



**Hochschule
Augsburg** University of
Applied Sciences

Vorlesung: Betriebssysteme

Prozesse und Threads

Prof. Dr. Lothar Braun, Dr.-Ing. Volodymyr Brovkov
Sommersemester 2024

Prozesse

Threads

Programmierung: Prozesse und Threads in POSIX-Systemen

Aktivitäten im Betriebssystem

Scheduling

Prozesse

Threads

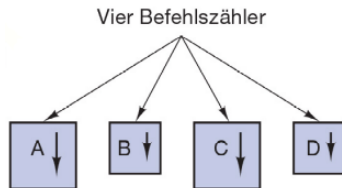
Programmierung: Prozesse und Threads in POSIX-Systemen

Aktivitäten im Betriebssystem

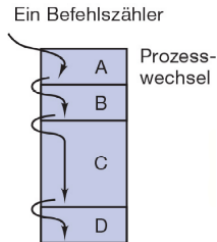
Scheduling

- **Hauptaufgabe des Betriebssystems: Ausführen von Programmen**
 - Gewünscht: Mehrere Programme gleichzeitig ausführen
- **Prozess: Instanz eines Programms in der Ausführung**
 - Speichert relevante Informationen über die Ausführung eines Programms
 - Jede Instanz eines Programms erhält eigenen Prozess
 - Prozess kann einer CPU zugeordnet und von ihr ausgeführt werden
- **Multiprogrammierung**
 - Prozess ist notwendige Voraussetzung für parallele Ausführung mehrerer Programme
 - Das Betriebssystem muss Zustand über Ablauf eines Programms speichern
 - Bei Ausführung eines anderen Programms
 - muss Zustand des aktuellen Prozesses gespeichert werden
 - muss Zustand des anderen Prozesses geladen werden

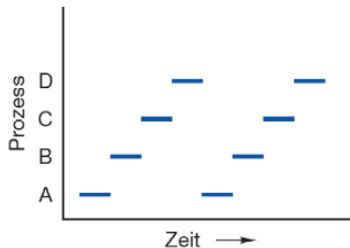
- **Jeder Prozessor (bzw. Kern) kann nur ein Programm gleichzeitig ausführen**
 - *Schneller* Wechsel zwischen Prozessen notwendig um Benutzer das Gefühl zu geben alle Programme werden gleichzeitig ausgeführt



Jeder Prozess hat eigenen
Befehlszähler



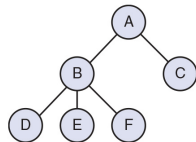
Wechsel zwischen Prozessen



Zeitlicher Ablauf aus Sicht der
Prozesse

- **Jedes Programm das auf einem Betriebssystem läuft ist ein eigener Prozess**
 - Vom Nutzer gestartete Programme
 - Programme die im Rahmen des Bootvorgang gestartet werden
 - Prozesse die Aufgaben für das Betriebssystem übernehmen
- **Zeitpunkte zu denen Prozesse erzeugt werden:**
 - Systemstart
 - Aufruf von Systemaufrufen zur Erzeugung neuer Prozesse von existierendem Prozess
 - Nutzer startet ein neues Programm
 - Start eines Batch-Jobs

- **Ein Prozess wird immer von einem anderen Prozess gestartet**
 - Ausnahme: Start des *init*-Prozess durch Betriebssystem
- **Jeder Prozess hat einen *Parent*-Prozess**
 - Der *Parent*-Prozess ist der Prozess, der den aktuellen Prozess gestartet hat
 - Der neu gestartete Prozess wird *Child*-Prozess genannt
- **Prozesse eines Systems bilden einen *Prozess-Baum***



Beispiel eines Prozessbaums

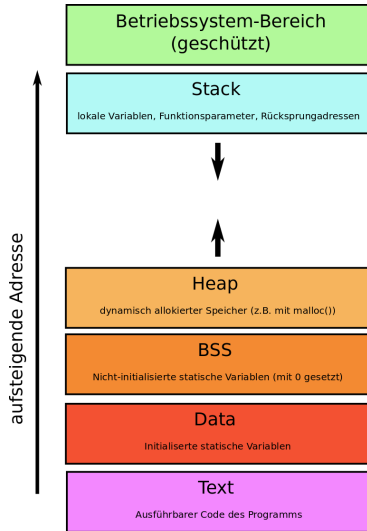
$\frac{7}{7} \cdot \frac{7}{7}$

Ausschnitt des Prozess-Baumes unter Ubuntu

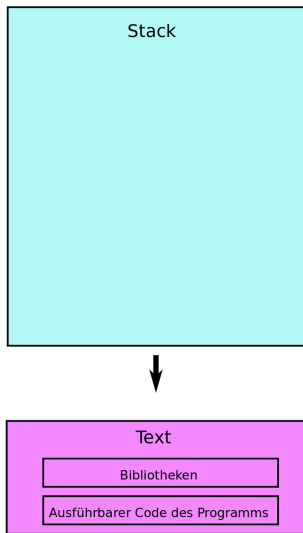
Ausschnitt des Prozess-Baumes unter Windows

- **Verschiedene Zustände oder Aktionen können zum Ende von Prozessen führen**
- **Häufige Gründe für das Ende von Prozessen**
 - Normales Programmende durch das Programm selbst (freiwillig)
 - Fehlerzustand im Programm nach dem sich das Programm selbst beendet (freiwillig)
 - Fehlerzustand im Programm mit Programmabbruch durch Betriebssystem (unfreiwillig)
 - Abbruch durch ein externes Programm (unfreiwillig)

- **Betriebssystem generiert neuen Prozess**
 - Anlegen der Verwaltungsinformationen im Betriebssystem: *Process Control Block*
 - Anlegen eines eigenen Adressraums für den Prozess
- **Laden des Programm-Codes und Initialisierung des Speichers**
 - Programm-Datei enthält Informationen über verschiedene Speicherbereiche
 - Linux/Unix: *Executable and Linking Format* (ELF)
 - Windows: *Portable Executable* (PE)
- **Übergabe der Kontrolle an den neuen Prozess**
 - Betriebssystem schaltet in den Benutzer-Modus
 - *Program Counter* (PC) wird an den *Entry Point* des Programms gesetzt



- **Jeder Prozess hat eigenen *virtuellen Adressraum***
 - Prozess kann Daten anderer Prozesse nicht "sehen"
 - Details in späterer Vorlesung
- **Relevante Speicherbereiche**
 - Text Segment
 - Heap Segment
 - Stack Segment
 - Betriebssystem-Bereich (geschützt)

