OS øving 1

The process abstraction

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 reason this must happen is that in order to start a new process, the kernel has to copy the program into memory. Then the kernel sets the program counter to the first instruction of the process. Additionally sets the pointer to the user stack base, then finally switches to user mode. This process is necessary because the user and kernel levels have different access levels, which is done for security reasons.

2. Download the latest Linux kernel source code from https://kernel.org and unpack it. Use a web search engine to help identify the file in the source tree that contains the process descriptor structure (hint: its name is task struct). List the field name from this structure that:
(a) Stores the process ID
(b) Keeps track of accumulated virtual memory
Use the Linux command-line tool top to explore other fields relating to running processes. Can you match them to field names in the process descriptor task struct? Name two such fields (besides those listed above).

a) The field that stores the process id is pid

```
	unsigned			in_thrashing:1;
#endif

	unsigned long		atomic_flags; /* Flags requiring atomic access. */

	struct restart_block		restart_block;

	pid_t			pid;
	pid_t			tgid;

```

b) The field that stores the accumulated virtual memory is Acct_vm_mem

```
#endif
#ifdef CONFIG_TASK_XACCT
	/* Accumulated RSS usage: */
	u64			acct_rss_mem1;
	/* Accumulated virtual memory usage: */
	u64			acct_vm_mem1;
	/* stime + utime since last update: */
	u64			acct_timexpd;
#endif
```

```
    PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
   1018 sverre    20   0 6308928 423004 145016 S   5,5   7,1   8:47.68 gnome-shell
   2187 sverre    20   0  839828  58952  42212 S   0,6   1,0   0:21.88 gnome-terminal-
    540 systemd+  20   0   14824   6784   6016 S   0,3   0,1   0:11.00 systemd-oomd
  25747 root      20   0       0      0      0 I   0,3   0,0   0:06.99 kworker/5:0-events
  28130 root      20   0       0      0      0 I   0,3   0,0   0:03.00 kworker/u16:0-events_freezable_power_
  28879 sverre    20   0   24764   4608   3712 R   0,3   0,1   0:00.47 top
      1 root      20   0  168116  12976   8240 S   0,0   0,2   0:04.71 systemd
```

PR - Prio

```

	int				prio;
	int				static_prio;
	int				normal_prio;
	unsigned int			rt_priority;

```

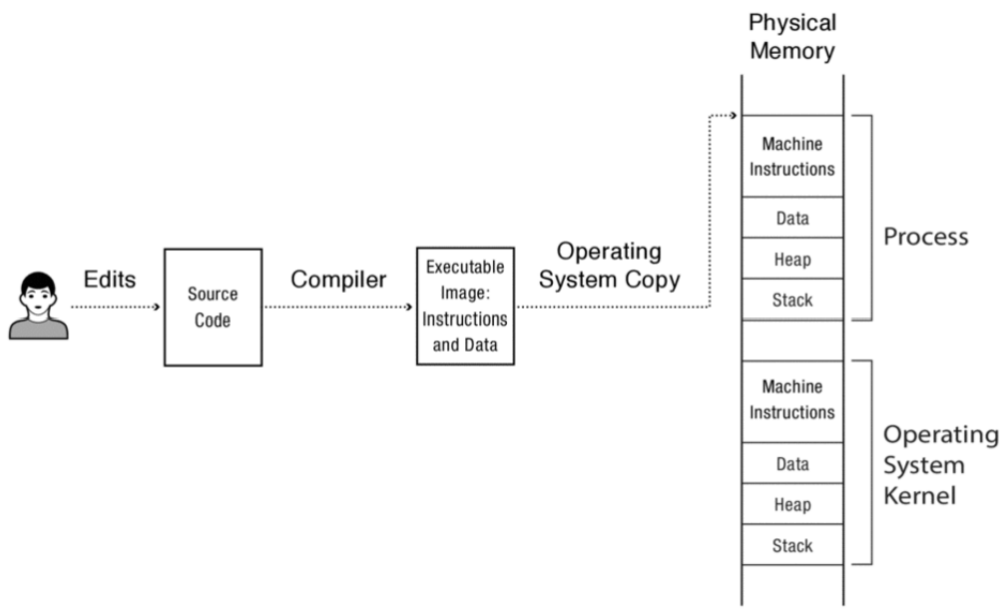
STATE - __state

```
struct task_struct {
#ifdef CONFIG_THREAD_INFO_IN_TASK
	/*
	 * For reasons of header soup (see current_thread_info()), this
	 * must be the first element of task_struct.
	 */
	struct thread_info		thread_info;
#endif
	unsigned int			__state;
```

