# 370 Assignment 3 Answer

elee353, 840454023

## 1.1.

The program header defines segments in the address space of a process running that ELF executable file. The segments are projected in virtual memory at execution time. The program header tells the kernel or the runtime linker what to load into memory and how to find dynamic linking information.

On the other hand, the sections header defines sections for compile-time linking. Each section belongs to a segment. The sections may or may not be visible, i.e. mapped into memory at execution time. The sections header tells the link editor how to resolve symbols, and how to group similar byte streams from different elf binary objects.

## 1.2.

```
elee353@en-368402:~/Desktop/370_A3$ ./hello
```

```
Hello elee353!
00400000-00401000 r-xp 00000000 00:2c 1047432194
/afs/ec.auckland.ac.nz/users/e/l/elee353/unixhome/Desktop/370_A3/hello
00600000-00601000 r--p 00000000 00:2c 1047432194
/afs/ec.auckland.ac.nz/users/e/l/elee353/unixhome/Desktop/370_A3/hello
00601000-00602000 rw-p 00001000 00:2c 1047432194
/afs/ec.auckland.ac.nz/users/e/l/elee353/unixhome/Desktop/370_A3/hello
014bc000-014dd000 rw-p 00000000 00:00 0                  [heap]
7f8bd3d66000-7f8bd3f26000 r-xp 00000000 103:05 918115    /lib/x86_64-linux-gnu/libc-2.23.so
7f8bd3f26000-7f8bd4126000 ---p 001c0000 103:05 918115    /lib/x86_64-linux-gnu/libc-2.23.so
7f8bd4126000-7f8bd412a000 r--p 001c0000 103:05 918115    /lib/x86_64-linux-gnu/libc-2.23.so
7f8bd412a000-7f8bd412c000 rw-p 001c4000 103:05 918115    /lib/x86_64-linux-gnu/libc-2.23.so
7f8bd412c000-7f8bd4130000 rw-p 00000000 00:00 0
7f8bd4130000-7f8bd4156000 r-xp 00000000 103:05 918020    /lib/x86_64-linux-gnu/ld-2.23.so
7f8bd4300000-7f8bd4303000 rw-p 00000000 00:00 0
7f8bd4353000-7f8bd4355000 rw-p 00000000 00:00 0
7f8bd4355000-7f8bd4356000 r--p 00025000 103:05 918020    /lib/x86_64-linux-gnu/ld-2.23.so
7f8bd4356000-7f8bd4357000 rw-p 00026000 103:05 918020    /lib/x86_64-linux-gnu/ld-2.23.so
7f8bd4357000-7f8bd4358000 rw-p 00000000 00:00 0
7ffd0649a000-7ffd064bb000 rw-p 00000000 00:00 0          [stack]
7ffd064e7000-7ffd064e9000 r--p 00000000 00:00 0          [vvar]
7ffd064e9000-7ffd064eb000 r-xp 00000000 00:00 0          [vdso]
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0  [vsyscall]
0 1 2 3 4 5 6 7 8 9 Goodbye world.
```

## 1.3.

```
elee353@en-368402:~/Desktop/370_A3$ readelf -h hello
```

```
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF64
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           Advanced Micro Devices X86-64
  Version:                           0x1
  Entry point address:               0x4005b0
  Start of program headers:          64 (bytes into file)
  Start of section headers:          6976 (bytes into file)
  Flags:                             0x0
  Size of this header:               64 (bytes)
  Size of program headers:           56 (bytes)
  Number of program headers:         9
  Size of section headers:           64 (bytes)
  Number of section headers:         31
  Section header string table index: 28
```

An ELF file for an executable program contains a program header table near the start of the file, after the ELF header; each entry in this table provides information that is needed to run the program.

The ELF magic identifies the file as an ELF executable.
The class field identifies the file is in 64-bit format.
The data field identifies the file should be interpreted using little endianness.
The version field is set to 1. This means this file is the original version of ELF.

The OS/ABI field Identifies the target operating system ABI for this file. This is UNIX System V.
The ABI version of 0 indicates System V.
The type field indicates this object is executable.

The machine field specifies the required architecture for this field. ‘Advanced Micro Devices X86-64’ is the intended architecture. This architecture is the 64-bit version of the x86 instruction set.