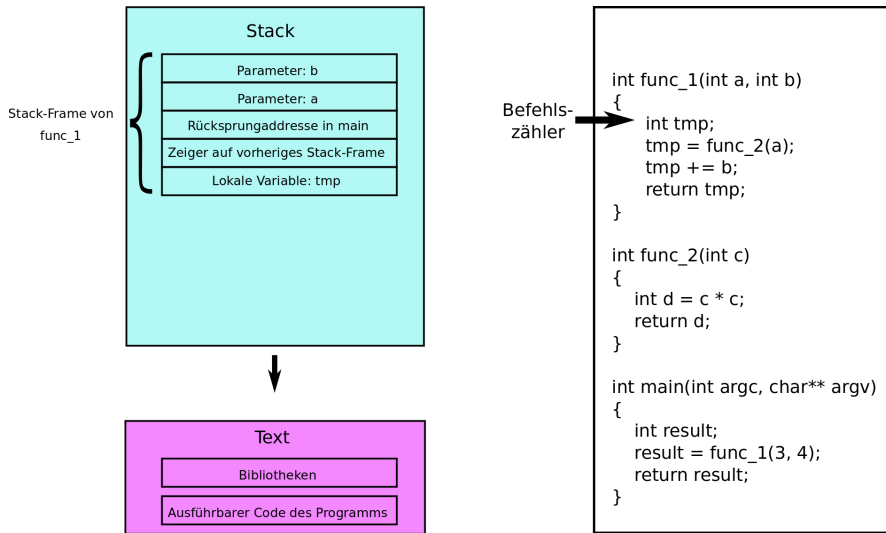


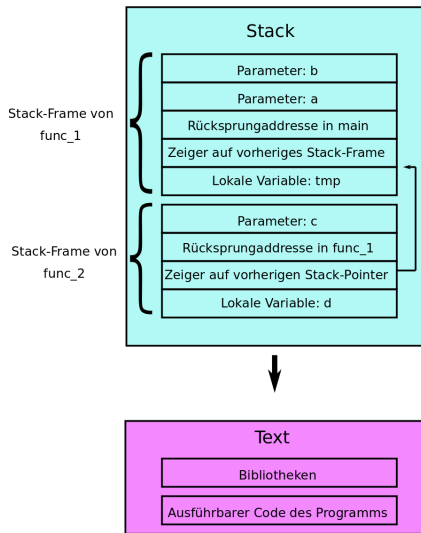
Befehls-
zähler

```
int func_1(int a, int b)
{
    int tmp;
    tmp = func_2(a);
    tmp += b;
    return tmp;
}

int func_2(int c)
{
    int d = c * c;
    return d;
}

int main(int argc, char** argv)
{
    int result;
    result = func_1(3, 4);
    return result;
}
```





Befehls-
zähler

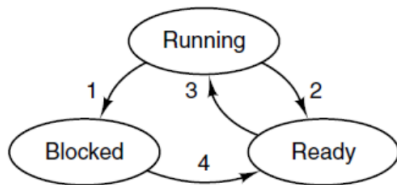


```
int func_1(int a, int b)
{
    int tmp;
    tmp = func_2(a);
    tmp += b;
    return tmp;
}

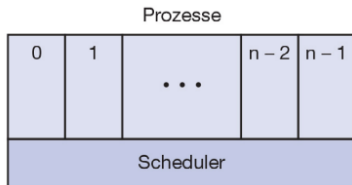
int func_2(int c)
{
    int d = c * c;
    return d;
}

int main(int argc, char** argv)
{
    int result;
    result = func_1(3, 4);
    return result;
}
```

- Prozesse können folgende Zustände einnehmen:

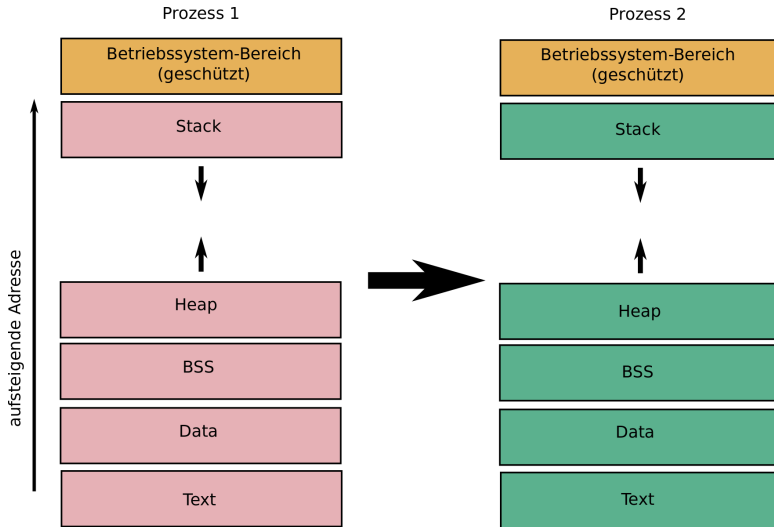


- *Running*: Prozess wird aktuell vom Prozessor ausgeführt
- *Blocked*: Ein Prozess wartet auf das Ende einer Ein-/Ausgabe Operation
- *Ready*: Der Prozess ist bereit zu laufen und wartet auf Aktivierung



- Der **Scheduler** eines Betriebssystems trifft die Entscheidung welcher Prozess läuft
- Grundlagen des **Scheduler**
 - Low-Level Komponente eines jeden Betriebssystems
 - Wird in regelmäßigen Abständen aufgerufen und prüft welcher Prozess laufen soll
 - Wird nach *Interrupts* aufgerufen um zu prüfen welcher Prozess laufen soll
 - Wählt nächsten Prozess aus Prozessen im Zustand *Ready* aus
 - Viele Scheduler-Algorithmen existieren und werden später diskutiert

- Das Umschalten zwischen mehreren Prozessen wird *Kontextwechsel* genannt.
- **Kontextwechsel bei Kontrolle der CPU durch Betriebssystem**
 - Details im Abschnitt *Aktivitäten im Betriebssystem*
- **Daten über Prozesse müssen bei *Kontextwechsel* gespeichert und geladen werden**
 - *Scheduler* verwendet dafür eine *Prozesstabelle*
 - Ein Eintrag pro Prozess: *Prozesskontrollblock* bzw. *Process Control Block (PCB)*





