Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich
Spring Term 2015

Operating Systems and Networks
Assignment 1

Assigned on: **19th February 2015**
Due by: **27th February 2015**

1 General Operating Systems Questions

a) *What is the purpose of having a kernel?*

Answer: A kernel is a piece of software which provides save multiplexing/access of the underlying hardware.

(a) What is the least functionality a kernel has to provide usually (Hint: Usually a minimal kernel provides three properties)?

Answer: A kernel provides at least basic scheduling, some form of message passing (or the setup of channels) and protection (only the kernel sets up page table entries)

(b) What does this functionality provide to the rest of the system?

Answer: This functionality provides a secure environment to the user-space. The underlying hardware is multiplexed. Every process can get memory, but cannot overwrite any other process's memory. Basic scheduling makes sure, that every process gets some amount of CPU cycles. Basic messaging makes sure that you can build applications which act as servers. Those servers export functionality/services which can be used by other applications.

(c) Where does the rest of the system reside?

Answer: The rest of the system lives in user space which means in libraries and applications. Applications may provide services to other applications.

(d) How does the rest of the system interact with the kernel?

Answer: The user-space part of the system interacts via system calls with the kernel. System calls are the "gateway" to the kernel.

(e) Why does it need to interact with the kernel?

Answer: Getting more memory means setting up page table entries which can only be done by the kernel. Setting up a message channel cannot be done in user-space, if user-space processes are completely isolated. In general, user-space libraries or applications have to interact with the kernel, if they need some resources which can only be accessed by the kernel.

Answer: **Note:** The description above is about the minimum functionality which usually is provided by a kernel (microkernel). Linux is a monolithic kernel. It provides a lot more functionality to the user-space.

b) *The kernel can do and access everything. It exports functionality to user applications by syscalls. Does that mean that every user application can execute code in the kernel by doing a syscall?*

Answer: In theory yes, but the real answer is no. System calls can be performed by every user-space application. However the kernel has to check arguments passed by the calling user application. It also has to check whether the user launching the application is "privileged" enough to do some operations. In Linux we have the notion of root. For some operations, the kernel checks whether the calling process has root rights.

c) *Can you compile libc without the kernel sources?* **Answer:** You can compile glibc without kernel sources. However you need the kernel headers. The reason is that there is an interface between the kernel and the user-space library `glibc`. The library has to know the system calls.

2 fork()

Creating new processes in the Unix/Linux world is done using fork(). fork() clones an existing process and adds it to the runqueue, rather then really creating a new one. Since fork clones a process, they both execute the line after the fork() call. Now they need to distinguish whether they are parent or child process. This can be done by checking the return value of fork(): to the parent process, fork() returns the PID of the child process, to the child process, fork() returns 0

2.1 Playing with fork()

2.1.1 Calling fork() once

Create a program which forks itself once. The parent process should output "Im the parent and my child's PID is <pid>". The child should output "I'm the child and my PID is <pid>". Do the PIDs match?

2.1.2 fork() multiple times

What do you expect to happen here? Please explain what you think will happen.

```
int main(int argc, char **argv) {
    while(1) { fork(); }
}
```

Answer: If you execute this code, your computer will be (almost) dead. Every child forks new children in a loop and this new children fork new children in a loop as well. This consumes too many resources in a very short time. Not only memory, but also CPU cycles, page table entries, descriptors...