



---

# Betriebssysteme I

WS 2019/2020

Roland Wismüller  
Betriebssysteme / verteilte Systeme  
roland.wismueller@uni-siegen.de  
Tel.: 0271/740-4050, Büro: H-B 8404

Stand: 14. November 2019



---

# Betriebssysteme I

WS 2019/2020

31.10.2019

Roland Wismüller  
Betriebssysteme / verteilte Systeme  
roland.wismueller@uni-siegen.de  
Tel.: 0271/740-4050, Büro: H-B 8404

Stand: 14. November 2019

# Betriebssysteme I

WS 2019/2020

## 2 Prozesse und Threads

## 2 Prozesse und Threads ...



- ➔ Begriffsklärung
- ➔ Thread-/Prozeßmodell und -zustände
- ➔ Implementierung von Prozessen und Threads
- ➔ Implementierungsvarianten für Threads
- ➔ Thread-Schnittstellen

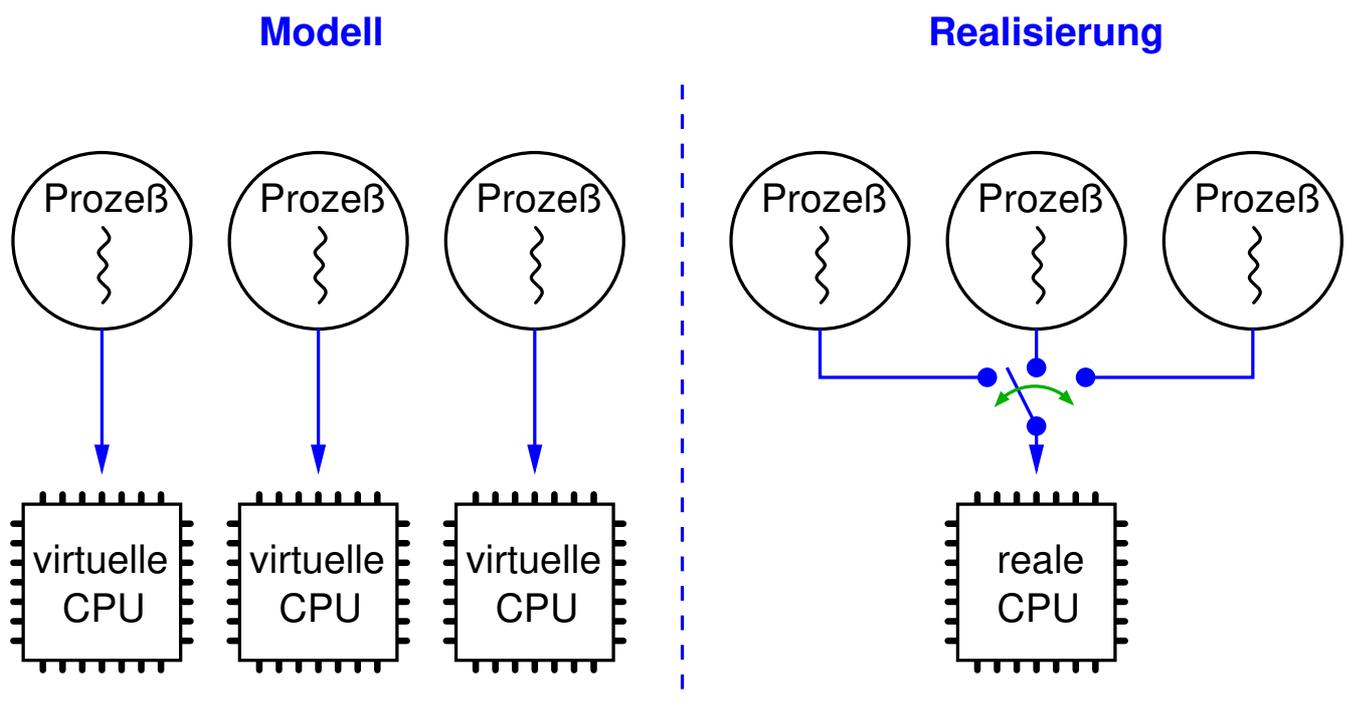
- ➔ Tanenbaum 2.1 - 2.2
- ➔ Stallings 3.1 - 3.2

## 2.1 Begriffsklärung



- ➔ Ein Prozeß ist ein Programm in Ausführung
- ➔ Wunsch: ein Rechner soll mehrere Programme „gleichzeitig“ ausführen können
- ➔ Konzeptuell: jeder Prozeß
  - wird durch eine eigene, virtuelle CPU ausgeführt
    - **nebenläufige** (quasi-parallele) Abarbeitung der Prozesse
    - hat seinen eigenen (virtuellen) Adreßraum („Speicher“, ↗ 6)
- ➔ Real: (jede) CPU schaltet zwischen den Prozessen hin und her
  - **Multiprogrammierung, Mehrprogrammbetrieb**
  - Umschalten durch Umladen der CPU-Register (incl. PC)
  - Beachte: Annahmen über die Geschwindigkeit der Ausführung sind nicht zulässig

## 2.1 Begriffsklärung ...



## 2.1 Begriffsklärung ...



### Ein (klassischer) Prozeß besitzt zwei Aspekte:

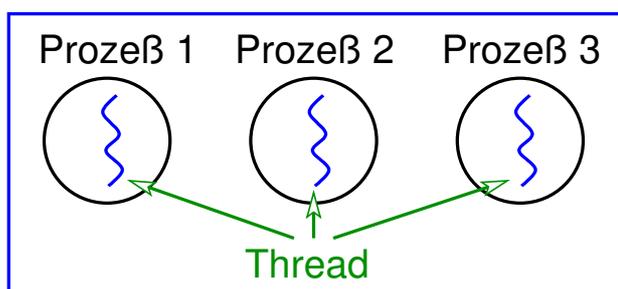
- ➔ Einheit des Ressourcenbesitzes
  - eigener (virtueller) Adreßraum
  - allgemein: Besitz / Kontrolle von Ressourcen (Hauptspeicher, Dateien, E/A-Geräte, ...)
  - BS übt Schutzfunktion aus
- ➔ Einheit der Ablaufplanung / Ausführung
  - Ausführung folgt einem Weg (*Trace*) durch ein Programm
  - verzahnt mit der Ausführung anderer Prozesse
  - BS entscheidet über Zuteilung des Prozessors
- ➔ vgl. Definition aus 1.5.1