- ***PID - Process ID, PPID - Parent PID***
- **Zustand des Prozesses**
 - Register der CPU, aktueller Programmzähler
- **Informationen über Speicher**
 - Geladener Programmcode
 - Stack, Heap, Shared Memory
 - Spezialbereiche für Ein-/Ausgabegeräte
- **Informationen über verwendete Betriebssystem-Ressourcen**
 - Geöffnete Dateien, Netzwerkverbindungen, usw.
- **Interne Management-Informationen für das Betriebssystem**
 - Priorität des Prozesses, zugeordnete Warteschlange, Benutzer-ID, ...
- **Linux: struct task_struct ist Umsetzung des Konzepts unter Linux:**
https://elixir.bootlin.com/linux/latest/A/ident/task_struct

Prozesse

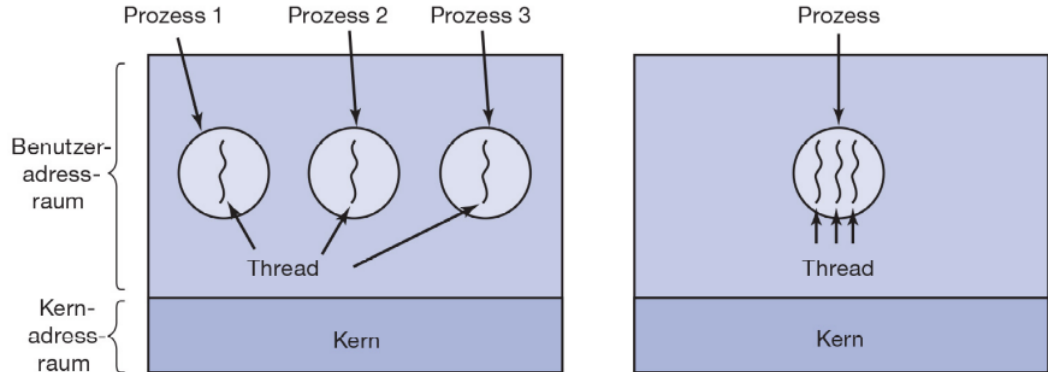
Threads

Programmierung: Prozesse und Threads in POSIX-Systemen

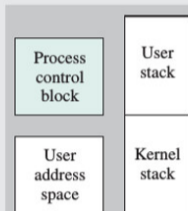
Aktivitäten im Betriebssystem

Scheduling

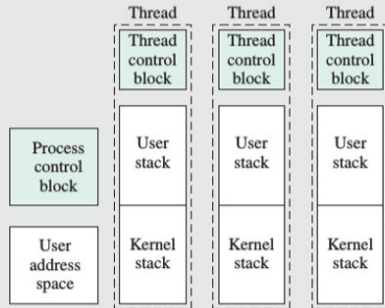
- **Prozess: Ausführungs-Kontext und Betriebsmittel**
 - Betriebsmittel:
 - Eigener Adressraum, geöffnete Dateien, Netzwerkverbindungen
 - Ausführungskontext:
 - Befehlszähler, Prozessor-Register, ...
 - Stack mit Variablen und Parametern von Funktionen sowie Rücksprungadressen
- **Thread: Nur Ausführungs-Kontext, keine eigenen Betriebsmittel**
 - Ausführungskontext jedes Threads ist eigener Ausführungs-Kontext
 - Alle Betriebsmittel sind geteilt
- **Wesentlicher Unterschied:**
 - Prozesse können ohne Freigabe nicht auf Daten anderer Prozesse zugreifen
 - Threads teilen sich alle relevanten Daten (insbesondere Speicher)



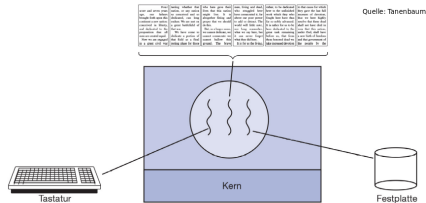
Single-threaded
process model



Multithreaded
process model



- **Kommunikation zwischen Threads und Prozessen**
 - Threads teilen sich Speicher und können leicht Daten austauschen
 - Prozesse können nicht auf Daten anderer Prozesse zugreifen:
Interprozess-Kommunikation notwendig
- **Geschwindigkeit von Start und Ende**
 - Prozesse: Neue Betriebsmittel müssen angelegt werden → langsam
 - Threads: Nur anlegen eines neuen Ausführungs-Kontextes → schnell
- **Geschwindigkeit des Kontextwechsels**
 - Prozesse: Wechsel der Betriebsmittel inklusive Cache-Invalidierung → langsam
 - Threads: Nur Wechsel der Stacks und *Program Counter*/Register → schnell
- **Schutz vor Abstürzen**
 - Prozesse: Absturz eines Prozesses hat keine Auswirkungen auf anderen Prozess
 - Threads: Beendigung eines Threads durch Betriebssystem → alle Threads des Prozesses beendet



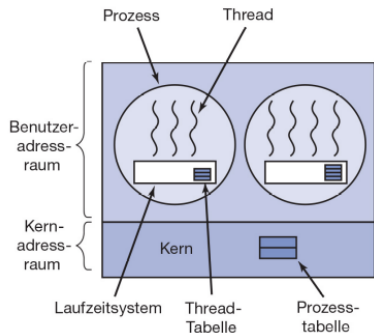
Quelle: Tanenbaum

Typischer Anwendungsfall für Threads

- **Aufteilung von Arbeit im Vordergrund und Hintergrund**
- **Bearbeitung von mehreren in sich abgeschlossenen Aufgaben, z.B. Webservern**
- **Auslastung mehrerer Prozessoren bei aufwändigen Rechenaufgaben**

- **Implementierung in der Anwendung**

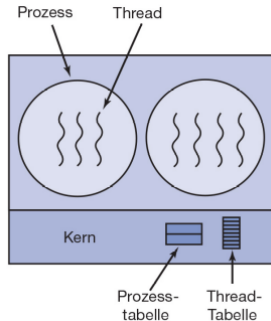
- Thread-Stack in der Anwendung
- Scheduling durch die Anwendung



Implementierung von Threads in der Anwendung

- **Implementierung im Betriebssystem**

- Verwaltung durch Betriebssystem
- Thread-Stacks im Betriebssystem



Implementierung von Threads im Betriebssystem

- **Implementierung in der Anwendung**

- Kann auf Betriebssystemen umgesetzt werden, die keine Threads unterstützen
- Programmierung muss umgestellt werden: keine blockierende Systemaufrufe
- Große Anzahl an langsamen nicht-blockierende Systemaufrufen
- Threads werden nicht durch Timer-Interrupts automatisch unterbrochen

