Betriebssysteme und nebenläufige Programmierung

SoSe 2025

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

Stand: 24. April 2025

Betriebssysteme und nebenläufige Programmierung

SoSe 2025

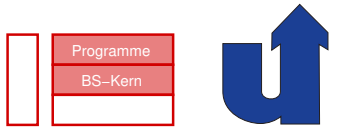
2 Prozesse und Threads

2 Prozesse und Threads ...

- Begriffsklärung
- Nebenläufige Programmierung
- Thread-/Prozeßmodell und -zustände
- Implementierung von Prozessen und Threads
- Realisierungsvarianten für Threads
- Schnittstellen zur Nutzung von Threads

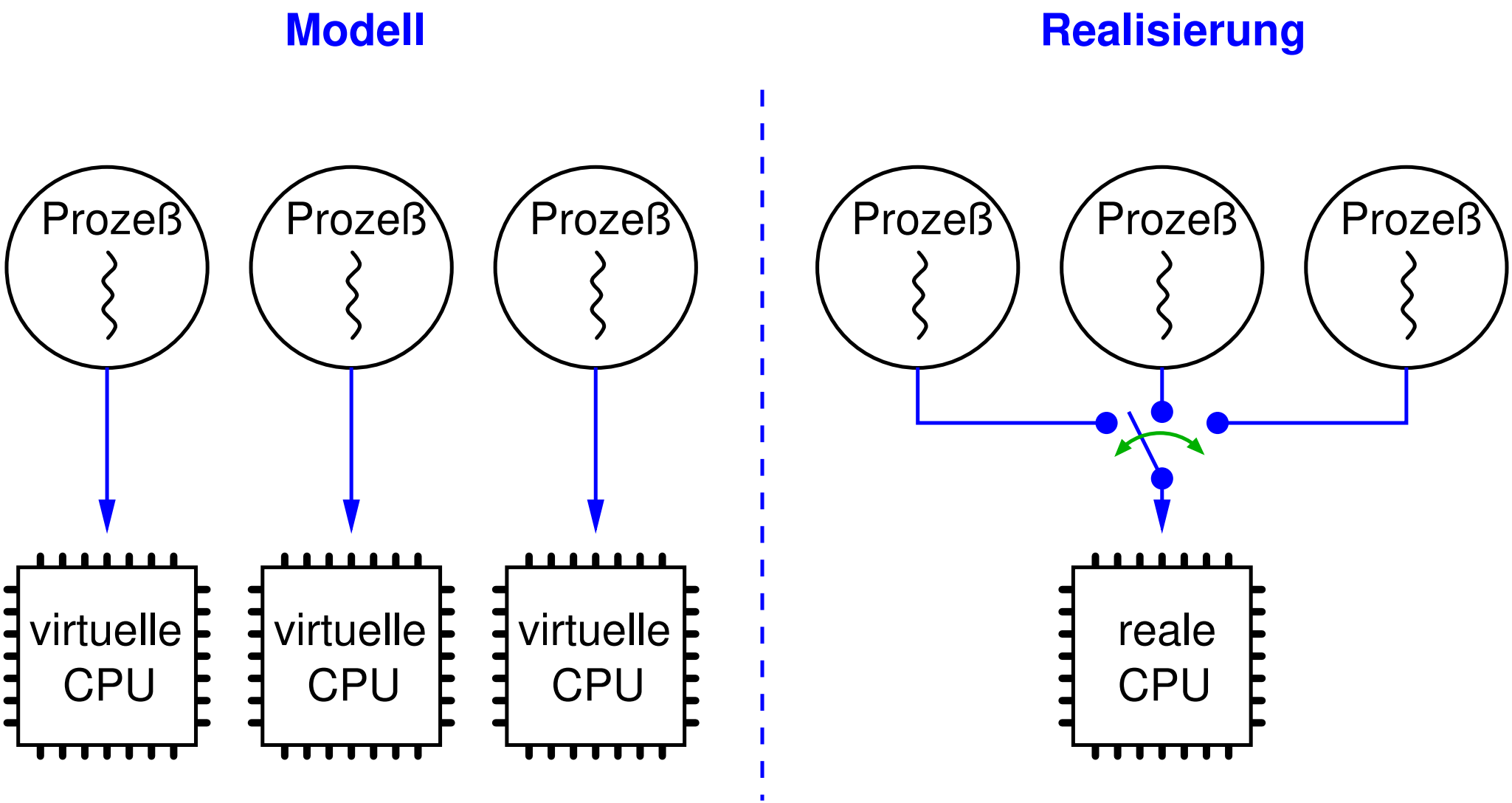
- Tanenbaum 2.1 - 2.2
- Stallings 3.1 - 3.2

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“, ☞ **8**)
- 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 ...

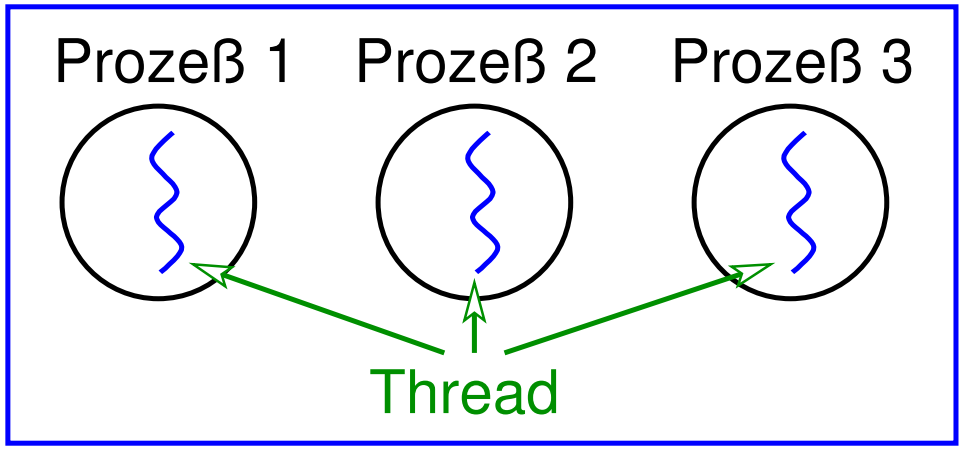


Ein (klassischer) Prozeß besitzt zwei Aspekte:

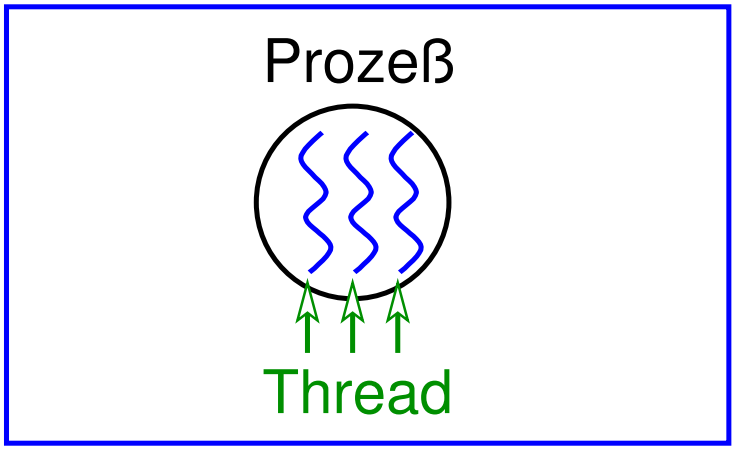
- Einheit des Ressourcenbesitzes
  - eigener (virtueller) Adreßraum
  - allgemein: Besitz / Kontrolle von Ressourcen (Hauptspeicher, Dateien, E/A-Geräte, ...)
  - BS übt Schutzfunktion aus
- Einheit der Ablaufplanung / Ausführung
  - Ausführung folgt einem Weg (*Trace*) durch ein Programm
  - verzahnt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

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

Betriebssysteme und nebenläufige Programmierung

SoSe 2025

02.05.2025

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

Stand: 24. April 2025

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

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?

Beispiel: GUI-Programmierung ...