## 2.1 Begriffsklärung ...

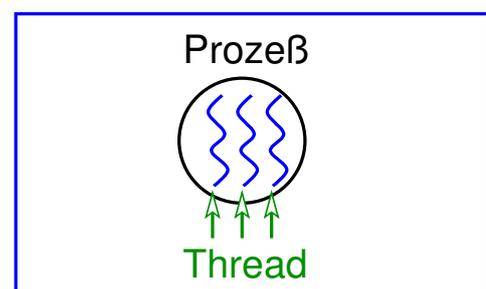


### In heutigen BSen: Trennung der Aspekte

- ➔ **Prozeß**: Einheit des Ressourcenbesitzes und Schutzes
- ➔ **Thread**: Einheit der Ausführung (Prozessorzuteilung)
  - Ausführungsfaden, leichtgewichtiger Prozeß
- ➔ Damit: innerhalb eines Prozesses auch mehrere Threads möglich
  - d.h., Anwendungen können mehrere nebenläufige Aktivitäten besitzen



3 (klassische) Prozesse



3 Threads in einem Prozeß

## 2.1 Begriffsklärung ...



### Vorteile bei der Nutzung mehrerer Threads in einer Anwendung

- ➔ Nebenläufige Programmierung möglich: mehrere Kontrollflüsse
- ➔ Falls ein Thread auf Ein-/Ausgabe wartet: die anderen können weiterarbeiten
- ➔ Kürzere Reaktionszeit auf Benutzereingaben
- ➔ Bei Multiprozessor-Systemen (bzw. mit Hyperthreading): echt parallele Abarbeitung der Threads möglich

## 2.1 Begriffsklärung ...



### Beispiel: GUI-Programmierung

- ➔ Sequentielles Programm (1 Thread):

```
while (true) {  
    ComputeStep();           // z.B. Animationsschritt  
    if (QueryEvent()) {     // Ereignis angekommen?  
        e = ReceiveEvent(); // Ereignis abholen  
        ProcessEvent(e);    // und bearbeiten  
    }  
}
```

- ➔ Verzahnung von Berechnung und Ereignisbehandlung
- ➔ **Polling** von Ereignissen: wann / wie oft?

## 2.1 Begriffsklärung ...



### Beispiel: GUI-Programmierung ...

➔ Nebenläufiges Programm mit 2 Threads:

Thread 1:

```
while (true) {  
    ComputeStep();  
}
```

Thread 2:

```
while (true) {  
    e = ReceiveEvent();  
    ProcessEvent(e);  
}
```

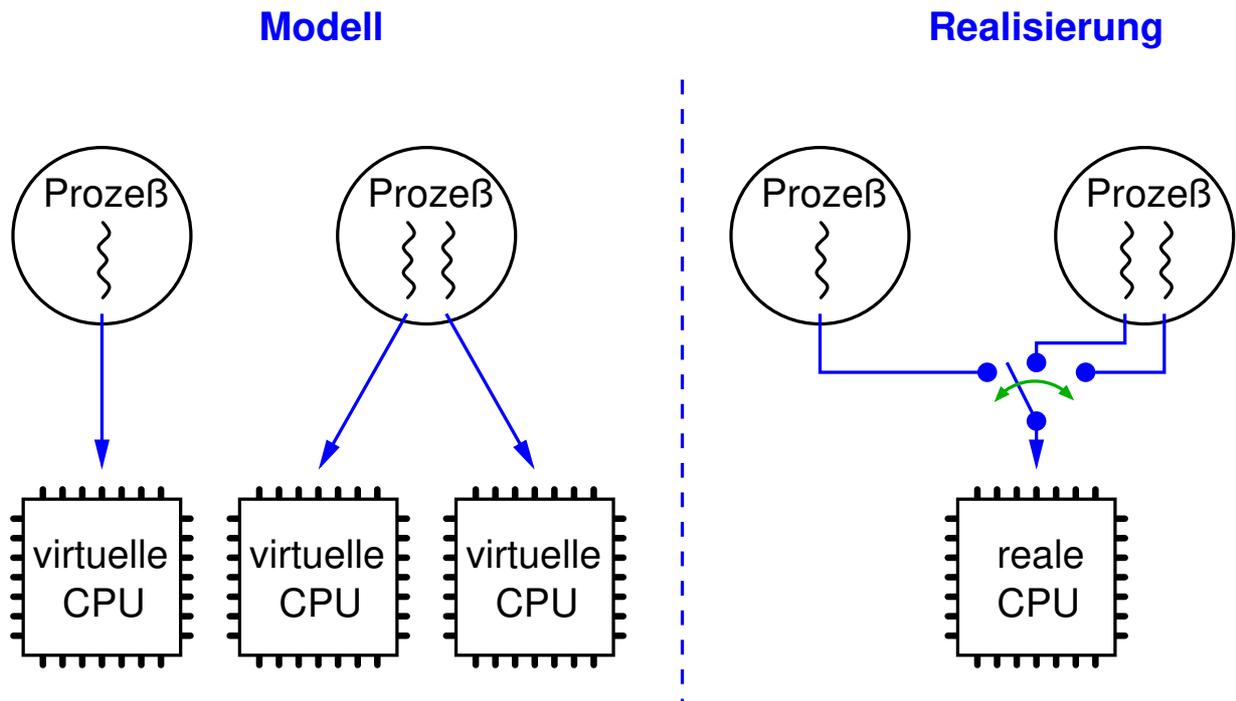
- ➔ einfachere Programmstruktur
- ➔ aber: Zugriff auf gemeinsame Variable erfordert Synchronisation (☞ **3.1**)