- **Implementierung im Betriebssystem**

- Betriebssystem übernimmt Scheduling, bei blockieren eines Threads oder Interrupt
- Einfacherer Programmierung auf Anwendungsebene

Prozesse

Threads

Programmierung: Prozesse und Threads in POSIX-Systemen

Aktivitäten im Betriebssystem

Scheduling

- **Prozesse können jederzeit Prozesse und Threads starten**
 - Prozesse müssen immer über das Betriebssystem gestartet werden → *Systemaufruf*
 - Start von Threads durch Betriebssystem oder in der Anwendung werden → *Bibliothek*
- **Systemaufrufe / Interfaces können frei durch das Betriebssystem definiert werden**
- **Standard-Interfaces existieren: POSIX**

- **Funktion: `pid_t fork();`**
 - `fork()` erzeugt einen neuen Prozess. Neuer Prozess ist Kopie des alten Prozesses
 - Rückgabewert der Funktion:
 - Im *Parent*: PID des neuen Prozesses
 - Im *Child*: Rückgabewert ist 0
- **Funktion: `int execve(const char* filename, char *const argv[], char *const envp[]);`**
 - Startet ein Programm im aktuellen Prozess
 - Altes Programm wird dabei überschrieben und Kontext wird entfernt
- **Funktion: `pid_t wait(int* wstatus);`**
 - Warten bis ein beliebiger Kind-Prozess beendet wird

- Weitere Funktionen rund um Prozess-Erzeugung existieren
- Dokumentation findet sich in *man pages* auf Linux-Systemen

```
FORK(2)                                Linux Programmer's Manual                                FORK(2)

NAME
    fork - create a child process

SYNOPSIS
    #include <sys/types.h>
    #include <unistd.h>

    pid_t fork(void);

DESCRIPTION
    fork() creates a new process by duplicating the calling process. The new process is referred to as the child process. The calling process is referred to as the parent process.

    The child process and the parent process run in separate memory spaces. At the time of fork() both memory spaces have the same content. Memory writes, file mappings (mmap(2)), and unmappings (munmap(2)) performed by one of the processes do not affect the other.

    The child process is an exact duplicate of the parent process except for the following points:

    * The child has its own unique process ID, and this PID does not match the ID of any existing process group (setpgid(2)) or session.

    * The child's parent process ID is the same as the parent's process ID.
```

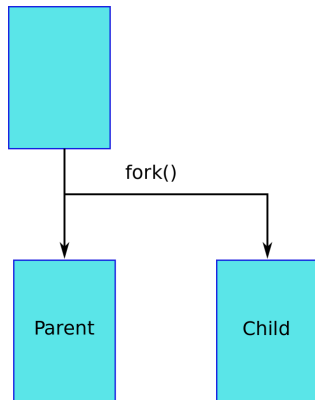
Aufruf des Befehls: *man fork*

```
#include <unistd.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <stdio.h>
#include <string.h>

int main(int argc, char** argv)
{
    pid_t pid; int status;

    pid = fork();
    if (pid == 0) {
        printf("Child: This is the child process!\n");
    } else if (pid > 0) {
        printf("Parent: This is the parent process!\n");
        printf("Parent: Waiting for child to terminate!\n");
        waitpid(pid, &status, 0);
        printf("Parent: Child has terminated!\n");
    } else {
        fprintf(stderr, "Error during fork: %s\n",
                strerror(errno));
    }
    return 0;
}
```

Ablauf bei Fork



- **POSIX Interface für Threads ist recht umfassend. Wichtigste Funktionen:**
- **`int pthread_create(pthread_t *thread, const pthread_attr_t *attr, void *(*start_routine) (void *), void *arg);`**
 - Startet einen neuen Thread und führt die Funktion *start_routine* aus
 - *start_routine* bekommt einen Pointer auf das Argument *em *arg* beliebiger Daten
- **`int pthread_join(pthread_t thread, void **retval);`**
 - Wartet auf das Ende eines Threads
 - Rückgabewerte des Threads an der Adresse **retval*
- **Deklaration einer Thread-Funktion: `void *thread_function(void* arg);`**

```
#include <stdio.h>
#include <pthread.h>

void* thread_func(void* data) {
    printf("Thread: I'm a thread!\n");
}

int main(int argc, char** argv)
{
    pthread_t thread_id;

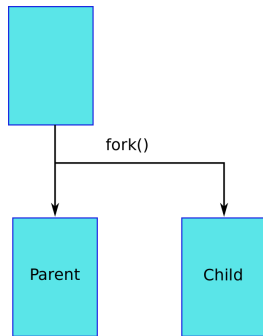
    printf("Main Thread: Starting Thread!\n");
    if (-1 == pthread_create(&thread_id, NULL,
                           &thread_func, NULL)) {
        fprintf(stderr, "Error creating thread!\n");
        return -1;
    }
    printf("Main thread: Started thread\n");
    printf("Main thread: waiting for thread to finish ...\n");
    pthread_join(thread_id, NULL);
    printf("Main thread: Thread finished!\n");

    return 0;
}
```

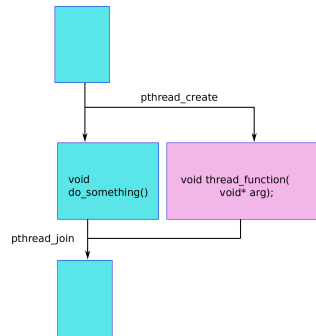
Kompilierung:

```
gcc -pthread thread-example.c
```

- **Prozesse sind eine Kopie des Programms: laufen unabhängig voneinander**
- **Threads sind Aufgaben die innerhalb eines Prozesses laufen**



