OBL1 - Operating Systems

By: John Ivar Eriksen

1 - The process abstraction

1.1

> Briefly describe what happens when a process is started from a program on disk. A mode switch from kernel- to user-mode must happen. Explain why this is necessary.

The first thing that happens is that the operating system (OS) initialize a process controll block (PCB) to represent this new process.

Next, the OS allocates memory for the new process, and the proceeds to load the program into the computer's memory (RAM).

The OS then allocates a user-level stack for user-level executions, and a kernel level stack for system calls, interrups and processor exceptions.

The mode switch from kernel- to user mode must happen in order to isolate the process from having full kernel access. This has to do with permissions and privileges of processes. If a process was allowed to run in kernel mode, it would have access to take control of the processes used to control and allocate system resources such as memory, CPU time and execute privileged instructions. This would enable manipulation of hardware and reading of memory, which in turn could be used by either malicous, or buggy, software in some detrimental way. E.g. being able to read memory would enable the reading of sensitive information such as passwords and encryption keys, or taking up all system resources for itself.

1.2

The struct `task_struct` is defined in `include\linux\sched.h` header file, in line `738` as per kernel version `linux-6.5.2`, 2023-09-08 (subject to change, i.e. varies between kernel versions).

a) The field name from this struct that stores the process ID:

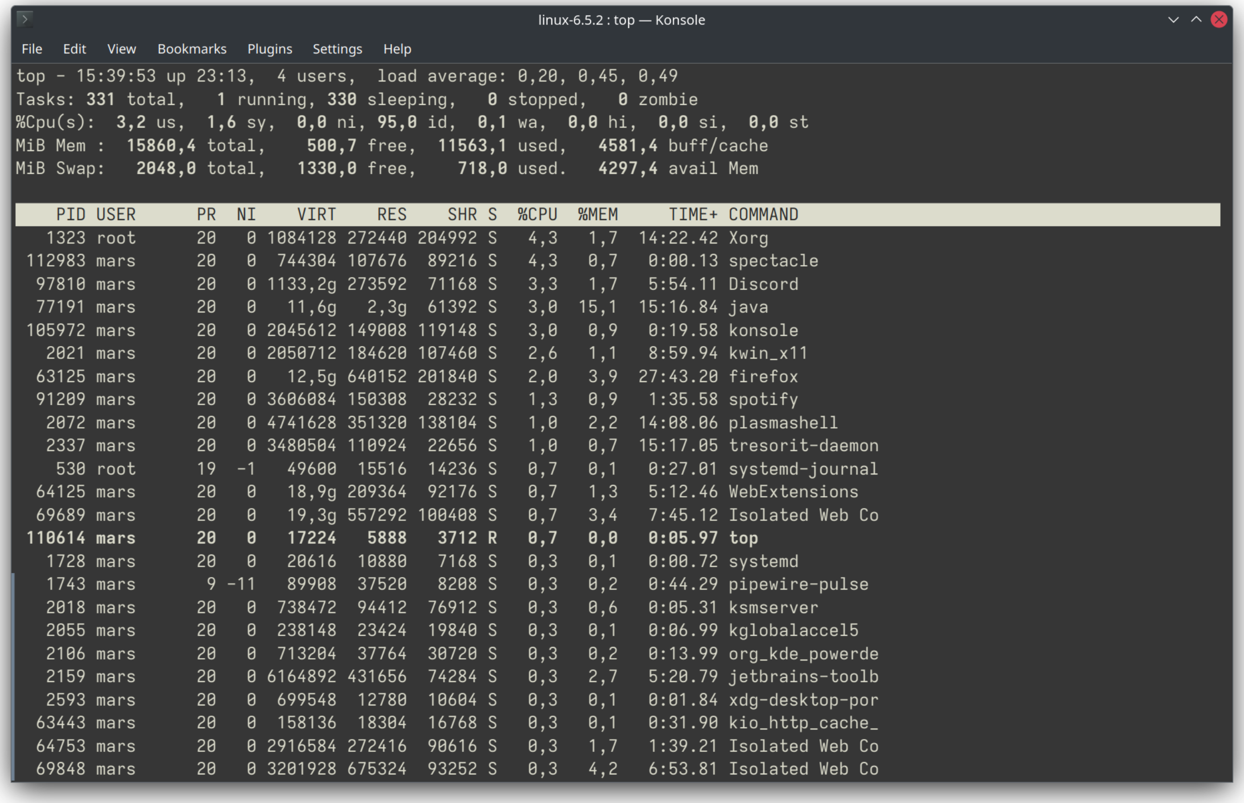
The fields named `pid` are responsible for storing process ID, which is what the acronym stands for (**p**process **id**).
There are many entries related `pid` in the header file.