## 2.1 Begriffsklärung ...



### Realisierung von Threads

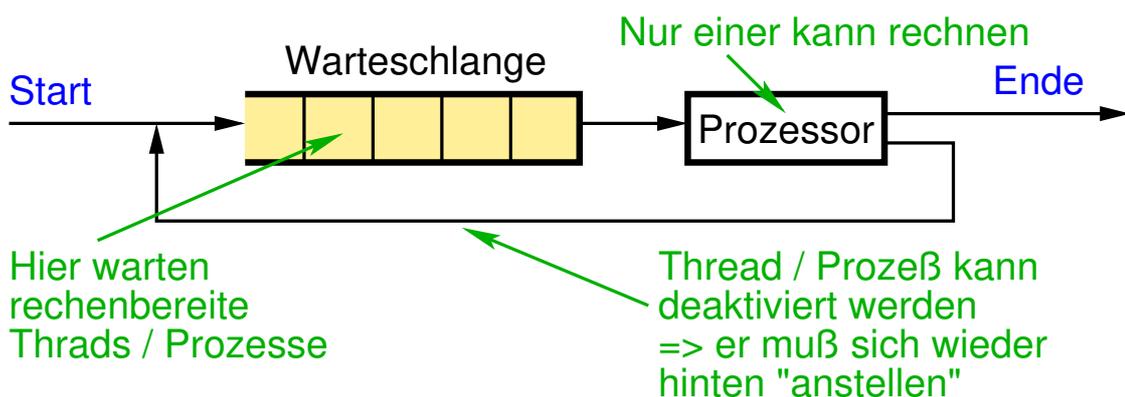
- ➔ Heute meist direkt durch das BS
  - ➔ andere Alternativen: siehe **2.5**
- ➔ Konzeptuell: jeder Thread
  - ➔ wird durch eine eigene, virtuelle CPU ausgeführt
    - ➔ nebenläufige (quasi-parallele) Abarbeitung der Threads
  - ➔ nutzt alle anderen Ressourcen seines Prozesses (u.a. den virtuellen Adreßraum) gemeinsam mit dessen anderen Threads
- ➔ Real: (jede) CPU schaltet zwischen den Threads hin und her
  - ➔ **Multithreading**
  - ➔ Umschalten durch Umladen der CPU-Register (incl. PC)



## 2.2 Thread-/Prozeßzustände

(Animierte Folie)

### Ein einfaches Thread/Prozeß-Modell

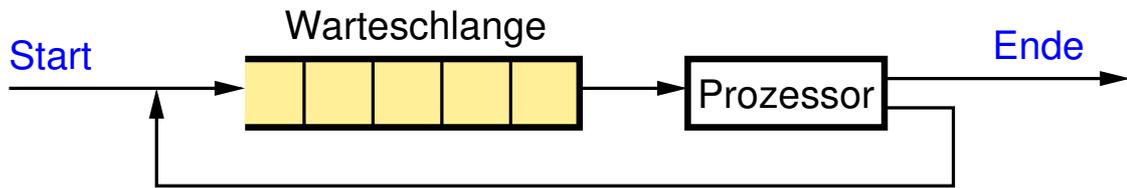


- ➔ Anmerkung: das Modell gilt sowohl für Threads als auch für klassische Prozesse (mit genau einem Thread)

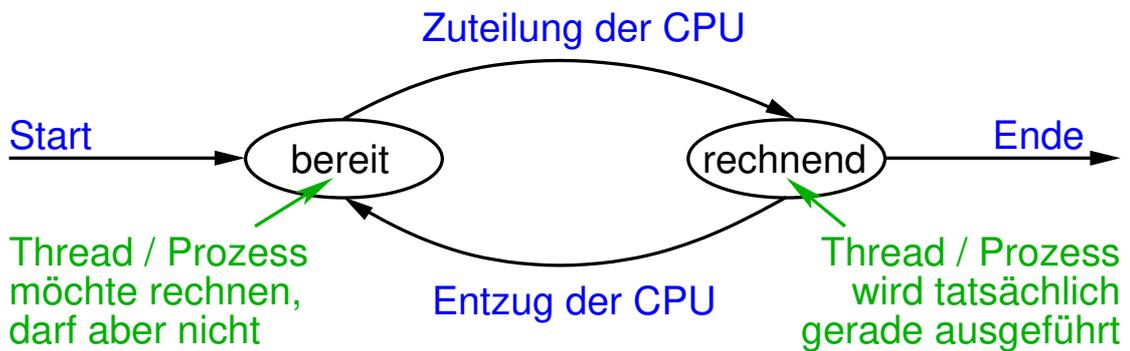
## 2.2 Thread-/Prozeßzustände ...



### Ein einfaches Thread/Prozeß-Modell



### Zustandsgraph eines Threads/Prozesses

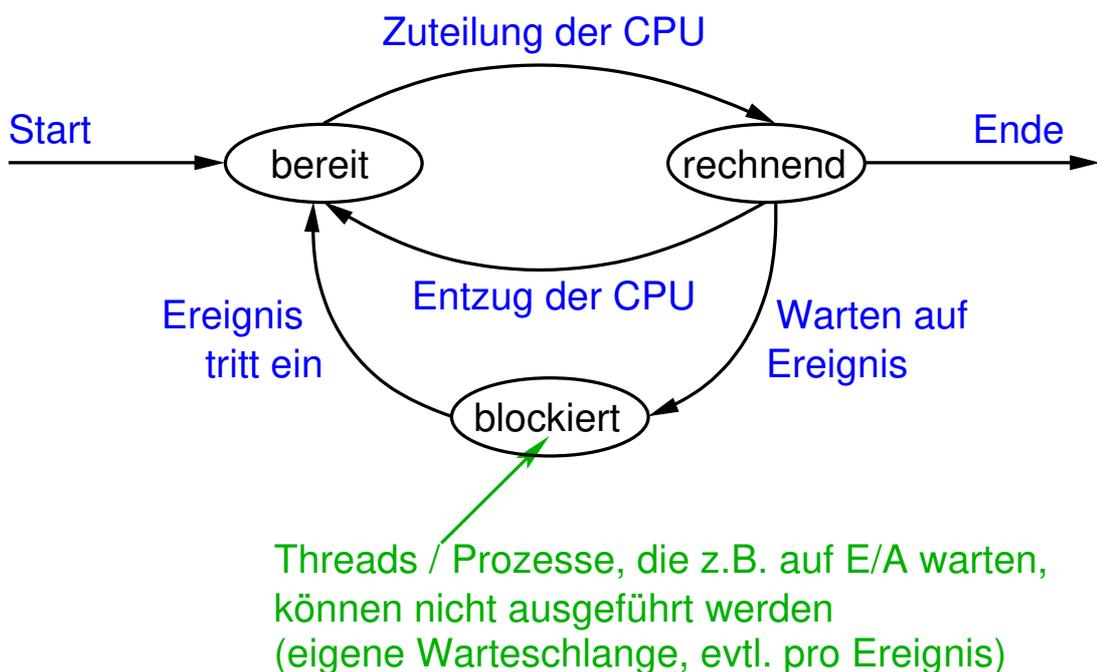


## 2.2 Thread-/Prozeßzustände ...



(Animierte Folie)