- Nebenläufiges Programm mit 2 Threads:

Thread 1:

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

Thread 2:

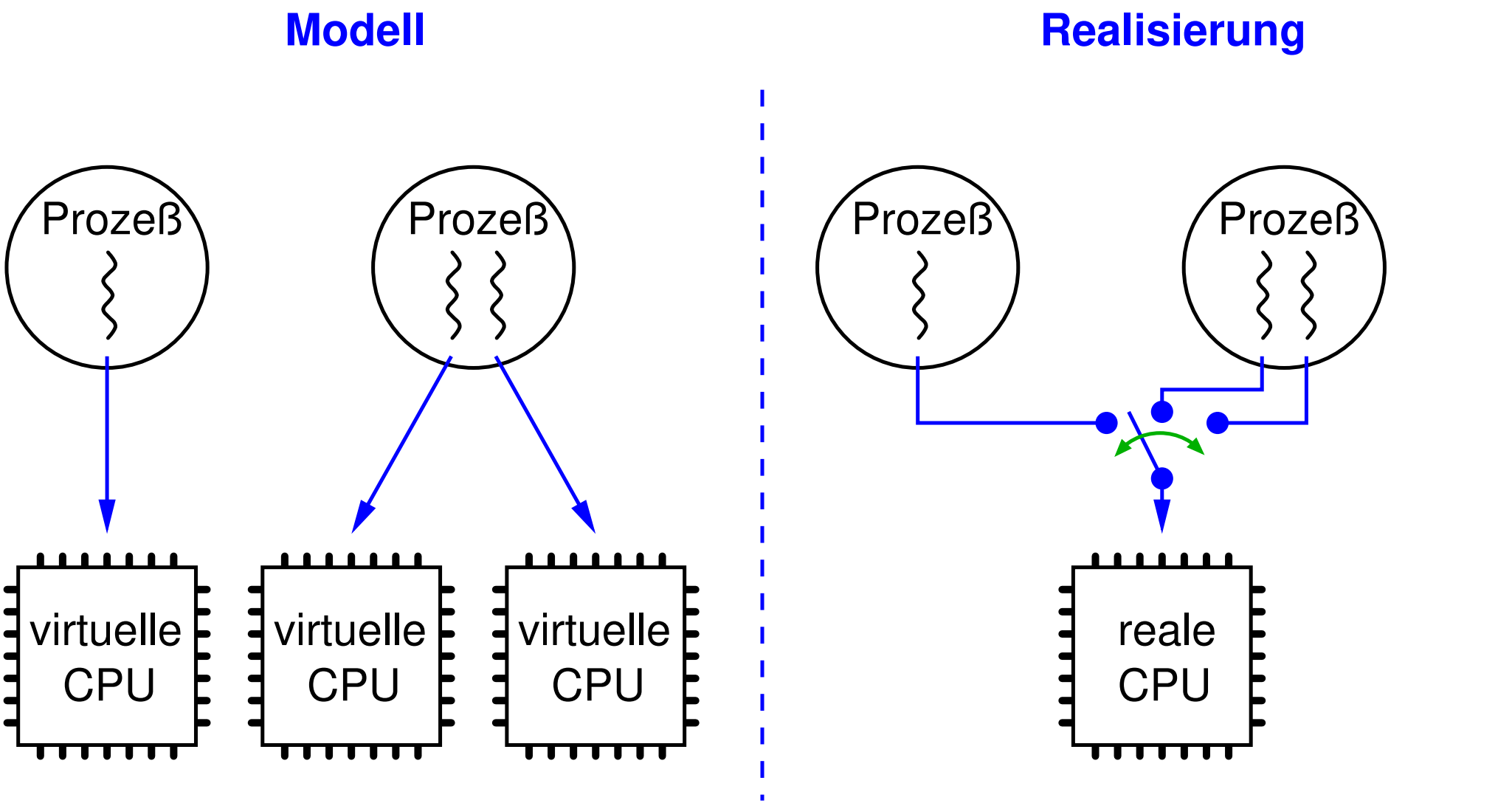
```
while (true) {
    e = ReceiveEvent();
    ProcessEvent(e);
}
```

- einfachere Programmstruktur
- aber: Zugriff auf gemeinsame Variable erfordert Synchronisation (☞ **3**)

Realisierung von Threads

- Heute meist direkt durch das BS
  - andere Alternativen: siehe Ergänzungsfolien
- 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)

2.1 Begriffsklärung ...

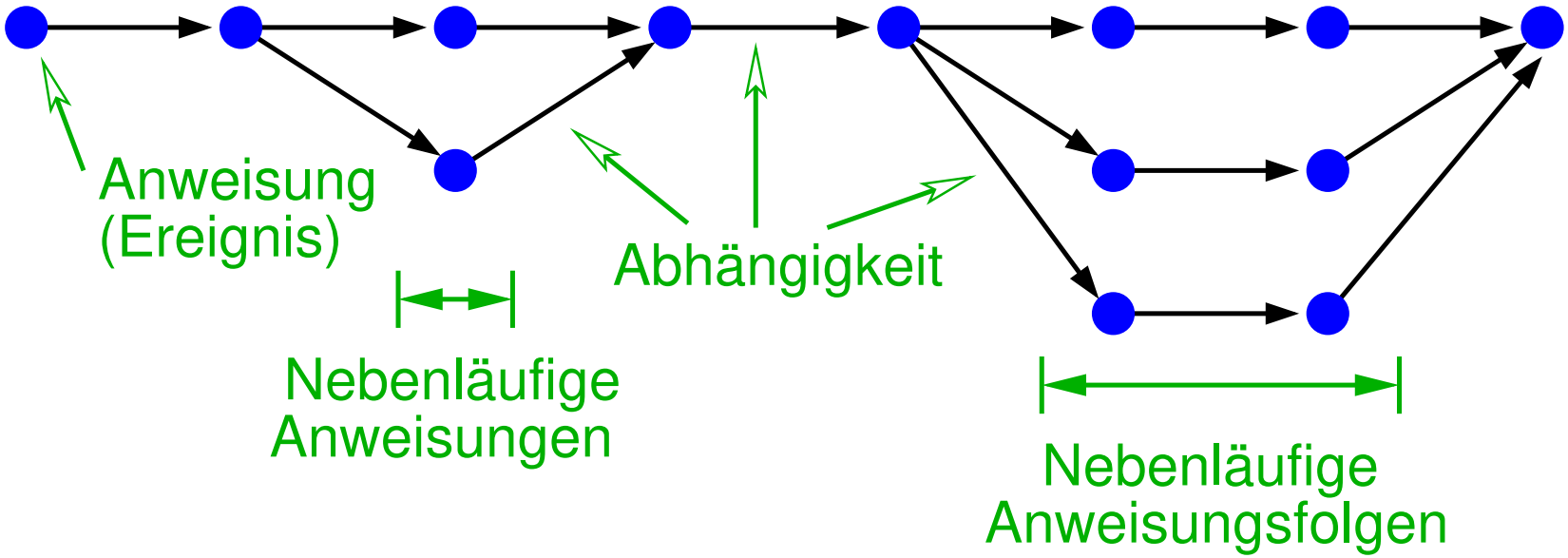


2.2 Nebenläufige Programmierung

- Einfache Programme arbeiten rein **sequentiell**
  - die Anweisungen werden streng nacheinander abgearbeitet
- Oft will man Aktivitäten definieren, die „gleichzeitig“ ausgeführt werden können
  - z.B. Abspielen von Bild und Ton eines Videos
  - z.B. Drucken im Hintergrund
- Wichtige Unterscheidung:
  - zwei Aktivitäten sind **parallel**, wenn sie gleichzeitig ausgeführt werden
  - zwei Aktivitäten sind **nebenläufig**, wenn sie parallel ausgeführt werden **können**
    - d.h., wenn es keine kausalen Abhängigkeiten zwischen ihnen gibt

Graphische Darstellung

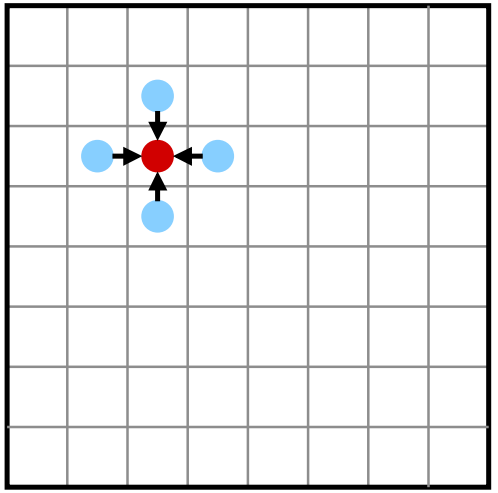
- Aktivitäten (Ereignisse, Anweisungen, ...) werden als Knoten gezeichnet
- Abhängigkeiten als gerichtete Kanten
- Beispiel:



- Mathematisch entspricht das einer Halbordnung

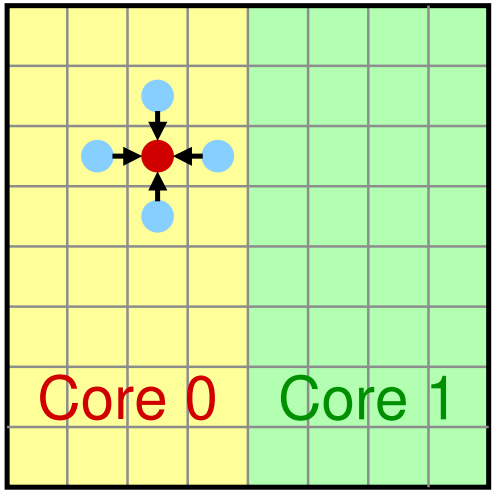
Motivationen für nebenläufige Abarbeitung

- Gleichzeitiges Rechnen
  - Beschleunigung von Programmen durch gleichzeitige Nutzung mehrerer CPU-Cores
  - Parallelverarbeitung
  - bei rechenintensiven Programmen
  - Beispiel: Simulation des Wärmeausgleichs (Jacobi-Verfahren)
    - iteratives Verfahren
    - ersetze Temperatur eines Punktes durch Mittelwert der Nachbarpunkte



Motivationen für nebenläufige Abarbeitung

- Gleichzeitiges Rechnen
  - Beschleunigung von Programmen durch gleichzeitige Nutzung mehrerer CPU-Cores
  - Parallelverarbeitung
  - bei rechenintensiven Programmen
  - Beispiel: Simulation des Wärmeausgleichs (Jacobi-Verfahren)
    - iteratives Verfahren
    - ersetze Temperatur eines Punktes durch Mittelwert der Nachbarpunkte
    - einfache Aufteilung der Arbeit auf mehrere Bearbeiter möglich



Motivationen für nebenläufige Abarbeitung ...