b) The field name from this struct that keeps track of accumulated virtual memory:

The fields named "mm" are responsible for the virtual memory. The name ("mm") is derived from the term "memory management".

C) Name two other fields found via `top`.

Looking at the display output of `top`, I decided on the fields `PR` and `NI`, i.e. 3rd and 4th column.



```
linux-6.5.2 : top — Konsole
File  Edit  View  Bookmarks  Plugins  Settings  Help
top - 15:39:53 up 23:13,  4 users,  load average: 0,20, 0,45, 0,49
Tasks: 331 total,   1 running, 330 sleeping,   0 stopped,   0 zombie
%Cpu(s):  3,2 us,  1,6 sy,  0,0 ni, 95,0 id,  0,1 wa,  0,0 hi,  0,0 si,  0,0 st
MiB Mem :  15860,4 total,    500,7 free,  11563,1 used,   4581,4 buff/cache
MiB Swap:   2048,0 total,   1330,0 free,    718,0 used.   4297,4 avail Mem

    PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
   1323 root      20   0 1084128 272440 204992 S   4,3   1,7  14:22.42 Xorg
 112983 mars      20   0  744304 107676  89216 S   4,3   0,7   0:00.13 spectacle
  97810 mars      20   0 1133,2g 273592  71168 S   3,3   1,7   5:54.11 Discord
  77191 mars      20   0   11,6g   2,3g  61392 S   3,0  15,1  15:16.84 java
 105972 mars      20   0 2045612 149008 119148 S   3,0   0,9   0:19.58 konsole
   2021 mars      20   0 2050712 184620 107460 S   2,6   1,1   8:59.94 kwin_x11
  63125 mars      20   0   12,5g 640152 201840 S   2,0   3,9  27:43.20 firefox
  91209 mars      20   0 3606084 150308  28232 S   1,3   0,9   1:35.58 spotify
   2072 mars      20   0 4741628 351320 138104 S   1,0   2,2  14:08.06 plasmashell
   2337 mars      20   0 3480504 110924  22656 S   1,0   0,7  15:17.05 tresorit-daemon
    530 root      19  -1   49600  15516  14236 S   0,7   0,1   0:27.01 systemd-journal
  64125 mars      20   0   18,9g 209364  92176 S   0,7   1,3   5:12.46 WebExtensions
  69689 mars      20   0   19,3g 557292 100408 S   0,7   3,4   7:45.12 Isolated Web Co
 110614 mars      20   0   17224   5888   3712 R   0,7   0,0   0:05.97 top
   1728 mars      20   0   20616  10880   7168 S   0,3   0,1   0:00.72 systemd
   1743 mars       9 -11   89908  37520   8208 S   0,3   0,2   0:44.29 pipewire-pulse
   2018 mars      20   0  738472  94412  76912 S   0,3   0,6   0:05.31 ksmserver
   2055 mars      20   0  238148  23424  19840 S   0,3   0,1   0:06.99 kglobalaccel5
   2106 mars      20   0  713204  37764  30720 S   0,3   0,2   0:13.99 org_kde_powerde
   2159 mars      20   0 6164892 431656  74284 S   0,3   2,7   5:20.79 jetbrains-toolb
   2593 mars      20   0  699548  12780  10604 S   0,3   0,1   0:01.84 xdg-desktop-por
  63443 mars      20   0  158136  18304  16768 S   0,3   0,1   0:31.90 kio_http_cache_
  64753 mars      20   0 2916584 272416  90616 S   0,3   1,7   1:39.21 Isolated Web Co
  69848 mars      20   0 3201928 675324  93252 S   0,3   4,2   6:53.81 Isolated Web Co
```

1. `PR` (Priority)
   - `PR` refers to "priority", and is found under the field name `static_prio`.
   - `static_prio` represents the initial priority assigned to the process by the kernel, and is the priority "in the moment" from the point of view of the task scheduler.
2. `NI` (Nice Value)
   - `NI` refers to "nice value", and is found under the field name `normal_prio`.
   - This value acts as a suggestion to the kernel of what priority the process "should" have, and acts to influence the priority given by the kernel itself.
   - The priority of a given process is usually illustrated, and simplified, as the sum `PR = 20 + NI`. The lower the number, the higher the priority.
   - However, the kernel can change the PR value regardless of NI value (but not the NI value itself) if needed.