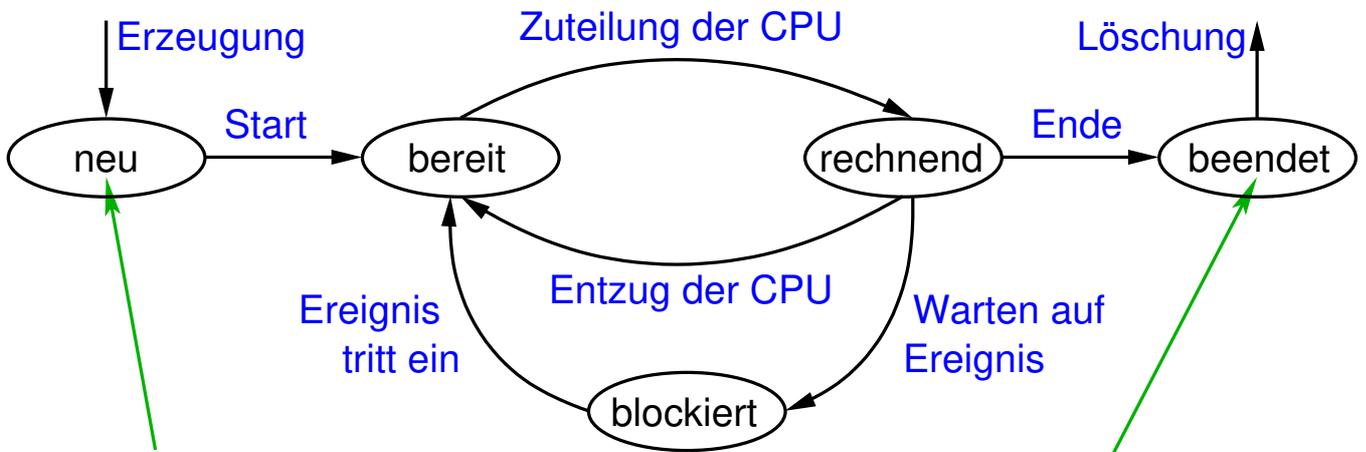
### Zustandsgraph für ein erweitertes Thread/Prozeß-Modell



## 2.2 Thread-/Prozeßzustände ...



### Zustandsgraph für ein erweitertes Thread/Prozeß-Modell ...



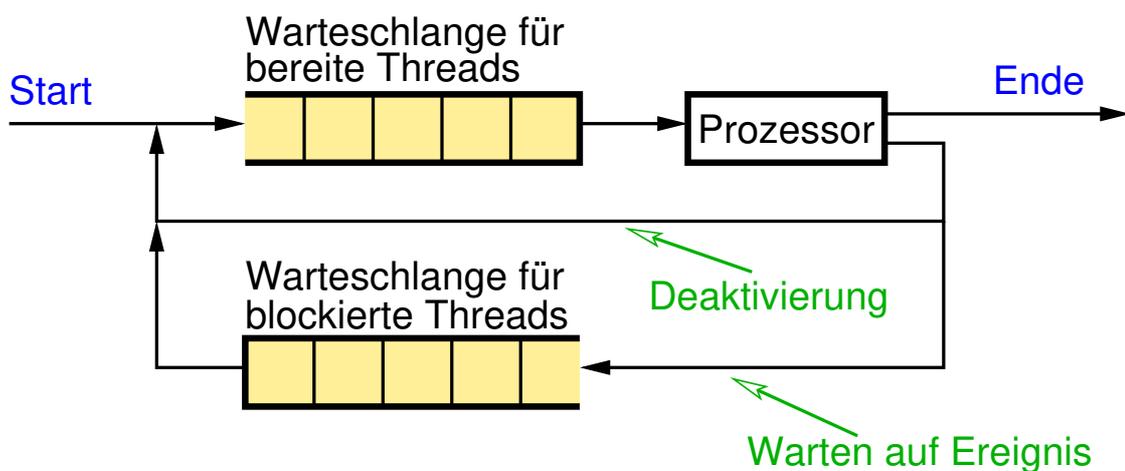
Die Verwaltungsdatenstruktur für den Thread / Prozeß ist bereits angelegt, der Thread / Prozeß selbst existiert aber noch nicht

Thread / Prozeß ist terminiert, Verwaltungsdaten sind noch vorhanden (z.B. zum Auslesen des Exit-Status)