- Gleichzeitiges Warten
  - optimale Nutzung der einzelnen CPU-Cores
  - Vermeidung von Leerlauf der Cores
  - bei E/A-lastigen Programmen
  - Beispiel: Drucken im Texteditor
    - während auf den Drucker gewartet wird, sollten weiterhin Benutzereingaben möglich sein

Abhängigkeiten in Programmen

- Frage: wann können zwei Aktivitäten (z.B. Anweisungen) eines Programms nebenläufig ausgeführt werden?
- Antwort: wenn es keine **Abhängigkeit** zwischen ihnen gibt
- Eine Anweisung S' ist abhängig von einer Anweisung S, wenn das Programm nur dann korrekt arbeitet, wenn S' **nach** S ausgeführt wird
- Beispiel:

S_1: `var1 = 8;`
S_2: `var2 = 5;`
S_3: `sum = var1 + var2;`
S_4: `System.out.println(sum);`

Abhängigkeiten in Programmen

- Frage: wann können zwei Aktivitäten (z.B. Anweisungen) eines Programms nebenläufig ausgeführt werden?
- Antwort: wenn es keine **Abhängigkeit** zwischen ihnen gibt
- Eine Anweisung S' ist abhängig von einer Anweisung S, wenn das Programm nur dann korrekt arbeitet, wenn S' **nach** S ausgeführt wird
- Beispiel:

S_2: `var2 = 5;`
S_1: `var1 = 8;`
S_3: `sum = var1 + var2;`
S_4: `System.out.println(sum);`

Vertauschen ändert nichts am Ergebnis

  - S_1 und S_2 sind unabhängig

Abhängigkeiten in Programmen

- Frage: wann können zwei Aktivitäten (z.B. Anweisungen) eines Programms nebenläufig ausgeführt werden?
- Antwort: wenn es keine **Abhängigkeit** zwischen ihnen gibt
- Eine Anweisung S' ist abhängig von einer Anweisung S, wenn das Programm nur dann korrekt arbeitet, wenn S' **nach** S ausgeführt wird
- Beispiel:

S_1: `var1 = 8;`
S_3: `sum = var1 + var2;`
S_2: `var2 = 5;`
S_4: `System.out.println(sum);`

nicht vertauschbar!

  - S_1 und S_2 sind unabhängig, S_3 ist von S_1 und S_2 abhängig

Abhängigkeiten in Programmen

- Frage: wann können zwei Aktivitäten (z.B. Anweisungen) eines Programms nebenläufig ausgeführt werden?
- Antwort: wenn es keine **Abhängigkeit** zwischen ihnen gibt
- Eine Anweisung S' ist abhängig von einer Anweisung S, wenn das Programm nur dann korrekt arbeitet, wenn S' **nach** S ausgeführt wird
- Beispiel:

```
S1: var1 = 8;
S2: var2 = 5;
S4: System.out.println(sum);
S3: sum = var1 + var2;
```

nicht vertauschbar!

  - S_1 und S_2 sind unabhängig, S_3 ist von S_1 und S_2 abhängig, S_4 ist von S_3 abhängig

Arten von Abhängigkeiten

- Wir betrachten zwei Anweisungen S, S' in einem imperativen Programm
- Festlegung: S ist die gemäß Programm zuerst auszuführende Anweisung
- **Echte Abhängigkeit** von S nach S':
  - S' nutzt das Rechenergebnis von S
  - d.h. S schreibt in eine Variable, die S' liest
- Beispiel:

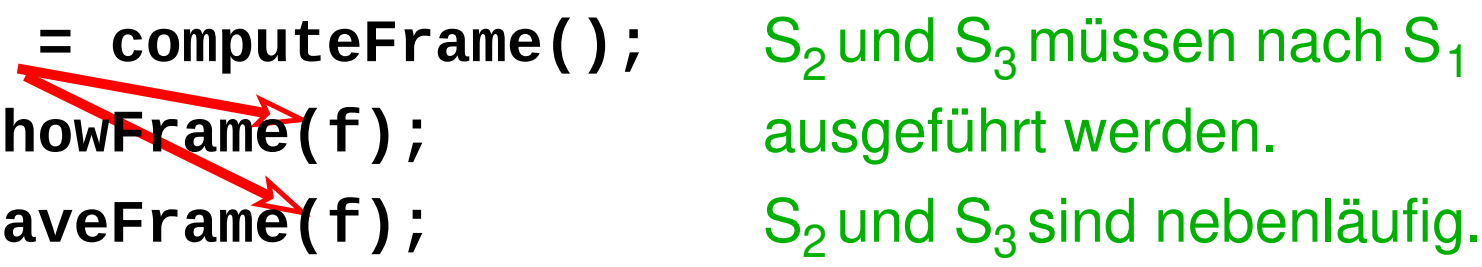
S_1: `f = computeFrame();`

S_2: `showFrame(f);`

S_3: `saveFrame(f);`

Arten von Abhängigkeiten

- Wir betrachten zwei Anweisungen S, S' in einem imperativen Programm
- Festlegung: S ist die gemäß Programm zuerst auszuführende Anweisung
- **Echte Abhängigkeit** von S nach S':
  - S' nutzt das Rechenergebnis von S
  - d.h. S schreibt in eine Variable, die S' liest
- Beispiel:



Arten von Abhängigkeiten ...

- **Anti-Abhängigkeit** von S nach S':
  - S liest eine Variable, die S' überschreibt
- **Ausgabe-Abhängigkeit** von S nach S':
  - S schreibt eine Variable, die S' überschreibt
- Beispiel:

S_1: `f = computeFrame();`
S_2: `showFrame(f);`
S_3: `f = compress(f);`
S_4: `saveFrame(f);`

Arten von Abhängigkeiten ...

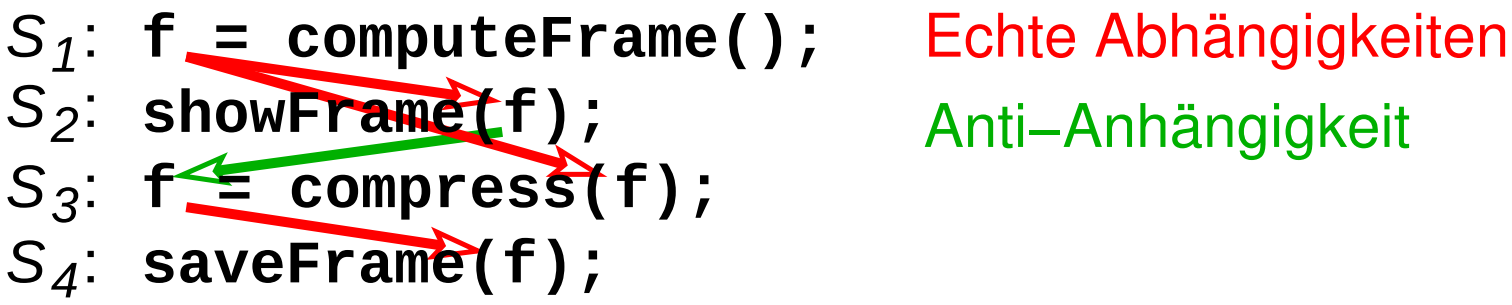
- **Anti-Abhängigkeit** von S nach S':
  - S liest eine Variable, die S' überschreibt
- **Ausgabe-Abhängigkeit** von S nach S':
  - S schreibt eine Variable, die S' überschreibt
- Beispiel:

```
S1: f = computeFrame();
S2: showFrame(f);
S3: f = compress(f);
S4: saveFrame(f);
```

Echte Abhängigkeiten

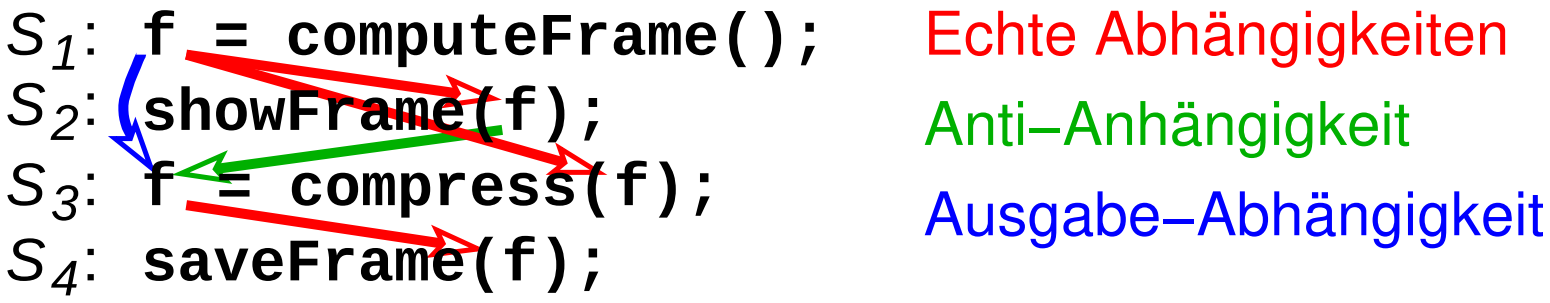
Arten von Abhängigkeiten ...

