VORLESUNG 1: Read Chapter 1 from *"Computer Systems: A Programmer's Perspective"* and Discuss two things you find particularly interesting

***http://csapp.cs.cmu.edu/2e/ch1-preview.pdf**

1. Kernel virtual memory

> "Kernel virtual memory. The kernel is the part of the operating system that is always resident in memory. The top region of the address space is reserved for the kernel. Application programs are not allowed to read or write the contents of this area or to directly call functions defined in the kernel code"

Ich fand interessant, dass Kernel in jedem Virtual Memory eigenen geschützten Bereich hat, auf den Prozesse nicht zugreifen können. Das ist auch logisch, denn beim Systemaufruf Kernel sofort auf eigene Daten zugreifen können muss und indem Kerner und User-Speicher denselben Virtual Memory teilen, entfällt zusätzliches Umschalten der Page Tables.

Dann habe ich mich gefragt "Wenn im gleichen virtuellen Adressraum des Prozesses ein geschützter Bereich für den Kernel existiert, besteht dann nicht ein Risiko, dass ein User-Mode-Prozess theoretisch Zugriff auf diesen Kernel-Bereich bekommen könnte?"

Ich habe recherchiert und erfahren, dass dieses Sicherheitsrisiko durch Kombination von MMU, Page Tables, und Prozessor-Priviledge-Rings minimiert wird.

1. Priviledge-Rings: der Prozessor arbeitet in verschiedenen Privilege Levels (Ring 0: Kernel-Mode, Ring 3: User-Mode) und nur Code, der im Ring 0 läuft, kann auf diesen geschützten Kernel-Bereich zugreifen. Wechsel in Ring 0 ist nur über einen syscall/interrupts/exceptions möglich. Noch wird das ganze durch spezielle Datenstruktur Interrupt Descriptor Table (IDT) geschützt, die jeden möglichen interrupt oder exception mit einer spezifischen Funktion (Interrupt-Handler) verknüpft und der Prozessor lässt nur Sprünge zu den in der IDT hinterlegten Adressen zu! Direktes Springen zum Kernel Code ist somit nicht möglich
2. MMU (Memory Management Unit: schützt den Kernel-Speicher in Virtual Memory durch Isolierung des Speicherraums und Zugriffsrechte in den Page Tables. Wenn ein Prozess versucht eine Kernel-Funktion direkt auszuführen oder auf KErnel Speicher zuzugreifen, prüft MMU in den Page Tables vom Prozess:

- **Ist denn die virtuelle Adresse gültig?**
  - Wenn sie nicht existiert (*Present-Bit = 0*):
    - → **Page Fault Exception:** MMU meldet OS, dass die Adresse ungültig ist.
- **Ist die Seite mit dem User/Supervisor-Bit als nur Supervisor (Ring 0) markiert?**
  - Wenn ja und der Prozes ist im User-Mode (Ring 3):
    - → **Segmentation Fault Exception:** MMU blockiert den Zugriff und meldet exception

2. Netzwerk als I/O-Device Abstraktion

> „From the point of view of an individual system, the network can be viewed as just another I/O device.“

Diese Idee von Abstraktion, das Netzwerk funktional als ein Ï/O device"behaltelt wird fand ich interessant! Das macht auch Sinn - die Idee hinter I/O-Geräten ist, Daten zu empfangen und Daten zu senden. Das Netzwerk erfüllt dieselbe Funktion, bzw. die Schnittstelle zum Netzwerk (Netzwerkadapter, *Network Interface Card* (NIC)).

Genau wie bei anderen I/O-Geräten (darunter auch für mich, als für Nicht-Informatik-Bachelor-Studentin weniger offensichtliche I/O-Geräte wie Dateien und Speichermedien) funktioniert der Zugriff auf Netzwerke durch folgende für alle I/O devices ähnliche Mechanismen:

1. **API**: Netzwerksockets werden in POSIX-kompatiblen OS wie Dateien behandelt (mit Standard-APIs wie read(), write() und close())
    - Socket wird wie eine Datei über einen Deskriptor identifiziert (in File Descriptor Table (im Kernel) verwaltet)!
2. **Netzwerkadapter als I/O-Controller**: Adapter (im allgemeinen) sorgen dafür, dass OS nicht mit den Hardware-Details arbeiten muss - Netzwerkadapter empfängt ein Ethernet-Paket, überprüft Prüfsummen und übergibt das Paket an OS. Modene NICs können sogar dadurch CPU entlasten!
3. **Pufferung und Datenflusskontrolle**: OS setzt für Netzwerkanfragen Puffer ein, genauso wie es das bei Festplatten tut