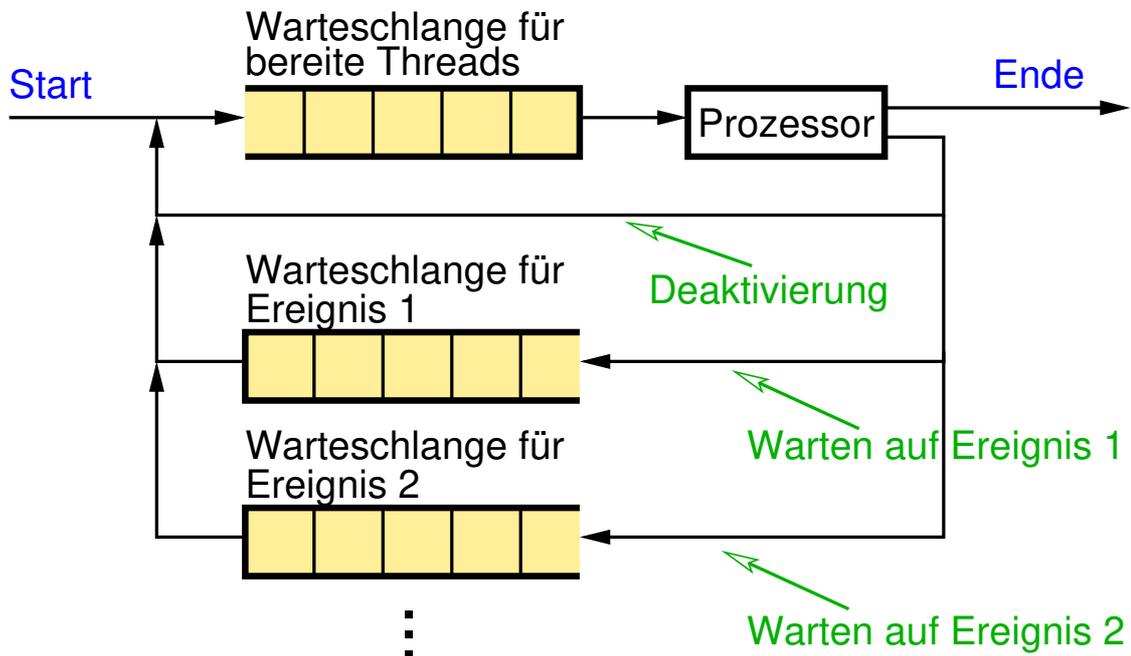
## 2.2 Thread-/Prozeßzustände ...



### Eine Warteschlange für alle blockierten Threads



### Eine Warteschlange pro Ereignis



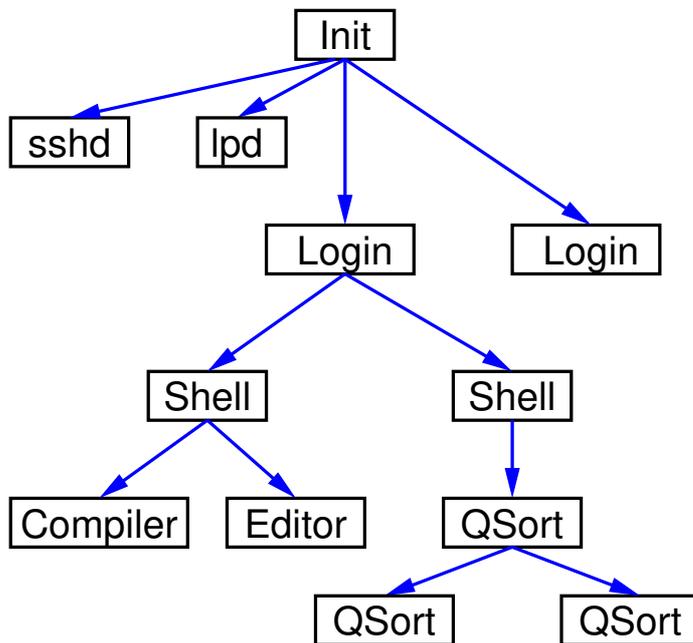
## 2.3 Erzeugung und Terminierung von Prozessen

### Gründe für Prozeßerzeugung:

- ➔ Initialisierung des Systems
  - ➔ Hintergrundprozesse (*Daemons*) für BS-Dienste
- ➔ Benutzeranfrage
  - ➔ Interaktive Anmeldung, Start eines Programms
- ➔ Erzeugung durch Systemaufruf eines bestehenden Prozesses
- ➔ Initiierung eines *Batch-Jobs*
  - ➔ in Mainframe-BSen
- ➔ Technisch wird ein neuer Prozeß (fast) immer durch einen Systemaufruf (z.B. `fork` bzw. `CreateProcess`) erzeugt
  - ➔ führt zu **Prozeßhierarchie**



### Beispiel: Prozeßhierarchie unter UNIX



Initialisierungsprozeß

Daemons

Für jeden Benutzer  
ein Login-Prozeß

Mehrere Kommando-  
interpreter (Shells) pro  
Benutzer

Anwendungen



### Gründe für Prozeßterminierung:

- ➔ Freiwillig
  - ➔ durch Aufruf von z.B. `exit` bzw. `ExitProcess`
  - ➔ normal oder wegen Fehler
- ➔ Unfreiwillig (Abbruch durch BS)
  - ➔ wegen schwerwiegender Fehler, z.B. Speicherüberschreitung, Ausnahme, E/A-Fehler, Schutzverletzung
  - ➔ durch andere (berechtigte) Prozesse, über Systemaufruf (z.B. `kill` bzw. `TerminateProcess`)
  - ➔ Teilweise ist noch eine Reaktion des Prozesses auf das Ereignis möglich (☞ **3.2.4**: Signale)



- ➔ BS pflegt **Prozeßtable** mit Informationen über alle Prozesse
  - ➔ der Eintrag für einen Prozeß heißt **Prozeßkontrollblock**
- ➔ Analog: **Threadtable** für alle Threads
  - ➔ Eintrag: **Threadkontrollblock**
- ➔ Prozeßadressraum, Prozeßkontrollblock und Threadkontrollblöcke beschreiben einen Prozeß vollständig
- ➔ Typische Elemente des Prozeßkontrollblocks:
  - ➔ Prozeßidentifikation, Zustands- und Ressourceninformation
- ➔ Typische Elemente des Threadkontrollblocks:
  - ➔ Threadidentifikation, Zustandsinformation
  - ➔ Scheduling- und Prozessorstatus-Information



### Inhalt des Prozeßkontrollblocks

- ➔ Prozeßidentifikation
  - ➔ Kennung des Prozesses und des Elternprozesses
  - ➔ Benutzerkennung
  - ➔ Liste der Kennungen aller Threads
- ➔ Zustandsinformation
  - ➔ Priorität, verbrauchte CPU-Zeit, ...
- ➔ Verwaltungsinformation
  - ➔ Daten für Interprozeßkommunikation (☞ 3)
  - ➔ Prozeßprivilegien
  - ➔ Tabellen für Speicherabbildung (Speicherverwaltung, ☞ 6)
  - ➔ Ressourcenbesitz und -nutzung
    - ➔ offene Dateien, Arbeitsverzeichnis, ...