- **Anti-Abhängigkeit** von S nach S':
  - S liest eine Variable, die S' überschreibt
- **Ausgabe-Abhängigkeit** von S nach S':
  - S schreibt eine Variable, die S' überschreibt
- Beispiel:



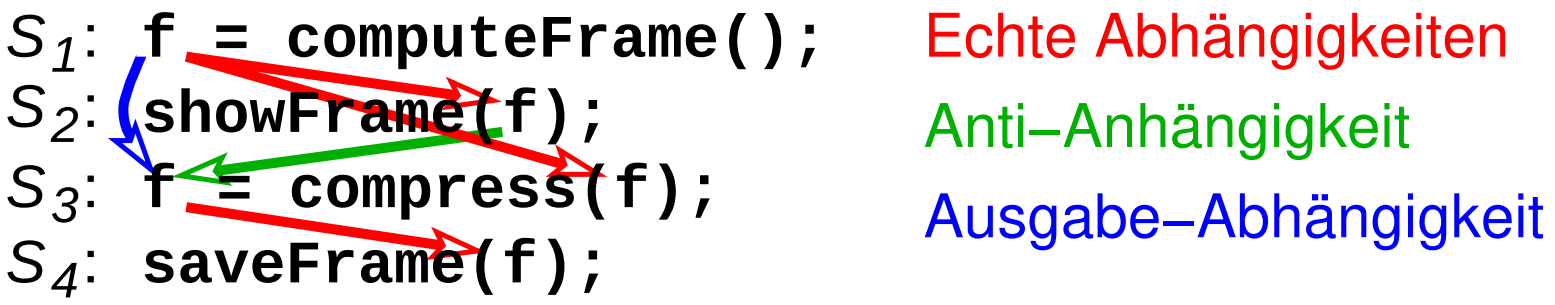
Arten von Abhängigkeiten ...

- **Anti-Abhängigkeit** von S nach S':
  - S liest eine Variable, die S' überschreibt
- **Ausgabe-Abhängigkeit** von S nach S':
  - S schreibt eine Variable, die S' überschreibt
- Beispiel:



Arten von Abhängigkeiten ...

- **Anti-Abhängigkeit** von S nach S':
  - S liest eine Variable, die S' überschreibt
- **Ausgabe-Abhängigkeit** von S nach S':
  - S schreibt eine Variable, die S' überschreibt
- Beispiel:



- Anti- und Ausgabe-Abh. gibt es nur in imperativen Sprachen
  - werden durch Überschreiben von Variablen verursacht
  - sind immer durch Umbenennung von Variablen zu entfernen

Arten von Abhängigkeiten ...

- **Anti-Abhängigkeit** von S nach S':
  - S liest eine Variable, die S' überschreibt
- **Ausgabe-Abhängigkeit** von S nach S':
  - S schreibt eine Variable, die S' überschreibt
- Beispiel:

```
S1: f = computeFrame();
S2: showFrame(f);
S3: c = compress(f);
S4: saveFrame(c);
```

- Anti- und Ausgabe-Abh. gibt es nur in imperativen Sprachen
  - werden durch Überschreiben von Variablen verursacht
  - sind immer durch Umbenennung von Variablen zu entfernen

Arten von Abhängigkeiten ...

- **Anti-Abhängigkeit** von S nach S':
  - S liest eine Variable, die S' überschreibt
- **Ausgabe-Abhängigkeit** von S nach S':
  - S schreibt eine Variable, die S' überschreibt
- Beispiel:

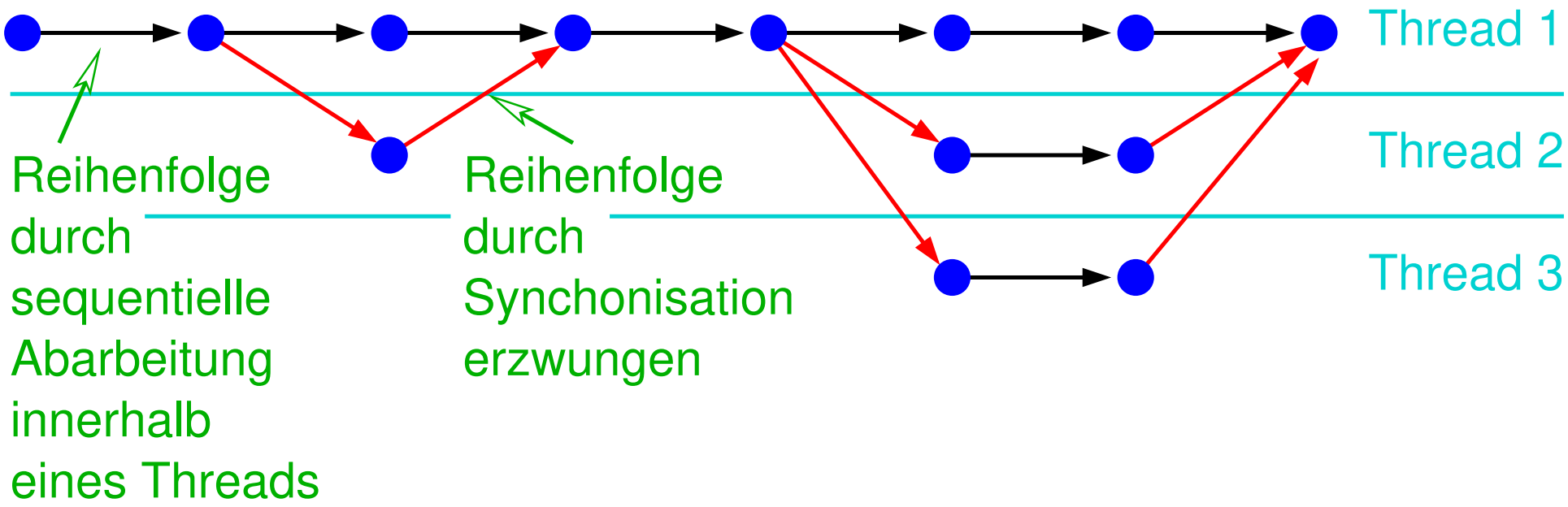
```
S1: f = computeFrame();
S2: showFrame(f);
S3: c = compress(f);
S4: saveFrame(c);
```