Process memory and segments

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

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



2. Briefly describe the purpose of each segment. Why is address 0x0 unavailable to the process?
    a. Text segment (Machine instructions) -
        i. The purpose of machine instructions is to tell the processor to perform machine operations.
    b. Data segment (Data) -
        i. The purpose to data segments is store and organize saved information
    c. Stack -
        i. The purpose of the stack is to store temporary variables that are created by a function
    d. Heap -
        i. The purpose of the heap is to store the objects that are created during the execution of a program
    e. 0x0
        i. The address 0x0 is unavailable as it is system dependent

3. What are the differences between a global, static, and local variable?
    a. Global
        i. Variable that can be accessed by any method in the system, even those outside the class in which it is defined
    b. Static
        i. A variable that gets initialized during the start of the program, in effect extending its lifetime to be equal to the runtime of the program.
    c. Local
        i. Variable that can only be accessed by method in the class in which it is defined
4. Given the following code snippet, show which segment each of the variables (var1, var2, var3) belong to.

```c
#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);
}
```

    a.
        i. Var1 = Global , Var2 = Local , var3 = Local (Static?)

Program code

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.

```
Oppgave3  Oppgave3.c
sverre@Sverre:~/os/2023/oblig1$ size Oppgave3
   text    data     bss     dec     hex filename
   1799     616       8    2423     977 Oppgave3
sverre@Sverre:~/os/2023/oblig1$
```

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

```
sverre@Sverre:~/os/2023/oblig1$ objdump -f Oppgave3

Oppgave3:     file format elf64-x86-64
architecture: i386:x86-64, flags 0x00000150:
HAS_SYMS, DYNAMIC, D_PAGED
start address 0x00000000000010a0

sverre@Sverre:~/os/2023/oblig1$
```

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.
   a. The function at this address is the _start method. The _start method initializes the program, and calls upon the program's main function.

4)

```
sverre@Sverre:~/os/2023/oblig1$ ./Oppgave3
Address: 70006014; Value: 0
Address: 96e6904c; Value: 1
Address: 96e69050; Address: 70ef92a0; Value: 2
sverre@Sverre:~/os/2023/oblig1$
```

```
sverre@Sverre:~/os/2023/oblig1$ ./Oppgave3
Address: 78930014; Value: 0
Address: 75b4011c; Value: 1
Address: 75b40120; Address: 796aa2a0; Value: 2
sverre@Sverre:~/os/2023/oblig1$
```

```
sverre@Sverre:~/os/2023/oblig1$ ./Oppgave3
Address: 35542014; Value: 0
Address: 21650c0c; Value: 1
Address: 21650c10; Address: 35a132a0; Value: 2
sverre@Sverre:~/os/2023/oblig1$
```

The reason the addresses change with each consecutive run is that the addresses are selected at random from the pool of free memory addresses. This is a process called ASLR and it works to make it more difficult to exploit security holes.

The stack

1.
   ```
   sverre@sverre-VirtualBox: ~
   sverre@sverre-VirtualBox:~$ ulimit -s
   8192
   sverre@sverre-VirtualBox:~$
   ```
   a. The default size of the stack for the linux system is 8192 bytes.

2.
   ```
   sverre@Sverre:~/os/2023/stackoverflow$ ./stackoverflow
   main() frame address @ 0x6f841c20
   Segmentation fault
   sverre@Sverre:~/os/2023/stackoverflow$
   ```
   a. When attempting to run the program we get a Segmentation fault. The func method calls itself recursively, but has no way of breaking out of the recursion, resulting in more and more recursive calls until the amount of calls exceeds the allotted default space in the stack, and the program aborts.

3. What does this number tell you about the stack? How does this relate to the default stack size you found using the ulimit command?
   a. The number we get is the maximum amount of lines the function can print before a stack overflow occurs.

    b. The relation between the number and the stack is that the number shows the amount of space in memory each function call requires.
    c. The result of ./stackoverflow | grep func | wc -l:

```
sverre@Sverre:~/os/2023/stackoverflow$ ./stackoverflow | grep func | wc -l
349123
sverre@Sverre:~/os/2023/stackoverflow$
```

4. How much stack memory (in bytes) does each recursive function call occupy?
    a. We can calculate how much space each call takes by calculating:
        i. (349123 / 2) / 8192 = 21.30877
        ii. We then round this up to 22
        iii. Each recursive call takes up 22 bytes