ec 08                sub    $0x8,%esp
40  804836e:       83 e4 f0                and    $0xfffffff0,%esp
    8048371:       b8 00 00 00 00          mov    $0x0,%eax
    8048376:       83 c0 0f                add    $0xf,%eax
    8048379:       83 c0 0f                add    $0xf,%eax
    804837c:       c1 e8 04                shr    $0x4,%eax
45  804837f:       c1 e0 04                shl    $0x4,%eax
    8048382:       29 c4                   sub    %eax,%esp
    8048384:       b8 00 00 00 00          mov    $0x0,%eax
    8048389:       c9                      leave
    804838a:       c3                      ret
50  804838b:       90                      nop
    804838c:       90                      nop
    804838d:       90                      nop
    804838e:       90                      nop
    804838f:       90                      nop
55
   08048390 <__libc_csu_init>:
    8048390:       55                      push   %ebp
```

```
    8048391:       89 e5                    mov    %esp,%ebp
    [...]
60
   08048400 <__libc_csu_fini>:
    8048400:       55                       push   %ebp
    [...]
```

Above we investigate the very simplest program. Using readelf we can see that the entry point is the `_start` function in the binary. At this point we can see in the disassembley some values are pushed onto the stack. The first value, `0x8048400` is the `__libc_csu_fini` function; `0x8048390` is the `__libc_csu_init` and then finally `0x8048368`, the `main()` function. After this the value `__libc_start_main` function is called.

`__libc_start_main` is defined in the glibc sources `sysdeps/generic/libc-start.c`. The file function is quite complicated and hidden between a large number of defines, as it needs to be portable across the very wide number of systems and architectures that glibc can run on. It does a number of specific things related to setting up the C library which the average programmer does not need to worry about. The next point where the library calls back into the program is to handle `init` code.

`init` and `fini` are two special concepts that call parts of code in shared libraries that may need to be called before the library starts or if the library is unloaded respectively. You can see how this might be useful for library programmers to setup variables when the library is started, or to clean up at the end. Originally the functions `_init` and `_fini` were looked for in the library; however this became somewhat limiting as everything was required to be in these functions. Below we will examine just how the `init`/`fini` process works.

At this stage we can see that the `__libc_start_main` function will receive quite a few input paramaters on the stack. Firstly it will have access to the program arguments, environment variables and auxiliary vector from the kernel. Then the initalization function will have pushed onto the stack addresses for functions to handle `init`, `fini`, and finally the address of the main function it's self.

We need some way to indicate in the source code that a function should be called by `init` or `fini`. With gcc we use *attributes* to label two functions as *constructors* and *destructors* in our main program. These terms are more commonly used with object oriented languages to describe object life cycles.

Example 8.16. Constructors and Destructors

```
 1
            $ cat test.c
   #include <stdio.h>

 5 void __attribute__((constructor)) program_init(void)  {
     printf("init\n");
   }

   void  __attribute__((destructor)) program_fini(void) {
10   printf("fini\n");
   }

   int main(void)
   {
```

```
15   return 0;
   }

   $ gcc -Wall  -o test test.c

20 $ ./test
   init
   fini

   $ objdump --disassemble ./test | grep program_init
25 08048398 <program_init>:

   $ objdump --disassemble ./test | grep program_fini
   080483b0 <program_fini>:

30 $ objdump --disassemble ./test

   [...]
   08048280 <_init>:
    8048280:       55                      push   %ebp
35  8048281:       89 e5                   mov    %esp,%ebp
    8048283:       83 ec 08                sub    $0x8,%esp
    8048286:       e8 79 00 00 00          call   8048304 <call_gmon_start>
    804828b:       e8 e0 00 00 00          call   8048370 <frame_dummy>
    8048290:       e8 2b 02 00 00          call   80484c0 <__do_global_ctors_aux>
40  8048295:       c9                      leave
    8048296:       c3                      ret
   [...]

   080484c0 <__do_global_ctors_aux>:
45  80484c0:       55                      push   %ebp
    80484c1:       89 e5                   mov    %esp,%ebp
    80484c3:       53                      push   %ebx
    80484c4:       52                      push   %edx
    80484c5:       a1 2c 95 04 08          mov    0x804952c,%eax
50  80484ca:       83 f8 ff                cmp    $0xffffffff,%eax
    80484cd:       74 1e                   je     80484ed <__do_global_ctors_aux+0x2d>
    80484cf:       bb 2c 95 04 08          mov    $0x804952c,%ebx
    80484d4:       8d b6 00 00 00 00       lea    0x0(%esi),%esi
    80484da:       8d bf 00 00 00 00       lea    0x0(%edi),%edi
55  80484e0:       ff d0                   call   *%eax
    80484e2:       8b 43 fc                mov    0xfffffffc(%ebx),%eax
    80484e5:       83 eb 04                sub    $0x4,%ebx
    80484e8:       83 f8 ff                cmp    $0xffffffff,%eax
    80484eb:       75 f3                   jne    80484e0 <__do_global_ctors_aux+0x20>
60  80484ed:       58                      pop    %eax
    80484ee:       5b                      pop    %ebx
    80484ef:       5d                      pop    %ebp
    80484f0:       c3                      ret
    80484f1:       90                      nop
65  80484f2:       90                      nop
    80484f3:       90                      nop


   $ readelf --sections ./test
70 There are 34 section headers, starting at offset 0xfb0:

   Section Headers:
     [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
     [ 0]                   NULL            00000000 000000 000000 00      0   0  0
75   [ 1] .interp           PROGBITS        08048114 000114 000013 00   A  0   0  1
     [ 2] .note.ABI-tag     NOTE            08048128 000128 000020 00   A  0   0  4
     [ 3] .hash             HASH            08048148 000148 00002c 04   A  4   0  4
     [ 4] .dynsym           DYNSYM          08048174 000174 000060 10   A  5   1  4
     [ 5] .dynstr           STRTAB          080481d4 0001d4 00005e 00   A  0   0  1
80   [ 6] .gnu.version      VERSYM          08048232 000232 00000c 02   A  4   0  2
     [ 7] .gnu.version_r    VERNEED         08048240 000240 000020 00   A  5   1  4
     [ 8] .rel.dyn          REL             08048260 000260 000008 08   A  4   0  4
     [ 9] .rel.plt          REL             08048268 000268 000018 08   A  4  11  4
     [10] .init             PROGBITS        08048280 000280 000017 00  AX  0   0  4
```

```
 85   [11] .plt              PROGBITS        08048298 000298 000040 04  AX  0   0  4
      [12] .text             PROGBITS        080482e0 0002e0 000214 00  AX  0   0 16
      [13] .fini             PROGBITS        080484f4 0004f4 00001a 00  AX  0   0  4
      [14] .rodata           PROGBITS        08048510 000510 000012 00   A  0   0  4
      [15] .eh_frame         PROGBITS        08048524 000524 000004 00   A  0   0  4
 90   [16] .ctors            PROGBITS        08049528 000528 00000c 00  WA  0   0  4
      [17] .dtors            PROGBITS        08049534 000534 00000c 00  WA  0   0  4
      [18] .jcr              PROGBITS        08049540 000540 000004 00  WA  0   0  4
      [19] .dynamic          DYNAMIC         08049544 000544 0000c8 08  WA  5   0  4
      [20] .got              PROGBITS        0804960c 00060c 000004 04  WA  0   0  4
 95   [21] .got.plt          PROGBITS        08049610 000610 000018 04  WA  0   0  4
      [22] .data             PROGBITS        08049628 000628 00000c 00  WA  0   0  4
      [23] .bss              NOBITS          08049634 000634 000004 00  WA  0   0  4
      [24] .comment          PROGBITS        00000000 000634 00018f 00      0   0  1
      [25] .debug_aranges    PROGBITS        00000000 0007c8 000078 00      0   0  8
100   [26] .debug_pubnames   PROGBITS        00000000 000840 000025 00      0   0  1
      [27] .debug_info       PROGBITS        00000000 000865 0002e1 00      0   0  1
      [28] .debug_abbrev     PROGBITS        00000000 000b46 000076 00      0   0  1
      [29] .debug_line       PROGBITS        00000000 000bbc 0001da 00      0   0  1
      [30] .debug_str        PROGBITS        00000000 000d96 0000f3 01  MS  0   0  1
105   [31] .shstrtab         STRTAB          00000000 000e89 000127 00      0   0  1
      [32] .symtab           SYMTAB          00000000 001500 000490 10     33  53  4
      [33] .strtab           STRTAB          00000000 001990 000218 00      0   0  1
    Key to Flags:
      W (write), A (alloc), X (execute), M (merge), S (strings)
110   I (info), L (link order), G (group), x (unknown)
      O (extra OS processing required) o (OS specific), p (processor specific)

    $ objdump --disassemble-all --section .ctors ./test

115 ./test:     file format elf32-i386

    Contents of section .ctors:
     8049528 ffffffff 98830408 00000000          ............