2 - Process memory and segments

> The memory region allocated to a process contains the following segments.
> - Text segment
> - Data segment
> - Stack
> - Heap

2.1

> 1. Sketch the organisation of a process' address space. Start with high addresses at the top, and the lowest address (0x0) at the bottom.

The address space of a process is organized as follows:

```
High addresses, 0xFFFFFFFF
-----------------------------
|Stack                      |
|---------------------------|
|Random address offset      |
|---------------------------|
|Heap (grows up)            |
|---------------------------|
|Data                       |
|---------------------------|
|Text                       |
-----------------------------
Lowest addresses, 0x0
```

The layout contains some more segments, but these are the highlights. There's also some random address offsets for the Stack and Heap to ensure that they aren't found in easily predicable locations.

https://manybutfinite.com/post/anatomy-of-a-program-in-memory/

2.2

> 2. Briefly describe the purpose of each segment. Why is address 0x0 unavailable to the process?

Text segment

- Store binary image of the process, i.e. the executable code.

Data segment

- Global and static variables initialized by the program.

Stack

- Contains data that we know are going to be used.
- Local variables.
- Function call management and information. E.g. return addresses.

Heap

- Contains data that may be dynamically allocated while the program is running.
- Variables that may or may not be used.

The address 0x0 is unavailable to the process because it's reserved to act as a **null pointer**. E.g. when a process is trying to request memory, say 1 MB, but there's not enough available, the program will be sent the 0x0 address instead. This in turn is interpreted as "null pointer", and instead of causing a crash or other critical errors, the program simply receives this null pointer and can then throw a more begning error message.

2.3.1

> 3. What are the differences between a global, static, and local variable?

Global variable

- Declared outside of any other functions/methods.
- A global "scope", i.e. not limited to any particular function, and accessible and modifiable by any functions.
- Automatically initialized to `0` when they are declared.

Static variable

- Scope is limited to the function/method, or block, in which it is declared.
- Retains it's value between function calls.
- Only initialized once.
- Automatically initialized to `0`, unless initialized to something else.

Local variable

- Scope is limited to the method/function, or block, where it is defined.
- Available only inside the functions in which they are defined.
- Only exists until the function is executed. Once finished, the variable(s) are destroyed.

2.3.2

> Given the following code snippet, show which segment each of the variables (var1, var2, var3) belong to.

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

int var1 = 0;
void main()
{
   int var2 = 1;
   int *var3 = (int *)malloc(sizeof(int));
   // Note, since we are using malloc(), var3 will be a
   // pointer into the heap!
   // So the question is, where is the pointer stored?
   *var3 = 2;
   printf("Address: %x; Value: %d\n", &var1, var1);
   printf("Address: %x; Value: %d\n", &var2, var2);
```

```
    printf("Address: %x; Address: %x; Value: %d\n", &var3, var3, *var3);
}
```

Output:

```
Address: 19f63014; Value: 0
Address: c0a8a2fc; Value: 1
Address: c0a8a300; "Address : 1a9592a0; Value: 2
```

var1

- Global variable; declared outside any functions.
- Typically stored in `data` segment.

var2

- Local variable; declared inside the `main` function.
- Typically stored on the `stack` segment.

var3

- Dynamically allocated variable, using `malloc` (**m**emory **alloc**ation).
- Stored on the `head` segment.

3 - Program code

3.1

> 1. Compile the example given above using gcc mem.c -o mem. Determine the sizes of the text, data,
> and bss segments using the command-line tool size.

Output of `size mem`:

```
text     data     bss     dec     hex filename
1801      616       8    2425     979 mem
```

3.2

> 2. Find the start address of the program using objdump -f mem.

Output of `objdump -f mem`:

```
mem:     file format elf64-x86-64
architecture: i386:x86-64, flags 0x00000150:
HAS_SYMS, DYNAMIC, D_PAGED
start address 0x00000000000010a0
```

3.3

> 3. Disassemble the compiled program using objdump -d mem. Capture the output and find the name of the function at the start address. Do a web search to find out what this function does, and why it is useful.

Output of `objdump -d mem`:

```
mem:     file format elf64-x86-64


Disassembly of section .init:

0000000000001000 <_init>:
    1000:       f3 0f 1e fa             endbr64
    1004:       48 83 ec 08             sub    $0x8,%rsp
    1008:       48 8b 05 d9 2f 00 00    mov    0x2fd9(%rip),%rax        # 3fe8 <__gmon_start__@Base>
    100f:       48 85 c0                test   %rax,%rax
    1012:       74 02                   je     1016 <_init+0x16>
    1014:       ff d0                   call   *%rax
    1016:       48 83 c4 08             add    $0x8,%rsp
    101a:       c3                      ret

Disassembly of section .plt:

0000000000001020 <.plt>:
    1020:       ff 35 8a 2f 00 00       push   0x2f8a(%rip)        # 3fb0 <_GLOBAL_OFFSET_TABLE_+0x8>
    1026:       ff 25 8c 2f 00 00       jmp    *0x2f8c(%rip)        # 3fb8 <_GLOBAL_OFFSET_TABLE_+0x10>
    102c:       0f 1f 40 00             nopl   0x0(%rax)
    1030:       f3 0f 1e fa             endbr64
    1034:       68 00 00 00 00          push   $0x0
    1039:       e9 e2 ff ff ff          jmp    1020 <_init+0x20>
    103e:       66 90                   xchg   %ax,%ax
    1040:       f3 0f 1e fa             endbr64
    1044:       68 01 00 00 00          push   $0x1
    1049:       e9 d2 ff ff ff          jmp    1020 <_init+0x20>
    104e:       66 90                   xchg   %ax,%ax
    1050:       f3 0f 1e fa             endbr64
    1054:       68 02 00 00 00          push   $0x2
    1059:       e9 c2 ff ff ff          jmp    1020 <_init+0x20>
    105e:       66 90                   xchg   %ax,%ax

Disassembly of section .plt.got:

0000000000001060 <__cxa_finalize@plt>:
    1060:       f3 0f 1e fa             endbr64
    1064:       ff 25 8e 2f 00 00       jmp    *0x2f8e(%rip)        # 3ff8 <__cxa_finalize@GLIBC_2.2.5>
```

```
    106a:       66 0f 1f 44 00 00       nopw   0x0(%rax,%rax,1)

Disassembly of section .plt.sec:

0000000000001070 <__stack_chk_fail@plt>:
    1070:       f3 0f 1e fa             endbr64
    1074:       ff 25 46 2f 00 00       jmp    *0x2f46(%rip)        # 3fc0 <__stack_chk_fail@GLIBC_2.4>
    107a:       66 0f 1f 44 00 00       nopw   0x0(%rax,%rax,1)

0000000000001080 <printf@plt>:
    1080:       f3 0f 1e fa             endbr64
    1084:       ff 25 3e 2f 00 00       jmp    *0x2f3e(%rip)        # 3fc8 <printf@GLIBC_2.2.5>
    108a:       66 0f 1f 44 00 00       nopw   0x0(%rax,%rax,1)

0000000000001090 <malloc@plt>:
    1090:       f3 0f 1e fa             endbr64
    1094:       ff 25 36 2f 00 00       jmp    *0x2f36(%rip)        # 3fd0 <malloc@GLIBC_2.2.5>
    109a:       66 0f 1f 44 00 00       nopw   0x0(%rax,%rax,1)

Disassembly of section .text:

00000000000010a0 <_start>:
    10a0:       f3 0f 1e fa             endbr64
    10a4:       31 ed                   xor    %ebp,%ebp
    10a6:       49 89 d1                mov    %rdx,%r9
    10a9:       5e                      pop    %rsi
    10aa:       48 89 e2                mov    %rsp,%rdx
    10ad:       48 83 e4 f0             and    $0xfffffffffffffff0,%rsp
    10b1:       50                      push   %rax
    10b2:       54                      push   %rsp
    10b3:       45 31 c0                xor    %r8d,%r8d
    10b6:       31 c9                   xor    %ecx,%ecx
    10b8:       48 8d 3d ca 00 00 00    lea    0xca(%rip),%rdi        # 1189 <main>
    10bf:       ff 15 13 2f 00 00       call   *0x2f13(%rip)        # 3fd8 <__libc_start_main@GLIBC_2.34>
    10c5:       f4                      hlt
    10c6:       66 2e 0f 1f 84 00 00    cs nopw 0x0(%rax,%rax,1)
    10cd:       00 00 00

00000000000010d0 <deregister_tm_clones>:
    10d0:       48 8d 3d 39 2f 00 00    lea    0x2f39(%rip),%rdi        # 4010 <__TMC_END__>
    10d7:       48 8d 05 32 2f 00 00    lea    0x2f32(%rip),%rax        # 4010 <__TMC_END__>
    10de:       48 39 f8                cmp    %rdi,%rax
    10e1:       74 15                   je     10f8 <deregister_tm_clones+0x28>
```

```
    10e3:       48 8b 05 f6 2e 00 00    mov    0x2ef6(%rip),%rax        # 3fe0 <_ITM_deregisterTMCloneTable@Base>
    10ea:       48 85 c0                test   %rax,%rax
    10ed:       74 09                   je     10f8 <deregister_tm_clones+0x28>
    10ef:       ff e0                   jmp    *%rax
    10f1:       0f 1f 80 00 00 00 00    nopl   0x0(%rax)
    10f8:       c3                      ret
    10f9:       0f 1f 80 00 00 00 00    nopl   0x0(%rax)

0000000000001100 <register_tm_clones>:
    1100:       48 8d 3d 09 2f 00 00    lea    0x2f09(%rip),%rdi        # 4010 <__TMC_END__>
    1107:       48 8d 35 02 2f 00 00    lea    0x2f02(%rip),%rsi        # 4010 <__TMC_END__>
    110e:       48 29 fe                sub    %rdi,%rsi
    1111:       48 89 f0                mov    %rsi,%rax
    1114:       48 c1 ee 3f             shr    $0x3f,%rsi
    1118:       48 c1 f8 03             sar    $0x3,%rax
    111c:       48 01 c6                add    %rax,%rsi
    111f:       48 d1 fe                sar    %rsi
    1122:       74 14                   je     1138 <register_tm_clones+0x38>
    1124:       48 8b 05 c5 2e 00 00    mov    0x2ec5(%rip),%rax        # 3ff0 <_ITM_registerTMCloneTable@Base>
    112b:       48 85 c0                test   %rax,%rax
    112e:       74 08                   je     1138 <register_tm_clones+0x38>
    1130:       ff e0                   jmp    *%rax
    1132:       66 0f 1f 44 00 00       nopw   0x0(%rax,%rax,1)
    1138:       c3                      ret
    1139:       0f 1f 80 00 00 00 00    nopl   0x0(%rax)

0000000000001140 <__do_global_dtors_aux>:
    1140:       f3 0f 1e fa             endbr64
    1144:       80 3d c5 2e 00 00 00    cmpb   $0x0,0x2ec5(%rip)        # 4010 <__TMC_END__>
    114b:       75 2b                   jne    1178 <__do_global_dtors_aux+0x38>
    114d:       55                      push   %rbp
    114e:       48 83 3d a2 2e 00 00    cmpq   $0x0,0x2ea2(%rip)        # 3ff8 <__cxa_finalize@GLIBC_2.2.5>
    1155:       00
    1156:       48 89 e5                mov    %rsp,%rbp
    1159:       74 0c                   je     1167 <__do_global_dtors_aux+0x27>
    115b:       48 8b 3d a6 2e 00 00    mov    0x2ea6(%rip),%rdi        # 4008 <__dso_handle>
    1162:       e8 f9 fe ff ff          call   1060 <__cxa_finalize@plt>
    1167:       e8 64 ff ff ff          call   10d0 <deregister_tm_clones>
    116c:       c6 05 9d 2e 00 00 01    movb   $0x1,0x2e9d(%rip)        # 4010 <__TMC_END__>
    1173:       5d                      pop    %rbp
    1174:       c3                      ret
```

