Betriebssysteme

Modularer/monolithischer Kernel

(Modular(Vor/Nachteile): +kleiner kernel, -deps problem Monolithisch(Vor/Nachteile): +consistency, -Bloatware

Aufgaben eines Betriebssystems

Ressourcenmanagement CPU/Festplatte/ RAM Hardware Abstraktion I/O Devices

Linux Commands

Erstellung von Prozessen

fork():
Childprozess erstellen, läuft an gleicher Zeile+1 Code aus (nach fork)
Ersetzt den Parent-Prozess mit neuem Child, Child führt ein neues programm aus.
exec(): Ersetzt den Parent-Prozess mit neuem Child, Child führt ein neues programm aus.

Boot

Booting an OS - BIOS

1. BIOS durchläuft POST (Power On Self Test)
2. Master Boot Record(MBR) wird eingelesen und ausgeführt
3. MBR wählt einen Bootloader(GRUB) aus und lädt ihn a. GRUB gestartet, Treiber aber Geräte b. Initialisiert OS Management Structure c. Erstellt System Services d. OS startet Prozess für graphical login etc.

Linux Booting - UEFI

1. System startet, EFW initialization
2. Grand Unified Bootloader (GRUB) Liest OS ein -> Device drivers + filesystem modules (initramfs)
3. Kernel (Linux OS): OS Set-up Code
4. UNIT Process (Run levels)
5. User Promt (Shell or GUI)

Bios vs UEFI

Das UEFI ist im gegensatz zum BIOS ein eigenes kleines Betriebssystem. BIOS braucht den MBR und UEFI eine UEFI Partition.
MBR wird mit GPT ersetzt für UEFI. UEFI nutzt eine Architektur unabhängige VM, dadurch lassen sich EFI binaries(EFI system partio.) (.efi) ausführen. UEFI ist modular

SystemD

- If unit1 has *Wants=unit2* as a dependency, when unit1 is run, unit2 will be run as well. But whether unit2 starts successfully does not affect unit1 running successfully.
- When unit1 has *Requires=unit2*, however, again both units will run, but if unit2 does not succeed, unit1 is also deactivated. This happens regardless of whether the processes of unit1 would otherwise have worked fine.

Run process in Linux

Runlevel	Target Units	Description
0	runlevel0.target poweroff.target	Shutdown/ power off the System
1	runlevel1.target rescue.target	Set up rescue shell
2-4	runlevel2/3/4.target multi-user.target	Set up a non-graphical multi-user system
5	runlevel5.target graphical.target	Set up graphical multi-user system
6	runlevel6.target reboot.target	Shut down and reboot the system

SystemD dependencies:

- If unit A has **Wants=B** as a dependency, when unit B is not run, it will be run. If unit B fails, unit A will start running regardless of whether B is running successfully.
- If unit A has **Requires=B** as a dependency, when unit B is not run, it will be run. If unit B fails or is stopped, unit A will be stopped.

Order:

- If unit1 has the directive **Before=unit2**, then if both units are run, unit1 will be executed fully before unit2 starts.
- If unit1 has the directive **After=unit2**, then if both units are run, unit2 will be executed fully before unit1 starts.

Unit file (ends in .service)

```
[Unit]
Description=Foo

[Service]
ExecStart=/usr/sbin/foo-daemon

[Install]
WantedBy=multi-user.target
```

Prozesse und Threads

Prozess

Ein Prozess ist ein Programm in Ausführung.
Prozessstatus: Ready, Running oder Blocked
`ps -ax` (print PID mit Status)>> D(Deep Sleeping), R(Running), S(Sleeping), T(Traced = Angehalten), Z(Zombie)
Attribute
Register, MemManagement, codedate, stacksegment, file management(welches File offen)
PID / parent PID
PID = Process ID von einem Prozess
PPID = prozess welcher den Prozess gestartet hat
Context Switch (wann):
Context Switch: State Save und dann State Restore **wenn zwischen Prozessen auf der CPU geswitched wird.** Zustand eines Prozesses wird gespeichert in das PCB geschrieben und vom neuen der letzte Stand von seinem PCB Eintrag geholt.
Mode Switch: wenn man vom User mode in den kernel mode switched und was erledigen zu können und anschließend wieder in den User mode wechselt.

Process table und PCB

Process table: Für jeden PID gibt es einen PCB Eintrag mit Informationen zum prozess.
PCB(Process Control Block): Hier stehen Informationen zu einem Prozess: State, memory, etc.

Threads

Sharing den Speicher eines Prozesses und müssen einem Prozess zugehören.
Kernel kann nur Prozess nicht die Threads davon.

Precondition:
2 Threads wollen gleiche Variable benutzen

Vorteile user lvl Threads
Vorteil: effizient, «günstig» zu erstellen
Nachteil: IO Zugriff Block würde alle Threads vom Prozess anhalten

Vorteile Kernel lvl Threads
Vorteil: Kernel kennt Threads, gut für viele I/O calls, blocking sind uninterruntable
Nachteil: Threadwechsel im Prozess kostet 2 Modeswitches, langsamer

Source Code

https://github.com/josias-r/zhaw-sem5-summaries