120
```

The last value pushed onto the stack for the `__libc_start_main` was the initialisation function `__libc_csu_init`. If we follow the call chain through from `__libc_csu_init` we can see it does some setup and then calls the `_init` function in the executable. The `_init` function eventually calls a function called `__do_global_ctors_aux`. Looking at the disassembley of this function we can see that it appears to start at address `0x804952c` and loop along, reading an value and calling it. We can see that this starting address is in the `.ctors` section of the file; if we have a look inside this we see that it contains the first value `-1`, a function address (in big endian format) and the value zero.

The address in big endian format is `0x08048398`, or the address of `program_init` function! So the format of the `.ctors` section is firstly a -1, and then the address of functions to be called on initialisation, and finally a zero to indicate the list is complete. Each entry will be called (in this case we only have the one function).

Once `__libc_start_main` has completed with the `_init` call it *finally* calls the `main()` function! Remember that it had the stack setup initially with the arguments and environment pointers from the kernel; this is how main gets it's `argc, argv[], envp[]` arguments. The process now runs and the setup phase is complete.

A similar process is enacted with the `.dtors` for destructors when the program exits. `__libc_start_main` calls these when the `main()` function completes.

As you can see, a lot is done before the program gets to start, and even a little after you think it is finished!

Chapter 9. Dynamic Linking

Code Sharing

We know that for the operating system code is considered read only, and separate from data. It seems logical then that if programs can not modify code and have large amounts of common code, instead of replicating it for every executable it should be shared between many executables.

With virtual memory this can be easily done. The physical pages of memory the library code is loaded into can be easily referenced by any number of virtual pages in any number of address spaces. So while you only have one physical copy of the library code in system memory, every process can have access to that library code at any virtual address it likes.

Thus people quickly came up with the idea of a *shared library* which, as the name suggests, is shared by multiple executables. Each executable contains a reference essentially saying "I need library foo". When the program is loaded, it is up to the system to either check if some other program has already loaded the code for library foo into memory, and thus share it by mapping pages into the executable for that physical memory, or otherwise load the library into memory for the executable.

This process is called *dynamic linking* because it does part of the linking process "on the fly" as programs are executed in the system.

Dynamic Library Details

Libraries are very much like a program that never gets started. They have code and data sections (functions and variables) just like every executable; but no where to start running. They just provide a library of functions for developers to call.

Thus ELF can represent a dynamic library just as it does an executable. There are some fundamental differences, such as there is no pointer to where execution should start, but all shared libraries are just ELF objects like any other executable.

The ELF header has two mutually exclusive flags, `ET_EXEC` and `ET_DYN` to mark an ELF file as either an executable or a shared object file.

Including libraries in an executable

Compilation

When you compile your program that uses a dynamic library, object files are left with references to the library functions just as for any other external reference.

You need to include the *header* for the library so that the compiler knows the specific types of the functions you are calling. Note the compiler only needs to know the types

associated with a function (such as, it takes an `int` and returns a `char *`) so that it can correctly allocate space for the function call.[1]

Linking

Even though the *dynamic linker* does a lot of the work for shared libraries, the traditional linker still has a role to play in creating the executable.

The traditional linker needs to leave a pointer in the executable so that the dynamic linker knows what library will satisfy the dependencies at runtime.

The `dynamic` section of the executable requires a `NEEDED` entry for each shared library that the executable depends on.

Again, we can inspect these fields with the `readelf` program. Below we have a look at a very standard binary, `/bin/ls`

Example 9.1. Specifying Dynamic Libraries

```
 1
              $ readelf --dynamic /bin/ls

   Dynamic segment at offset 0x22f78 contains 27 entries:
 5   Tag        Type                         Name/Value
    0x0000000000000001 (NEEDED)             Shared library: [librt.so.1]
    0x0000000000000001 (NEEDED)             Shared library: [libacl.so.1]
    0x0000000000000001 (NEEDED)             Shared library: [libc.so.6.1]
    0x000000000000000c (INIT)               0x4000000000001e30
10  ... snip ...
```

You can see that it specifies three libraries. The most common library shared by most, if not all, programs on the system is `libc`. There are also some other libraries that the program needs to run correctly.

Reading the ELF file directly is sometimes useful, but the usual way to inspect a dynamically linked executable is via `ldd`. `ldd` "walks" the dependencies of libraries for you; that is if a library depends on another library, it will show it to you.

Example 9.2. Looking at dynamic libraries

```
 1
              $ ldd /bin/ls
           librt.so.1 => /lib/tls/librt.so.1 (0x2000000000058000)
           libacl.so.1 => /lib/libacl.so.1 (0x2000000000078000)
 5         libc.so.6.1 => /lib/tls/libc.so.6.1 (0x2000000000098000)
           libpthread.so.0 => /lib/tls/libpthread.so.0 (0x20000000002e0000)
           /lib/ld-linux-ia64.so.2 => /lib/ld-linux-ia64.so.2 (0x2000000000000000)
           libattr.so.1 => /lib/libattr.so.1 (0x2000000000310000)
   $ readelf --dynamic /lib/librt.so.1
10
   Dynamic segment at offset 0xd600 contains 30 entries:
```

[1] This has not always been the case with the C standard. Previously, compilers would assume that any function it did not know about returned an `int`. On a 32 bit system, the size of a pointer is the same size as an `int`, so there was no problem. However, with a 64 bit system, the size of a pointer is generally twice the size of an `int` so if the function actually returns a pointer, its value will be destroyed. This is clearly not acceptable, as the pointer will thus not point to valid memory. The C99 standard has changed such that you are required to specify the types of included functions.

```
     Tag        Type                         Name/Value
    0x0000000000000001 (NEEDED)             Shared library: [libc.so.6.1]
    0x0000000000000001 (NEEDED)             Shared library: [libpthread.so.0]
15  ... snip ...
```

We can see above that `libpthread` has been required from somewhere. If we do a little digging, we can see that the requirement comes from `librt`.

The Dynamic Linker

The dynamic linker is the program that manages shared dynamic libraries on behalf of an executable. It works to load libraries into memory and modify the program at runtime to call the functions in the library.

ELF allows executables to specify an *interpreter*, which is a program that should be used to run the executable. The compiler and static linker set the interpreter of executables that rely on dynamic libraries to be the dynamic linker.

Example 9.3. Checking the program interpreter

```
 1
          ianw@lime:~/programs/csbu$ readelf --headers /bin/ls

  Program Headers:
 5  Type           Offset             VirtAddr           PhysAddr
                   FileSiz            MemSiz              Flags  Align
    PHDR           0x0000000000000040 0x4000000000000040 0x4000000000000040
                   0x0000000000000188 0x0000000000000188  R E    8
    INTERP         0x00000000000001c8 0x40000000000001c8 0x40000000000001c8
10                 0x0000000000000018 0x0000000000000018  R      1
        [Requesting program interpreter: /lib/ld-linux-ia64.so.2]
    LOAD           0x0000000000000000 0x4000000000000000 0x4000000000000000
                   0x0000000000022e40 0x0000000000022e40  R E    10000
    LOAD           0x0000000000022e40 0x6000000000002e40 0x6000000000002e40
15                 0x0000000000001138 0x00000000000017b8  RW     10000
    DYNAMIC        0x0000000000022f78 0x6000000000002f78 0x6000000000002f78
                   0x0000000000000200 0x0000000000000200  RW     8
    NOTE           0x00000000000001e0 0x40000000000001e0 0x40000000000001e0
                   0x0000000000000020 0x0000000000000020  R      4
20  IA_64_UNWIND   0x0000000000022018 0x4000000000022018 0x4000000000022018
                   0x0000000000000e28 0x0000000000000e28  R      8
```

You can see above that the interpreter is set to be /lib/ld-linux-ia64.so.2, which is the dynamic linker. When the kernel loads the binary for execution, it will check if the `PT_INTERP` field is present, and if so load what it points to into memory and start it.

We mentioned that dynamically linked executables leave behind references that need to be fixed with information that isn't available until runtime, such as the address of a function in a shared library. The references that are left behind are called *relocations*.

Relocations

The essential part of the dynamic linker is fixing up addresses at runtime, which is the only time you can know for certain where you are loaded in memory. A relocation can

simply be thought of as a note that a particular address will need to be fixed at load time. Before the code is ready to run you will need to go through and read all the relocations and fix the addresses it refers to to point to the right place.

Table 9.1. Relocation Example

Address	Action
0x123456	Address of symbol "x"
0x564773	Function X

There are many types of relocation for each architecture, and each types exact behaviour is documented as part of the ABI for the system. The definition of a relocation is quite straight forward.

Example 9.4. Relocation as defined by ELF

```
 1
            typedef struct {
    Elf32_Addr    r_offset;  <--- address to fix
    Elf32_Word    r_info;    <--- symbol table pointer and relocation type
 5 }

   typedef struct {
     Elf32_Addr    r_offset;
     Elf32_Word    r_info;
10   Elf32_Sword   r_addend;
   } Elf32_Rela
```

The `r_offset` field refers to the offset in the file that needs to be fixed up. The `r_info` field specifies the type of relocation which describes what exactly must be done to fix this code up. The simplest relocation usually defined for an architecture is simply the value of the symbol. In this case you simply substitute the address of the symbol at the location specified, and the relocation has been "fixed-up".

The two types, one with an addend and one without specify different ways for the relocation to operate. An addend is simply something that should be added to the fixed up address to find the correct address. For example, if the relocation is for the symbol `i` because the original code is doing something like `i[8]` the addend will be set to 8. This means "find the address of `i`, and go 8 past it".

That addend value needs to be stored somewhere. The two solutions are covered by the two forms. In the REL form the addend is actually store in the program code in the place where the fixed up address should be. This means that to fix up the address properly, you need to first read the memory you are about to fix up to get any addend, store that, find the "real" address, add the addend to it and then write it back (over the addend). The RELA format specifies the addend right there in the relocation.

The trade offs of each approach should be clear. With REL you need to do an extra memory reference to find the addend before the fixup, but you don't waste space in the binary because you use relocation target memory. With RELA you keep the addend with

the relocation, but waste that space in the on disk binary. Most modern systems use RELA relocations.

Relocations in action

The example below shows how relocations work. We create two very simple shared libraries and reference one from in the other.

Example 9.5. Specifying Dynamic Libraries

