Linking and Loading Exercise

Second Year Computing Laboratory
Department of Computing
Imperial College London

Summary

This first lab exercise is intended to support the *Compilers*, *Software Engineering*, *Computer Architecture* and *Operating Systems* courses. The initial investigation should help you to understand the purpose of compilers, assemblers, linkers and loaders in the generation of executable files. This exercise will also give you a chance to practice writing makefiles and simple scripts (skills that will be very helpful in this year's laboratory programme) as well as introducing you to the format of the second year lab exercises. In particular, how you will receive provided/skeleton files, how you should test your work and how you will make your final submissions.

Submit by 19:00 on Friday 14th October 2016

What To Do:

You need to work through this exercise sheet, trying out the suggested commands and answering the questions where appropriate. In the final section of this exercise sheet you will also be asked to write a simple makefile and a simple script (in a scripting language of your choice). Your answers to questions 1-5 should not need to be more than a paragraph each.

This exercise can *only* be done on a Unix operating system. It is recommended that you use one of the lab Linux machines, or that you SSH into one of the lab machines (not one of the `shell` servers!). The tools that you will be using will probably not be present on other operating systems.

If you choose to use your own machine you should ensure that your C compiler `gcc` is version 4.6 or later (type `gcc -v` to find out which version of the compiler is installed on your machine). Earlier versions of `gcc` may not give the expected answers to some of the questions below.

Getting information about Linux

There is a lot of information on-line about the Linux commands, for example:
`http://www.oreillynet.com/linux/cmd/`
provides an alphabetical directory of Linux commands. There are also a number of useful commands from within the Linux terminal. The `whatis` command provides a one line summary of what a command is for. For example:

```
prompt> whatis as
as (1)               - the portable GNU assembler.

prompt> whatis gcc
gcc (1)              - GNU project C and C++ compiler
```

The `man` command gives a lot more detailed information about a Linux command, including all of the options available for that command and often some example uses of the command.

Getting the files required for the exercise

You have each been provided with a Git repository on the department's `GitLab` server that contains the files required for this exercise. To obtain this skeleton repository you will need to clone it into your local workspace. You can do this with the following command:

```
prompt> git clone https://gitlab.doc.ic.ac.uk/lab1617_autumn/linkload_<login>.git
```

replacing `<login>` with your normal college login. You will be prompted for your normal college username and password. This will create a directory in your current location called `linkload_<login>`. This is generally the way that we will hand out all lab exercises this year.

You should work on the files in your local workspace, making regular commits and pushes back to this Git repository. Recall that you will first need to add any new/modified files to your local Git workspace with

```
prompt> git add <filename>
```

You can then commit your changes to your local index with

```
prompt> git commit -m "your commit message here"
```

Finally you will need to push these changes from your local index to the Git repository with

```
prompt> git push origin master
```

You can check that a push succeeded by looking at the state of your repository using the `GitLab` webpages:
`https://gitlab.doc.ic.ac.uk/`
(you will need to login with your normal college username and password).

You are of course free to utilise the more advanced features of Git such as branching and tagging. Further details can be found in your first year notes and at:
`https://workspace.imperial.ac.uk/computing/Public/files/Git-Intro.pdf`.

Important: Your final submission will be taken from this `GitLab` repository, so make sure that you push your work to it correctly. If in any doubt, come and see me in my office (room 228) or during one of the lab sessions. It is your responsibility to ensure that you submit the correct version of your work.

The provided files for this exercise are:

`hello.s`	a simple Intel assembly program.
`chello.c`	a C version of the same program.
`writeexit.s`	an assembly file the provides some support for `chello.c`.
`simple.sh`	a simple example bash script.
`simple.rb`	a simple example ruby script.
`example.c`	a more complex C program for experimenting with optimisations.
`answers.txt`	a skeleton submission file (pdfs may also be submitted).
`makefile`	a skeleton makefile for you to edit.
`last_touch`	a skeleton script for you to edit.

Working with Assembly Code

The principles of Linux assembler programs are similar to those of other assembler programs you may have seen, although the syntax may differ slightly. The file `hello.s`, shown in Figure 1 contains a simple "Hello World" program.

- The `.text` directive on line 1 of the program marks the start of the instructions.
- The `.globl` directive on line 2 makes the label `_start` accessible outside of the section it is defined in.
- Constants in the assembler source are written with a leading dollar sign '`$`'.
- The prefix '`0x`' indicates that a number is written in hexadecimal.

- Register names start with '`%`', the following '`e`' indicates that only 32 bits of the register are used.
- Command operands are written in the order: source, destination.
- The `int` instruction, found on lines 8 and 11, is a software interrupt (or trap) used to make requests to the Linux operating system. We want to let the program jump to the OS, but it would not be safe to let it jump in just anywhere. So `int` is a protected procedure call, "call OS function number N", where the OS has set up a table that maps N to the actual functions to be called. The value stored in register `eax` indicates which system call is to be made. Depending on the call further information may be passed to the operating system in other registers.
- The `exit` system call, set-up on line 10 and called on line 11, terminates the process making the call. When the process exists the OS returns a one-byte exit code to the parent process (usually the shell from which it was called). This is sometimes used to indicate whether an operation succeeded or not.

```
1          .text
2          .globl _start
3  _start:
4          movl   $0x4, %eax        # eax = code for 'write' system call
5          movl   $1, %ebx          # ebx = file descriptor to standard output
6          movl   $message, %ecx    # ecx = pointer to the message
7          movl   $13, %edx         # edx = length of the message
8          int    $0x80             # make the system call
9          movl   $0x0, %ebx        # the status returned by 'exit'
10         movl   $0x1, %eax        # eax = code for 'exit' system call
11         int    $0x80             # make the system call
12         .data
13         .globl message
14 message:
15         .string "Hello world!\n" # The message as data
```

Figure 1: The "Hello World" assembly code in `hello.s`

Many of the 32 bit instructions work in x86_64 programs too and `hello.s` should still work as expected on a 64 bit architecture. Later in this exercise we will see an example of a program written using x86_64 instructions.