### Inhalt des Threadkontrollblocks

- ➔ Threadidentifikation
  - ➔ Kennung des Threads
  - ➔ Kennung des zugehörigen Prozesses
- ➔ Scheduling- und Zustandsinformation
  - ➔ Threadzustand (bereit, rechnend, blockiert, ...)
  - ➔ ggf. Ereignis, auf das der Thread wartet
  - ➔ Priorität, verbrauchte CPU-Zeit, ...
- ➔ Prozessorstatus-Information
  - ➔ Datenregister
  - ➔ Steuer- und Statusregister: PC, PSW, ...
  - ➔ Kellerzeiger (SP)



### Elemente von Prozessen und Threads

Elemente pro Prozeß	Elemente pro Thread
Adreßraum	Befehlszähler
geöffnete Dateien	Register
Kindprozesse	Keller*
Signale	Zustand (bereit, ...)
Privilegien	
...	
...	* genauer: Kellerzeiger

- ➔ (Bei Verwendung höherer Programmiersprachen: lokale Variable sind pro Thread, globale pro Prozeß)



### Ablauf einer Prozeßerzeugung

- ➔ Eintrag mit eindeutiger Kennung in Prozeßtabelle erzeugen
- ➔ Zuteilung von Speicherplatz an den Prozeß
  - ➔ für Programmcode, Daten, und Keller
  - ➔ (siehe später: **6.** Speicherverwaltung)
- ➔ Initialisierung des Prozeßkontrollblocks
  - ➔ Ressourcen evtl. von Elternprozeß geerbt
- ➔ Erzeugung und Initialisierung eines Threadkontrollblocks
  - ➔ PC und SP (und alle anderen Register)
  - ➔ Threadzustand: bereit
  - ➔ Prozeß startet mit genau einem Thread
- ➔ Einhängen des Threads in die Bereit-Warteschlange



### Ablauf eines Threadwechsels

- 1.** Prozessorstatus im Threadkontrollblock sichern
  - ➔ PC, PSW, SP, andere Register
- 2.** Thread- und Prozeßkontrollblock aktualisieren
  - ➔ Threadzustand, Grund der Deaktivierung, Buchhaltung, ...
- 3.** Thread in entsprechende Warteschlange einreihen
- 4.** Nächsten bereiten Thread auswählen (☞ **5.** Scheduling)
- 5.** Threadkontrollblock des neuen Threads aktualisieren
  - ➔ Zustand auf „rechnend“ setzen
- 6.** Falls neuer Thread in anderem Prozess liegt:
  - ➔ Aktualisierung der Speicherverwaltungsstrukturen (☞ **6**)
- 7.** Prozessorstatus aus neuem Threadkontrollblock laden



### Anmerkungen

- ➔ Beim Threadwechsel **innerhalb desselben Prozesses** entfällt die Aktualisierung der Speicherverwaltungsstrukturen
  - ➔ Threads im selben Prozeß haben gemeinsamen Speicher
- ➔ BS (Scheduler) entscheidet direkt, welcher **Thread** als nächstes rechnen soll
  - ➔ falls nötig, wird dann auch der Prozeß mit umgeschaltet
  - ➔ Scheduler kann/sollte die Zuordnung von Threads zu Prozessen bei der Entscheidung berücksichtigen
    - ➔ z.B. wegen unterschiedlicher Kosten



### Wann erfolgt ein Threadwechsel?

- ➔ Threadwechsel kann immer dann erfolgen, wenn BS die Kontrolle erhält:
  - ➔ bei Systemaufruf (z.B. E/A)
    - ➔ Thread gibt Kontrolle (d.h. Prozessor) freiwillig ab
  - ➔ bei Ausnahme (z.B. unzulässigem Befehl)
    - ➔ Prozeß wird ggf. beendet
    - ➔ evtl. auch Behandlung der Ausnahme durch BS
  - ➔ bei Interrupt (z.B. E/A-Gerät, Timer)
    - ➔ Behandlung des Interrupts erfolgt im BS
    - ➔ **periodischer Timer-Interrupt** stellt sicher, daß kein Thread die CPU monopolisieren kann



### Ablauf beim Systemaufruf

- ➔ Durch Hardware: Einsprung ins BS (Systemmodus)
- ➔ Ablauf im BS:
  - Sichern des gesamten Prozessorstatus (1)
  - Thread in Zustand „bereit“ setzen (2,3)
  - Ausführung bzw. Initiierung des Auftrags
    - evtl. Thread in Zustand „blockiert“ setzen (2,3)
  - Sprung zum Scheduler (4-7)
    - Aktivieren eines (anderen) Threads
      - dieser Thread wird nach Rückkehr in den Benutzermodus fortgesetzt



### Ablauf bei Ausnahme

- ➔ Durch Hardware: Einsprung ins BS (Systemmodus)
- ➔ Ablauf im BS:
  - Sichern des gesamten Prozessorstatus (1)
  - je nach Art der Ausnahme:
    - Beenden des Prozesses (2)
    - Blockieren des Threads (2,3)  
(z.B. bei Seitenfehler, ➔ **6.3.2**: dyn. Seitenersetzung)
    - Behebung der Ursache der Ausnahme (2,3)
  - Sprung zum Scheduler (4-7)





### Realisierung durch BS-Kern: Diskussion

➔ Vorteile:

- ➔ bei Blockierung eines Threads kann BS einen anderen Thread desselben Prozesses auswählen
- ➔ bei Mehrprozessorsystemen: echte Parallelität innerhalb eines Prozesses möglich