```
 1
              $ cat addendtest.c
   extern int i[4];
   int *j = i + 2;
 5
   $ cat addendtest2.c
   int i[4];

   $ gcc -nostdlib -shared -fpic -s -o addendtest2.so addendtest2.c
10 $ gcc -nostdlib -shared -fpic -o addendtest.so addendtest.c ./addendtest2.so

   $ readelf -r ./addendtest.so

   Relocation section '.rela.dyn' at offset 0x3b8 contains 1 entries:
15   Offset          Info           Type           Sym. Value    Sym. Name + Addend
   0000000104f8  000f00000027 R_IA64_DIR64LSB   0000000000000000 i + 8
```

We thus have one relocation in `addendtest.so` of type `R_IA64_DIR64LSB`. If you look this up in the IA64 ABI, the acronym can be broken down to

1. *R_IA64* : all relocations start with this prefix.

2. *DIR64* : a 64 bit direct type relocation

3. *LSB* : Since IA64 can operate in big and little endian modes, this relocation is little endian (least significant byte).

The ABI continues to say that that relocation means "the value of the symbol pointed to by the relocation, plus any addend". We can see we have an addend of 8, since `sizeof(int) == 4` and we have moved two int's into the array (`*j = i + 2`). So at runtime, to fix this relocation you need to find the address of symbol `i` and put it's value, plus 8 into `0x104f8`.

Position Independence

In an *executable* file, the code and data segment is given a specified base address in virtual memory. The executable code is not shared, and each executable gets its own fresh address space. This means that the compiler knows exactly where the data section will be, and can reference it directly.

Libraries have no such guarantee. They can know that their data section will be a specified *offset* from the base address; but exactly where that base address is can only be known at runtime.

Consequently all libraries must be produced with code that can execute no matter where it is put into memory, known as *position independent code* (or PIC for short). Note that the data section is still a fixed offset from the code section; but to actually find the address of data the offset needs to be added to the load address.

Global Offset Tables

You might have noticed a critical problem with relocations when thinking about the goals of a shared library. We mentioned previously that the big advantage of a shared library with virtual memory is that multiple programs can use the code in memory by sharing of pages.

The problem stems from the fact that libraries have no guarantee about where they will be put into memory. The dynamic linker will find the most convenient place in virtual memory for each library required and place it there. Think about the alternative if this were *not* to happen; every library in the system would require its own chunk of virtual memory so that no two overlapped. Every time a new library were added to the system it would require allocation. Someone could potentially be a hog and write a *huge* library, not leaving enough space for other libraries! And chances are, your program doesn't ever want to use that library anyway.

Thus, if you modify the code of a shared library with a relocation, that code no longer becomes sharable. We've lost the advantage of our shared library.

Below we explain the mechanism for doing this.

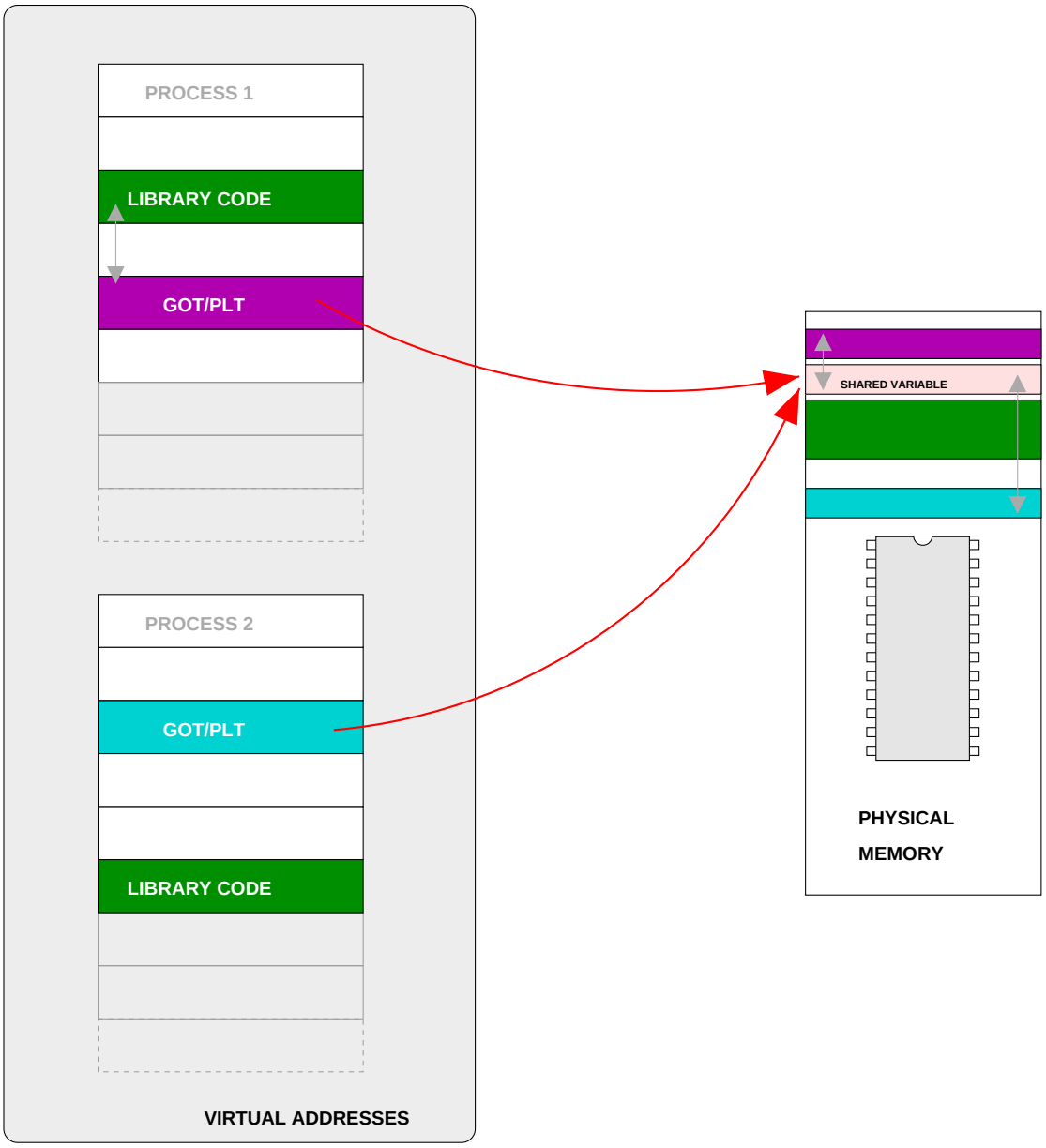
The Global Offset Table

So imagine the situation where we take the value of a symbol. With only relocations, we would have the dynamic linker look up the memory address of that symbol and re-write the code to load that address.

A fairly straight forward enhancement would be to set aside space in our binary to hold the address of that symbol, and have the dynamic linker put the address there rather than in the code directly. This way we never need to touch the code part of the binary.

The area that is set aside for these addresses is called the Global Offset Table, or GOT. The GOT lives in a section of the ELF file called `.got`.

Figure 9.1. Memory access via the GOT



The GOT is private to each process, and the process must have write permissions to it. Conversely the library code is shared and the process should have only read and execute permissions on the code; it would be a serious security breach if the process could modify code.

The GOT in action

Example 9.6. Using the GOT

```
1
                $ cat got.c
  extern int i;

5 void test(void)
```

```
   {
          i = 100;
   }

10 $ gcc -nostdlib  -shared -o got.so ./got.c

   $ objdump --disassemble ./got.so

   ./got.so:     file format elf64-ia64-little
15
   Disassembly of section .text:

   0000000000000410 <test>:
    410:   0d 10 00 18 00 21       [MFI]       mov r2=r12
20  416:   00 00 00 02 00 c0                   nop.f 0x0
    41c:   81 09 00 90                         addl r14=24,r1;;
    420:   0d 78 00 1c 18 10       [MFI]       ld8 r15=[r14]
    426:   00 00 00 02 00 c0                   nop.f 0x0
    42c:   41 06 00 90                         mov r14=100;;
25  430:   11 00 38 1e 90 11       [MIB]       st4 [r15]=r14
    436:   c0 00 08 00 42 80                   mov r12=r2
    43c:   08 00 84 00                         br.ret.sptk.many b0;;

   $ readelf --sections ./got.so
30 There are 17 section headers, starting at offset 0x640:

   Section Headers:
     [Nr] Name              Type             Address           Offset
          Size              EntSize          Flags  Link  Info  Align
35   [ 0]                   NULL             0000000000000000  00000000
          0000000000000000  0000000000000000           0     0     0
     [ 1] .hash             HASH             0000000000000120  00000120
          00000000000000a0  0000000000000004   A       2     0     8
     [ 2] .dynsym           DYNSYM           00000000000001c0  000001c0
40        00000000000001f8  0000000000000018   A       3     e     8
     [ 3] .dynstr           STRTAB           00000000000003b8  000003b8
          000000000000003f  0000000000000000   A       0     0     1
     [ 4] .rela.dyn         RELA             00000000000003f8  000003f8
          0000000000000018  0000000000000018   A       2     0     8
45   [ 5] .text             PROGBITS         0000000000000410  00000410
          0000000000000030  0000000000000000  AX       0     0     16
     [ 6] .IA_64.unwind_inf PROGBITS         0000000000000440  00000440
          0000000000000018  0000000000000000   A       0     0     8
     [ 7] .IA_64.unwind     IA_64_UNWIND     0000000000000458  00000458
50        0000000000000018  0000000000000000  AL       5     5     8
     [ 8] .data             PROGBITS         0000000000010470  00000470
          0000000000000000  0000000000000000  WA       0     0     1
     [ 9] .dynamic          DYNAMIC          0000000000010470  00000470
          0000000000000100  0000000000000010  WA       3     0     8
55   [10] .got              PROGBITS         0000000000010570  00000570
          0000000000000020  0000000000000000 WAp       0     0     8
     [11] .sbss             NOBITS           0000000000010590  00000590
          0000000000000000  0000000000000000   W       0     0     1
     [12] .bss              NOBITS           0000000000010590  00000590
60        0000000000000000  0000000000000000  WA       0     0     1
     [13] .comment          PROGBITS         0000000000000000  00000590
          0000000000000026  0000000000000000           0     0     1
     [14] .shstrtab         STRTAB           0000000000000000  000005b6
          000000000000008a  0000000000000000           0     0     1
65   [15] .symtab           SYMTAB           0000000000000000  00000a80
          0000000000000258  0000000000000018          16    12     8
     [16] .strtab           STRTAB           0000000000000000  00000cd8
          0000000000000045  0000000000000000           0     0     1
   Key to Flags:
70   W (write), A (alloc), X (execute), M (merge), S (strings)
     I (info), L (link order), G (group), x (unknown)
     O (extra OS processing required) o (OS specific), p (processor specific)
```

Above we create a simple shared library which refers to an external symbol. We do not know the address of this symbol at compile time, so we leave it for the dynamic linker to fix up at runtime.

But we want our code to remain sharable, in case other processes want to use our code as well.

The disassembly reveals just how we do this with the `.got`. On IA64 (the architecture which the library was compiled for) the register `r1` is known as the *global pointer* and always points to where the `.got` section is loaded into memory.

If we have a look at the `readelf` output we can see that the `.got` section starts 0x10570 bytes past where library was loaded into memory. Thus if the library were to be loaded into memory at address 0x6000000000000000 the `.got` would be at 0x6000000000010570, and register `r1` would always point to this address.

Working backwards through the disassembly, we can see that we store the value 100 into the memory address held in register `r15`. If we look back we can see that register 15 holds the value of the memory address stored in register 14. Going back one more step, we see we load this address is found by adding a small number to register 1. The GOT is simply a big long list of entries, one for each external variable. This means that the GOT entry for the external variable `i` is stored 24 bytes (that is 3 64 bit addresses).

Example 9.7. Relocations against the GOT

```
1
              $ readelf --relocs ./got.so

  Relocation section '.rela.dyn' at offset 0x3f8 contains 1 entries:
5   Offset          Info           Type           Sym. Value    Sym. Name + Addend
  000000010588  000f00000027 R_IA64_DIR64LSB   0000000000000000 i + 0
```

We can also check out the relocation for this entry too. The relocation says "replace the value at offset 10588 with the memory location that symbol i is stored at".

We know that the `.got` starts at offset 0x10570 from the previous output. We have also seen how the code loads an address 0x18 (24 in decimal) past this, giving us an address of 0x10570 + 0x18 = 0x10588 ... the address which the relocation is for!

So before the program begins, the dynamic linker will have fixed up the relocation to ensure that the value of the memory at offset 0x10588 is the address of the global variable `i`!

Libraries

The Procedure Lookup Table

Libraries may contain many functions, and a program may end up including many libraries to get its work done. A program may only use one or two functions from each library of the many available, and depending on the run-time path through the code may use some functions and not others.

As we have seen, the process of dynamic linking is a fairly computationally intensive one, since it involves looking up and searching through many tables. Anything that can be done to reduce the overheads will increase performance.

The Procedure Lookup Table (PLT) facilitates what is called *lazy binding* in programs. Binding is synonymous with the fix-up process described above for variables located in the GOT. When an entry has been "fixed-up" it is said to be "bound" to its real address.

As we mentioned, sometimes a program will include a function from a library but never actually call that function, depending on user input. The process of binding this function is quite intensive, involving loading code, searching through tables and writing memory. To go through the process of binding a function that is not used is simply a waste of time.

Lazy binding defers this expense until the actual function is called by using a PLT.

Each library function has an entry in the PLT, which initially points to some special dummy code. When the program calls the function, it actually calls the PLT entry (in the same was as variables are referenced through the GOT).

This dummy function will load a few parameters that need to be passed to the dynamic linker for it to resolve the function and then call into a special lookup function of the dynamic linker. The dynamic linker finds the real address of the function, and writes that location into the calling binary over the top of the dummy function call.

Thus, the next time the function is called the address can be loaded without having to go back into the dynamic loader again. If a function is never called, then the PLT entry will never be modified but there will be no runtime overhead.

The PLT in action

Things start to get a bit hairy here! If nothing else, you should begin to appreciate that there is a fair bit of work in resolving a dynamic symbol!

Let us consider the simple "hello World" application. This will only make one library call to `printf` to output the string to the user.

Example 9.8. Hello World PLT example

```
 1
              $ cat hello.c
   #include <stdio.h>

 5 int main(void)
   {
           printf("Hello, World!\n");
           return 0;
   }
10
   $ gcc -o hello hello.c

   $ readelf --relocs ./hello

15 Relocation section '.rela.dyn' at offset 0x3f0 contains 2 entries:
     Offset          Info           Type           Sym. Value    Sym. Name + Addend
   6000000000000ed8  000700000047 R_IA64_FPTR64LSB  0000000000000000 _Jv_RegisterClasses + 0
   6000000000000ee0  000900000047 R_IA64_FPTR64LSB  0000000000000000 __gmon_start__ + 0
```

```
20 Relocation section '.rela.IA_64.pltoff' at offset 0x420 contains 3 entries:
     Offset          Info           Type           Sym. Value    Sym. Name + Addend
   6000000000000f10  000200000081 R_IA64_IPLTLSB    0000000000000000 printf + 0
   6000000000000f20  000800000081 R_IA64_IPLTLSB    0000000000000000 __libc_start_main + 0
   6000000000000f30  000900000081 R_IA64_IPLTLSB    0000000000000000 __gmon_start__ + 0
25
```

We can see above that we have a `R_IA64_IPLTLSB` relocation for our `printf` symbol. This is saying "put the address of symbol printf into memory address 0x6000000000000f10". We have to start digging deeper to find the exact procedure that gets us the function.

Below we have a look at the disassembly of the `main()` function of the program.

Example 9.9. Hello world main()

```
 1
              4000000000000790 <main>:
   4000000000000790:       00 08 15 08 80 05       [MII]       alloc r33=ar.pfs,5,4,0
   4000000000000796:       20 02 30 00 42 60                   mov r34=r12
 5 400000000000079c:       04 08 00 84                         mov r35=r1
   40000000000007a0:       01 00 00 00 01 00       [MII]       nop.m 0x0
   40000000000007a6:       00 02 00 62 00 c0                   mov r32=b0
   40000000000007ac:       81 0c 00 90                         addl r14=72,r1;;
   40000000000007b0:       1c 20 01 1c 18 10       [MFB]       ld8 r36=[r14]
10 40000000000007b6:       00 00 00 02 00 00                   nop.f 0x0
   40000000000007bc:       78 fd ff 58                         br.call.sptk.many b0=4000000000000520 <_ini
   40000000000007c0:       02 08 00 46 00 21       [MII]       mov r1=r35
   40000000000007c6:       e0 00 00 00 42 00                   mov r14=r0;;
   40000000000007cc:       01 70 00 84                         mov r8=r14
15 40000000000007d0:       00 00 00 00 01 00       [MII]       nop.m 0x0
   40000000000007d6:       00 08 01 55 00 00                   mov.i ar.pfs=r33
   40000000000007dc:       00 0a 00 07                         mov b0=r32
   40000000000007e0:       1d 60 00 44 00 21       [MFB]       mov r12=r34
   40000000000007e6:       00 00 00 02 00 80                   nop.f 0x0
20 40000000000007ec:       08 00 84 00                         br.ret.sptk.many b0;;
```

The call to 0x4000000000000520 must be us calling the `printf` function. We can find out where this is by looking at the sections with `readelf`.

Example 9.10. Hello world sections

```
 1
              $ readelf --sections ./hello
   There are 40 section headers, starting at offset 0x25c0:

 5 Section Headers:
     [Nr] Name              Type             Address           Offset
          Size              EntSize          Flags  Link  Info  Align
     [ 0]                   NULL             0000000000000000  00000000
          0000000000000000  0000000000000000           0     0     0
10 ...
     [11] .plt              PROGBITS         40000000000004c0  000004c0
          00000000000000c0  0000000000000000  AX       0     0     32
     [12] .text             PROGBITS         4000000000000580  00000580
          00000000000004a0  0000000000000000  AX       0     0     32
15   [13] .fini             PROGBITS         4000000000000a20  00000a20
          0000000000000040  0000000000000000  AX       0     0     16
     [14] .rodata           PROGBITS         4000000000000a60  00000a60
          000000000000000f  0000000000000000   A       0     0     8
```

```
     [15] .opd              PROGBITS         4000000000000a70  00000a70
20        0000000000000070  0000000000000000   A       0     0     16
     [16] .IA_64.unwind_inf PROGBITS         4000000000000ae0  00000ae0
          00000000000000f0  0000000000000000   A       0     0     8
     [17] .IA_64.unwind     IA_64_UNWIND     4000000000000bd0  00000bd0
          00000000000000c0  0000000000000000  AL      12     c     8
25   [18] .init_array       INIT_ARRAY       6000000000000c90  00000c90
          0000000000000018  0000000000000000  WA       0     0     8
     [19] .fini_array       FINI_ARRAY       6000000000000ca8  00000ca8
          0000000000000008  0000000000000000  WA       0     0     8
     [20] .data             PROGBITS         6000000000000cb0  00000cb0
30        0000000000000004  0000000000000000  WA       0     0     4
     [21] .dynamic          DYNAMIC          6000000000000cb8  00000cb8
          00000000000001e0  0000000000000010  WA       5     0     8
     [22] .ctors            PROGBITS         6000000000000e98  00000e98
          0000000000000010  0000000000000000  WA       0     0     8
35   [23] .dtors            PROGBITS         6000000000000ea8  00000ea8
          0000000000000010  0000000000000000  WA       0     0     8
     [24] .jcr              PROGBITS         6000000000000eb8  00000eb8
          0000000000000008  0000000000000000  WA       0     0     8
     [25] .got              PROGBITS         6000000000000ec0  00000ec0
40        0000000000000050  0000000000000000 WAp       0     0     8
     [26] .IA_64.pltoff     PROGBITS         6000000000000f10  00000f10
          0000000000000030  0000000000000000 WAp       0     0     16
     [27] .sdata            PROGBITS         6000000000000f40  00000f40
          0000000000000010  0000000000000000 WAp       0     0     8
45   [28] .sbss             NOBITS           6000000000000f50  00000f50
          0000000000000008  0000000000000000  WA       0     0     8
     [29] .bss              NOBITS           6000000000000f58  00000f50
          0000000000000008  0000000000000000  WA       0     0     8
     [30] .comment          PROGBITS         0000000000000000  00000f50
50        00000000000000b9  0000000000000000           0     0     1
     [31] .debug_aranges    PROGBITS         0000000000000000  00001010
          0000000000000090  0000000000000000           0     0     16
     [32] .debug_pubnames   PROGBITS         0000000000000000  000010a0
          0000000000000025  0000000000000000           0     0     1
55   [33] .debug_info       PROGBITS         0000000000000000  000010c5
          00000000000009c4  0000000000000000           0     0     1
     [34] .debug_abbrev     PROGBITS         0000000000000000  00001a89
          0000000000000124  0000000000000000           0     0     1
     [35] .debug_line       PROGBITS         0000000000000000  00001bad
60        00000000000001fe  0000000000000000           0     0     1
     [36] .debug_str        PROGBITS         0000000000000000  00001dab
          00000000000006a1  0000000000000001  MS       0     0     1
     [37] .shstrtab         STRTAB           0000000000000000  0000244c
          000000000000016f  0000000000000000           0     0     1
65   [38] .symtab           SYMTAB           0000000000000000  00002fc0
          0000000000000b58  0000000000000018          39    60     8
     [39] .strtab           STRTAB           0000000000000000  00003b18
          0000000000000479  0000000000000000           0     0     1
   Key to Flags:
70   W (write), A (alloc), X (execute), M (merge), S (strings)
     I (info), L (link order), G (group), x (unknown)
     O (extra OS processing required) o (OS specific), p (processor specific)
```

That address is (unsurprisingly) in the `.plt` section. So there we have our call into the PLT! But we're not satisfied with that, let's keep digging further to see what we can uncover. We disassemble the `.plt` section to see what that call actually does.

Example 9.11. Hello world PLT

```
 1
              40000000000004c0 <.plt>:
   40000000000004c0:       0b 10 00 1c 00 21       [MMI]       mov r2=r14;;
   40000000000004c6:       e0 00 08 00 48 00                   addl r14=0,r2
```

```
 5 40000000000004cc:       00 00 04 00                           nop.i 0x0;;
   40000000000004d0:       0b 80 20 1c 18 14       [MMI]         ld8 r16=[r14],8;;
   40000000000004d6:       10 41 38 30 28 00                     ld8 r17=[r14],8
   40000000000004dc:       00 00 04 00                           nop.i 0x0;;
   40000000000004e0:       11 08 00 1c 18 10       [MIB]         ld8 r1=[r14]
10 40000000000004e6:       60 88 04 80 03 00                     mov b6=r17
   40000000000004ec:       60 00 80 00                           br.few b6;;
   40000000000004f0:       11 78 00 00 00 24       [MIB]         mov r15=0
   40000000000004f6:       00 00 00 02 00 00                     nop.i 0x0
   40000000000004fc:       d0 ff ff 48                           br.few 40000000000004c0 <_init+0x50>;;
15 4000000000000500:       11 78 04 00 00 24       [MIB]         mov r15=1
   4000000000000506:       00 00 00 02 00 00                     nop.i 0x0
   400000000000050c:       c0 ff ff 48                           br.few 40000000000004c0 <_init+0x50>;;
   4000000000000510:       11 78 08 00 00 24       [MIB]         mov r15=2
   4000000000000516:       00 00 00 02 00 00                     nop.i 0x0
20 400000000000051c:       b0 ff ff 48                           br.few 40000000000004c0 <_init+0x50>;;
   4000000000000520:       0b 78 40 03 00 24       [MMI]         addl r15=80,r1;;
   4000000000000526:       00 41 3c 70 29 c0                     ld8.acq r16=[r15],8
   400000000000052c:       01 08 00 84                           mov r14=r1;;
   4000000000000530:       11 08 00 1e 18 10       [MIB]         ld8 r1=[r15]
25 4000000000000536:       60 80 04 80 03 00                     mov b6=r16
   400000000000053c:       60 00 80 00                           br.few b6;;
   4000000000000540:       0b 78 80 03 00 24       [MMI]         addl r15=96,r1;;
   4000000000000546:       00 41 3c 70 29 c0                     ld8.acq r16=[r15],8
   400000000000054c:       01 08 00 84                           mov r14=r1;;
30 4000000000000550:       11 08 00 1e 18 10       [MIB]         ld8 r1=[r15]
   4000000000000556:       60 80 04 80 03 00                     mov b6=r16
   400000000000055c:       60 00 80 00                           br.few b6;;
   4000000000000560:       0b 78 c0 03 00 24       [MMI]         addl r15=112,r1;;
   4000000000000566:       00 41 3c 70 29 c0                     ld8.acq r16=[r15],8
35 400000000000056c:       01 08 00 84                           mov r14=r1;;
   4000000000000570:       11 08 00 1e 18 10       [MIB]         ld8 r1=[r15]
   4000000000000576:       60 80 04 80 03 00                     mov b6=r16
   400000000000057c:       60 00 80 00                           br.few b6;;

40
```

Let us step through the instructions. Firstly, we add 80 to the value in r1, storing it in r15. We know from before that r1 will be pointing to the GOT, so this is saying "store in r15 80 bytes into the GOT". The next thing we do is load into r16 the value stored in this location in the GOT, and post increment the value in r15 by 8 bytes. We then store r1 (the location of the GOT) in r14 and set r1 to be the value in the next 8 bytes after r15. Then we branch to r16.

In the previous chapter we discussed how functions are actually called through a function descriptor which contains the function address and the address of the global pointer. Here we can see that the PLT entry is first loading the function value, moving on 8 bytes to the second part of the function descriptor and then loading that value into the op register before calling the function.

But what exactly are we loading? We know that r1 will be pointing to the GOT. We go 80 bytes past the got (0x50)

Example 9.12. Hello world GOT

```
 1
              $ objdump --disassemble-all ./hello
   Disassembly of section .got:

 5 6000000000000ec0 <.got>:
           ...
   6000000000000ee8:       80 0a 00 00 00 00                     data8 0x02a000000
   6000000000000eee:       00 40 90 0a                           dep r0=r0,r0,63,1
```

```
   6000000000000ef2:       00 00 00 00 00 40       [MIB] (p20) break.m 0x1
10 6000000000000ef8:       a0 0a 00 00 00 00                   data8 0x02a810000
   6000000000000efe:       00 40 50 0f                         br.few 6000000000000ef0 <_GLOBAL_OFFSET_TAB
   6000000000000f02:       00 00 00 00 00 60       [MIB] (p58) break.m 0x1
   6000000000000f08:       60 0a 00 00 00 00                   data8 0x029818000
   6000000000000f0e:       00 40 90 06                         br.few 6000000000000f00 <_GLOBAL_OFFSET_TAB
15 Disassembly of section .IA_64.pltoff:

   6000000000000f10 <.IA_64.pltoff>:
   6000000000000f10:       f0 04 00 00 00 00       [MIB] (p39) break.m 0x0
   6000000000000f16:       00 40 c0 0e 00 00                   data8 0x03b010000
20 6000000000000f1c:       00 00 00 60                         data8 0xc000000000
   6000000000000f20:       00 05 00 00 00 00       [MII] (p40) break.m 0x0
   6000000000000f26:       00 40 c0 0e 00 00                   data8 0x03b010000
   6000000000000f2c:       00 00 00 60                         data8 0xc000000000
   6000000000000f30:       10 05 00 00 00 00       [MIB] (p40) break.m 0x0
25 6000000000000f36:       00 40 c0 0e 00 00                   data8 0x03b010000
   6000000000000f3c:       00 00 00 60                         data8 0xc000000000
```

0x6000000000000ec0 + 0x50 = 0x6000000000000f10, or the `.IA_64.pltoff` section. Now we're starting to get somewhere!

We can decode the objdump output so we can see exactly what is being loaded here. Swapping the byte order of the first 8 bytes `f0 04 00 00 00 00 00 40` we end up with `0x4000000000004f0`. Now that address looks familiar! Looking back up at the assemble output of the PLT we see that address.

The code at `0x4000000000004f0` firstly puts a zero value into r15, and then branches back to `0x40000000000004c0`. Wait a minute! That's the start of our PLT section.

We can trace this code through too. Firstly we save the value of the global pointer (`r2`) then we load three 8 byte values into `r16`, `r17` and finally, `r1`. We then branch to the address in `r17`. What we are seeing here is the actual call into the dynamic linker!

We need to delve into the ABI to understand exactly what is being loaded at this point. The ABI says two things -- dynamically linked programs must have a special section (called the `DT_IA_64_PLT_RESERVE` section) that can hold three 8 byte values. There is a pointer where this reserved area in the dynamic segment of the binary.

Example 9.13. Dynamic Segment

```
 1

   Dynamic segment at offset 0xcb8 contains 25 entries:
     Tag        Type                         Name/Value
 5  0x0000000000000001 (NEEDED)             Shared library: [libc.so.6.1]
    0x000000000000000c (INIT)               0x4000000000000470
    0x000000000000000d (FINI)               0x4000000000000a20
    0x0000000000000019 (INIT_ARRAY)         0x6000000000000c90
    0x000000000000001b (INIT_ARRAYSZ)       24 (bytes)
10  0x000000000000001a (FINI_ARRAY)         0x6000000000000ca8
    0x000000000000001c (FINI_ARRAYSZ)       8 (bytes)
    0x0000000000000004 (HASH)               0x4000000000000200
    0x0000000000000005 (STRTAB)             0x4000000000000330
    0x0000000000000006 (SYMTAB)             0x4000000000000240
15  0x000000000000000a (STRSZ)              138 (bytes)
    0x000000000000000b (SYMENT)             24 (bytes)
    0x0000000000000015 (DEBUG)              0x0
    0x0000000070000000 (IA_64_PLT_RESERVE)  0x6000000000000ec0 -- 0x6000000000000ed8
    0x0000000000000003 (PLTGOT)             0x6000000000000ec0
```

```
20  0x0000000000000002 (PLTRELSZ)           72 (bytes)
    0x0000000000000014 (PLTREL)             RELA
    0x0000000000000017 (JMPREL)             0x4000000000000420
    0x0000000000000007 (RELA)               0x40000000000003f0
    0x0000000000000008 (RELASZ)             48 (bytes)
25  0x0000000000000009 (RELAENT)            24 (bytes)
    0x000000006ffffffe (VERNEED)            0x40000000000003d0
    0x000000006fffffff (VERNEEDNUM)         1
    0x000000006ffffff0 (VERSYM)             0x40000000000003ba
    0x0000000000000000 (NULL)               0x0
30
```

Do you notice anything about it? It's the same value as the GOT. This means that the first three 8 byte entries in the GOT are actually the reserved area; thus will always be pointed to by the global pointer.

When the dynamic linker starts it is its duty to fill these values in. The ABI says that the first value will be filled in by the dynamic linker giving this *module* a unique ID. The second value is the global pointer value for the dynamic linker, and the third value is the address of the function that finds and fixes up the symbol.

Example 9.14. Code in the dynamic linker for setting up special values (from libc `sysdeps/ia64/dl-machine.h`)

```
 1
              /* Set up the loaded object described by L so its unrelocated PLT
     entries will jump to the on-demand fixup code in dl-runtime.c.  */

 5 static inline int __attribute__ ((unused, always_inline))
   elf_machine_runtime_setup (struct link_map *l, int lazy, int profile)
   {
     extern void _dl_runtime_resolve (void);
     extern void _dl_runtime_profile (void);
10
     if (lazy)
       {
         register Elf64_Addr gp __asm__ ("gp");
         Elf64_Addr *reserve, doit;
15
         /*
          * Careful with the typecast here or it will try to add l-l_addr
          * pointer elements
          */
20       reserve = ((Elf64_Addr *)
                    (l->l_info[DT_IA_64 (PLT_RESERVE)]->d_un.d_ptr + l->l_addr));
         /* Identify this shared object.  */
         reserve[0] = (Elf64_Addr) l;

25       /* This function will be called to perform the relocation.  */
         if (!profile)
           doit = (Elf64_Addr) ((struct fdesc *) &_dl_runtime_resolve)->ip;
         else
           {
30           if (GLRO(dl_profile) != NULL
                && _dl_name_match_p (GLRO(dl_profile), l))
              {
                /* This is the object we are looking for.  Say that we really
                   want profiling and the timers are started.  */
35              GL(dl_profile_map) = l;
              }
            doit = (Elf64_Addr) ((struct fdesc *) &_dl_runtime_profile)->ip;
           }