```
    1175:	0f 1f 00             	nopl   (%rax)
    1178:	c3                   	ret
    1179:	0f 1f 80 00 00 00 00 	nopl   0x0(%rax)

0000000000001180 <frame_dummy>:
    1180:	f3 0f 1e fa          	endbr64
    1184:	e9 77 ff ff ff       	jmp    1100 <register_tm_clones>

0000000000001189 <main>:
    1189:	f3 0f 1e fa          	endbr64
    118d:	55                   	push   %rbp
    118e:	48 89 e5             	mov    %rsp,%rbp
    1191:	48 83 ec 20          	sub    $0x20,%rsp
    1195:	64 48 8b 04 25 28 00 	mov    %fs:0x28,%rax
    119c:	00 00 
    119e:	48 89 45 f8          	mov    %rax,-0x8(%rbp)
    11a2:	31 c0                	xor    %eax,%eax
    11a4:	c7 45 ec 01 00 00 00 	movl   $0x1,-0x14(%rbp)
    11ab:	bf 04 00 00 00       	mov    $0x4,%edi
    11b0:	e8 db fe ff ff       	call   1090 <malloc@plt>
    11b5:	48 89 45 f0          	mov    %rax,-0x10(%rbp)
    11b9:	48 8b 45 f0          	mov    -0x10(%rbp),%rax
    11bd:	c7 00 02 00 00 00    	movl   $0x2,(%rax)
    11c3:	8b 05 4b 2e 00 00    	mov    0x2e4b(%rip),%eax        # 4014 <var1>
    11c9:	89 c2                	mov    %eax,%edx
    11cb:	48 8d 05 42 2e 00 00 	lea    0x2e42(%rip),%rax        # 4014 <var1>
    11d2:	48 89 c6             	mov    %rax,%rsi
    11d5:	48 8d 05 2c 0e 00 00 	lea    0xe2c(%rip),%rax        # 2008 <_IO_stdin_used+0x8>
    11dc:	48 89 c7             	mov    %rax,%rdi
    11df:	b8 00 00 00 00       	mov    $0x0,%eax
    11e4:	e8 97 fe ff ff       	call   1080 <printf@plt>
    11e9:	8b 55 ec             	mov    -0x14(%rbp),%edx
    11ec:	48 8d 45 ec          	lea    -0x14(%rbp),%rax
    11f0:	48 89 c6             	mov    %rax,%rsi
    11f3:	48 8d 05 0e 0e 00 00 	lea    0xe0e(%rip),%rax        # 2008 <_IO_stdin_used+0x8>
    11fa:	48 89 c7             	mov    %rax,%rdi
    11fd:	b8 00 00 00 00       	mov    $0x0,%eax
    1202:	e8 79 fe ff ff       	call   1080 <printf@plt>
    1207:	48 8b 45 f0          	mov    -0x10(%rbp),%rax
    120b:	8b 08                	mov    (%rax),%ecx
    120d:	48 8b 55 f0          	mov    -0x10(%rbp),%rdx
    1211:	48 8d 45 f0          	lea    -0x10(%rbp),%rax
    1215:	48 89 c6             	mov    %rax,%rsi
    1218:	48 8d 05 01 0e 00 00 	lea    0xe01(%rip),%rax        # 2020 <_IO_stdin_used+0x20>
    121f:	48 89 c7             	mov    %rax,%rdi
    1222:	b8 00 00 00 00       	mov    $0x0,%eax
    1227:	e8 54 fe ff ff       	call   1080 <printf@plt>
```

```
    122c:	90                   	nop
    122d:	48 8b 45 f8          	mov    -0x8(%rbp),%rax
    1231:	64 48 2b 04 25 28 00 	sub    %fs:0x28,%rax
    1238:	00 00 
    123a:	74 05                	je     1241 <main+0xb8>
    123c:	e8 2f fe ff ff       	call   1070 <__stack_chk_fail@plt>
    1241:	c9                   	leave  
    1242:	c3                   	ret    

Disassembly of section .fini:

0000000000001244 <_fini>:
    1244:	f3 0f 1e fa          	endbr64 
    1248:	48 83 ec 08          	sub    $0x8,%rsp
    124c:	48 83 c4 08          	add    $0x8,%rsp
    1250:	c3                   	ret    
```

The function at the start address i `main`, located at `0x1189` (`0000000000001189`). This is the entry point of the program, i.e. where execution of the program code begins.

The "lower" addresses are part of the programs initilization and setup, i.e. preparing the system for running the program.

3.4

> 4. Run the program several times (hint: running a program from the current directory is done using the syntax ./mem). The addresses change between consecutive runs. Why?

The addresses change between consecutive runs due to randomization of the address space. This is a security feature to prevent e.g. malicous software from predicting where any given process will store its data in memory. This feature is called Address Space Layout Ranomization, or ASLR for short.

4 - The stack

Consider the following C program:

```c
#include <stdio.h>
#include <stdlib.h>
// Changed pointers from 0x%08x to %p due to 64bit system.
void func()
{
   char b = 'b';
   /*long localvar = 2;
   printf("func() with localvar @ %p\n", (void*)&localvar);
   printf("func() frame address @ %p\n", (void*)__builtin_frame_address(0));
   localvar++;*/
   b = 'a';
   func();
}

```

```c
int main()
{
   printf("main() frame address @ %p\n", __builtin_frame_address(0));
   func();
   exit(0);
}
```

4.1

> 1. Compile the example given above using gcc stackoverflow.c -o stackoverflow.

I had to change the pointers from `0x%08x` to `%p` due to running Linux on a 64 bit system, not in a 32 bit VM. A comment was made in the code pasted above.

4.2

> 2. Determine the default size of the stack for your Linux system. Hint: use the ulimit command (a web search or running the command ulimit --help will help find the appropriate command-line flags).

Output of `ulimit -s`:

```
8192
```

The output show the size of the stack in kilobytes.

4.3

> 3. Run the program. Describe your observations and find the cause of the error.

Output from running the `./stackoverflow` C-file:

```
main() frame address @ 0x7ffd5689fe00
Segmentation fault (core dumped)
```

The segmentation fault, or `segfault`, is a fault condition thrown by the memory protection system to notify the OS that the program has attempted to access a restricted memory area ("memory access violation"). This is kind of fault is often the result of programming mistakes that result in stack overflow error.

The reason for this is usually that the program is stuck in an infinite loop, where function calls or writing are running without any conditions under which it will terminate. It will continue using up stack space until it runs out, and since it's still calling functions it will try to write access memory outside the bounds of stack. This violates the restrictions on memory access, and is terminated with a `segfault`.

The reason for this, in this program, is the recursive calls on `func()`. Since there's no conditions set in `func()` where it will self-terminate, or other wise funish or end, the recursive funtion calls will run in an infinite loop.

4.4

> 4. Run the program and pipe the output to grep and wc -l:
> `./stackoverflow | grep func | wc -l`
> What does this number tell you about the stack? How does this relate to the default stack size you found using the ulimit command?

Output from running the program as is resultet in the output `0`.

Assuming there is supposed to be something more to see, uncommenting line 9,

```
printf("func() frame address @ %p\n", (void*)__builtin_frame_address(0));
```

results in the output `261712.`

Further uncommenting line 6, 8 and 11:

```
long localvar = 2;
printf("func() with localvar @ %p\n", (void*)&localvar);
localvar++;
```

outputs `349022`.

The `wc -l` command outputs "word count, lines". If we focus on the second output, this means that a line with the string `func` was output 261 712 times before the segfault occured. In short, the stack could fit 261 712 lines of the string `func() frame address @ %p\n` into the stack.

4.5

> 5. How much stack memory (in bytes) does each recursive function call occupy?

- Stack size (binary): 8 192 KiB * 1024 = 8 388 608 bytes
- Stack size (SI): 8192 kB * 1000 = 8 192 000 bytes
- Lines output: 261 712

Calculating stack frame size using binary size:

$$\frac{Total\ stack\ size,\ bytes}{Number\ of\ calls} = \frac{8\,388\,608}{261\,712} = 32.053\ byte$$

The calculation show that each recursive call occupy about 32 bytes of stack space.