Using the Assembler

As discussed above, the `as` command is the Linux assembler. You can assemble `hello.s` with the following command:

```
prompt> as hello.s -o hello.o
```

The '`-o`' option specifies the name of the output file, in this case `hello.o`. This resulting file contains relocatable binary machine instructions, initialised values and other data. If you want to determine the type of a file you can use the command `file` which identifies the type of a file by analysing its contents. For example, running the command:

```
prompt> file hello.o
```

results in:

```
hello.o: ELF 64-bit LSB relocatable, x86-64, version 1 (SYSV), not stripped
```

The result tells us that this executable code is stored in ELF, the *Executable and Linkable Format.* We can also see that `hello.o` is relocatable, meaning it contains information allowing it to be loaded at any required address.

Each executable file starts with a fixed size *ELF header.* The ELF header is made up of three parts: the main ELF header, the program headers and the section headers. The actual program code can be found between the program headers and the section headers, along with any data constants used by the program and some other program meta-data. The structure of the ELF header file is described in more detail in the appendix.

Looking at hello.o

You can examine the contents of a non-text file like `hello.o` using the Linux octal-dumping utility `od`. For example, you can run the command:

```
prompt> od -xc -Ax hello.o
```

The '`-xc`' option requests the output be displayed in hexadecimal and ASCII and the '`-Ax`' option requests that all addresses be displayed in hexadecimal format.

The result of running this command is included in the appendix with annotations describing the information stored in the ELF header file. The first column of the result shows the start address for the line within the file. The remainder of the line shows the 16 bytes from the file at this address.

An important part of the ELF header file is the symbol table which provides a list of the labels (procedure names and variable names) used in the program. Rather than having to look through the output from `od` in gory detail, you can use the Linux symbol names utility `nm` to list all of the symbols defined or used in an ELF header file. For example, running the command:

```
prompt> nm hello.o
```

should show that the only labels defined in this program are '`_start`' and '`message`'.

Another useful way of examining an executable file is to use the `objdump` utility. Running the command with the '`--disassemble`' option:

```
prompt> objdump --disassemble hello.o
```

displays the executable part of the file as assembler code. Running the command with the '`--all-headers`' option:

```
prompt> objdump --all-headers hello.o
```

displays a summary of the information stored in the ELF header file, including the symbol table and the relocation records for the file.

Using the Linker

All but the simplest programs are constructed from multiple source files which are compiled and assembled at different times. To make the sections of a program reusable, the assembler outputs the binary code in a form which can be loaded at any address in memory (that is, it is relocatable). When the assembler writes the binary code for an instruction that uses a label, the address of that label is not yet known. To cope with this, the assembler writes the address as '`0`' and then creates a relocation record for that address. This relocation record is added to the assembler file and indicates that the instruction must be changed when the label's address is known. To see this you can compare the instructions in the disassembled `hello.o` with the original source file `hello.s`.

The purpose of the Linux *linker* `ld` is to resolve all of the relocation records in a program and produce a single file with a fixed start address. In our running "Hello World" example, all that needs to be done is to relocate the text and program data so that they reside at the correct start address. We do this by running the following command:

```
prompt> ld -N hello.o -o hello
```

The '`-N`' option tells the linker to set the text and data sections to be readable and writable. The command takes the code from `hello.o` and relocates it for loading at an address where the Linux loader expects it to be. It writes the resulting executable program to the file `hello`. This file is also in the ELF format, but examining it shows that it now has a new entry point and a few other parts will also have changed. You should examine `hello` using the `od` and `objdump` utilities.

Running the Program

You can now run the "Hello World" program with the command:

```
prompt> ./hello
```

The operating system's loader will look at the data structure contained in the file `hello` checking the machine type and examining the header to determine what kind of environment the process will require. Once the process has been set up, the instructions and initialised data will be loaded into free memory and the operating system will branch to the program's entry point. For a more detailed discussion of this process, see the manual page for the `execve` system call.

Question 1. How big (how many bytes) is the program code section of the file `hello.o`? Explain how you arrived at your answer, including any mathematical conversions you had to make.

Question 2. How many relocation records are there in the file `hello.o`? Name them and explain how you arrived at your answer.

Question 3. How many relocation records are there in the file `hello` (the result of linking `hello.o`)? Explain how you arrived at your answer.

