Vorlesung Betriebssysteme WS 2020/21

Aufgabenblatt 1 zu Unit OS2 vom 13. November 2020

(Vorstellung der Lösungen in der Vorlesung am 26.11.2020)

Die Aufgaben auf diesem und den folgenden Übungsblättern sollen Ihnen helfen, Ihr Selbststudium zu strukturieren und Ihnen ein Gefühl dafür geben, welchen Wissensstand Sie bisher in der Veranstaltung erlangt haben sollten. Um die gemeinsame Zeit mit den Tutoren optimal nutzen zu können, sollten Sie versuchen, bis zum jeweiligen Tutorengespräch möglichst viele der Aufgaben zu bearbeiten.

Die Bearbeitung sollte in einer Art und Weise erfolgen, dass a) Sie und Ihre Gruppe gut mit den Ergebnissen arbeiten können und b) sich im Gespräch mit den Tutoren über den Stoff der Aufgaben austauschen können. Es bestehen keine weiteren formalen Anforderungen an die Aufgabenbearbeitung unsererseits.

1. Definieren Sie mit Ihren eigenen Worten den Begriff *Betriebssystem*.

   Beschreiben Sie außerdem die Rolle, die ein Betriebssystem in einem Rechner erfüllt. Welche Aufgaben hat es? Welche Designziele liegen der Entwicklung von Betriebssystemen zugrunde?

2. Erklären Sie die Begriffe *Prozess*, *Thread*, *Programm* und *Shell*, und gehen Sie dabei auf die Zusammenhänge zwischen diesen Begriffen ein.

3. Moderne Betriebssysteme unterscheiden in der Ausführung von Code immer zwischen einem *User-* und einem *Kernelmode*. Wozu dient diese Trennung? Beschreiben Sie, welche Eigenschaften Code hat, der in den jeweiligen Modi läuft. Welche Betriebssystemkomponenten können Sie typischerweise im User- bsw. im Kernelmode finden und warum? Unterscheiden Sie hierbei zwischen *Monolithischen* und *Mikrokern* Betriebssystemen.

4. Beschreiben Sie, was ein Interrupt ist, und wie das System mit ankommenden Interrupts umgeht. Welche Rolle spielen Interrupts in modernen Computersystemen?

5. Setzen Sie sich programmatisch mit Systemaufrufen, User- und Kernelmode, sowie privilegierten Instruktionen auseinander. Bearbeiten Sie dazu mindestens eine der folgenden Teilaufgaben:

   (a) Betrachten Sie den Programmrahmen zur Ausführung von Systemaufrufen:
   `https://gitlab.hpi.de/osm-teaching/operatingsystems2020-labs/syscall_invoke`

   Vervollständigen Sie den Programmrahmen, sodass Sie einen Systemaufruf zur Funktion 'getpid' ausführen können, ohne die dafür existierende Bibliotheksfunktion zu verwenden.

   Der bereitstehende Programmrahmen nimmt dabei für Sie die erste Hürde, und unterscheidet zwischen Windows und Linux, und stellt bereits einen Teil des Assembler Codes bereit, der auf Windows 10 64 Bit, bzw. Windows 7 32 Bit benötigt wird.

   Machen Sie sich dabei mit dem *C-Präprozessor*, sowie den Präprozessorbefehlen `#if` und `#ifdef` vertraut.

   Auf Linux finden sie im Makro `__NR_getpid`[1] die Nummer des `getpid` Systemaufrufs. Unter Windows nutzen Sie die Nummer der Funktion `NtQueryInformationProcess`[2].

   Beurteilen Sie, welchen Einfluss die direkte Verwendung von Systemaufrufen ohne die Bibliotheksfunktionen auf die Portabilität eines C Programmes hat.