Echte Abhängigkeiten

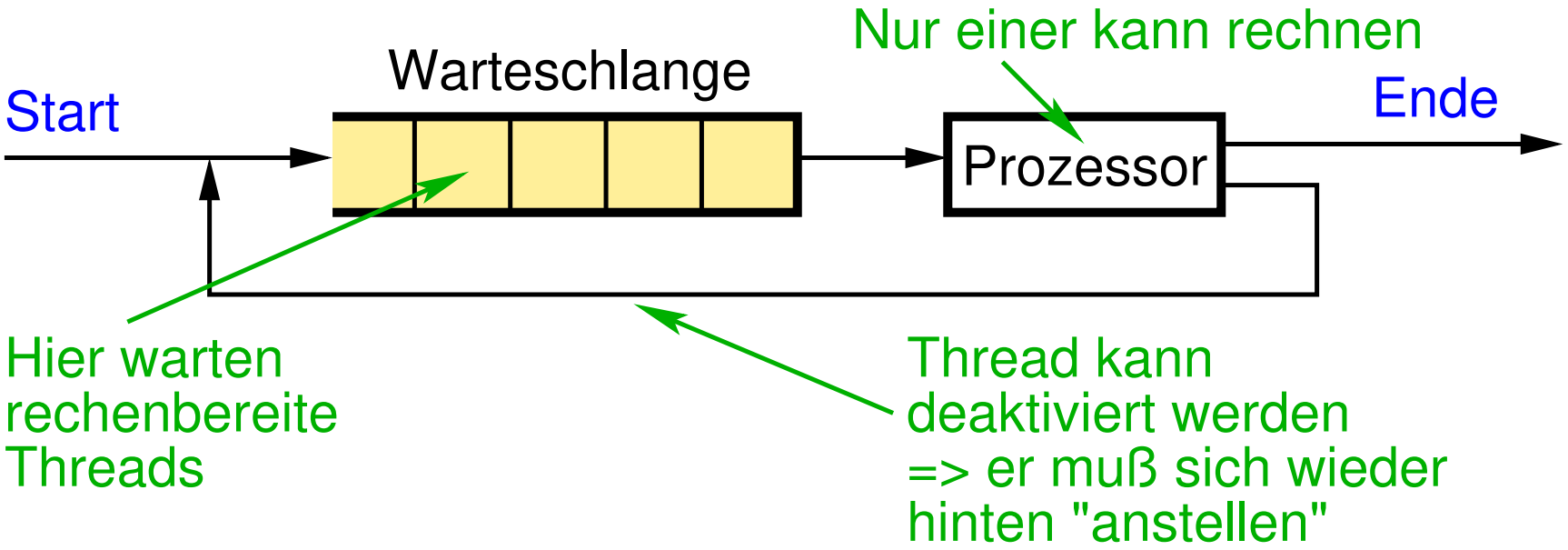
- Anti- und Ausgabe-Abh. gibt es nur in imperativen Sprachen
  - werden durch Überschreiben von Variablen verursacht
  - sind immer durch Umbenennung von Variablen zu entfernen

Abhängigkeiten und Synchronisation

- Abhängigkeit zwischen S und S' bedeutet nicht, dass S und S' im selben Thread ausgeführt werden müssen
- Aber: Synchronisation nötig, um Reihenfolge zu erzwingen
- Im Eingangsbeispiel:

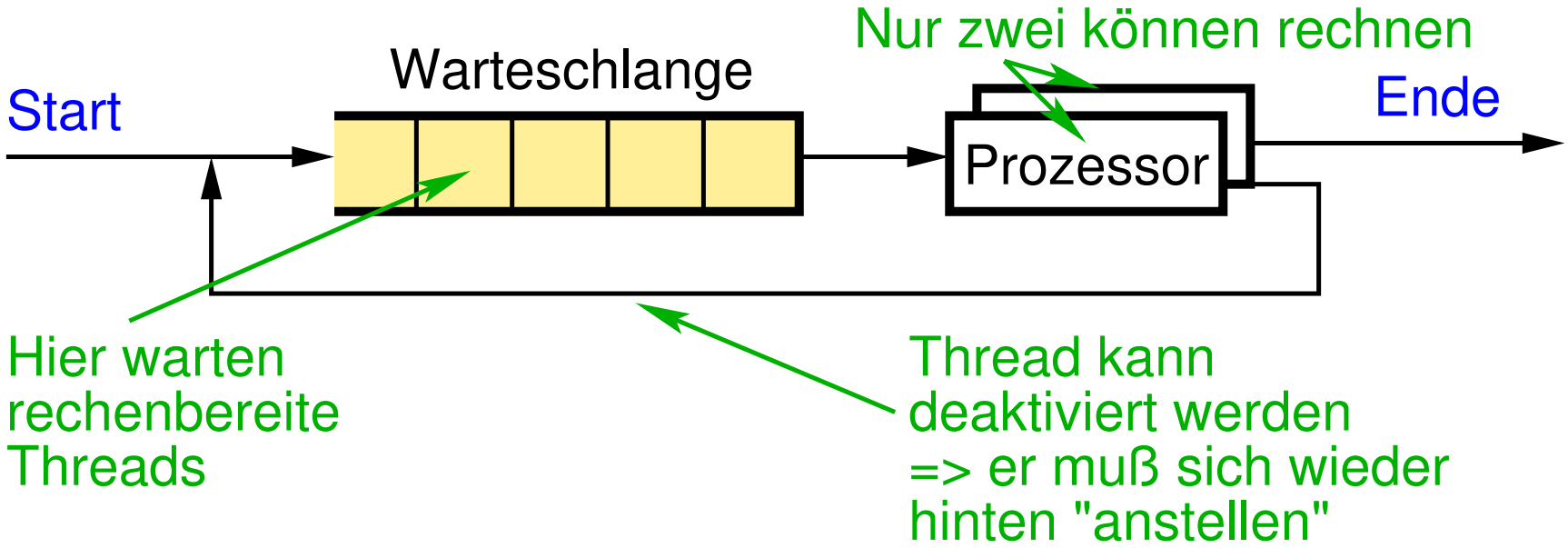


Ein einfaches Thread-Modell



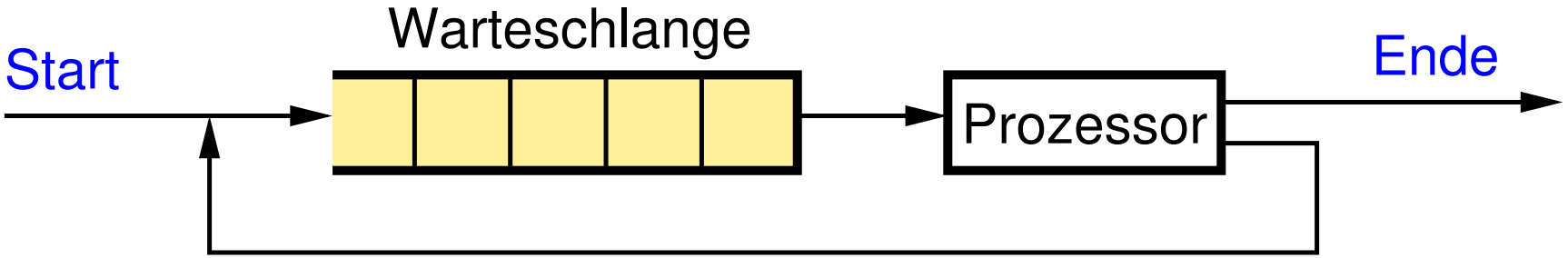
- Anmerkung: alle Modelle in diesem Abschnitt gelten in der selben Form auch für klassische Prozesse (mit genau einem Thread)

Ein einfaches Thread-Modell

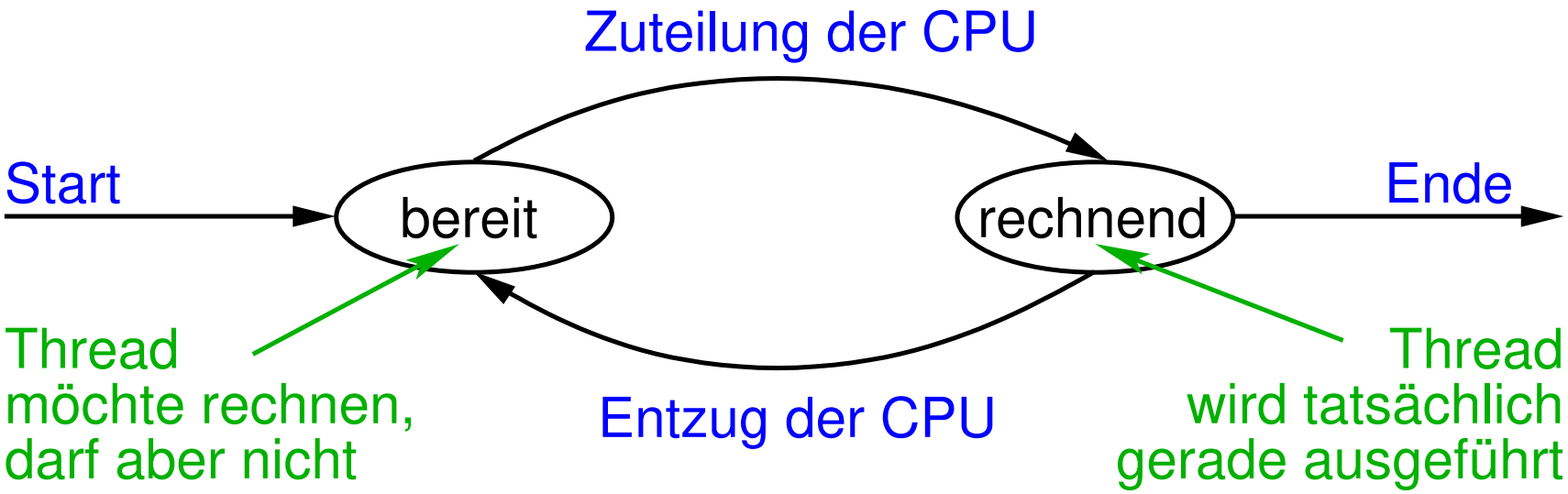


- Anmerkung: alle Modelle in diesem Abschnitt gelten in der selben Form auch für klassische Prozesse (mit genau einem Thread)

