SOFTENG 370 2021
Operating Systems

Assignment 3 - ELF & Memory

Worth 10%

due date 11:59pm Friday the 28th of May

Part 1

This part of the assignment is concerned with compiled executable files and how they are loaded into memory. There are many details involved in how this happens we are only going to look at some of them.

Compile the `say_hello.c` program.

```
cc say_hello.c -o say_hello
```

Run it to make sure everything is ok.

```
./say_hello
```

For all of these instructions or questions **show the relevant output** you get when you run the commands. If there is a lot of output you should include the sections you need to answer the questions.

Programs compiled on modern Linux systems place executable and shared code into ELF files (Executable and Linkable Format). ELF files have a strict structure.

The first part of an ELF file is the header.

```
readelf -h say_hello
```

1.a) What is the architecture the file is compiled to run on? [0.5]

1.b) What is the type? Why is the type not EXEC (executable - see man ELF)? [1]

The next parts are the segments.

```
readelf -l say_hello
```

1.c) Using one sentence for each type, describe the different segment types you see. You do not need to describe the GNU extensions (starting with GNU). [2]

1.d) From the output above or otherwise what is the interpreter in the ELF header? [0.5]

1.e) Given that the program is compiled why does it have an interpreter? [1]

1.f) In which of the program segments do you think the `main` function is stored? Give a reason. [1]

Sections are also important parts of an ELF file.

```
readelf -SW say_hello
```

We will return to the sections in a later question. For the moment examine the output. You don't need to include this in your submission.

1.g) Describe the main differences between how program segment headers and section headers in ELF files are used. Include any web links you used to find your answer. [2]

Now to look at the code stored in our ELF file.

For this section you are not expected to completely understand the assembly language code. Here are just a few things to help. The `LEA` instruction loads an address into a register or memory

location. The `RDI` register and other registers are used to pass integer parameters to functions when they are called.

e.g. `lea 0xe6(%rip),%rdi`

This works out the address by adding `E6` to the current value of the instruction pointer (`RIP` - like the PC in other instruction sets). The `RIP` will always hold the address of the next instruction following this one. So if the next address is `654`, the address stored by `LEA` in `RDI` will be `654 + E6 = 73A`.

`CALLQ` is a function call. Because we want position independent code the destination address of all calls are relative to `RIP`.

The `objdump` command allows us to inspect the code in the ELF file.

```
objdump -d say_hello
```

This disassembles the code inside the ELF file.

2.a) In the header information in question 1 one of the values is the entry point address for the file. Locate the disassembled instructions from the relevant section in the `objdump` output. Include them in your answer. [0.5]

2.b) In a general way (you don't have to give details) explain how the `main` function is reached from this code. [2]

2.c) Show the assembly language which corresponds to the `main` function. [0.5]

2.d) When the `main` function returns, where does it return to? (In general, not the address.) [1]

Now look at the memory associated with the program as it runs. Start the program but leave it waiting for your input (and in another terminal get the process id via the `ps a` command). Process information is stored in the `/proc` directory.

If the process id is 18020 all the process information can be found in the `/proc/18020` directory.

3.a) Find the regions of memory mapped to this process. Show the output. [0.5]

```
cat /proc/18020/maps
```

3.b) See the files behind them. Show the output. [0.5]

```
ls -l /proc/18020/map_files/
```

3.c) Comparing this information with information from the ELF file, where in memory was the `main` function when the program was running? Explain how you worked that out. [2]

Possible references:

https://en.wikipedia.org/wiki/Executable_and_Linkable_Format

https://www.cs.stevens.edu/%7Ejschauma/631/elf.html

https://lwn.net/Articles/631631/

Further information can be gathered from the `man` pages on `readelf`, and `objdump`.

Also read the `/proc/[pid]/maps` section in `man proc`.

Part 2

4. [4]

We can keep track of free memory in a similar way to free blocks on disk. We could use a bitmap where 1 bit represents a free frame of memory or we could maintain a linked list of free frames. The linked list is maintained as a list of nodes where each node has a frame number, and a pointer to the next node. Assume these pointers and numbers are 64 bits each.

Calculate the space requirements using each of those methods for the following system, give the answers in bytes:

32 GB (2^{35} bytes) of RAM in 8 KB (2^{13} bytes) frames. Assume that memory is currently being used in alternating chunks of size 1 MB (2^{20} bytes). i.e. The first MB of RAM is used, the second is free, the third is used, etc.

How much of that space (for both approaches) would normally be in kernel memory? Why?

How would an extents version compare to the two approaches in the question above?

5. [2]

The number of instructions executed between page faults is directly proportional to the number of page frames allocated to a program. Double the amount of memory allocated to a process and you double the length of time between page faults. Suppose that a normal instruction takes 1 nanosecond, but if a page fault occurs, it takes 1,000,000 nanoseconds (i.e. 1msec) to handle the fault. If a program which is allocated *n* page frames takes 100 seconds to run, during which time it gets 50,000 page faults, how many page frames should be allocated to the program in order to have it finish in 60 seconds? Explain your working.

6. [4]

Given a machine with 5 frames show the contents of each frame using the following replacement algorithms on the reference string. **Also say how many page faults would occur for each algorithm.** Pages are brought in on demand and the initial load of a page counts as a page fault.

a) FIFO - First In First Out

b) LRU - Least Recently Used

c) LFU - Least Frequently Used (if there are multiple pages with the same lowest frequency choose in a FIFO manner)

d) Optimal (if there are multiple pages which are not used again choose in a FIFO manner)

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

Lay out your answer for each algorithm like this. A zero means the frame is free, an empty cell means the content of the frame is the same as in the previous step.

Submitting the assignment

Put all of the output requested and the answers to questions in a single file for upload via Canvas. The file must be readable to TurnItIn e.g. not an image (otherwise you may get zero for the assignment).

Make sure your name and upi is included in the file you submit.

Any work you submit must be your work and your work alone – see the School and University policies on academic integrity.