Prozesse



Threads

Prozesse

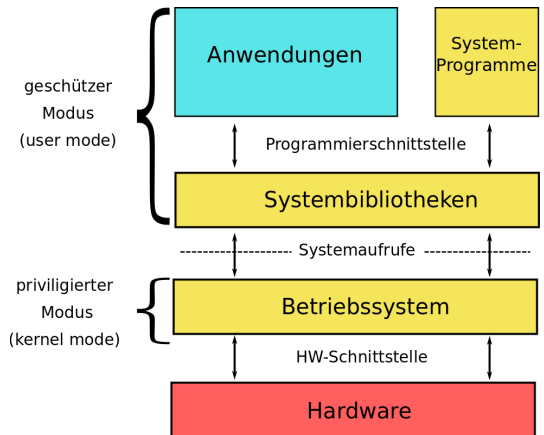
Threads

Programmierung: Prozesse und Threads in POSIX-Systemen

Aktivitäten im Betriebssystem

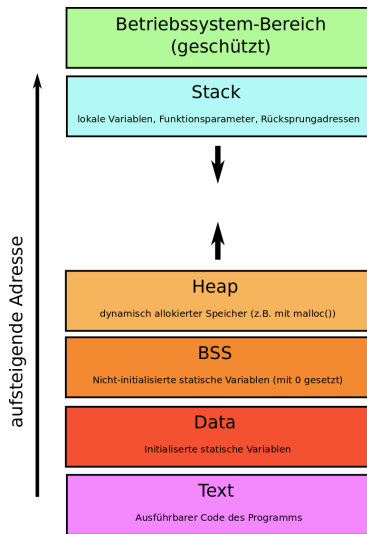
Scheduling

- **Ausführen von Anwendungen**
- **Ausführen von Systemaufrufen**
- **Behandlung von Ausnahmen (Exception Handling)**
- **Betriebssystem-interne Verwaltungsaufgaben**

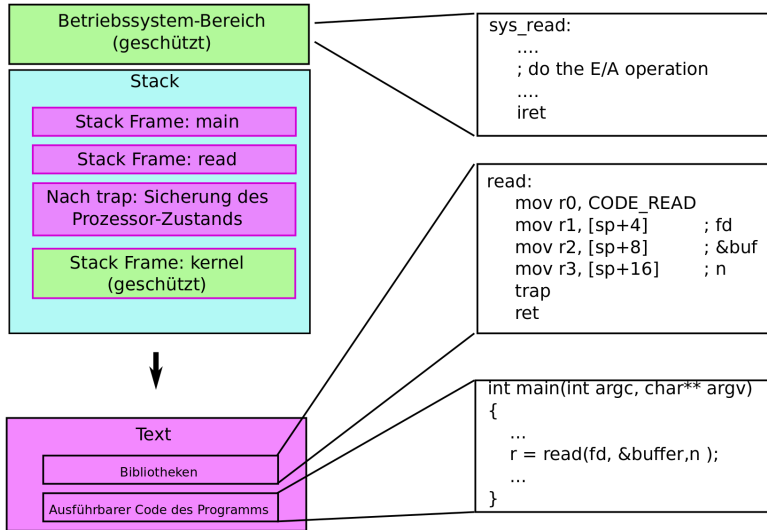


- **Privilegierter Modus / Kernmodus**
 - Prinzipiell unbeschränkter Zugriff auf alle Ressourcen
 - Exceptions können auftreten wenn Ressourcen nicht vorhanden sind
- **Benutzer-Modus**
 - Eingeschränkter Modus der nur erlaubte Aktionen durchführen darf
 - Alle Benutzerprogramme laufen in diesem Modus
 - Auch Anwendungen des Administrators / root
 - Betriebssystem erlaubt nur eingeschränkten Zugriff auf Ressourcen:
 - Speicher
 - Ein/Ausgabe-Geräte
 - Instruktionen im Prozessor
 - Bei Zuwiderhandlung: Exception
- **Hardware unterstützt Trennung der beiden Modi**

- **Betriebssystem übernimmt Aufgaben im Auftrag eines Benutzerprogramms**
- **Problem:**
 - Benutzerprogramm hat eingeschränkte Rechte
 - Betriebssystem benötigt vollen Zugriff auf Hardware
- **Lösung:**
 - Spezieller Prozessor-Befehl schaltet Modus um
 - Führt Code an einer bestimmten Adresse des Betriebssystems aus
 - x86-Instruktion: *int*, *trap*

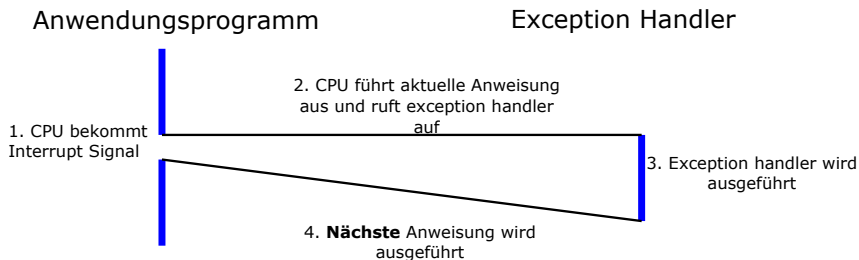


- **Programm möchte Systemaufruf ausführen**
 - `open('/root/.ssh/id_rsa', 'r');`
- **Programm `open()` aus Systembibliothek auf**
 - `open()` legt Argumente für Funktion auf den Stack
 - Führt Assembler-Instruktion *trap* aus
- **Kontrolle und Code-Pfad geht ins Betriebssystem über**
 - Betriebssystem prüft ob Anwendungsprozess Aktion durchführen kann
 - Führt Aktion mit Betriebssystem-Rechten aus und produziert Ergebnis
 - Gibt Betriebssystemrechte ab und kehrt zu Prozess zurück

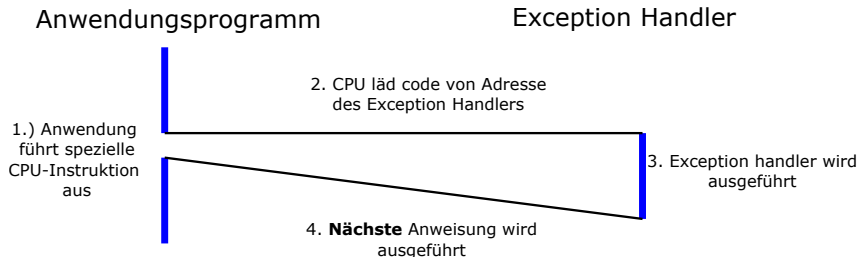


- **Ausnahmen / Exceptions unterbrechen den Programmfluss**
 - CPU muss sich um ein Ereignis kümmern und diese behandeln
 - Nicht verwandt mit Exceptions in Java/C++/...
- **Es existieren vier Arten von Exceptions**
 - Interrupt
 - Trap
 - Fault
 - Abort

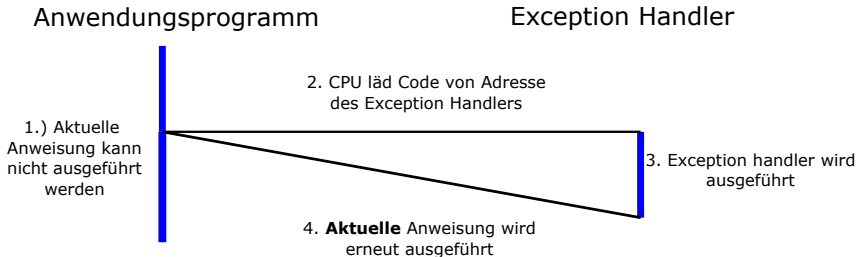
- Treten *asynchron* auf wenn ein Gerät sich bei der CPU meldet
- Beispiel: Tastatureingabe, Daten von Festplatte liegen vor, ...