Working with C Code

The C version of our simple "Hello World" program, shown in Figure 2, is written in the file `chello.c`. The program imports two externally-defined functions `writeA` and `exitA` which are defined in `writeexit.s`, a Linux x86_64 assembly program shown in Figure 3. Note that in x86_64 programs, to reduce the need to use the stack for parameter passing, the first three integer parameters of any function (including system calls) are placed into the `%edx`, `%esi` and `%eax` registers.

```c
extern void writeA(int, char [], int );
extern void exitA(int);

char message [] = "Hello World!\n";
#define MESSAGELENGTH 13
int main(void){
  writeA(1, message, MESSAGELENGTH);
  exitA(0);
}
```

Figure 2: The "Hello World" C program in `chello.c`

We now look at the steps that are required to construct a running program from these components. In practice, much of this process can be automated, but it is important that you understand what is happening to your code. The first step is to use the C compiler `gcc` to translate our C program into an assembler program. Run the command:

```
prompt> gcc -S chello.c
```

The '`-S`' option tells the compiler to stop after producing the assembly version of the file. You may find it interesting to compare the assembly generated in `chello.s` with that in `hello.s`.

The next step is to assemble `chello.s`. As before we do this with the command:

```
prompt> as chello.s -o chello.o
```

The output of this command will be a file called `chello.o` which is in the ELF format. You should examine this file using the utilities discussed in the Assembly Code section above.

The third step in the process of building our C program is to assemble the support file `writeexit.s` which we do as follows:

```
prompt> as writeexit.s -o writeexit.o
```

```
  .text
# an implementation of write and exit in assembly
#
  .globl writeA
  .type writeA, @function
  # the string length is passed in %edx
  # the string  address is passed in %esi
  # the number of the stream is passed in  %edi
writeA:
  .cfi_startproc
        pushq  %rbp               # save register ebx
        movl   %esi,%ecx          # move the string address to ecx
        movl   $0x4,%eax          # write system call
        int    $0x80
        popq   %rbp               # restore ebx
        ret
  .cfi_endproc
        .globl exitA
exitA:
        pushq   %rbp             # save %rbp
        movl    %edi,%ebx        # move the number to return to %ebx
        movl   $0x1,%eax
        int    $0x80

  .globl _start
# the program is started by a call to _start

_start:
# call the C code
  call  main
  .globl main
```

Figure 3: The assembly code in `writeexit.s`

Finally, we are in a position to link the two files using the Linux linker:

```
prompt> ld -N chello.o writeexit.o -o chello
```

This results in an executable file `chello` in the ELF format. You can now run this program with the command:

```
prompt> ./chello
```

and should see the text `Hello World`! output to the terminal.

Question 4. What happens if you try to link `chello.o` using `ld` as you did with `hello.o` (that is by running the command: `ld -N chello.o -o chello`)? Explain what has happened and how the `nm` utility can be used to confirm this.

Question 5. The last instruction in `hello.s` is an `exit` system call. What would happen if you assembled, linked and ran this program with this instruction commented out? (*Hint*: you might want to try this) Explain why this happens. Does the program still terminate?

Makefiles

In the previous section we saw how to build a runnable program from the C code in `chello.c` and the assembly code in `writeexit.s`. However, this approach is both cumbersome and time consuming. We could write a single line command that will do all of these steps for us, for example:

```
prompt> gcc -c chello.c -o chello.o && as writeexit.s -o writeexit.o
&& ld -N chello.o writeexit.o -o chello
```

Here the '-c' option tells the compiler to stop before attempting to link the file and the && operator runs the first command and then, if its exit code reports success, runs the next command. You can then use the up arrow in the terminal to find this command to avoid having to type it afresh each time you want to use it. However, this doesn't allow you to switch machines and also wastes time recompiling files that you may not have changed. The correct solution is to create a makefile for your program.

Linux provides a program 'make' that keeps track of how to build your program from its source files. The information required for this program to run is contained in a file, typically called 'makefile'. The syntax of a makefile is relatively simple. Variable definitions (with variable names in capitals) can be written as follows:

```
CODE=foo.c bar.c
```

Of more importance is the definition of rules which tell the make program what file is being made, what files it is made from and how to build it. For example:

```
main: foo.c bar.c
	gcc foo.c bar.c -o main
```

tells the make program how to build the file main. This build can be run with the command:

```
prompt> make main
```

The target file is followed by a colon which is then followed by the dependencies, the files that the target file depends on. In this case main needs to be rebuilt whenever foo.c or bar.c are modified. The last part of the rule is the command that make should run in order to create the target from its dependencies. In this case main is built using the gcc command.

Important: the line with the gcc command on it is prefixed by a tab character and *not* a number of spaces. If you put spaces in instead then the make program will throw a "missing separator" error.

Makefiles normally consist of multiple rules, each explaining how to build some part of the program. If make is run with no arguments, it will attempt to build the first rule encountered in the makefile (this may recursively trigger other make rules, based on the rule dependencies). Rules are terminated by a blank line, allowing you to trigger multiple commands in one rule. Rules can also make use of variables, for instance using the definition of CODE from above we could rewrite our rule for main as:

```
main: $(CODE)
	gcc $(CODE) -o main
```

where the syntax $(...) is used to access the values of variables. Using variables makes it easier to modify your makefiles when you add in extra dependencies or rules.

It is common practice to specify an all target as the first rule in a makefile (this usually redirects make to a particular rule in the makefile) and also a clean target, as the last rule (this usually deletes all compiled code in the project). For example:

```
all: main

  ...

clean:
	rm -rf *.o
```

Note, however, that if files of these names (all or clean) exist in the directory, then the make command will try to rebuild them and this will result in an error. To avoid this issue we can specify make targets as .PHONY, which tells make that they are not project files. For example, we could write:

```
.PHONY: all
```

so that the command make never tries to build a file called all.