   (b) Betrachten Sie den Programmrahmen zur Ausführung von privilegierten Instruktionen im User- und Kernelmode:
   `https://gitlab.hpi.de/osm-teaching/operatingsystems2020-labs/rdmsr`

   Vervollständigen Sie den Programmrahmen und untersuchen Sie das unterschiedliche Verhalten von Programmen in Kernel- und Usermode, wenn Sie:

   - Eine privilegierte CPU-Instruktion ausführen (z.B. `rdmsr` auf x86_64)
   - Einen Programmierfehler machen (z.B. ungültiger Zeigerzugriff)

   Beachten Sie dabei, dass die Ausgabe von `printk` im Linux Kernel mithilfe des Kommandozeilenbefehls `dmesg` im Usermode angezeigt werden kann.

   Moderne Linux-Systeme versuchen außerdem, einfache Fehler (*Oops*) abzufangen. Sie können dies (temporär) deaktivieren:

   ```
   sudo sh -c echo 1 > /proc/sys/kernel/panic_on_oops
   ```

[1] Auf Ihrem System vermutlich `/usr/include/asm/unistd.h`
[2] `https://j00ru.vexillium.org/syscalls/nt/32/`

6. Sie haben sich mit der Funktionsweise von Interrupts und Systemaufrufen vertraut gemacht. Beschreibne Sie im Detai, welche Schritte das Betriebssystem vollzieht, wenn Sie in der Eingabeaufforderung (`cmd.exe` auf Windows, oder die Shell auf Unix-ähnlichen Systemen) einen Befehl zur Anzeige eines Verzeichnisinhaltes ausführen (`dir` bzw. `ls`).

   Denken Sie dabei zum Beispiel darüber nach, was passiert wenn Sie als Nutzer eine Taste auf der Tastatur drücken, wie die Buchstaben auf dem Bildschirm erscheinen, und wie das Betriebssystem zum Starten des `dir` oder `ls` Programms einen neuen Prozess starten muss, der sich schließlich um die Bearbeitung Ihrer Anfrage kümmert, und eine Ausgabe erzeugt.

   Versuchen Sie, mithilfe eines Diagramms zu illustrieren an welchen Stellen in diesem Prozess das Betriebssystem Aufgaben erledigen muss, damit dieser scheinbar einfache Ablauf funktionieren kann.

7. Windows unterscheidet Programmen nach der Zugehörigkeit zu verschiedenen *Persönlichkeiten* mithilfe des Konzepts der *Subsysteme*. Beschreiben Sie, was ein Subsystem ist, welche Aufgaben im System es erfüllt, und aus welchen Komponenten es aufgebaut ist.

   Betrachten Sie den Programmrahmen zu Subsystemen:
   `https://gitlab.hpi.de/osm-teaching/operatingsystems2020-labs/environment_subsystem`

   Der Programmrahmen implementiert eine Annäherung an das Konzept der Subsysteme auf dem Betriebssystem GNU/Linux. Beschreiben Sie, wie die Komponenten des Programmrahmens funktionieren und vergleichen Sie deren Funktionalität mit den echten Windows Subsystemen.

8. (optional) Betrachten Sie den Programmrahmen zum File I/O in C:
   `https://gitlab.hpi.de/osm-teaching/operatingsystems2020-labs/ebcdic2ascii`

   Vervollständigen Sie den Programmrahmen unter Linux oder macOS, sodass Sie eine *EBCDIC*-kodierte Textdatei nach *ASCII* konvertieren können.

   Ihr Programm soll den Pfad zu einer EBCDIC-kodierten Textdatei als Kommandozeilenparameter erwarten. Diese Datei soll nach ASCII konvertiert und auf der Standardausgabe ausgegeben werden. Beachten Sie dabei insbesondere, dass EBCDIC-kodierte Dateien traditionell nur einen *Carriage Return* als Zeilenende verwenden, und keine *Line Feed*-Zeichen enthalten. Wie gehen Sie in Ihrer Ausgabe damit um?

   Achten Sie in Ihrer Implementierung außerdem auf die korrekte Behandlung von Laufzeit- und Eingabefehlern. Geben Sie im Fehlerfall sinnvolle Fehlermeldungen aus, zum Beispiel durch Verwendung der Funktionen `perror` oder `strerror`.