40       reserve[1] = doit;
```

```
          reserve[2] = gp;
        }

      return lazy;
45 }
```

We can see how this gets setup by the dynamic linker by looking at the function that does this for the binary. The `reserve` variable is set from the PLT_RESERVE section pointer in the binary. The unique value (put into `reserve[0]`) is the address of the *link map* for this object. Link maps are the internal representation within `glibc` for shared objects. We then put in the address of `_dl_runtime_resolve` to the second value (assuming we are not using profiling). `reserve[2]` is finally set to gp, which has been found from r2 with the `__asm__` call.

Looking back at the ABI, we see that the `relocation index` for the entry must be placed in `r15` and the unique identifier must be passed in `r16`.

`r15` has previously been set in the stub code, before we jumped back to the start of the PLT. Have a look down the entries, and notice how each PLT entry loads `r15` with an incremented value? It should come as no surprise if you look at the relocations the `printf` relocation is number zero.

`r16` we load up from the values that have been initialised by the dynamic linker, as previously discussed. Once that is ready, we can load the function address and global pointer and branch into the function.

What happens at this point is the dynamic linker function `_dl_runtime_resolve` is run. It finds the relocation; remember how the relocation specified the name of the symbol? It uses this name to find the right function; this might involve loading the library from disk if it is not already in memory, or otherwise sharing the code.

The relocation record provides the dynamic linker with the address it needs to "fix up"; remember it was in the GOT and loaded by the initial PLT stub? This means that after the first time the function is called, the *second* time it is loaded it will get the direct address of the function; short circuiting the dynamic linker.

Summary

You've seen the *exact* mechanism behind the PLT, and consequently the inner workings of the dynamic linker. The important points to remember are

- Library calls in your program actually call a stub of code in the PLT of the binary.
- That stub code loads an address and jumps to it.
- Initially, that address points to a function in the dynamic linker which is capable of looking up the "real" function, given the information in the relocation entry for that function.
- The dynamic linker re-writes the address that the stub code reads, so that the next time the function is called it will go straight to the right address.

Working with libraries and the linker

The presence of the dynamic linker provides both some advantages we can utilise and some extra issues that need to be resolved to get a functional system.

Library versions

One potential issue is different versions of libraries. With only static libraries there is much less potential for problems, as all library code is built directly into the binary of the application. If you want to use a new version of the library you need to recompile it into a new binary, replacing the old one.

This is obviously fairly impractical for common libraries, the most common of course being libc which is included in most all applications. If it were only available as a static library any change would require every single application in the system be rebuilt.

However, changes in the way the dynamic library work could cause multiple problems. In the best case, the modifications are completely compatible and nothing externally visible is changed. On the other hand the changes might cause the application to crash; for example if a function that used to take an `int` changes to take an `int *`. Worse, the new library version could have changed semantics and suddenly start silently returning different, possibly wrong values. This can be a very nasty bug to try and track down; when an application crashes you can use a debugger to isolate where the error occurs whilst data corruption or modification may only show up in seemingly unrelated parts of the application.

The dynamic linker requires a way to determine the version of libraries within the system so that newer revisions can be identified. There are a number of schemes a modern dynamic linker can use to find the right versions of libraries.

`sonames`

Using `sonames` we can add some extra information to a library to help identify versions.

As we have seen previously, an application lists the libraries it requires in `DT_NEEDED` fields in the dynamic section of the binary. The actual library is held in a file on disc, usually in `/lib` for core system libraries or `/usr/lib` for optional libraries.

To allow multiple versions of the library to exist on disk, they obviously require differing file names. The `soname` scheme uses a combination of names and file system links to build a hierarchy of libraries.

This is done by introducing the concept of *major* and *minor* library revisions. A minor revision is one wholly backwards compatible with a previous version of the library; this usually consists of only bug fixes. A major revision is therefore any revision that is not compatible; e.g. changes the inputs to functions or the way a function behaves.

As each library revision, major or minor, will need to be kept in a separate file on disk, this forms the basis of the library hierarchy. The library name is by convention

`libNAME.so.MAJOR.MINOR`[2]. However, if every application were directly linked against this file we would have the same issue as with a static library; every time a minor change happened we would need to rebuild the application to point to the new library.

What we really want to refer to is the *major* number of the library. If this changes, we reasonably are required to recompile our application, since we need to make sure our program is still compatible with the new library.

Thus the `soname` is the `libNAME.so.MAJOR`. The `soname` should be set in the `DT_SONAME` field of the dynamic section in a shared library; the library author can specify this version when they build the library.

Thus each minor version library file on disc can specify the same major version number in it's `DT_SONAME` field, allowing the dynamic linker to know that this particular library file implements a particular major revision of the library API and ABI.

To keep track of this, an application called ldconfig is commonly run to create symbolic links named for the major version to the latest minor version on the system. ldconfig works by running through all the libraries that implement a particular major revision number, and then picks out the one with the highest minor revision. It then creates a symbolic link from `libNAME.so.MAJOR` to the actual library file on disc, i.e. `libNAME.so.MAJOR.MINOR`.

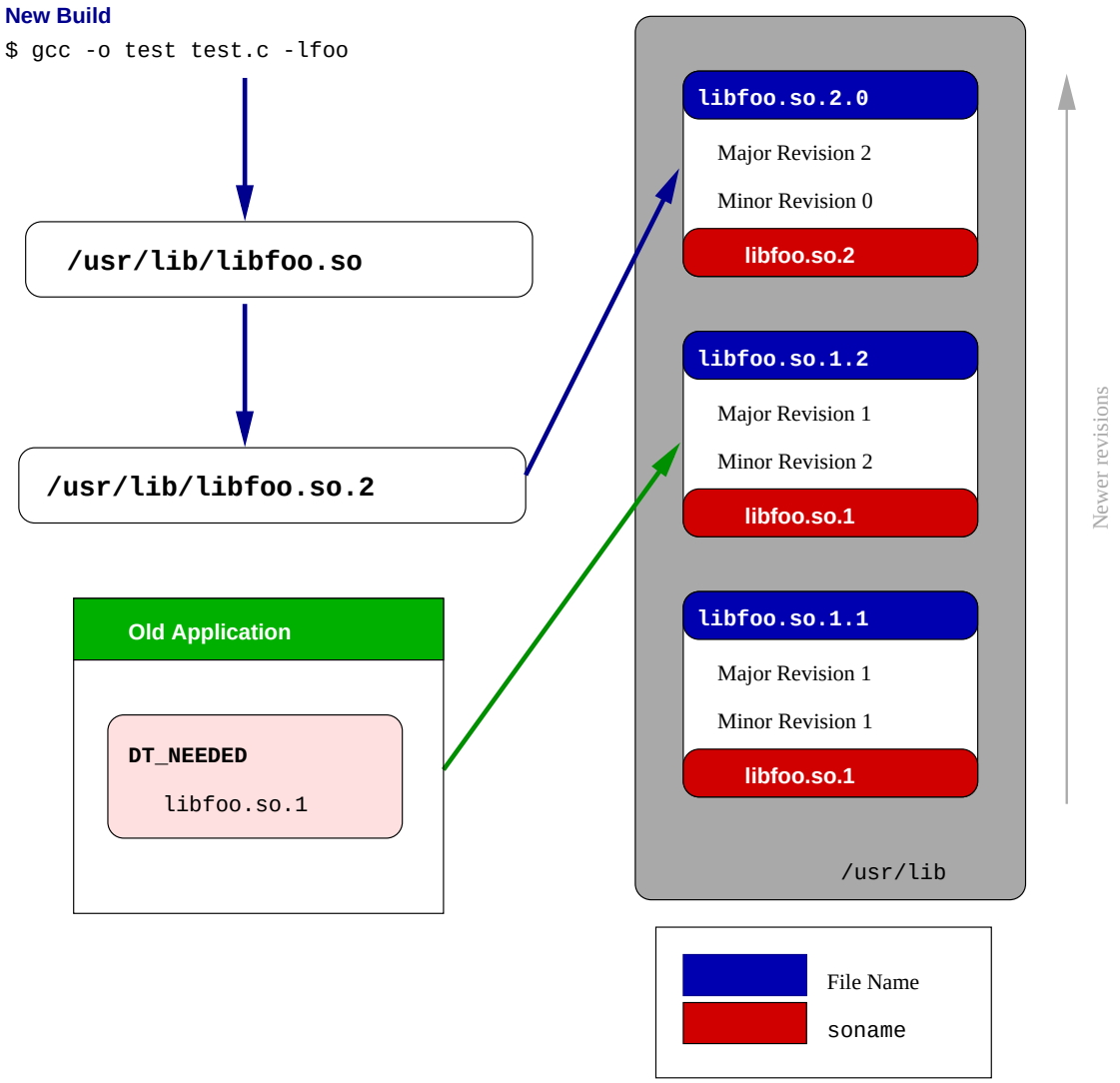
XXX : talk about libtool versions

The final piece of the hierarchy is the *compile name* for the library. When you compile your program, to link against a library you use the `-lNAME` flag, which goes off searching for the `libNAME.so` file in the library search path. Notice however, we have not specified any version number; we just want to link against the latest library on the system. It is up to the installation procedure for the library to create the symbolic link between the compile `libNAME.so` name and the latest library code on the system. Usually this is handled by your package management system (dpkg or rpm). This is not an automated process because it is possible that the latest library on the system may not be the one you wish to always compile against; for example if the latest installed library were a development version not appropriate for general use.

The general process is illustrated below.

[2] You can optionally have a *release* as a final identifier after the minor number. Generally this is enough to distinguish all the various versions library.

Figure 9.2. sonames



How the dynamic linker looks up libraries

When the application starts, the dynamic linker looks at the DT_NEEDED field to find the required libraries. This field contains the soname of the library, so the next step is for the dynamic linker to walk through all the libraries in its search path looking for it.

This process conceptually involves two steps. Firstly the dynamic linker needs to search through all the libraries to find those that implement the given soname. Secondly the file names for the minor revisions need to be compared to find the latest version, which is then ready to be loaded.

We mentioned previously that there is a symbolic link setup by ldconfig between the library soname and the latest minor revision. Thus the dynamic linker should need to only follow that link to find the correct file to load, rather than having to open all possible libraries and decide which one to go with each time the application is required.

Since file system access is so slow, ldconfig also creates a *cache* of libraries installed in the system. This cache is simply a list of `sonames` of libraries available to the dynamic linker and a pointer to the major version link on disk, saving the dynamic linker having to read entire directories full of files to locate the correct link. You can analyse this with /sbin/ldconfig -p; it actually lives in the file `/etc/ldconfig.so.cache`. If the library is not found in the cache the dynamic linker will fall back to the slower option of walking the file system, thus it is important to re-run ldconfig when new libraries are installed.

Finding symbols

We've already discussed how the dynamic linker gets the address of a library function and puts it in the PLT for the program to use. But so far we haven't discussed just *how* the dynamic linker finds the address of the function. The whole process is called *binding*, because the symbol name is bound to the address it represents.

The dynamic linker has a few pieces of information; firstly the *symbol* that it is searching for, and secondly a list of libraries that that symbol might be in, as defined by the `DT_NEEDED` fields in the binary.

Each shared object library has a section, marked `SHT_DYNSYM` and called `.dynsym` which is the minimal set of symbols required for dynamic linking -- that is any symbol in the library that may be called by an external program.

Dynamic Symbol Table

In fact, there are three sections that all play a part in describing the dynamic symbols. Firstly, let us look at the definition of a symbol from the ELF specification

Example 9.15. Symbol definition from ELF

```
 1
              typedef struct {
             Elf32_Word    st_name;
             Elf32_Addr    st_value;
 5           Elf32_Word    st_size;
             unsigned char st_info;
             unsigned char st_other;
             Elf32_Half    st_shndx;
   } Elf32_Sym;
10
```

Table 9.2. ELF symbol fields

Field	Value
`st_name`	An *index to the string table*
`st_value`	Value - in a relocatable shared object this holds the offset from the section of index given in `st_shndx`
`st_size`	Any associated size of the symbol

Field	Value
st_info	Information on the binding of the symbol (described below) and what type of symbol this is (a function, object, etc).
st_other	Not currently used
st_shndx	Index of the section this symbol resides in (see st_value

As you can see, the actual string of the symbol name is held in a separate section (.dynstr; the entry in the .dynsym section only holds an index into the string section. This creates some level of overhead for the dynamic linker; the dynamic linker must read all of the symbol entries in the .dynsym section and then follow the index pointer to find the symbol name for comparison.

To speed this process up, a third section called .hash is introduced, containing a *hash table* of symbol names to symbol table entries. This hash table is pre-computed when the library is built and allows the dynamic linker to find the symbol entry much faster, generally with only one or two lookups.

Symbol Binding

Whilst we usually say the process of finding the address of a symbol refers is the process of binding that symbol, the *symbol binding* has a separate meaning.

The binding of a symbol dictates its external visibility during the dynamic linking process. A *local* symbol is not visible outside the object file it is defined in. A *global* symbol is visible to other object files, and can satisfy undefined references in other objects.

A *weak* reference is a special type of lower priority global reference. This means it is designed to be overridden, as we will see shortly.

Below we have an example C program which we analyse to inspect the symbol bindings.

Example 9.16. Examples of symbol bindings