## 1.4.

```
elee353@en-368402:~/Desktop/370_A3$ nm hello
```

```
000000000060106f B __bss_start
0000000000601060 D bye
0000000000601070 b completed.7585
0000000000601050 D __data_start
0000000000601050 W data_start
00000000004005e0 t deregister_tm_clones
```

```
0000000000400660 t __do_global_dtors_aux
0000000000600e18 t __do_global_dtors_aux_fini_array_entry
0000000000601058 D __dso_handle
0000000000600e28 d _DYNAMIC
000000000060106f D _edata
0000000000601078 B _end
00000000004007d4 T _fini
0000000000400680 t frame_dummy
0000000000600e10 t __frame_dummy_init_array_entry
0000000000400958 r __FRAME_END__
                 U getpid@@GLIBC_2.2.5
0000000000601000 d _GLOBAL_OFFSET_TABLE_
                 w __gmon_start__
000000000040080c r __GNU_EH_FRAME_HDR
0000000000400500 T _init
0000000000600e18 t __init_array_end
0000000000600e10 t __init_array_start
00000000004007e0 R _IO_stdin_used
                 w _ITM_deregisterTMCloneTable
                 w _ITM_registerTMCloneTable
0000000000600e20 d __JCR_END__
0000000000600e20 d __JCR_LIST__
                 w _Jv_RegisterClasses
00000000004007d0 T __libc_csu_fini
0000000000400760 T __libc_csu_init
                 U __libc_start_main@@GLIBC_2.2.5
00000000004006d8 T main
00000000004006a6 T numbers
                 U printf@@GLIBC_2.2.5
                 U puts@@GLIBC_2.2.5
0000000000400620 t register_tm_clones
                 U sprintf@@GLIBC_2.2.5
                 U __stack_chk_fail@@GLIBC_2.4
00000000004005b0 T _start
                 U system@@GLIBC_2.2.5
0000000000601070 D __TMC_END__
0000000000601074 B unused
```

| | |
|---|---|
| 0000000000601060 D bye | -D The char array bye is in the initialized data section.<br>This is at address 0000000000601060. |
| U getpid@@GLIBC_2.2.5 | -U The getpid function is undefined. This may be caused by it being part of a library. GNU Glibc version 2.2.5 is used. |
| 00000000004006d8 T main | -T The main function is in the text (code) section.<br>This is at address 00000000004006d8. |
| 00000000004006a6 T numbers | -T The numbers function is in the text (code) section. |

| | |
|---|---|
| | This is at address 00000000004006a6. |
| U printf@@GLIBC_2.2.5 | -U The printf function is undefined. This may be caused by it being part of a library. GNU Glibc version 2.2.5 is used. |
| U puts@@GLIBC_2.2.5 | -U The puts function is undefined. This may be caused by it being part of a library. GNU Glibc version 2.2.5 is used. |
| U sprintf@@GLIBC_2.2.5 | -U The sprintf function is undefined. This may be caused by it being part of a library. GNU Glibc version 2.2.5 is used. |
| U system@@GLIBC_2.2.5 | -U The system function is undefined. This may be caused by it being part of a library. GNU Glibc version 2.2.5 is used. |
| 0000000000601074 B unused | -B The unused variable is in the uninitialized data section (known as BSS). This is at address 0000000000601074. |

The undefined symbols are in libraries GNU Glibc version 2.2.5 and version 2.4.

An automatic variable such as ‘int i’ is created on the stack at runtime each time the program enters the block in which it is defined. The automatic variable then ceases to exist and leaves that block. Such a variable occupies no storage in the executable so it is not represented in the symbol table. It cannot be seen by the linker and lives on process’s stack.

## 1.5.

The .text section is a place where the compiler puts executable instructions. The .rodata section contains static constants rather than variables. This section is generally read-only and fixed size. As the consequence, this section is marked as executable with "X" onFlg field.

The .data section holds all the initialised variables inside a c program which doesn't live inside the stack. "Initialised" here means it is given an initial value. The data segment is read-write, since the values of variables can be altered at run time. The values for these uninitialised variables are initially stored within the read-only memory, typically within the .text section. They are then copied into the .data segment during the start-up routine of the program.