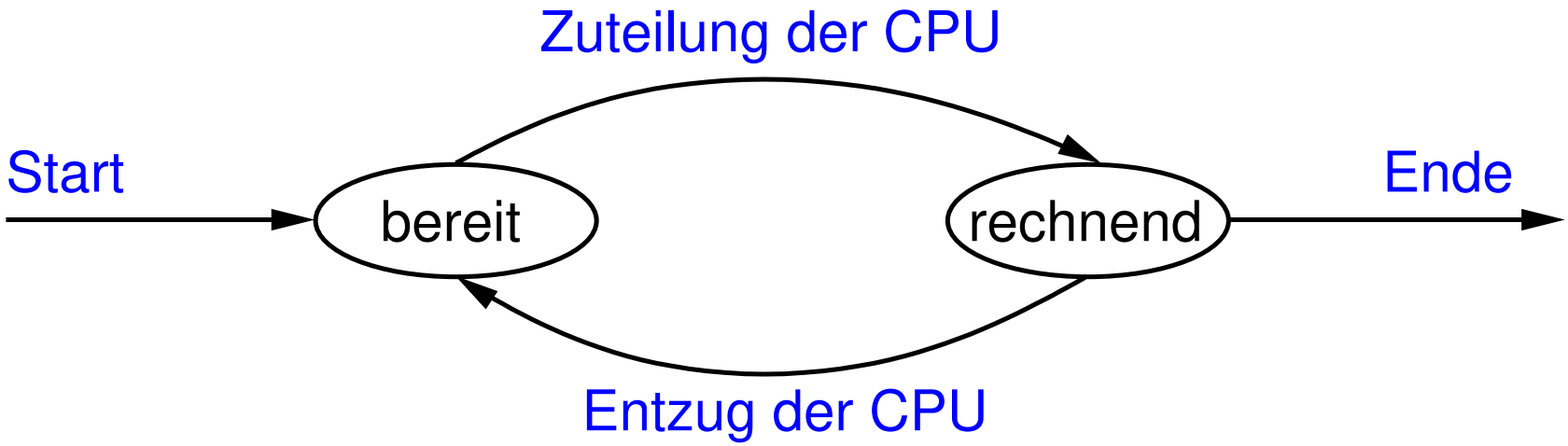
Ein einfaches Thread-Modell ...



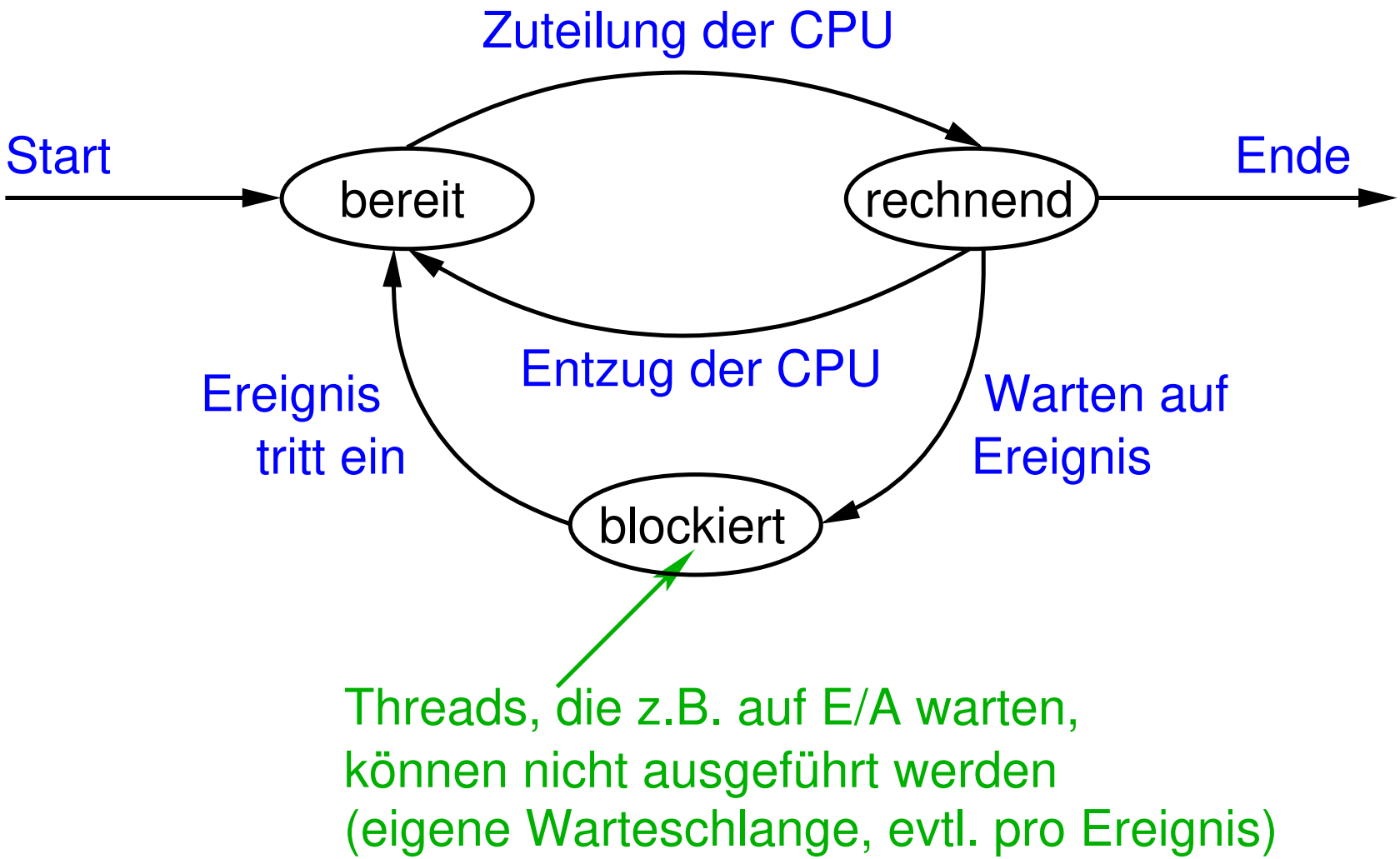
Zustandsgraph eines Threads



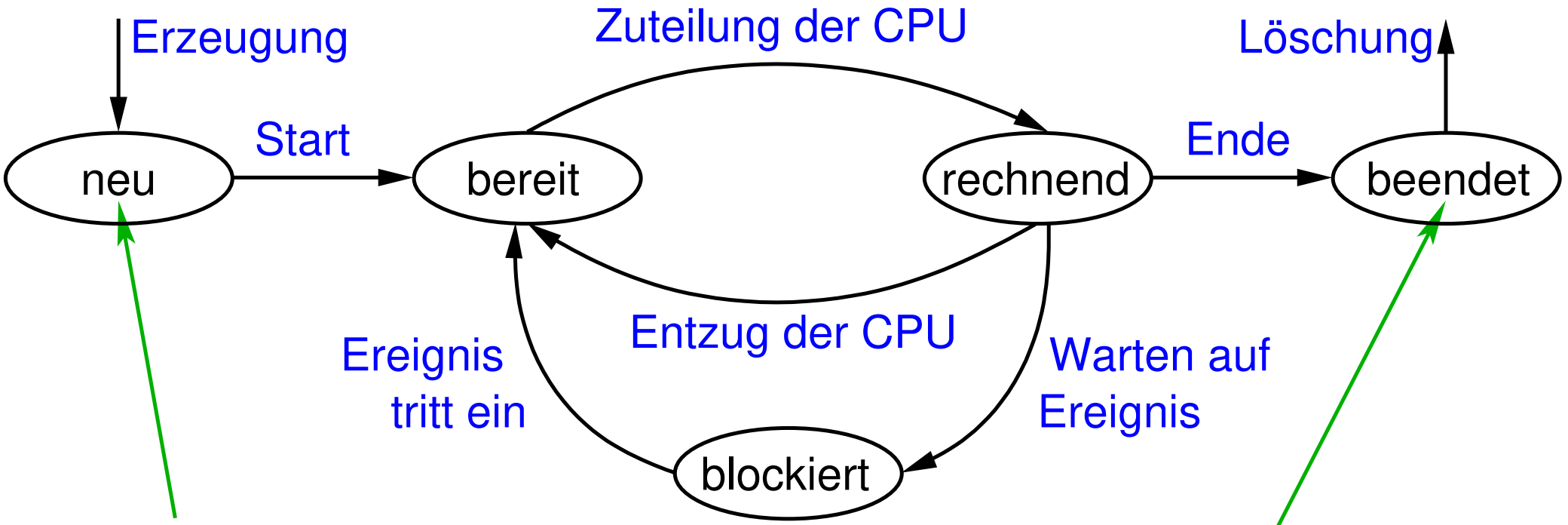
Zustandsgraph für ein erweitertes Thread-Modell