Question 6. Edit the provided file makefile so that running make will build the executable chello from the source files chello.c and writeexit.s as discussed in the previous section. You must also write a clean rule, so that running make clean will delete all compiled files. You must ensure that your makefile sets up the necessary dependencies to avoid unnecessary compilation of unmodified files. You may optionally make use of appropriate variables, but must keep the file's purpose clear.

Scripting

A script is a file containing a sequence of commands which are run when the script is executed. For example, most editors typically conclude their setup by running a user defined script that initialises the users preferred settings. Here we will give short overviews of Bash, a scripting language extension of the original Unix shell, and Ruby, a dynamic, open source programming language that you will encounter in the Software Engineering course. You can find more details on both of these languages in the appendix and on-line. For example, a basic Bash tutorial can be found at:
`http://linuxconfig.org/Bash_scripting_Tutorial`
and a basic Ruby tutorial can be found at:
`https://www.ruby-lang.org/en/documentation/quickstart`.

```
1  #!/bin/bash
2  # simple.sh
3
4  # A simple example of a bash script
5  echo "Running the bash script $0"
6  #read in input
7  name=$1
8  #find the current path
9  dir=‘pwd‘
10 echo "hello $name you are at the path $dir"
```

Figure 4: The Bash script `simple.sh`

As a simple example consider the provided Bash script `simple.sh`, shown in Figure 4. You can run this script with the command:

```
prompt> ./simple.sh "your name here"
```

This script makes use of the following properties of Bash:

- Anything following a hash sign `#` in a Bash script is treated as a comment (except the command interpreter name on line 1).
- Variables are defined by assigning to them for the first time. There must be no spaces in an assignment as spaces are used to split up statements. Double quotes `"..."` can be used to prevent spaces from being treated specially. The value of a variable is accessed via the dollar symbol, as in `$name`, and the inputs to a script are passed in the variables `$0`, `$1`, `$2`,... with `$0` containing the name of the script, `$1` containing the script's first argument, and so on.
- The output of a system call can be returned to the script, rather than the terminal, by enclosing it in back quotes, as in `‘pwd‘`.

```
#!/usr/bin/ruby
# simple.rb

# A simple example of a ruby script
puts "Running the ruby script #{$0}"
#read in input
name = ARGV[0]
#find the current path
dir = ‘pwd‘
puts "hello #{name} you are at the path #{dir}"
```

Figure 5: The Ruby script `simple.rb`

Similarly, a simple Ruby script is provided in the file `simple.rb`, shown in Figure 5. As before, you can run this script with the command:

```
prompt> ./simple.rb "your name here"
```

This script makes use of the following properties of Ruby:

- Anything following a hash sign `#` in a Ruby script is treated as a comment (except the command interpreter name on line 1).
- Variables are defined by assigning to them for the first time. Spacing is not important in Ruby. The value of a variable can be accessed within a string via the `#{...}` syntax, as in `#{name}`. The inputs to a program are passed in the `ARGV` array, with `ARGV[0]` containing the first argument, `ARGV[1]` the second argument, etc. The special variable `$0` contains the name of the script.
- The output of a system call can be returned to the script, rather than the terminal, by enclosing it in back quotes, as in `‘pwd‘`.

When writing scripts that include destructive updates, such as `rm`, it is advisable to first write the script with print statements, such as `echo` or `puts`, so that you can test it safely. Once you are sure the script is correct, you can then replace such print commands with the actual update commands.

Question 7. Sometimes, when working as a group, it can be frustrating trying to determine when or why a file was last changed. Your last task for this exercise is to write a script that will tell the user which files in a directory are under version control and which git user, if any, last modified these files (along with some information about when that modification was made). You should edit the script `last_touch` so that it fulfils this purpose (more details below).

Hints and Tips:

You can think of your `last_touch` script as a light-weight version of the `git blame` command. However, we only want to inspect the files in the repo, not the full revision history of their contents.

Your `last_touch` script should expect to be called as:

```
prompt> last_touch <target_path>
```

If given no arguments, you should treat `<target_path>` as the *current* working directory. If more than one argument is given, or `<target_path>` does not point to a directory, then the script should exit with a non-zero exit code and output a meaningful error message.

The script should output, in order of modification date (most recent first), information about all of the files at or beneath the target directory path. There should be no output for directories/folders.

The information for each file should be output one per line in the following format:

```
<datetime> [<revision_id>]: <file_path><padding>(<author>)
```

where:

- the `<datetime>` is the UTC time-stamp of when the file was *last* modified in a git commit that was *not* an automated merge;
- the `<revision_id>` is the first 5 characters of the SHA1 ID of this git commit;
- the `<file_path>` is the path to the file relative to the input `<target_path>`;
- the `<author>` is the git user who made the corresponding commit;
- and the `<padding>` gap between the `<file_path>` and `<author>` ensures that all of the authors are neatly left-aligned in the output.