The .bss section (Block Started by Symbol) is a section where all uninitialised variables are mapped. This is a read-write memory segment

1.6.

```
elee353@en-368402:~/Desktop/370_A3$ readelf -lW hello
```

```
Elf file type is EXEC (Executable file)
Entry point 0x4005b0
There are 9 program headers, starting at offset 64

Program Headers:
  Type           Offset   VirtAddr           PhysAddr           FileSiz  MemSiz   Flg Align
  PHDR           0x000040 0x0000000000400040 0x0000000000400040 0x0001f8 0x0001f8 R E 0x8
  INTERP         0x000238 0x0000000000400238 0x0000000000400238 0x00001c 0x00001c R   0x1
      [Requesting program interpreter: /lib64/ld-linux-x86-64.so.2]
  LOAD           0x000000 0x0000000000400000 0x0000000000400000 0x00095c 0x00095c R E 0x200000
  LOAD           0x000e10 0x0000000000600e10 0x0000000000600e10 0x00025f 0x000268 RW  0x200000
  DYNAMIC        0x000e28 0x0000000000600e28 0x0000000000600e28 0x0001d0 0x0001d0 RW  0x8
  NOTE           0x000254 0x0000000000400254 0x0000000000400254 0x000044 0x000044 R   0x4
  GNU_EH_FRAME   0x00080c 0x000000000040080c 0x000000000040080c 0x00003c 0x00003c R   0x4
  GNU_STACK      0x000000 0x0000000000000000 0x0000000000000000 0x000000 0x000000 RW  0x10
  GNU_RELRO      0x000e10 0x0000000000600e10 0x0000000000600e10 0x0001f0 0x0001f0 R   0x1

 Section to Segment mapping:
  Segment Sections...
   00
   01     .interp
   02     .interp .note.ABI-tag .note.gnu.build-id .gnu.hash .dynsym .dynstr .gnu.version .gnu.version_r .rela.dyn
.rela.plt .init .plt .plt.got .text .fini .rodata .eh_frame_hdr .eh_frame
   03     .init_array .fini_array .jcr .dynamic .got .got.plt .data .bss
   04     .dynamic
   05     .note.ABI-tag .note.gnu.build-id
   06     .eh_frame_hdr
   07
   08     .init_array .fini_array .jcr .dynamic .got
```

The last 3 hexadecimal digits of the OffSet match the last 3 hexadecimal digits of the virtual address. The more significant hexadecimal digits excluding the last 3 hexadecimal digits are masked when the virtual address is calculated. This makes them fit into a predetermined address space.

1.7.

| seg | type | Virtual address | size | Access (flg) | Map area (range of addresses) | file offset | mem Access |
|---|---|---|---|---|---|---|---|
| 2 | LOAD | 0x0000000000400000 | 0x00095c | R E | 0040000-00401000 | 0x000000 | r-xp |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 3 | LOAD | 0x00000000000600e10 | 0x000268 | RW | 00600000-00601000 | 0x000e10 | r--p |

1.8.

The .text section has flags AX. This means it has execute and allocate access rights. It has r-xp access permissions. This indicates read and execute permissions. The ‘p’ indicates copy on write.

The .rodata section has flags A. This means it has allocate access right. It has r--p access permissions. This indicates read permissions.

The .data section has flags WA. This means it has write and allocate access rights. It has rw-p access permissions. This indicates read and write permissions.

The .bss section has flags WA. This means it has write and allocate access rights. It has rw-p access permissions. This indicates read and write permissions.

# Part 2

## 2.1.

The x86 architecture CPUs are backward-compatible with 32-bit code, for example, Intel i7 CPUs.

A 64-bit program can access more memory than its 32-bit counterpart. A 64-bit environment takes twice as much memory compared to a 32-bit environment. This increases application startup time and shutdown time. In addition, most 64-bit processors can run and virtualise 32-bit programs.

The 32-bit has a limitation for RAM access. This means as soon as a program needs more working memory it has to swap out information in temporary free space created on the hard drive. The temporary free space is referred to as the swap file.

Moving from a 32-bit environment to a 64-bit environment have consequences on programs. The consequences are listed below.

64-bit Libraries
Using 32-bit libraries in 64-bit applications requiring marshalling data between the two environments. That slows down application speed considerably and introduces potential errors.

Data Access
Data issues include both data storage on disk, as well as in memory. Some data structures that work fine in a 32-bit environment may not work in a 64-bit environment. The issue can

be caused by the way the 64-bit environment packs the data into the structure. The data structure uses a different alignment in 64-bits. Nevertheless, even if all of the data elements are the same size, they are not in the same location in memory. Other problems include pointer size; a 64-bit pointer is twice the size of a 32-bit pointer.

Operating System Feature Access
For example, On the 64-bit version of the operating system, 32-bit applications can suddenly find their data moved and 64-bit applications may find it difficult to locate data generated by 32-bit counterparts.

