Operating systems

Assignment 1 – Ingebrigt Hovind

Worked on with Mads Lundegaard, but handed in individually

1 – The process abstraction

1. In order to start a program from disk, kernel mode is initially needed in order to deliver an interruption to the processor, telling it to finish what it is currently working on and then prioritize starting the new program. There are also several other processes which must happen in order to initialize this new program, several of these processes demand privileged access. These are, assigning the memory locations, accessing the actual location of the program on the disk and to specify the base and bound of the memory that will be assigned to this program. After the program has been started successfully the processor needs to make the switch to user mode, this is to ensure that the program we just started does not have the privileged access that was just used to start it. Using the principle of least privilege we can ensure a higher level of security.
2. Linux-5.8.3/include/linux/sched.h

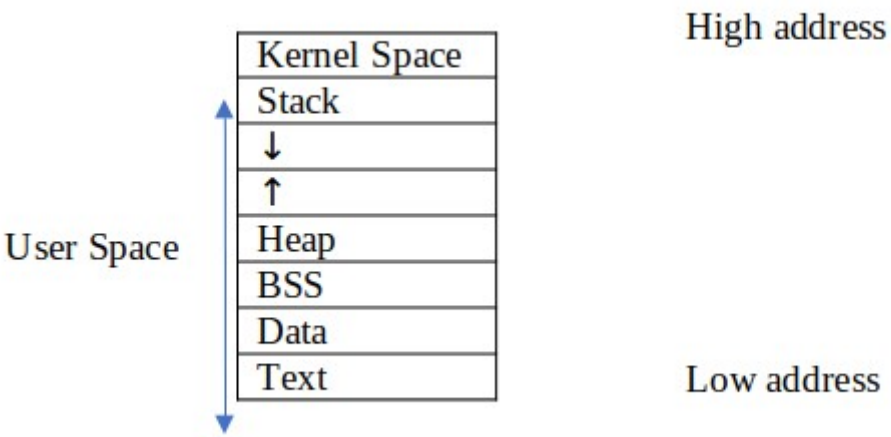
   A) Thread_PID

   B) acct_vm_mem1

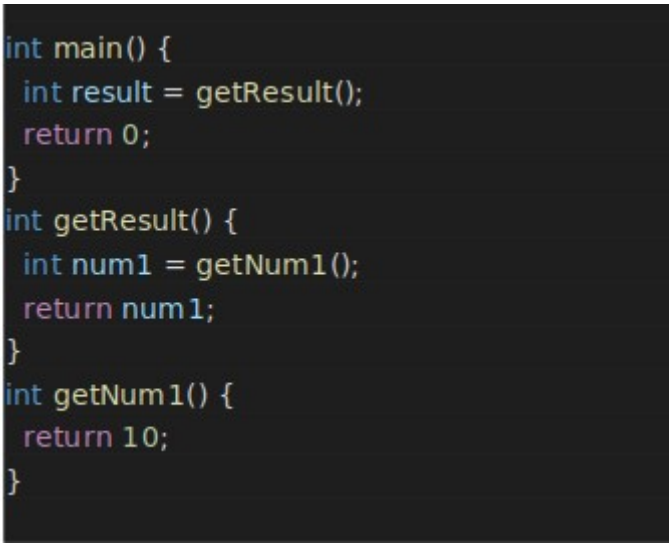
      1. The command name is stored in char comm[TASK_COMM_LEN]; on line 908 in sched.h
      2. PR is stored in unsigned int rt_priority on line 678 in sched.h

2 – Process memory and segments

1.



2.

- Text Segment
  - Also known as code segment. It is the part of the virtual address space that contain the executable instructions. The compiler will go through this segment with the program counter line for line when executing the program. The text segment is often read-only, to prevent a program from accidentally modifying its instructions.
- Data Segment
  - The data segment is the segment containing initialized static variables. It is divided into a read-only and a read/write space
- Stack Segment
  - Stack is the first segment of a process memory and is stored under the kernel space. The stack and heap are often located besides each other, with some space between them. Both the stack and the heap uses this space to grow towards each other when they need to extended memory space. The purpose of the stack is to store all the data needed by a function. The variables that are saved on the stack are only temporary, as long as the function "exists". The structure of the stack is "last in first out", which means that the last added "task" will be the first to be executed. The two operations available in LiFo (Last in, first out) is push and pop, where push is adding a new variable/function and pop is removing.
  - 

```
int main() {
 int result = getResult();
 return 0;
}
int getResult() {
 int num1 = getNum1();
 return num1;
}
int getNum1() {
 return 10;
}
```

In the code example above the stack flow will be like this:

Push	Push	Push	Pop	Pop
main()	getResult()	getNum1()	getResult()	main()
	main()	getResult()	main()	
		main()		

Since the stack can grow in size but still has a limited space to grow, problems may occur. A well known technique for solving computable problems in computer science is recursion. In recursion the function calls itself to solve a sub-problem, this means that both the original function and the new function created, are stored on the stack. Deep recursion may therefore cause stack overflow since all the function calls are saved on the stack.