```
 1
              $ cat test.c
   static int static_variable;

 5 extern int extern_variable;

   int external_function(void);

   int function(void)
10 {
          return external_function();
   }

   static int static_function(void)
15 {
          return 10;
   }

   #pragma weak weak_function
20 int weak_function(void)
```

```
   {
           return 10;
   }

25 $ gcc -c test.c
   $ objdump --syms test.o

   test.o:     file format elf32-powerpc

30 SYMBOL TABLE:
   00000000 l    df *ABS*  00000000 test.c
   00000000 l    d  .text  00000000 .text
   00000000 l    d  .data  00000000 .data
   00000000 l    d  .bss   00000000 .bss
35 00000038 l     F .text  00000024 static_function
   00000000 l    d  .sbss  00000000 .sbss
   00000000 l     O .sbss  00000004 static_variable
   00000000 l    d  .note.GNU-stack        00000000 .note.GNU-stack
   00000000 l    d  .comment       00000000 .comment
40 00000000 g     F .text  00000038 function
   00000000         *UND*  00000000 external_function
   0000005c  w    F .text  00000024 weak_function

   $ nm test.o
45          U external_function
   00000000 T function
   00000038 t static_function
   00000000 s static_variable
   0000005c W weak_function
50
```

Notice the use of `#pragma` to define the weak symbol. A `pragma` is a way of communicating extra information to the compiler; its use is not common but occasionally is required to get the compiler to do out of the ordinary operations.x

We inspect the symbols with two different tools; in both cases the binding is shown in the second column; the codes should be quite straight forward (are are documented in the tools man page).

Overriding symbols

It is often very useful for a programmer to be able to *override* a symbol in a library; that is to subvert the normal symbol with a different definition.

We mentioned that the order that libraries is searched is given by the order of the `DT_NEEDED` fields within the library. However, it is possible to insert libraries as the *last* libraries to be searched; this means that any symbols within them will be found as the final reference.

This is done via an environment variable called `LD_PRELOAD` which specifies libraries that the linker should load last.

Example 9.17. Example of `LD_PRELOAD`

```
 1
                $ cat override.c
   #define _GNU_SOURCE 1
   #include <stdio.h>
 5 #include <stdlib.h>
   #include <unistd.h>
```

```
   #include <sys/types.h>
   #include <dlfcn.h>

10 pid_t getpid(void)
   {
           pid_t (*orig_getpid)(void) = dlsym(RTLD_NEXT, "getpid");
           printf("Calling GETPID\n");

15         return orig_getpid();
   }

   $ cat test.c
   #include <stdio.h>
20 #include <stdlib.h>
   #include <unistd.h>

   int main(void)
   {
25         printf("%d\n", getpid());
   }

   $ gcc -shared -fPIC -o liboverride.so override.c -ldl
   $ gcc -o test test.c
30 $ LD_PRELOAD=./liboverride.so ./test
   Calling GETPID
   15187
```

In the above example we override the `getpid` function to print out a small statement when it is called. We use the `dlysm` function provided by `libc` with an argument telling it to continue on and find the *next* symbol called `getpid`.

Weak symbols over time

The concept of the *weak* symbol is that the symbol is marked as a lower priority and can be overridden by another symbol. Only if no other implementation is found will the weak symbol be the one that it used.

The logical extension of this for the dynamic loader is that all libraries should be loaded, and any weak symbols in those libraries should be ignored for normal symbols in any other library. This was indeed how weak symbol handling was originally implemented in Linux by glibc.

However, this was actually incorrect to the letter of the Unix standard at the time (*SysVr4*). The standard actually dictates that weak symbols should only be handled by the *static* linker; they should remain irrelevant to the dynamic linker (see the section on binding order below).

At the time, the Linux implementation of making the dynamic linker override weak symbols matched with SGI's IRIX platform, but differed to others such as Solaris and AIX. When the developers realised this behaviour violated the standard it was reversed, and the old behaviour relegated to requiring a special environment flag (*LD_DYNAMIC_WEAK*) be set.

Specifying binding order

We have seen how we can override a function in another library by *preloading* another shared library with the same symbol defined. The symbol that gets resolved as the final one is the last one in the order that the dynamic loader loads the libraries.

Libraries are loaded in the order they are specified in the `DT_NEEDED` flag of the binary. This in turn is decided from the order that libraries are passed in on the command line when the object is built. When symbols are to be located, the dynamic linker starts at the last loaded library and works backwards until the symbol is found.

Some shared libraries, however, need a way to override this behaviour. They need to say to the dynamic linker "look first inside me for these symbols, rather than working backwards from the last loaded library". Libraries can set the `DT_SYMBOLIC` flag in their dynamic section header to get this behaviour (this is usually set by passing the `-Bsymbolic` flag on the static linkers command line when building the shared library).

What this flag is doing is controlling *symbol visibility*. The symbols in the library can not be overridden so could be considered private to the library that is being loaded.

However, this loses a lot of granularity since the library is either flagged for this behaviour, or it is not. A better system would allow us to make some symbols private and some symbols public.

Symbol Versioning

That better system comes from symbol versioning. With symbol versioning we specify some extra input to the static linker to give it some more information about the symbols in our shared library.

Example 9.18. Example of symbol versioning

```
 1
                $ cat Makefile
   all: test testsym

 5 clean:
           rm -f *.so test testsym

   liboverride.so : override.c
           $(CC) -shared -fPIC -o liboverride.so override.c
10
   libtest.so : libtest.c
           $(CC) -shared -fPIC -o libtest.so libtest.c

   libtestsym.so : libtest.c
15         $(CC) -shared -fPIC -Wl,-Bsymbolic -o libtestsym.so libtest.c

   test : test.c libtest.so liboverride.so
           $(CC) -L. -ltest -o test test.c

20 testsym : test.c libtestsym.so liboverride.so
           $(CC) -L. -ltestsym -o testsym test.c

   $ cat libtest.c
   #include <stdio.h>
25
   int foo(void) {
           printf("libtest foo called\n");
           return 1;
   }
30
   int test_foo(void)
   {
           return foo();
   }
```

```
35
   $ cat override.c
   #include <stdio.h>

   int foo(void)
40 {
           printf("override foo called\n");
           return 0;
   }

45 $ cat test.c
   #include <stdio.h>

   int main(void)
   {
50         printf("%d\n", test_foo());
   }

   $ cat Versions
   {global: test_foo;  local: *; };
55
   $ gcc -shared -fPIC -Wl,-version-script=Versions -o libtestver.so libtest.c

   $ gcc -L. -ltestver -o testver test.c

60 $ LD_LIBRARY_PATH=. LD_PRELOAD=./liboverride.so ./testver
   libtest foo called

   100000574 l     F .text 00000054              foo
   000005c8 g     F .text 00000038              test_foo
65
```

In the simplest case as above, we simply state if the symbol is *global* or *local*. Thus in the case above the `foo` function is most likely a support function for `test_foo`; whilst we are happy for the overall functionality of the `test_foo` function to be overridden, if we do use the shared library version it needs to have unaltered access nobody should modify the support function.

This allows us to keep our *namespace* better organised. Many libraries might want to implement something that could be named like a common function like `read` or `write`; however if they all did the actual version given to the program might be completely wrong. By specifying symbols as *local* only the developer can be sure that nothing will conflict with that internal name, and conversely the name he chose will not influence any other program.

An extension of this scheme is *symbol versioning*. With this you can specify multiple versions of the same symbol in the same library. The static linker appends some version information after the symbol name (something like @VER) describing what version the symbol is given.

If the developer implements a function that has the same name but possibly a binary or programatically different implementation he can increase the version number. When new applications are built against the shared library, they will pick up the latest version of the symbol. However, applications built against earlier versions of the same library will be requesting older versions (e.g. will have older @VER strings in the symbol name they request) and thus get the original implementation. XXX : example

Chapter 10. I/O Fundamentals

File System Fundamentals

File System Funamentals

Networking Fundamentals

Networking Fundamentals

Glossary

A

Application Programming Interface	The set of variables and functions used to communicate between different parts of programs. See Also Application Binary Interface.
Application Binary Interface	A technical description of how the operating system should interface with hardware. See Also Application Programming Interface.

E

Extensible Markup Language	Some reasonable definition here. See Also Standardised Generalised Markup Language.
Standardised Generalised Markup Language	The grand daddy of all documents See Also Extensible Markup Language.

M

Mutually Exclusive	When a number of things are mutually exclusive, only one can be valid at a time. The fact that one of the things is valid makes the others invalid.
MMU	The *memory managment unit* component of the hardware architecture.

O

Open Source	Software distributed in source form under licenses guaranteeing anybody rights to freely use, modify, and redistribute the code.

S

Shell	The interface used to interact with the operating system.