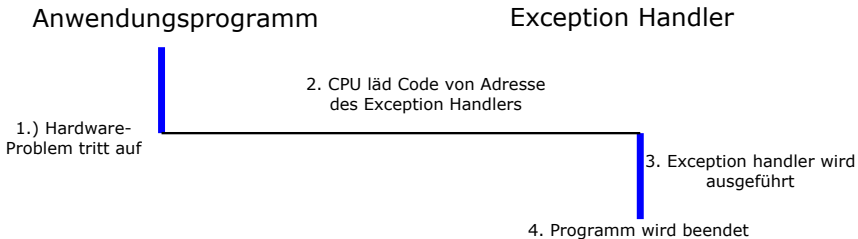
- Treten *synchron* auf, wenn eine Anwendung bestimmte CPU-Instruktion ausführt (z.B. *int 80*)
- Beispiele: Anwendung braucht mehr Speicher, Anwendung möchte Ein-/Ausgabe triggern ...



- Treten *synchron* auf, wenn eine Anwendung ein (potentiell) behebbares Problem aufwirft
- **Beispiele:**
 - Anwendung greift auf illegal Speicher zu (Segmentation Fault)
 - Anwendung greift auf Speicher zu, der nicht im RAM ist (Page Fault)



- Treten *synchron* auf, wenn CPU während Ausführung ein (nicht-behebbares) Problem feststellt
- Beispiele:
 - CPU stellt Speicher- oder Busfehler fest (Machine Check)
 - Exception Handler wurde nicht gefunden oder Exception im Exception Handler (Double Fault)



Typ	Grund	Synchronität	Folgeaktion
Interrupt	Signal von Ein/Ausgabegerät	Asynchron	Nächste Anweisung
Trap	Gewünschte von Anwendung	Synchron	Nächste Anweisung
Fault	(Potentiell) behebbarer Fehler	Synchron	Wiederholung aktuelle Anweisung (oder Programmabbruch)
Abort	Nicht-behebbarer Fehler	Synchron	Programmabbruch (und evtl. Kernel Panic)

- **CPU-Programmfluss wird unterbrochen (Interrupt, Trap, Fault, Abort)**
 - Sichern des aktuellen Befehlzählers und Prozessor-Statusflags
 - Wechsel in Kern-Modus (privilegierter Modus)
 - (Potentiell) Sperren für weitere Unterbrechungen
- **CPU untersucht Interrupt Table nach Exception Handler**
 - CPU kennt Interrupt-Nummer
 - Interrupt-Tabelle enthält Adressen der *Interrupt Service Routines* (ISR)
 - ISR entspricht einem Exception Handler für eine spezielle Exception
 - Struktur der Tabelle durch CPU-Hersteller vorgegeben
- x86 (32 oder 64 Bit): Adresse in Register IDTR
- x86 (Real Mode, 16 Bit Modus): *Interrupt Vector Table*
- x86 (Protected Mode (32 Bit), Long Mode (64 Bit)): *Interrupt Descriptor Table*
- CPU liest Adresse des ISR und springt zu ISR

- **Aufruf der *Interrupt Service Routine* (ISR)**
 - Umschalten auf Kern-Stack
 - Sichern der Prozessor-Register
 - **Behandlung des Interrupts durch den *Interrupt Handler***
 - Zurücksetzen der Prozessor-Register
 - Umschalten auf Prozess-Stack
- **Rückkehr in den alten Zustand (wenn möglich)**
 - Rücksetzen der Prozessor-Flags
 - Rückschalten in Benutzer-Modus
 - Rückkehr an alte Programmcode-Stelle mit altem Befehlszähler

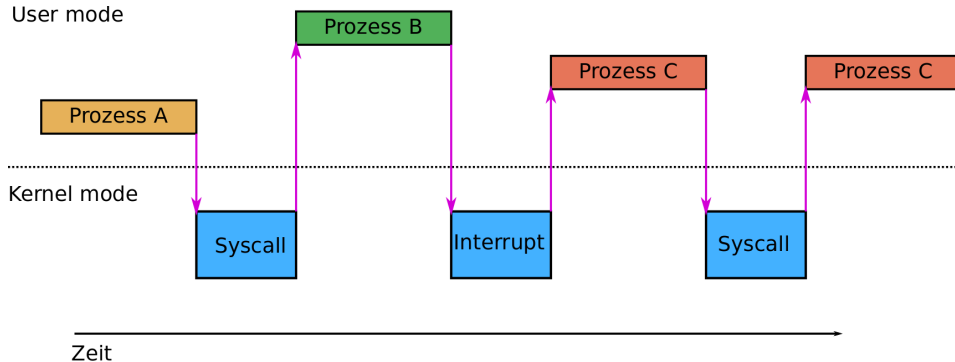
- **Manche Betriebssystem-Aufgaben können *später* erledigt werden**
 - Beispiel: Swapping
- **Kernel Threads können für Nebenläufigkeit im Kernel eingesetzt werden**
 - Threads die keinem Benutzer-Prozess angehören
 - Laufen mit Priorität des Betriebssystems

```
$ ps aux | head -5
USER      PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root         1  0.0  0.0 168580  9540 ?        Ss   Mär20    2:52 /sbin/init splash
root         2  0.0  0.0     0      0 ?        S    Mär20    0:00 [kthreadd]
root         3  0.0  0.0     0      0 ?        I<   Mär20    0:00 [rcu_gp]
root         4  0.0  0.0     0      0 ?        I<   Mär20    0:00 [rcu_par_gp]
```

Prozess-Ansicht Ubuntu

- **Wenn Betriebssystem Kontrolle übernimmt, kann ein Kontextwechsel stattfinden**
- **Mögliche Auslöser für Kontextwechsel**
 - Wenn aktueller Task (= Prozess oder Thread) blockiert
 - Wenn Blockade des aktuellen Tasks endet
 - Wenn Task neuen Task startet oder alter Task beendet
 - Beim Auftreten eines Interrupts
 - Interrupt durch Ein-/Ausgabegerät
 - Timer-Ablauf