Zustandsgraph für ein erweitertes Thread-Modell



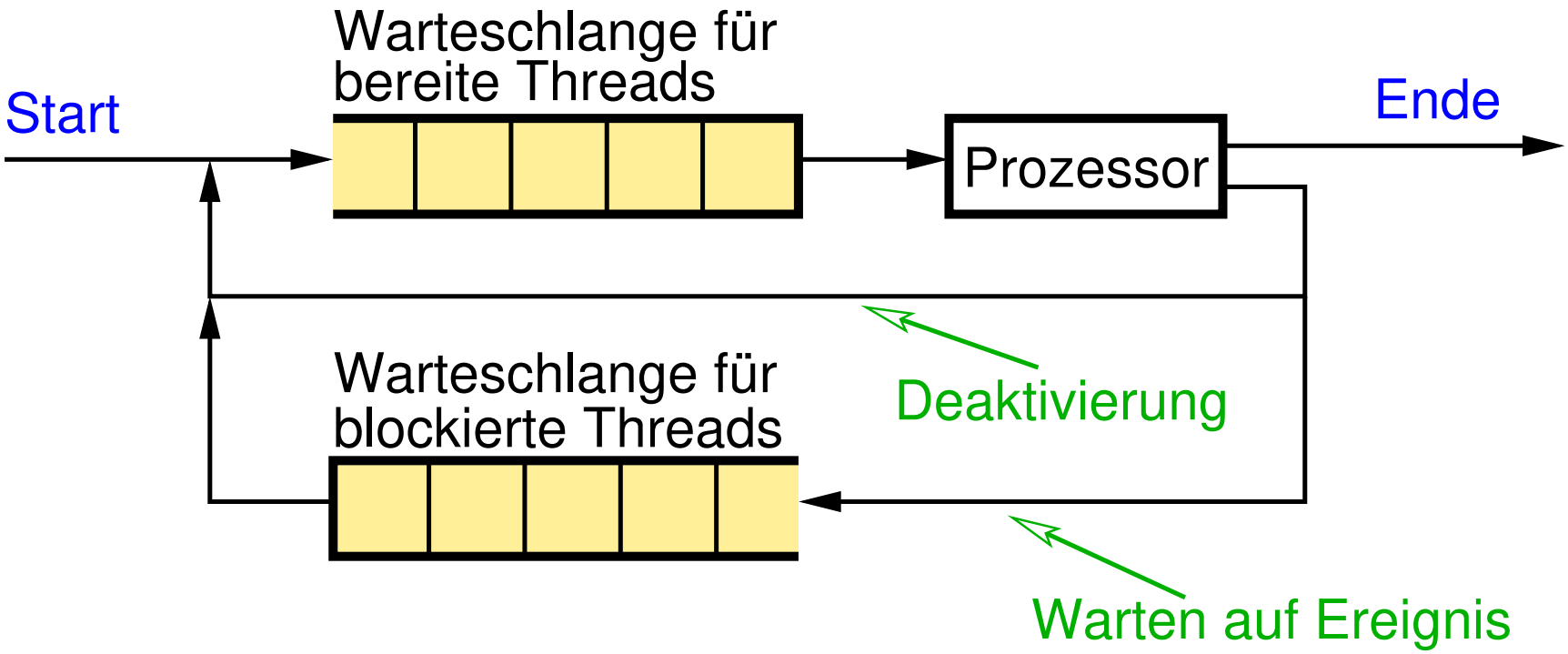
Zustandsgraph für ein erweitertes Thread-Modell ...



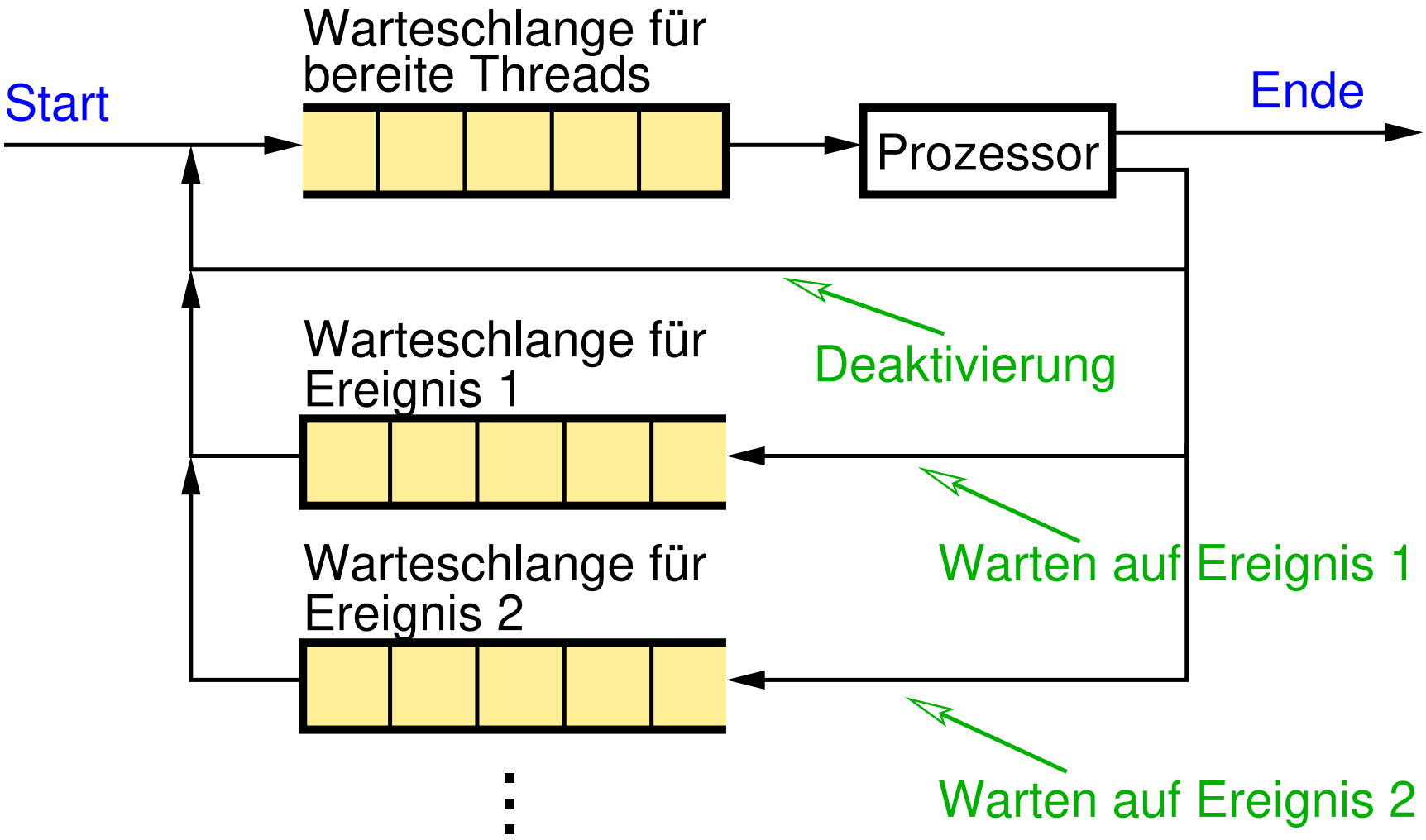
Die Verwaltungsdatenstruktur für den Thread ist bereits angelegt, der Thread selbst existiert aber noch nicht