Any files within the target directory structure that have *not* been checked into git should be output after this in the format:

```
[***not under version control***]: <file_path>
```

As an example, if your script were called on the provided git repository for this lab exercise *before* you have made any commits, then the output should be:

```
2016-09-30T15:36:13+00:00 [99b07]: makefile    (Mark Wheelhouse)
2016-09-30T15:36:13+00:00 [99b07]: writeexit.s (Mark Wheelhouse)
2016-09-30T15:36:13+00:00 [99b07]: simple.sh   (Mark Wheelhouse)
2016-09-30T15:36:13+00:00 [99b07]: simple.rb   (Mark Wheelhouse)
2016-09-30T15:36:13+00:00 [99b07]: last_touch  (Mark Wheelhouse)
2016-09-30T15:36:13+00:00 [99b07]: hello.s     (Mark Wheelhouse)
2016-09-30T15:36:13+00:00 [99b07]: example.c   (Mark Wheelhouse)
2016-09-30T15:36:13+00:00 [99b07]: chello.c    (Mark Wheelhouse)
2016-09-30T15:36:13+00:00 [99b07]: answers.txt (Mark Wheelhouse)
```

It is entirely up to you how you would like to approach this task and which utilities you want to use to help you. However, I would suggest that working out how to traverse a directory-structure and reading up on the `git log` command is a useful place to start. I also suggest that a date handling library may be useful and that you almost certainly need to do some string parsing to be able to extract data from the output of other utilities.

You are free to write your script in any scripting language provided on the Lab machines, but we recommend that you use Ruby or Bash. Your script should be written for the general case and will be tested on some arbitrary directory structures with files that are both under version control and not. You should also recall the importance of well-chosen comments to keep the meaning of your code clear.

Important: You are *never* permitted to submit binary executables for second year lab exercises.

Testing

As you work, you should *add*, *commit* and *push* your changes to your Git repository, as discussed above. You should also be carefully testing your work throughout the exercise.

You should be used to regularly testing your code on your development machine, but to help you ensure that your code will compile and run as expected in our testing environment, we have provided you with a Lab Testing Service: `LabTS`.

`LabTS` will clone your `GitLab` repository and run several automated test processes over your work. This will happen automatically after the deadline, but can also be requested during the course of the exercise (usually on a sub-set of the final tests).

You can access the `LabTS` webpages at:

`https://teaching.doc.ic.ac.uk/labts`

Note that you will be required to log-in with your normal college username and password.

If you click through to your `linkload_<login>` repository you will see a list of the different versions of your work that you have pushed. Next to each commit you will see a button that will allow you to request that this version of your work is run through the automated test process. If you click this button your work will be tested (this may take a few minutes) and the results will appear in the relevant column.

Important: It is **your** responsibility to ensure that your code behaves as expected in our automated test environment. Code that fails to compile/run in this environment will score **zero marks**. You should find that this environment behaves like the set-up found on our lab machines. If you are experiencing any problems in this regard then you should seek help from a lab demonstrator or the lab coordinator at the earliest opportunity.

Submission

Your `GitLab` repository should contain the final submission for your `makefile` and `last_touch` script. `LabTS` can be used to test any revision of your work that you wish. However, you will still need to submit a *revision id* to CATe so that we know which version of your code you consider to be your final submission.

Prior to submission, you should check the state of your `GitLab` repository using the `LabTS` webpages:
`https://teaching.doc.ic.ac.uk/labts`
If you click through to your `linkload_<login>` repository you will see a list of the different versions of your work that you have pushed. Next to each commit you will see a link to that commit on `GitLab` as well as a button to submit that version of your code to CATe. Pressing this button will redirect you to CATe (automatically submitting your revision id) and prompt you to upload an answers file and sign the usual "original work" disclaimer.

You should submit to CATe the version of your code that you consider to be "final". You can change this later by submitting a different version to CATe as usual. The CATe submission button on LabTS will be replaced with a green confirmation label if the submission has been successful.

You should submit your answers to questions 1-5 (either `answers.txt` or `answers.pdf`) and the chosen version of you code to CATe by 19:00 on Friday 14th October 2016.

Assessment

In total there are 35 marks available in this exercise. These are allocated as follows:

Question 1	5 marks
Question 2	4 marks
Question 3	2 marks
Question 4	4 marks
Question 5	4 marks
Question 6: `makefile`	6 marks
Question 7: `last_touch`	10 marks

Feedback on the exercise will be returned by Friday 28th October 2016.

EXTRA: Experimenting with Optimisation (Not Assessed)

So far we have run the C compiler in its default mode, which chooses to compile code quickly rather than aiming to produce good code. If you use the '`-O`' flag, then the compiler will try to optimise the code that it outputs. Now that you know how to run the C compiler and examine its output, you can experiment to see how clever it can be. For example,

- does it replace constant expressions with their values?
- does it replace code segments with shorter sequences of instructions?
- does it avoid recomputing an expression's value within a loop if that value cannot change during the loop?
- does it replace function calls by a copy of the function's body?

In order to perform these kinds of experiments we have provided you with a slightly more complex C program `example.c`, as shown in Figure 6. You can use the C compiler to compile `example.c` into assembly code at several different levels of optimisation. For example:

- `gcc -O -S example.c -o opt1`
- `gcc -O2 -S example.c -o opt2`
- `gcc -O2 -finline-functions -S example.c -o opt3`

You should examine the assembly code produced at the various levels of optimisation available. You may find that the code becomes rather complicated at the higher levels of optimisation.

```c
/*
 * This shows a procedure call with a single parameter.
 * It looks simple enough, but at high optimisation levels
 * produces some pretty weird code.
 */

int i;
int a[100];
int b=42;

void f(int c){
  for (i = 1; i < 100-2*25; i = i + 1){
    a[i] = c*3+10;
  }
}

int main(){
  f(b);
  return 0;
}
```

Figure 6: The C program in `example.c`

Appendix A - ELF Headers

Executable code on Linux is usually stored in ELF, the *Executable and Linkable Format*. ELF specifies a standard format for mapping your code on disk to a complete executable image in memory that consists of: your code; a stack; a heap (for malloc); and all the libraries you link against. There are three header areas in an ELF file: The main ELF file header; the program headers; and the section headers. The actual program code can be found between the program headers and the section headers, along with any data constants used by the program and some other program meta-data. An annotated ELF header file for the assembled `hello.o` is given in Figure 7. To save space, some of the sections are only described and not shown in full detail.