Math
Converting from 32-bits to 64-bits may introduce mathematical operation issues. This can be caused by sign extension when a value is converted to a signed or unsigned value.

Hardware Capacity and Supporting Hardware
A 64-bit application uses data elements that are twice as wide as a 32-bit application. Some hardware simply won't support 64-bit access. This is especially true for older hardware.

## 2.2. [4 marks]

Linked List:

Unused RAM due to alternating chunks = 16 GB / 2 = 2^33 bytes
Number of frames = 2^33 bytes / 2^13 (byte/frame) = 2^20 frames

Space = 2^20 frames * 32 bits * 2 (number and pointer) = 2^26 bits = 2^23 bytes = 8 MB

Bitmap:

All RAMs are used = 16Gb = 2^34 bytes
Number of frames = 2^34 bytes / 2^13 (byte/frame) = 2^21 frames

Bitmap space used = 2^21 * 1 bit = 2 ^ 21 bits = 2^18 bytes = 262 kB

For the linked list and the bitMap implementations, 100% of the space shown above is stored in kernel memory. This is solely managed by the kernel because managing free and used memory blocks is a vital functionality. This ensures security. If users can modify free blocks at will, they may be able to corrupt the system. The user may accidentally override memory.

Extent:
For the extent implementation, three 32-bit numbers are required to store 2 addresses and the block size. The three numbers are a location, a block count, and a link to the first block of the next extent.

3 numbers * 4 bytes = 12 bytes
8 Gb / 1Mb = 8192 total extents
Space = 12 bytes * 8192 extents = 98 kB = 98304 bytes

This implementation is very efficient because of the 1 Mb alternating chunks of RAM. This leads to many consecutive empty frames and consequently a small number of total extents. In comparison, the linked list implementation uses 8 Mb to represent each free frame.

In comparison to the linked list implementation, 1 extend is equivalent to 128 linked list nodes. The nodes are calculated by 1 megabyte / 8 kilobytes = 128 (nodes).
In comparison to the bitMap implementation, 1 extend is equivalent to 256 bitMap bits. Hence the extent implementation results in significant space saving.

## 2.3.

1024 processes = 2^10 processes

2^32 byte * 2^10 = 2^42 bytes = 4 TB swap space
This is unrealistic because we don't normally have 4 TB to spare for swap space.

## 2.4.

Assuming the 0.01 % page fault is of the overall time.

We have 99% TLB hit, 0.01% page fault, and 0.99% page table hit.

EAT = 99% TLB hit + 0.01% page fault + and 0.99% page table hit
= ( TLB 1 nanosec + memory access 50 nanosec) * 99%
+ 0.01% *( TLB 1 nanosec + page table access 100 nanosec + page replacement 5 milsec + memory access 50 nanosec)
+ 0.99% * (TLB 1 nanosec + page table access 100 nanosec + memory access 50 nanosec)
= 50.49 nanosec + 500.0151 nanosec + 1.4949 nanosec
= 552 nanosec

## 2.5.

Page entry in each table = 2^22 byte pages / 2^6 bytes = 2^16 entries

So 16 bits address space is needed.

Single level page entries = 2^64 addressable bytes / 2^22 bytes per page = 2^42 page table entries

ceiling(42/16) = 3 levels needed

The minimum number of pages needed for our process would be 5 pages. We know the program and data page is the lowest page, and the stack page is the highest page.

At the first level page table, 1 page would be required to store a reference to 2 page tables in the second level. At second level page table, 2 pages are needed to store page reference to the program and data page and the stack page in the third level. At the third level page

tablet, 2 pages are needed to store page reference to the program and data page and the stack page in disk memory.

## 2.6.

- Double the size of memory, half the page faults
- 60 sec to run the program including page faults
- Time to run program excluding page faults
  = 60 sec - 2 msec * 20,000 page faults = 20 sec
- 10000 page faults take 20000 msec
- Total time with more memory = 20 sec + 20 sec = 40 sec

## 2.7.

a) FIFO - First In First Out

9 page faults.

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

b) LRU - Least Recently Used

10 page faults

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

c) LFU - Least Frequently Used (if there are multiple pages with the same lowest frequency

choose in a FIFO manner)

11 page faults

| | 1 | 2 | 3 | 4 | 3 | 5 | 6 | 7 | 6 | 5 | 4 | 3 | 4 | 7 | 6 | 1 | 5 | 4 | 1 | 2 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

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

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

9 page faults

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