Joachim Grimen Westgaard

Assignment 1

1 The process abstraction

1. To start a new process from a program on a disk, the kernel must first allocate and initialize the PCB (process control block, also called the process descriptor). Then, the kernel must allocate memory for the process. After this, the kernel must copy the program from the disk onto the allocated memory area. In addition to this, it must allocate a user-level stack for the execution that will be handled at user-level, and a kernel-level stack for handling of the system calls, asynchronous and synchronous interrupts and processor exceptions.

   After completing the abovementioned steps, the kernel can begin running the program. To run the program, the kernel must also go through the following steps:

   - Copy arguments into the user memory. This essentially means that the potential arguments passed by the user when starting the program must be copied into user memory.

   - Initialize CPU registers, set the stack pointer and load the program's point of entry onto the PC (Program Counter)

   - Transfer control back to the user mode.

   The mode switch from the kernel-level to the user-level must happen due to several reasons:

   Security, since this will separate the process from the kernel after the kernel has completed the necessary steps to start the program. This will ensure that the program does not interfere with or corrupt the kernel moving forward with the process.

   Isolation in the sense that the process will now be contained in the user memory, to ensure that the process cannot access and modify memory locations of other

processes. This ensures that there will not occur any unwanted interactions between different processes.

Hardware protection, in the sense that it will render the process unable to directly access the hardware of the computer moving forward with the program.

In conclusion, the mode switch back to user mode must happen, amongst other reasons, due to security, protection of hardware and the kernel itself.

2.

a) The field from the structure in ***include/linux/sched.h*** that holds the process ID is called ***pid***.

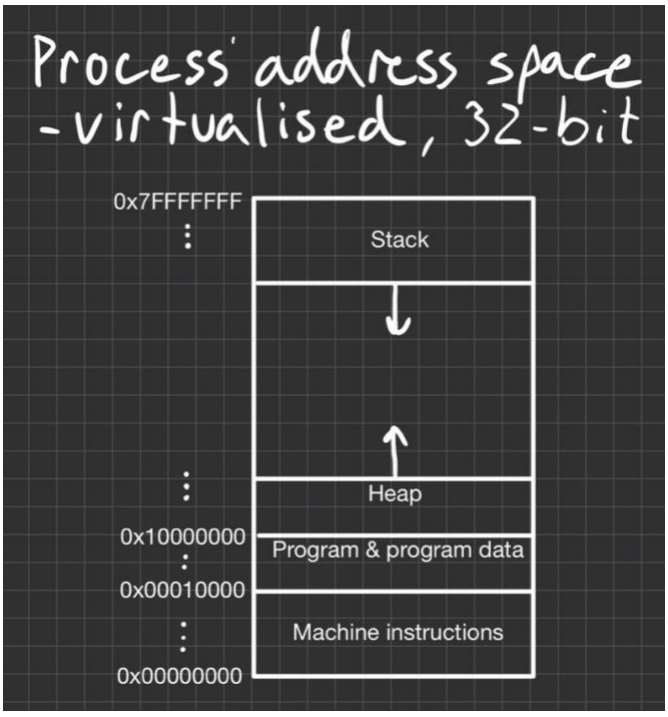
b) The accumulated virtual memory field name is ***acct_vm_mem1***.

NI (Nice Value) has the field name ***nice***.

PPID (Parent Process ID) has the field name ***ppid***.

2 Process memory and segments

1.



2. The picture above shows a 32-bit OS's virtualized process layout. Due to virtualization and the fact that every process can be tricked into believing it has the entire memory address space to itself, every process can start with the virtual address 0x00000000 and end with 0x7fffffff.

   Next, we will take a closer look at each segment, starting from the bottom.

   Machine instructions

   This segment is also referred to as "code" or "text". It is the segment of a process memory that holds onto the executable instructions for the program. It can be compared to the steps in a food recipe.

   Data

   The segment that represents the data in the process memory, holds the global and static variables that have been initialized by the programmer's code. It can be compared to the already bought and prepared items for the food recipe. For example, the line "int j = 666" will be stored here.

   There is a segment between the data and heap that is called "uninitialized data", where all the variables that hold the value 0 either by declaration or the kernel's decision, reside. For example the line "int j;" will reside here.

   Heap

   This is the area where dynamically allocated variables in a program are stored. Data structures that require a longer lifecycle than attainable in the stack, will be stored here.

   Stack

   This segment is continuously managed by the OS to store locally declared variables that will be handled with the LIFO order (Last In, First Out). It also stores the order of method execution/function calls.

The memory address 0x0 is unavailable due to it being the conventional reference for a “null pointer”. In other words, it is reserved to be the memory address for “a pointer to nothing”.

3. Global vs. static vs. local variables

A global variable refers to variables that have been assigned a global-scope in the program and other .c-files residing inside the same folder. This variable is available to all the other functions and logic within the different C-program, while a local variable is encapsulated inside of a function or logic. For example, a global variable in C will be declared outside of the main-method, and therefore be available to all logic inside of the main-method. It can also be available to many other methods, if it is declared ahead of them in the code. If we have two .c-files with the same global variable name, we will encounter an error, because the compiler will not be sure which one to use. This is where global static variables come in handy, because they will be “local” to the file that it is declared in.