The Main ELF File Header

The main elf header tells us where everything is located in the file and comes at the very beginning of the executable. It has the following format:

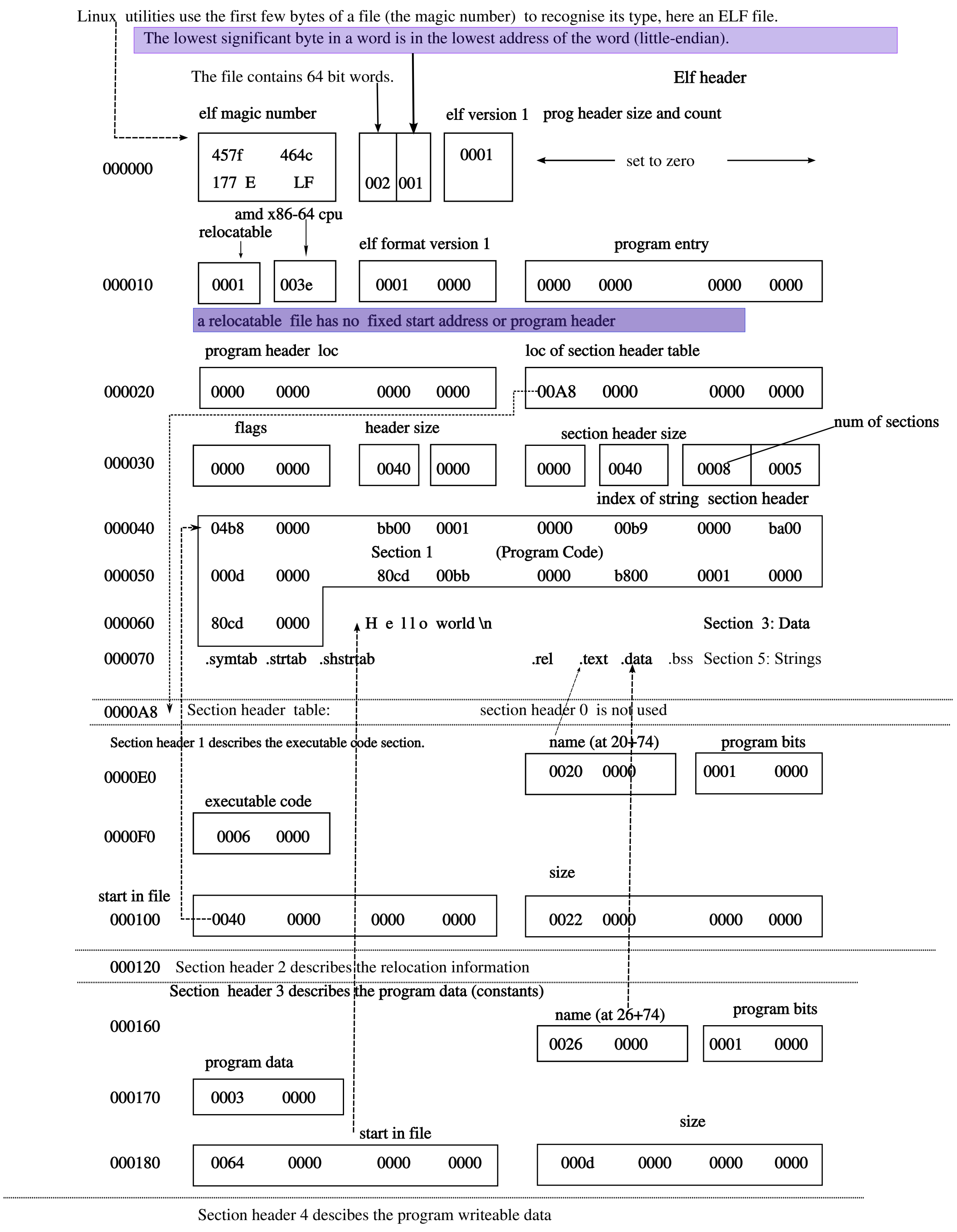
```
/* ELF File Header */
typedef struct
{
  unsigned char e_ident[EI_NIDENT];     /* Magic number and other info */
  Elf32_Half    e_type;                 /* Object file type */
  Elf32_Half    e_machine;              /* Architecture */
  Elf32_Word    e_version;              /* Object file version */
  Elf32_Addr    e_entry;                /* Entry point virtual address */
  Elf32_Off     e_phoff;                /* Program header table file offset */
  Elf32_Off     e_shoff;                /* Section header table file offset */
  Elf32_Word    e_flags;                /* Processor-specific flags */
  Elf32_Half    e_ehsize;               /* ELF header size in bytes */
  Elf32_Half    e_phentsize;            /* Program header table entry size */
  Elf32_Half    e_phnum;                /* Program header table entry count */
  Elf32_Half    e_shentsize;            /* Section header table entry size */
  Elf32_Half    e_shnum;                /* Section header table entry count */
  Elf32_Half    e_shstrndx;             /* Section header string table index */
} Elf32_Ehdr;
```