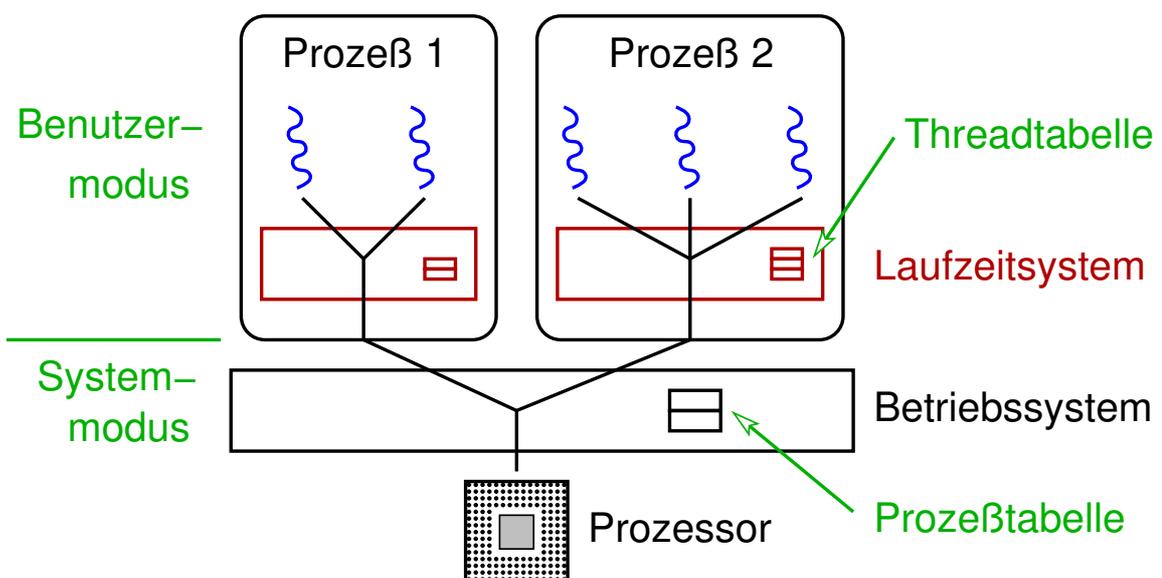
➔ Nachteil: hoher Overhead

- ➔ Threadwechsel benötigt Moduswechsel zum BS-Kern
- ➔ Erzeugen, Beenden, etc. benötigt Systemaufruf

## 2.5 Realisierungsvarianten für Threads ...



### Realisierung im Benutzeradreibraum



➔ Genutzt in frühen Thread-Implementierungen

### Realisierung im Benutzeradreibraum: Diskussion

➔ Vorteile:

- ➔ keine Unterstützung durch BS notwendig
- ➔ schnelle Threaderzeugung und Threadwechsel
  - ➔ z.B. Zeit für Erzeugung (BS: Solaris, CPU: Sparc2; alt!)

Benutzer-Thread	Kernel-Thread	Prozess
$52 \mu s$	$350 \mu s$	$1700 \mu s$

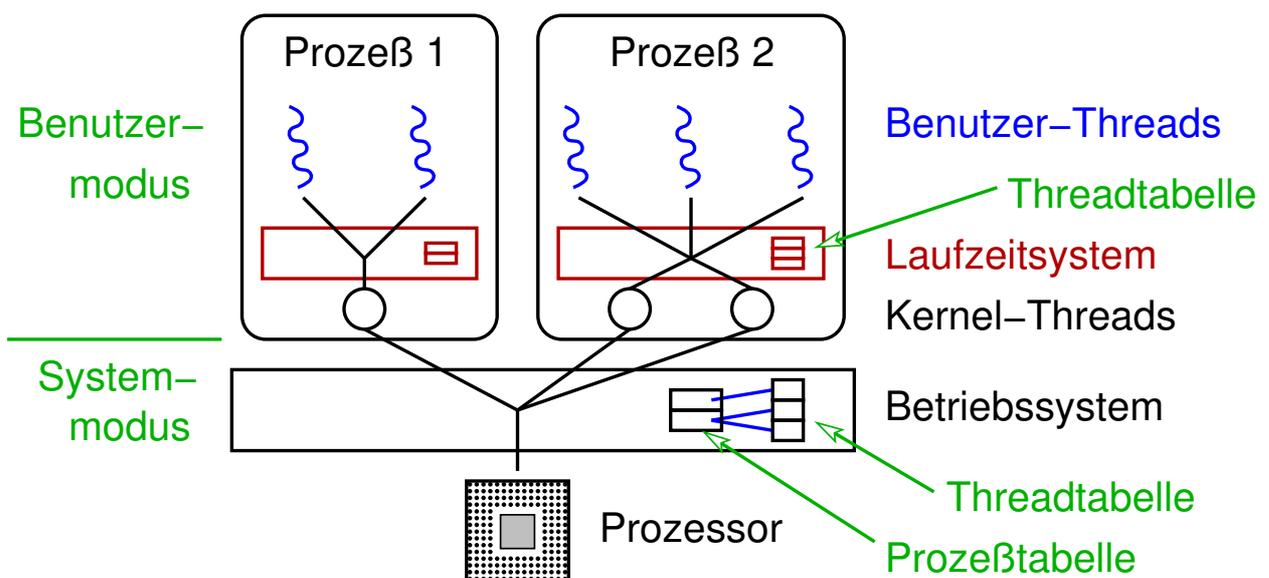
- ➔ individuelle Scheduling-Algorithmen möglich

➔ Nachteile:

- ➔ blockierender Systemaufruf blockiert alle Threads
  - ➔ macht eine Hauptmotivation für Threads zunichte
- ➔ Threads müssen Prozessor i.d.R. freiwillig abgeben
  - ➔ Threadwechsel erfolgt durch Bibliotheksfunktion

## 2.5 Realisierungsvarianten für Threads ...

### Hybride Realisierung



➔ Z.B. in alten Versionen von Solaris

➔ Heute: für Programme mit sehr vielen nebenläufigen Aktivitäten

# Betriebssysteme I

WS 2019/2020

07.11.2019

Roland Wismüller  
Betriebssysteme / verteilte Systeme  
roland.wismueller@uni-siegen.de  
Tel.: 0271/740-4050, Büro: H-B 8404

Stand: 14. November 2019

## 2.6 Schnittstellen zur Nutzung von Threads



- ➔ Alle heute gängigen BSe unterstützen (Kernel-)Threads
- ➔ Anwendungen nutzen jedoch i.d.R. nicht die Systemaufrufe, sondern höhere Programmierschnittstellen
- ➔ Beispiele:
  - ➔ POSIX Threads (Programmierbibliothek)
  - ➔ Java Threads (Sprachkonstrukt)