- **Betriebssysteme können Multitasking mit zwei Konzepten realisieren**
- **Kooperatives Multitasking**
 - Task kann nur freiwillig verdrängt werden: Systemaufruf oder *yield*-Operation
 - Vorteil: Einfach zu realisieren, keine Synchronisation notwendig
 - Nachteil: Lang laufender Prozess kann System blockieren
- **Verdrängendes Multitasking**
 - Prozess kann jederzeit durch Timer-Interrupt verdrängt werden
 - Vorteil: Kein blockieren durch Langläufer
 - Nachteil: Synchronisation notwendig



- **Entscheidung darüber welcher Prozess läuft abhängig von Scheduling-Strategie**

Prozesse

Threads

Programmierung: Prozesse und Threads in POSIX-Systemen

Aktivitäten im Betriebssystem

Scheduling

- **Aufgabe:**
 - Auswahl des Prozesses (bzw. Threads) der aktuell ausgeführt werden soll
- **Ziele des Auswahlprozesses:**
 - Abhängig von Hauptaufgabe des Systems
 - Stapelverarbeitung hat andere Anforderungen als interaktive Systeme

- **Alle Systeme**

- Fairness - jeder Prozess bekommt einen fairen Anteil
- Policy Enforcement - die gewählte Strategie wird umgesetzt
- Balance - Alle Ressourcen werden bestmöglich ausgenutzt

- **Stapelverarbeitung**

- Durchsatz - Maximieren der Jobs pro Stunde
- Durchlaufzeit - Minimieren der Zeit von Job eingegeben bis Ende
- CPU - CPU ist immer ausgelastet

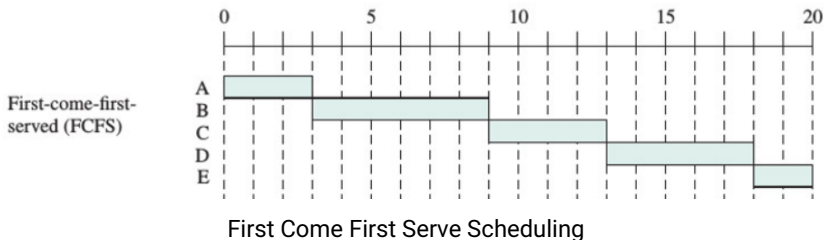
- **Interaktive Systeme**

- Antwortzeit - Schnelle Antwort an Anfragen
- Proportionalität - Erwartungshaltung

- **Echtzeitsysteme**

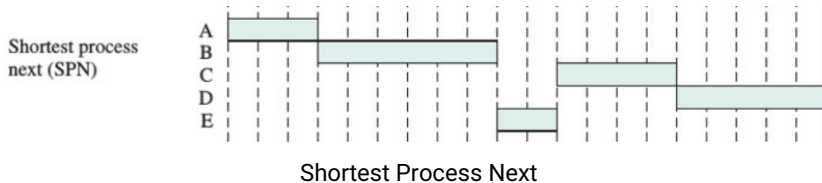
- Deadlines einhalten - Datenverlust vermeiden
- Vorhersagbarkeit - Qualitätseinbußen in Multimediasystemen vermeiden

- **Keine Verdrängung nach Zeitablauf**
 - Beispiel von Jobs: A, B, C, D, E (Startzeit: 0s, 2s, 4s, 6s, 8s)

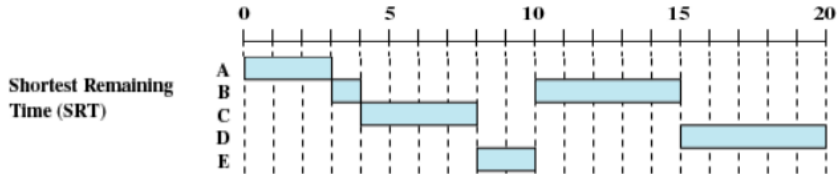


- **Keine Verdrängung nach Zeitablauf**

- Beispiel von Jobs: A, B, C, D, E (Startzeit: 0s, 2s, 4s, 6s, 8s)

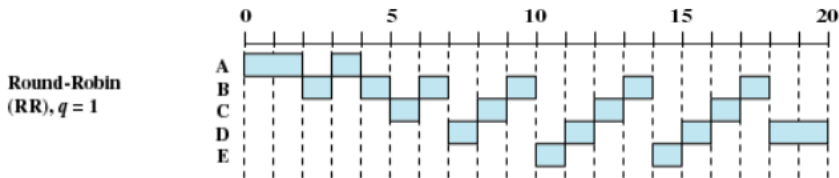


- **Mit Verdrängung nach Ablauf gewisser Zeit**
 - Beispiel von Jobs: A, B, C, D, E (Startzeit: 0s, 2s, 4s, 6s, 8s)



Shortest Remaining Time

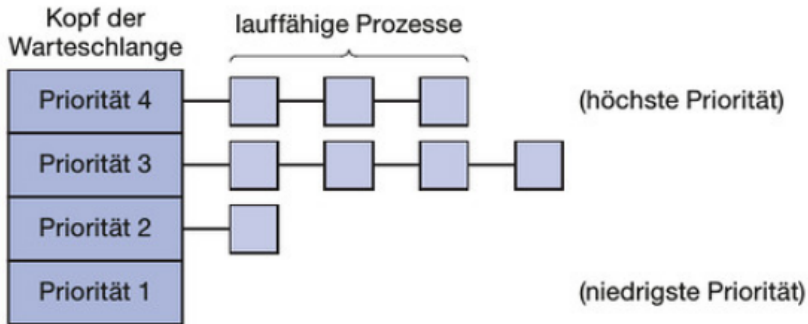
- **Mit Verdrängung nach Ablauf gewisser Zeit**
 - Beispiel von Jobs: A, B, C, D, E (Startzeit: 0s, 2s, 4s, 6s, 8s)



Round-Robin Scheduling

- **Frage: Wie lange werden Zeitscheiben gemacht?**
 - Zu kurz: Zu häufige Kontextwechsel
 - Zu lange: Lange Antwortzeiten
 - Oft gewählt: zwischen 20ms und 50ms

Quelle: Tanenbaum



- **Verdrängendes Scheduling mit Prioritäten**

- Potentielles Problem: *Verhungern* von Prozessen mit niedriger Priorität
- Verbesserung: dynamische Prioritäten