The Program Headers

The ELF program headers describe the sections of the program that contain executable program code that needs to be mapped into the program address space as it loads. Each program header has the following format:

```
/* Program segment header.  */

typedef struct
{
  Elf32_Word    p_type;                 /* Segment type */
  Elf32_Off     p_offset;               /* Segment file offset */
  Elf32_Addr    p_vaddr;                /* Segment virtual address */
  Elf32_Addr    p_paddr;                /* Segment physical address */
  Elf32_Word    p_filesz;               /* Segment size in file */
  Elf32_Word    p_memsz;                /* Segment size in memory */
  Elf32_Word    p_flags;                /* Segment flags */
  Elf32_Word    p_align;                /* Segment alignment */
} Elf32_Phdr;
```



Linux utilities use the first few bytes of a file (the magic number) to recognise its type, here an ELF file.
The lowest significant byte in a word is in the lowest address of the word (little-endian).
The file contains 64 bit words.
Elf header
elf magic number
elf version 1
prog header size and count
000000
457f 464c
177 E LF
002
001
0001
set to zero
amd x86-64 cpu
relocatable
elf format version 1
program entry
000010
0001
003e
0001 0000
0000 0000 0000 0000
a relocatable file has no fixed start address or program header
program header loc
loc of section header table
000020
0000 0000 0000 0000
00A8 0000 0000 0000
flags
header size
section header size
num of sections
000030
0000 0000
0040
0000
0000
0040
0008
0005
index of string section header
000040
04b8 0000 bb00 0001 0000 00b9 0000 ba00
Section 1 (Program Code)
000050
000d 0000 80cd 00bb 0000 b800 0001 0000
000060
80cd 0000
H e l l o world \n
Section 3: Data
000070
.symtab .strtab .shstrtab
.rel .text .data .bss Section 5: Strings
0000A8
Section header table:
section header 0 is not used
Section header 1 describes the executable code section.
name (at 20+74)
program bits
0000E0
0020 0000
0001 0000
executable code
0000F0
0006 0000
size
start in file
000100
0040 0000 0000 0000
0022 0000 0000 0000
000120 Section header 2 describes the relocation information
Section header 3 describes the program data (constants)
name (at 26+74)
program bits
000160
0026 0000
0001 0000
program data
000170
0003 0000
size
start in file
000180
0064 0000 0000 0000
000d 0000 0000 0000
Section header 4 descibes the program writeable data

Figure 7: The annotated ELF header for `hello.o`

The Section Headers

The ELF section headers describe the various named sections of an executable file. Each section has an entry in the section headers array, which is found at the end of the executable, and has the following format:

```
/* Section header.  */

typedef struct
{
  Elf32_Word    sh_name;                /* Section name (section table index) */
  Elf32_Word    sh_type;                /* Section type */
  Elf32_Word    sh_flags;               /* Section flags */
  Elf32_Addr    sh_addr;                /* Section virtual addr at execution */
  Elf32_Off     sh_offset;              /* Section file offset */
  Elf32_Word    sh_size;                /* Section size in bytes */
  Elf32_Word    sh_link;                /* Link to another section */
  Elf32_Word    sh_info;                /* Additional section information */
  Elf32_Word    sh_addralign;           /* Section alignment */
  Elf32_Word    sh_entsize;             /* Entry size if section holds table */
} Elf32_Shdr;
```

Appendix B - Bash Scripting

Bash is now quite an old scripting language with a rather rigid syntax. At its simplest, it can be thought of as a way of chaining together shell commands. However, for the experienced programmer, it is a very powerful tool. For those of you just starting out with script writing, you may find its similarity to the command-line reassuring.

We include here a few Bash script examples that make use of the main Bash constructs, namely regular expressions, tests, conditional statements, loops and functions. This should give you enough of an overview of Bash to be able to make a start on the `last_touch` script required for Question 7.

Regular Expressions

The normal regular expressions syntax, including `*` and `?`, can be used within a Bash script. Note that if a patten does not match any file then the result is the pattern itself, rather than an empty string. For example the code:

```
echo "all␣C␣files:" *.c
```

will output all of the files that end with the `.c` extension, or the string `*.c` if there are none. Note that `echo` sends its argument to the terminal, but all commands also send their output to the terminal by default. You can enclose a command in back-ticks, e.g. `files=‘*.c‘` to capture its output in a variable instead.

Tests

Bash comes with a number of tests that can be applied to data. As well as the standard logical operators: not !, and `&&`, or `||`, equals `=`, not equals =! and integer comparisons `-eq`, `-le`, `gt`, Bash also includes a number of more powerful tests such as:

- `-n` *operand* has non-zero length.
- `-z` *operand* has zero length.
- `-f` a file named *operand* exists.
- `-d` a directory named *operand* exists.
- `-r, -w, -x` a file named *operand* exists and has read/write/executable permission.
- `-nt, -ot` file *operand1* is newer/older than *operand2*.

Conditional Statements

Bash includes standard conditional (`if`) statements, for example:

```
if [ 1 -lt 2 ]
then
  echo "yes"
else
  echo "no"
fi
```

Each `if` statement must end with a matching `fi`, but the `else` branch is optional. The conditional test must be separated from the brackets `[ ]` by a space, or the test will evaluate incorrectly.

