Assignment 1

1 The process abstraction:

1. When a process is started from a program on disk, the kernel firstly allocates and initialize the process control block and allocates memory for the process. Then it copies the program from disk to the allocated memory from the previous step. Then it allocates a user-level stack for user-level execution and a kernel-stack for handling system calls, processor exceptions and interrupts. Then, order to start running the program, the kernel copies arguments into the user-stack and increments the user's stack pointer. Finally, there is a mode switch from kernel- to user-mode because only the kernel has permission to start new processes, but a process itself shouldn't have the same permissions as the kernel and we must therefore switch to user-mode before the process starts running.
2.
    a. ***pid_t pid*** stores the process ID
    b. ***struct mm_struct *mm*** keeps track of accumulated virtual memory

    int rt_priority equals top's "*PR*" and shows the scheduling priority of each process

    long state equals top's "*S*" and shows the state of the process.

2 Process memory and segments

1. *Figure 1* shows a good example of the organisation of a process' address space.
2. The text segment consists of program code. The data segment contains strings initialized from the program executable and global variables. The stack stores local variables and function parameters. The heap segment provides runtime memory allocation for data that must outlive the function doing the allocation. The address 0x0 is unavailable because most implementations of *null pointers* represent it with 0x0 and since a *null pointer* cannot be referenced, it's unavailable.

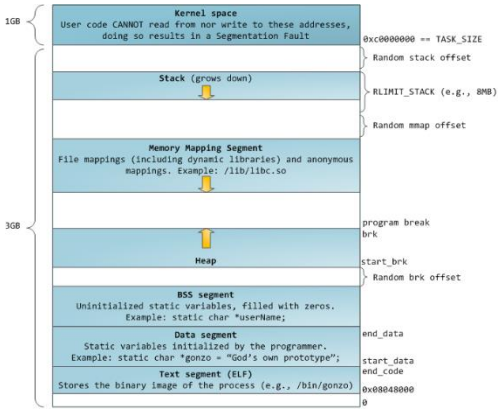

Figure 1: Organisation of a process' address space

3. Local variables are created when the function has started execution and is lost when the function terminates, on the other hand, Global variable is created as execution starts and is lost when the program ends. Local variables are stored on the stack whereas the Global variable are stored on a fixed location decided by the compiler which means that in other source files the same name refers to the same location in memory. Static global variables on the other hand are private to the file where they are defined and don't conflict with other files. Static local variables keep their memory throughout the program while they keep the scope like local variables.

var1 is stored in the data segment as it's a global variable. *var2* is a local variable and therefore belong to the stack. *var3* is a local variable and therefore also belongs to the stack even though it points into the heap.

3 Program code

1. Text: 1 895, data: 616, bss: 8
2. Start address: 0x00000000000005f0
3. The function at the start address is *xor* and it's used with *%ebp, %ebp* which is the frame register. The *xor*-function in this case clears the frame register which is useful because we're starting a new process
4. The addresses changes because of address virtualization and because there isn't another process of the same program running at the same time so the global variables don't use the same memory.

4 The stack

1.
2. ***ulimit -s*** says that the maximum stack size is 8 192 * 1 024 bytes (8MB)
3. The program runs a recursive function infinite times because *func()* calls itself every run. This causes a *stack overflow* error.
4. ***/stackoverflow | grep func | wc -l*** prints 523 685. ***grep func | wc -l*** prints the number of *printf* and since it's 2 per *func* call, the stack is big enough for 523 685 / 2 = 261 842 recursive calls.
5. 8 388 608 bytes / 261 842 recursive calls = 32 bytes per recursive call