- **Lotterie Scheduling**
 - Jeder Task bekommt Anzahl Lose und Lose werden zufällig gezogen
 - Priorisierung: Tasks mit hoher Priorität bekommen mehr Lose
- **Fair-Share Scheduling**
 - Rechenzeit wird zu gleichen Teilen an Tasks verteilt
- **Feedback Scheduling**
 - Tasks die noch nicht lange laufen werden bevorzugt

- **Anforderung: Antwortzeit muss garantiert werden**
 - **Periodische Vorgänge:** Ereignisse treten periodisch auf. Reaktion bis Ende der Periode
 - **Aperiodische Vorgänge:** Ereignisse treten jederzeit auf. Reaktion in bestimmter Zeit notwendig
- **Typen von Echtzeit-Systemen**
 - **Harte Echtzeit:** Deadline darf nicht verpasst werden (z.B. ABS, Airbagsteuerung im Auto)
 - **Weiche Echtzeit:** Seltenes verpassen der Frist akzeptabel (z.B. Audio/Video-Player)

- **Nicht jedes Scheduling-Problem für Echtzeit-Systeme ist lösbar**
 - Problem: wenn mehr Arbeit anfällt als Rechenzeit zur Verfügung steht
- **Periodische Echtzeit-Systeme sind nur unter bestimmten Bedingungen realisierbar**
 - Sei m die Anzahl der periodischen Ereignisse
 - Ein Ereignis i trete alle P_i Sekunden auf und benötigt C_i Sekunden zur Verarbeitung
 - Dann ist das System **zeitlich verwaltbar** wenn gilt:

$$\sum_{i=0}^m \frac{C_i}{P_i} \leq 1$$

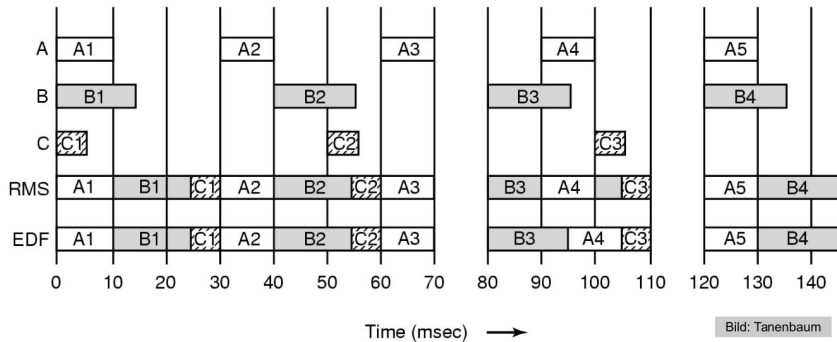
- **Bei aperiodischen Ereignissen müssen weitere Bedingungen erfüllt sein**

- **Rate-Monotonic Scheduling (RMS)**
 - Statisches Verfahren welches auf Prioritäten basiert
 - Priorität wird anhand der Frequenz der Ereignisse festgelegt
- **Earliest Deadline First (EDF)**
 - Dynamisches Verfahren bei der Prozessor immer Prozess mit der kürzesten Deadline wählt
 - Deadline muss für jeden Prozess bekannt sein

Beispiel Scheduling periodischer Ereignisse (1)



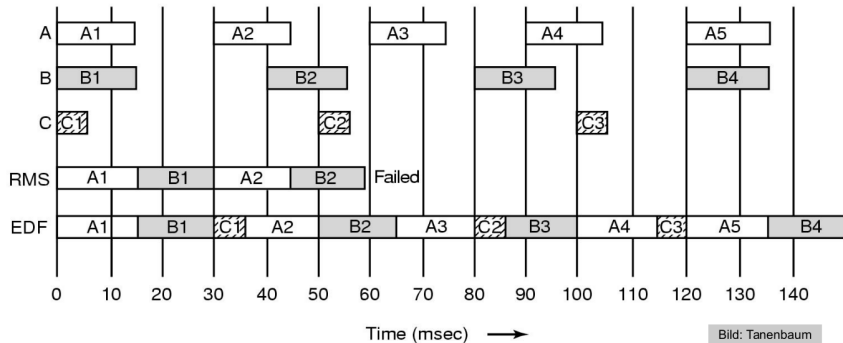
A: 10ms alle 30ms B: 15ms alle 40ms C: 5ms alle 50ms



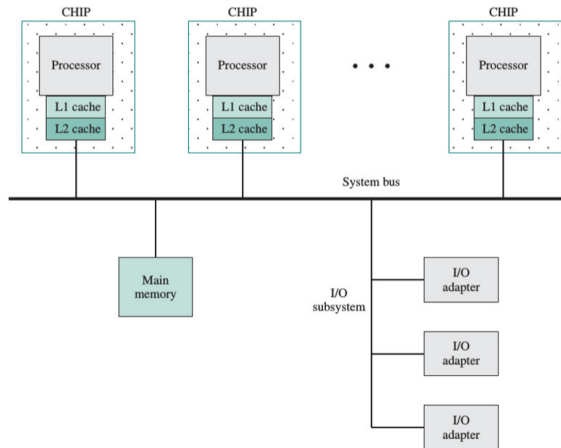
Beispiel Scheduling periodischer Ereignisse (2)



A: 15ms alle 30ms B: 15ms alle 40ms C: 5ms alle 50ms



- **Erweiterte Anforderungen an Scheduling**
- **Verteilung der Last**
- **Steuerung der Prozessorzuteilung**
 - Zentral? Verteilt?
- **Abhängigkeit: Speicherarchitektur**
 - Symmetrisch
 - NUMA (non-uniform memory architecture)



Architektur Symmetrischer Multiprozessor-Systeme

- **Globale Warteschlangen, verteiltes Scheduling**

- System-Funktionen gleichwertig auf alle Prozessoren verteilt
- Wenn Prozessor frei, wählt er nächsten Prozess aus Schlange
- **Nachteile:**
 - Synchronisation von Warteschlange notwendig
 - Potentiell häufiger Wechsel zwischen Prozessoren für Prozesse

- **Globale Warteschlangen, Master/Slave-Architektur**

- Systemfunktionen des Betriebssystems laufen auf einem Prozessor
- **Nachteil:** Überlastung des Master bremst ganzes System

- **Lokale Verwaltung**

- Jeder Prozessor verwaltet eigene Prozesse
- In größeren Zeitabschnitten Migration zum Lastenaustausch
- Prozesse verbleiben lange auf einem Prozessor

Fragen?