### POSIX Threads (PThreads)

- ➔ IEEE 1003.1c: Standardschnittstelle zur Programmierung mit Threads
  - ➔ (weitestgehend) betriebssystemunabhängig
  - ➔ in fast allen PC/Server-BSen verfügbar
- ➔ Programmiermodell:
  - ➔ bei Programmstart: genau ein (Master-)Thread
  - ➔ Master-Thread erzeugt andere Threads und wartet auf deren Beendigung
  - ➔ Prozeß terminiert bei Beendigung des Master-Threads



### PThreads: Funktionen zur Threadverwaltung (unvollständig)

- ➔ `pthread_create`: erzeugt neuen Thread
  - ➔ Parameter: Prozedur, die der Thread abarbeiten soll
  - ➔ Ergebnis: Thread-*Handle* (= Referenz)
- ➔ `pthread_exit`: eigene Terminierung (mit Ergebniswert)
- ➔ `pthread_cancel`: terminiert anderen Thread
- ➔ `pthread_join`: wartet auf Terminierung eines Threads, liefert Ergebniswert

### Code-Beispiel: Hello World mit PThreads in C

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

```
void * SayHello(void *arg) {  
    printf("Hello World!\n");  
    return NULL;  
}
```

Diese Funktion wird durch einen neuen Thread abgearbeitet

```
int main(int argc, char **argv) {  
    pthread_t t; // Erzeugung des Threads  
    if (pthread_create(&t, NULL, SayHello, NULL) != 0) {  
        /* FEHLER! */  
    }  
    pthread_join(t, NULL); // Warten auf Beendigung  
    exit(0);  
}
```

#### Anmerkungen zu Folie 119:

Die genauen Schnittstellen der verwendeten Funktionen sind:

- ➔ `int pthread_create(pthread_t *thread, pthread_attr_t *attr, void *(*start_routine)(void *), void *arg)`
  - `attr`: Thread-Attribute (i.W. BS-spezifisch)
  - `start_routine`: Prozedur, die der Thread abarbeiten soll
  - `arg`: Parameter, der `start_routine` übergeben wird
  - `thread`: Zeiger auf Ergebnis (Thread-Handle)
- ➔ `void pthread_exit(void *retval)`
  - `retval`: Ergebniswert
- ➔ `int pthread_join(pthread_t th, void **thread_return)`
  - `th`: Thread, auf den gewartet werden soll
  - `*thread_return`: Zeiger auf Ergebnis



### Java Threads

- ➔ Grundlage: die Klasse Thread
  - ➔ Konstruktoren (u.a.):
    - ➔ `Thread()`, `Thread(Runnable target)`
  - ➔ Methoden (u.a.):
    - ➔ `void start()` – startet einen Thread
    - ➔ `void join()` – wartet auf Ende des Threads
- ➔ Definieren von Threads
  - ➔ Deklaration einer neuen Klasse
    - ➔ als Unterklasse der Klasse Thread
    - ➔ oder: als Implementierung der Schnittstelle `Runnable`
  - ➔ Überschreiben / Implementieren der Methode `void run()` mit dem auszuführenden Code



### Code-Beispiel: Hello World mit Java Threads (1)

```
public class MyThread extends Thread
{
    public void run()
    {
        System.out.println("Hello World!");
    }

    public static void main(String[] args)
    {
        MyThread t = new MyThread();
        t.start();
        t.join();
    }
}
```

Diese Methode wird durch den neuen Thread abgearbeitet

`t.start();` Erzeugung des Threads

`t.join();` Warten auf Beendigung



### Code-Beispiel: Hello World mit Java Threads (2)

```
public class HelloWorld implements Runnable
{
    public void run()
    {
        System.out.println("Hello World!");
    }

    public static void main(String[] args)
    {
        HelloWorld runner = new HelloWorld();
        Thread t = new Thread(runner);
        t.start(); Erzeugung des Threads
        t.join(); Warten auf Beendigung
    }
}
```

Diese Methode wird durch den neuen Thread abgearbeitet



### Anmerkungen zu den Code-Beispielen

- ➔ Programmiermodell:
  - ➔ bei Programmstart: genau ein Thread
  - ➔ Prozeß terminiert erst, wenn alle Threads beendet sind
- ➔ Jeder Thread darf nur einmal über `start()` gestartet werden
- ➔ Übergabe von Parametern und Ergebnissen:
  - ➔ über Attribute des Thread bzw. Runnable-Objekts
  - ➔ Eingabeparameter werden vor `start()` gesetzt (z.B. durch Konstruktor)
  - ➔ Ergebnisse können nach `join()` z.B. durch Getter-Methoden ausgelesen werden

## 2.7 Zusammenfassung / Wiederholung



- ➔ Zwei Aspekte:
  - ➔ Prozeß: Einheit der Ressourcenverwaltung, Schutzseinheit
  - ➔ Thread: Einheit der Prozessorzuteilung
    - ➔ pro Prozeß mehrere Threads möglich
- ➔ Threadmodell
  - ➔ Zustände „rechnend“, „bereit“, „blockiert“ + andere
  - ➔ Warteschlangen
- ➔ Zum Prozeß gehören u.a.:
  - ➔ Adreßraum, geöffnete Dateien, Signale, Privilegien, ...
- ➔ Zum Thread gehören u.a.:
  - ➔ Befehlszähler, CPU-Register, Keller(zeiger), Scheduling-Zustand, ...

## 2.7 Zusammenfassung / Wiederholung ...



- ➔ Threadwechsel:
  - ➔ Umladen des Prozessorkontexts
  - ➔ bei Prozeßwechsel auch Wechsel der Speicherabbildung
  - ➔ kann bei Systemaufruf, Ausnahme und Interrupt erfolgen
- ➔ Realisierungsvarianten für Threads:
  - ➔ im BS-Kern, Benutzeradreßraum, hybrid
- ➔ Thread-Schnittstellen:
  - ➔ POSIX-Threads, Java Threads