Bash also includes case statements, for exmaple:

```
filename="somewords.txt"
ext=${filename##*\.}

case "$ext" in
  c) echo "its a C file" ;;
  o) echo "its an assembly file" ;;
  txt) echo "its a text file" ;;
esac
```

The case statement selects the *first* pattern that matches the value of the case. Each case condition is terminated by a close bracket `)`, each case is terminated by double semi-colon `;;` and the entire case statement must end with a matching `esac`.

Loops

Bash has a number of looping constructs. For loops iterate over a list of space separated items, while loops repeat ”while” a given condition is true and until loops repeat ”until” the condition is true. Some examples of their use are:

```
for X in 1 2 3
do
  echo $X
done

count=5
while [ $count -gt 0 ]
do
  echo "beep..."
  count=$(( $count - 1 ))
done
echo "boom!"

flag="unset"
until [ $flag == "set" ]
do
  echo "waiting..."
  flag="set"
done
echo "done"
```

Functions

Functions in Bash are simply subroutines. They may be defined in one of two ways:

```
function name {
  code...
}

name(){
  code...
}
```

They are called in the same way as commands, that is, there are no brackets around the arguments and the arguments are passed to the function body in the variables `$1`, `$2`, etc. For example:

```
function cylonEcho {
  echo "$1. By your command!"
}

cylonEcho "Destroy all humans"
```

A Bash function’s definition must not be empty and must precede the first call to it, otherwise the script will throw an error.

Appendix C - Ruby Scripting

In contrast to Bash, Ruby is a relatively new programming language. At its heart, Ruby is a powerful dynamic, open source programming language with a focus on simplicity and productivity. It has an elegant syntax that is natural to read and easy to write. However, due to its simplicity, Ruby is often employed for script writing. In fact, much of the lab infrastructure you will be using this year has been written in Ruby.

We include here a few Ruby script examples that make use of the main Ruby constructs, namely arrays, hashes, conditional staementes, loops, functions and classes. This should give you enough of an overview of Ruby to be able to make a start on the `last_touch` script required for Question 7.

Arrays

Declaring an array in Ruby is as easy as declaring a list in Haskell. However, Ruby provides a number of built in functions for arrays, such as `length`, and an inbuilt iteration operator `each`.

```
array = [1,2,3,4,5,6]
puts array.length
puts array[0]

array.each { |elem| puts "found #{elem} in the array" }

array.each do |elem|
  puts "found #{elem} in the array"
end
```

The use of the `each` operator in the above code snippets demonstrates two ways of writing “blocks” in Ruby. A block is a new scope with some scope-local variables assigned in each iteration of the block. In the above example, the `elem` variable gets assigned to each element of the array in turn.

The `puts` command prints to the terminal. To call system commands in Ruby, you either need to enclose them in backticks, as in Bash, or use the `system(...)` or `%x()` commands.

Hashes

Ruby has inbuil support for hashes, which are effectively sets of key to value pairs. These also have a very natural declaration syntax and inbuilt functionality, for example:

```
marks = { "alice" => 5, "bob" => 5, "eve" => 10 }
puts marks.length
puts marks["bob"]

marks.each {|name,score| puts "#{name} scored #{score}/10"}
marks.sort.reverse.each {|name,score| puts "#{name} scored #{score}/10"}
```

Note that the order of itteration through a hash is *not* gauranteed to be the same as the order of insertion into the hash.

Conditional Statements

Ruby has conditional statements that are very similar to other modern langauges, including `if` statements, `unless` statements and `case` statements. Some examples of their use are:

```
x = 1
if x > 2 then
  puts "x is bigger than 2"
elsif x < 2 and x != 0
  puts "I think x is 1"
else
  puts "I give up"
end
```

```
unless x == 1
  puts "This is just too hard"
else
  puts "I knew it was 1!"
end

case x
when 1
  puts "Is it still 1?!"
when -1
  puts "negative, that's cheating!"
when 2..10
  puts "I don't know, you win."
end
```

Note that the `elsif` and `else` branhes of an `if` statement are optional, as is the `else` branch of an `until` statement.

Loops

Again, Ruby's looping constructs are what you would expect to see in any modern language. We give a few examples below:

```
x = 1
while x < 3 do
  puts "#{x} is not magical"
  x += 1
end
puts "#{x} is the magic number!"

until x == 0 do
  x -= 1
end

for i in 0..10
  puts i
end
```

Functions

Functions or methods in Ruby can be defined and called as follows:

```
def mult(x,y)
  product = x * y
  return product
end

puts mult(3,5)
```

As with Bash, it is necessary to define a function/method before it is called. However, by default, Ruby returns the last expression it evaluated, so the above code could be rewritten as:

```
def mult(x,y)
  x * y
end

puts mult(3,5)
```

Classes

As an object oriented programming language, Ruby lets you define classes. The classes can have attributes and methods, as you would expect. For example:

```
Class Person
  def initialize(name, age)
    @name = name
    @age = age
  end

  def greeting
    puts "Hello, my name is #{@name}"
  end

bob = Person.new("Bob", 37)
bob.greeting
```

Note that the `initialize` method is use to set local variables `@name` and `@age` within the class. These variables can be accessed within the class's methods by using `name#@` as in the `greeting` method.