A local variable can for example refer to the variable “i” in a for-loop’s method signature. This variable will only be available to the code inside of the for-loop. In other words, this variable will be local-scoped, and unavailable to any code outside of the for-loop. Both global and local variables can have their values changed in a program’s lifecycle.

A static variable is a variable that will persist even when the program exits its scope. A function that increments a locally variable made static (with the static keyword in front of it) will store its incremented value for later, due to it not being stored in neither the heap nor stack. This essentially means that the variable can have a local scope for code references but also be able to store its value for later usage.

The variable “var1” is declared outside of any functions and will therefore be considered a global variable. This variable will be stored in the data-segment of the allocated memory. The “var2” is declared inside of the main function and will therefore be a local variable. This variable will be stored on the stack. The “var3” will be a pointer which points to a

memory address inside the heap-segment, but the pointer itself will be stored in the stack along with “var2”. We can verify this due to the memory address of the pointer “var3” having a memory address close to “var2”.

3 Program code

1. Sizes:
    a. text: 1877
    b. data: 632
    c. bss: 8
2. Start address: 0x00000000000006c0
3.

```
Disassembly of section .text:

00000000000006c0 <_start>:
 6c0:   d503201f        nop
 6c4:   d280001d        mov     x29, #0x0                       // #0
 6c8:   d280001e        mov     x30, #0x0                       // #0
 6cc:   aa0003e5        mov     x5, x0
 6d0:   f94003e1        ldr     x1, [sp]
 6d4:   910023e2        add     x2, sp, #0x8
 6d8:   910003e6        mov     x6, sp
 6dc:   f00000e0        adrp    x0, 1f000 <__FRAME_END__+0x1e630>
 6e0:   f947ec00        ldr     x0, [x0, #4056]
 6e4:   d2800003        mov     x3, #0x0                        // #0
 6e8:   d2800004        mov     x4, #0x0                        // #0
 6ec:   97ffffd1        bl      630 <__libc_start_main@plt>
 6f0:   97ffffe0        bl      670 <abort@plt>
```

The function “_start()” is the entry point of the program. This is where the program will start executing from. It initializes the program’s runtime and invokes the main function. It also, among others, prepare the argc/argv from the stack and prepare the stack before calling the main function.

4. The memory addresses are different for each run due to it being a secucrity-implementation for the safety of the memory space. Linux, among other OS, use ASLR (address space layout randomization) to randomize the memory addresses, even when executing the same program twice in a row. This is a safety feature because it secures processes from being maliciously found and targeted by potentially malevolent programs.

4 The stack

1. Compiled and ran.
2.

```
joachimgrimenwestgaard@debianJoachimGW:~/Documents$ ulimit -s
8192
```

The “ulimit -s” shows the maximum stack size in kB. In this case, the system has exactly 8 MB set as the maximum stack size.

3.

```
joachimgrimenwestgaard@debianJoachimGW:~/Documents$ ./stackoverflow
main() frame address ' 0xfca2cdd0
Segmentation fault
```

The segmentation fault appears due to the program being infinitely recursive. The function “func()” is called once in the main-function, and then recursively calls itself inside the function. The variable “b” is changed between “a” and “b” each time. This infinite function calling will eventually lead to the stack being overflown, thus creating a “Segmentation fault”. The memory region allocated to the stack will be filled, thus not allowing for any further function calls, which finally results in the exception that halts the program.

We can fix this issue by for example restricting the number of recursive calls or removing it entirely. I choose to remove the recursive calls entirely in my solution.

4. The number printed is 0, due to there being no matches for the word “func” in the program’s output. However, if we were to uncomment the two print statements inside the func()-method, we would see the amount of recursive calls this function completes. In our case, only the print statement inside the main-function prints to *stdout*, which does not include the word “func” in any way.

   That the number printed is 0, tells us that there are no memory spaces in the stack that are occupied with the character array value “func”. If we were to use “grep main” instead, we

would see the number 1 appear instead, due to the print statement in the main-function including the word “main”.

If we were to remove the commented out code, and run the command, we would receive the number “523565” due to it being the number of found prints that match the word “func”. Due to this number being reasonably larger than the allocated default stack size, this will mean that the stack size needs to be extended, which can finally result in a memory segmentation fault due to the memory region allocated to the stack being exhausted.
We can determine that the func()-function was called 523565 divided by 2 times, due to the word “func” being printed twice each call.

5. We can, among other viable methods, determine the number of bytes occupied by each call by looking at two neighboring addresses. Consider the two addresses

   0xf4418840 and 0xf4418820

   These two addresses are the last and second-to-last addresses occupied by the two last func()-calls. We know that the addresses are represented in hexadecimals, so we can look at the remainder when we subtract the lesser from the greater number, which results in the hexadecimal number 20. To convert this number into our normal 10-digit system, we must split the number into 2 and 0. The 2 can be converted by doing (2 * 16^1) and the 0 can be converted using (0 * 16^2). This is the same as converting from binary but using 16 instead of 2 when converting. This results in the number 32 + 0, which is equal to 32.

   This means that we can determine that each call occupies **32 bytes** inside the stack.

   We can also determine the rough number of recursive calls by dividing the stack size (8192 kB) with the number of recursive calls (roughly 523565/2). This equals around 31.3, which we can round up to 32 bytes.

   In conclusion, we can see that the stack limit found using *ulimit* was reached by the program and attempts of exceeding this limit resulted in the program running into a segmentation fault.