Operating Systems 2016/17
Solutions for Assignment 4

Prof. Dr. Frank Bellosa
Dipl.-Inform. Marc Rittinghaus

T-Question 4.1: Processes and Threads

a. A process creates a new kernel level thread. Name at least two structures in memory and one operation that the kernel needs to allocate or perform until the CPU executes the new thread. For each structure, denote if they are placed in kernel or user space. **2 T-pt**

Solution:
Data structures:

(a) Create a new thread control block (TCB) (kernel space)
(b) Allocate a stack (user memory)
(c) Depending on OS: Allocate a kernel stack (kernel memory)

Operations:

(a) Add the new thread to the (process's) list of threads
(b) Perform a context switch to the new thread

b. What command could you use in Linux to list the threads of a certain process? **1 T-pt**

Solution:
There are various ways to display the threads of a process. A simple one is to call: `ps -T -p <pid>`.

c. A process creates two threads T1 and T2. When T1 executes a recursive function, the process crashes with a stack overflow error. When instead T2 executes the same function, the exception does not occur. What might be the reason? **1 T-pt**

Solution:
When a thread is created, the operating system (OS) has to allocate a stack for the thread. On x86, stacks are typically placed at the top of the address space, because the stacks grow downwards when the respective threads execute. On the lower address range, the operating system thus places the process's heap so that it can grow upwards. How much each stack can expand, depends on where the OS places the stacks in the virtual address space (VAS).

A stack overflow exception occurs, when a stack reaches its maximum size and cannot grow any further. This might be due to configured limits or because the placement of the stacks in the virtual address space does not allow further growth. For the given scenario, the maximum size of T1's stack must be smaller than the one of T2's stack. In consequence, either the limits for T1's and T2's stacks are configured differently, or the stacks are placed in the VAS in such a way that the gap between the stacks is smaller than the gap between T2's stack and the heap – leaving T1 less space for stack growth.

d. Describe how dynamic shared libraries can be loaded at any virtual address. **2 T-pt**

Solution:
Dynamic shared libraries are loaded into the virtual address space of the process when the process starts. This is usually done by a component simply called 'loader' or 'dynamic loader'. In Windows the component is located in `ntdll.dll`*, which is*

mapped into each process by the kernel. At program start, the loader identifies all dependencies on shared libraries (which may have dependencies themselves) and loads the libraries into the virtual address space.

The position at which libraries are mapped depends on what other libraries have to be loaded and if security features such as address space layout randomization (ASLR) are active. Since compiled program code typically requires many references on data and other code in memory, it is a problem if the addresses are not known at compile or link time. This is the case with dynamic shared libraries.

For dynamic shared libraries to work, (1) the code of the shared library itself must be functional, and (2) external code needs a way to access the exported functionality of the shared library, both independent of the mapping address of the library.

The first aspect is solved by using position-independent code (PIC), that means the compiler only uses relative addressing when accessing data and code. This implies that data is usually placed at a fixed offset from the code that accesses it.

The second aspect can be solved by introducing a level of indirection. If a library A references an exported variable from library B, the loader must first load library B. Afterwards, it can load library A and place the address of the variable in a special table, the global offset table (GOT). Each library has a dedicated GOT that will hold the addresses of imported variables. Similar to data, the GOT is at a fixed known position to the code that accesses it, thus allowing the code to fetch the address from the GOT at runtime. Calling external functions works similarly. Instead of producing code that directly calls the imported function (as possible with static libraries), the compiler generates calls that will first fetch the address of the imported function from an address table.

Total:
6 T-pt