- Heap segment
  - The heap is also used for storing variables like the stack but it doesn't have the same LiFo method as the stack. This means that a variable in the heap can exist without being tied to a function, whereas a variable in the stack only "exists" a limited time. The variables stored in the stack is often global variables and objects. The space in the heap is not managed automatically by the CPU like the stack, this can cause fragmented memory and memory leaks. In the C language you have to free the variables that you have allocated yourself or else you will have memory leaks. Memory leak is a part of memory that is set aside and therefore is not available for other processes, but that is not being used. In languages like Java the freeing of objects is handled automatically by the garbage collector.
- The 0x0 address is unavailable for the process because the 0x0 address is used as a null pointer constant. When you assign a variable the value NULL it will point to the address 0x0. By reserving this address we can keep it consistent by always having this constant at the same address.

3.

1. Static variables remain in memory for the entire duration while the program is running. They are stored in the data segment of the memory after they are initialized. They are accessible within the scope where they are declared, but they remain in memory despite the program moving out of this scope.
2. Local variables they are stored on the stack. Local variables cease to exist once the function that created them is completed. They are only accessible within the scope where they are declared.
3. Global variables are stored in the data segment, they are accessible from anywhere within the program after their declaration. If they were declared using memory allocation such as malloc() then they are stored on the heap.

4.

2. var1 is global and is stored in the data segment
3. var2 is local and is stored on the stack
4. var3 is a pointer stored on the stack, as it is a local variable. It has been declared using memory location so it is pointing to a memory location on the heap

3 - Program code

1)

```
ingebrigt@ingebrigt-ThinkPad-L450:~/Documents/uni
Oeving 1/opg3$ size --format=Berkeley mem
   text    data     bss     dec     hex filename
   1978     616       8    2602     a2a mem
```

2)

```
ingebrigt@ingebrigt-ThinkPad-L450:~/Documents/u
Oeving 1/opg3$ objdump -f mem

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

3)

Endbr64 is the first function in the start address. This function is used to mark valid jump target addresses of indirect calls and jumps in the program. It is useful because it marks the spots that are valid for jumps and it won't break the program even if the processor does not have CET enforcement.
It stands for "End Branch 64 bit" -- or more precisely, Terminate Indirect Branch in 64 bit

```
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:       4c 8d 05 f6 01 00 00    lea    0x1f6(%rip),%r8        # 12b0 <__libc_csu_fini>
    10ba:       48 8d 0d 7f 01 00 00    lea    0x17f(%rip),%rcx        # 1240 <__libc_csu_init>
    10c1:       48 8d 3d c1 00 00 00    lea    0xc1(%rip),%rdi        # 1189 <main>
    10c8:       ff 15 12 2f 00 00       callq  *0x2f12(%rip)        # 3fe0 <__libc_start_main@GLIBC_2.2.5>
    10ce:       f4                      hlt
    10cf:       90                      nop
```

4)
The addresses change between consecutive runs due to Address Space Layout Randomization (ASLR). This is a memory protection mechanism where addresses used by components of a process are randomized so that it is harder for an attacker to reliably jump to, for example, a known vulnerable function in memory
.

4 – The stack

1. Ok
2. 8192 kibibytes

```
ingebrigt@ingebrigt-ThinkPad-L
Oeving 1/opg4$ ulimit -s
8192
```

3. The command line alternates in printing “func() frame address” and “func() with localvar”, both including a slightly different address each time. This keeps on happening until “segmentation fault (core dumped)” is printed out and the program stops.
    1. The root cause of the error is the infinite recursion in the “func()” function. This in addition to creating a new localvar variable every call makes the program overflow its stack with local variables , creating a stack overflow since none of the functions actually complete.
4. 523444
5. 8192000/523444 ≈ 32 Bytes