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

Eine Warteschlange für alle blockierten Threads



Eine Warteschlange pro Ereignis

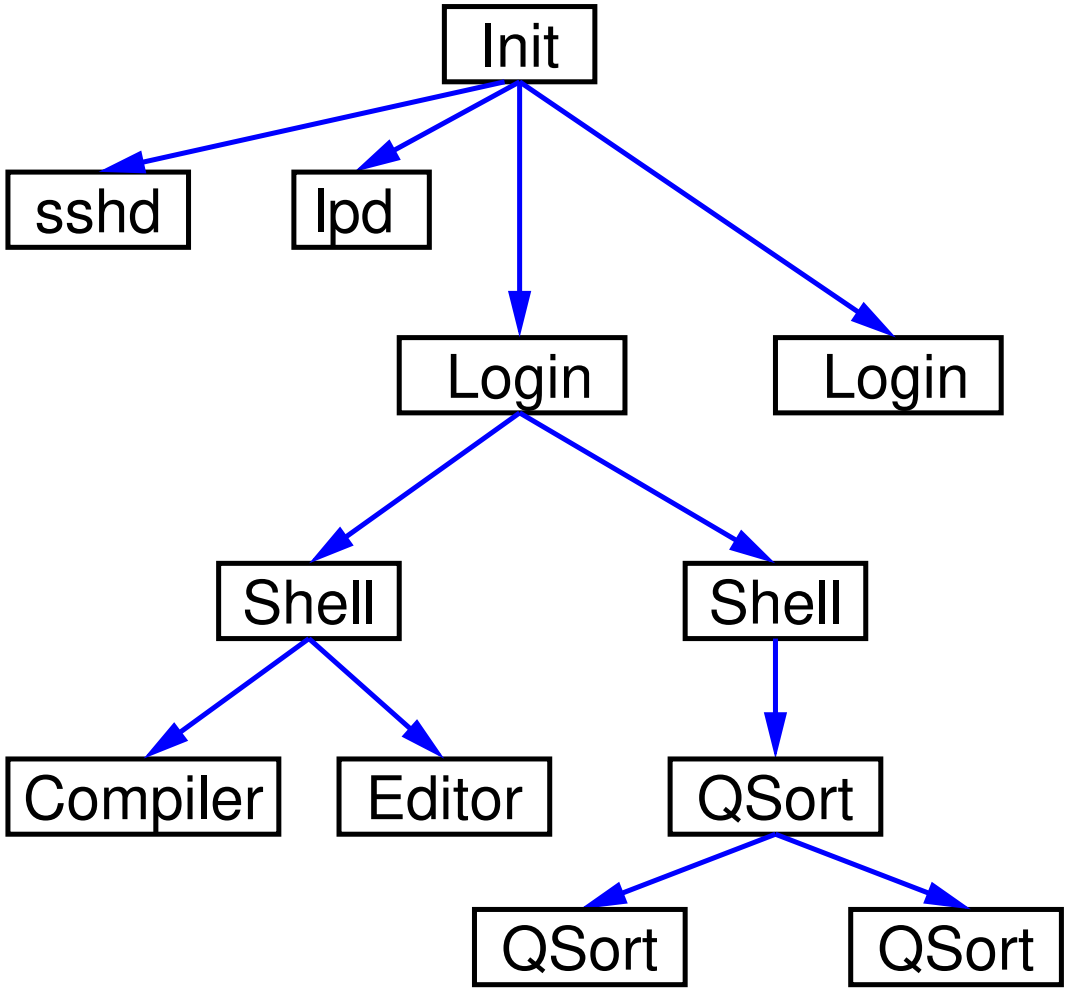


2.4 Erzeugung u. Terminierung von Prozessen

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

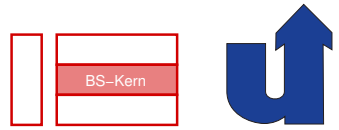
Für jeden Benutzer ein Login–Prozeß

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 (☞ **4.4**: Signale)



- BS pflegt **Prozeßtabelle** mit Informationen über alle Prozesse
  - der Eintrag für einen Prozeß heißt **Prozeßkontrollblock**
- Analog: **Threadtabelle** für alle Threads
  - Eintrag: **Threadkontrollblock**
- Prozeßadressraum, Prozeßkontrollblock und Threadkontroll-blöcke 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, 4**)
  - Prozeßprivilegien
  - Tabellen für Speicherabbildung (Speicherverwaltung, ☞ **8**)
  - Ressourcenbesitz und -nutzung
    - offene Dateien, Arbeitsverzeichnis, ...

Inhalt des Threadkontollblocks

- 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 physischem Adreßraum an den Prozeß
  - für Programmcode, Daten, und Keller
  - (siehe später: **8**. 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 und alle anderen Register
2. Thread- und Prozeßkontrollblock aktualisieren
   - ➥ Threadzustand, Grund der Deaktivierung, Buchhaltung, ...
3. Thread in entsprechende Warteschlange einreihen
4. Nächsten bereiten Thread auswählen (☞ **7**. Scheduling)
5. Threadkontrollblock des neuen Threads aktualisieren
   - ➥ Zustand auf „rechnend“ setzen
6. Falls neuer Thread in anderem Prozess liegt:
   - ➥ Aktualisierung der Speicherabbildung in der MMU (☞ **8**)
7. Prozessorstatus aus neuem Threadkontrollblock laden

Anmerkungen

- Beim Threadwechsel **innerhalb desselben Prozesses** entfällt die Aktualisierung der Speicherabbildung in der MMU
  - Threads im selben Prozeß haben gemeinsamen Adressraum
- 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

(vgl. Folie 38)

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

(Die Zahlen in Klammern beziehen sich auf Folie 109)

Ablauf bei Ausnahme

(vgl. Folie 37)

- 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, ☞ **8.3.2**: dyn. Seitenersetzung)
    - Behebung der Ursache der Ausnahme
  - Sprung zum Scheduler (4-7)

(Die Zahlen in Klammern beziehen sich auf Folie 109)

Ablauf bei Interrupt

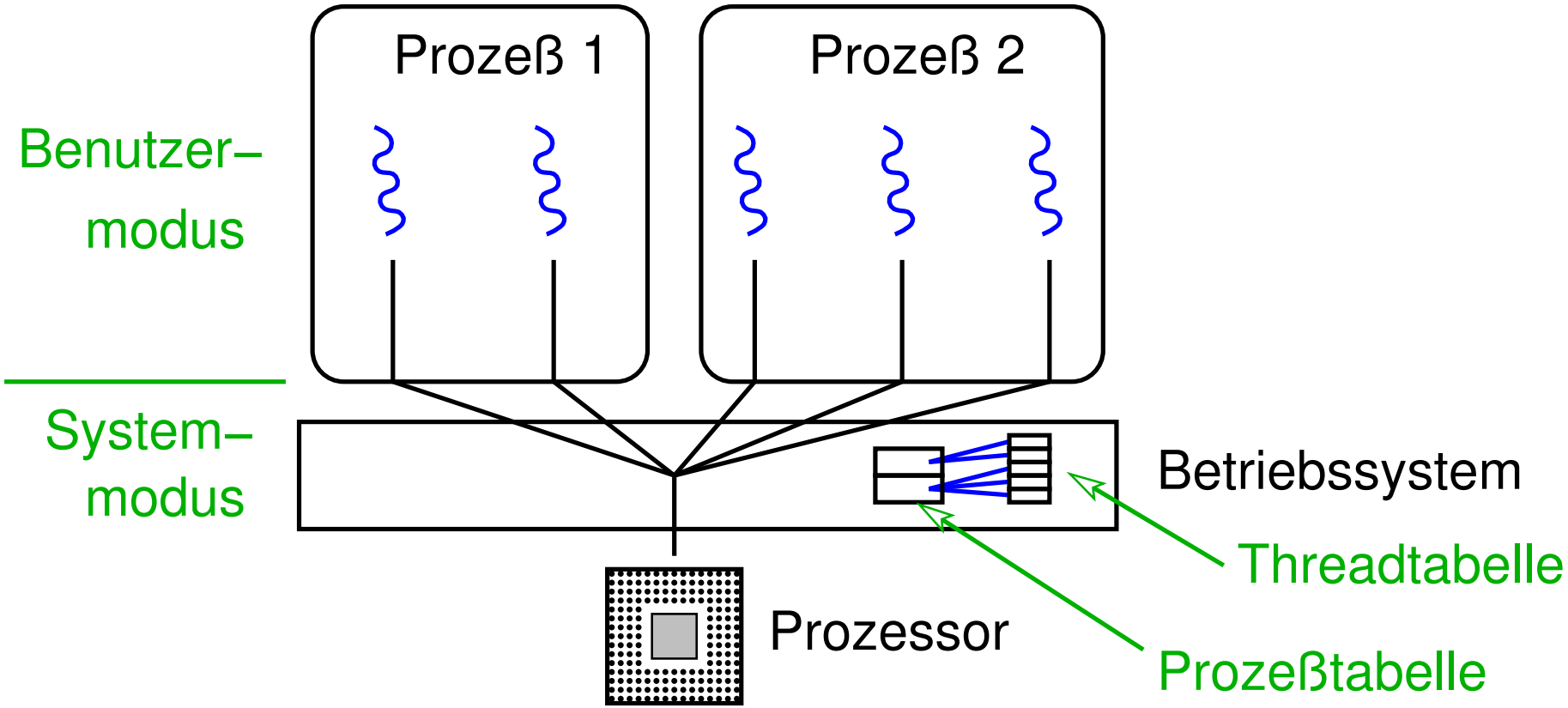
(vgl. Folie 39)

- Durch Hardware: Einsprung ins BS (Systemmodus)
- Ablauf im BS:
  - Sichern des gesamten Prozessorstatus (1)
  - aktuellen Thread auf „bereit“ setzen (2,3)
  - Ursache der Unterbrechung ermitteln
  - Ereignis (z.B. Ende der E/A) entsprechend behandeln
  - evtl. blockierte Threads wieder auf „bereit“ setzen
  - Sprung zum Scheduler (4-7)

(Die Zahlen in Klammern beziehen sich auf Folie 109)

2.6 Realisierungsvarianten für Threads

Realisierung durch BS-Kern

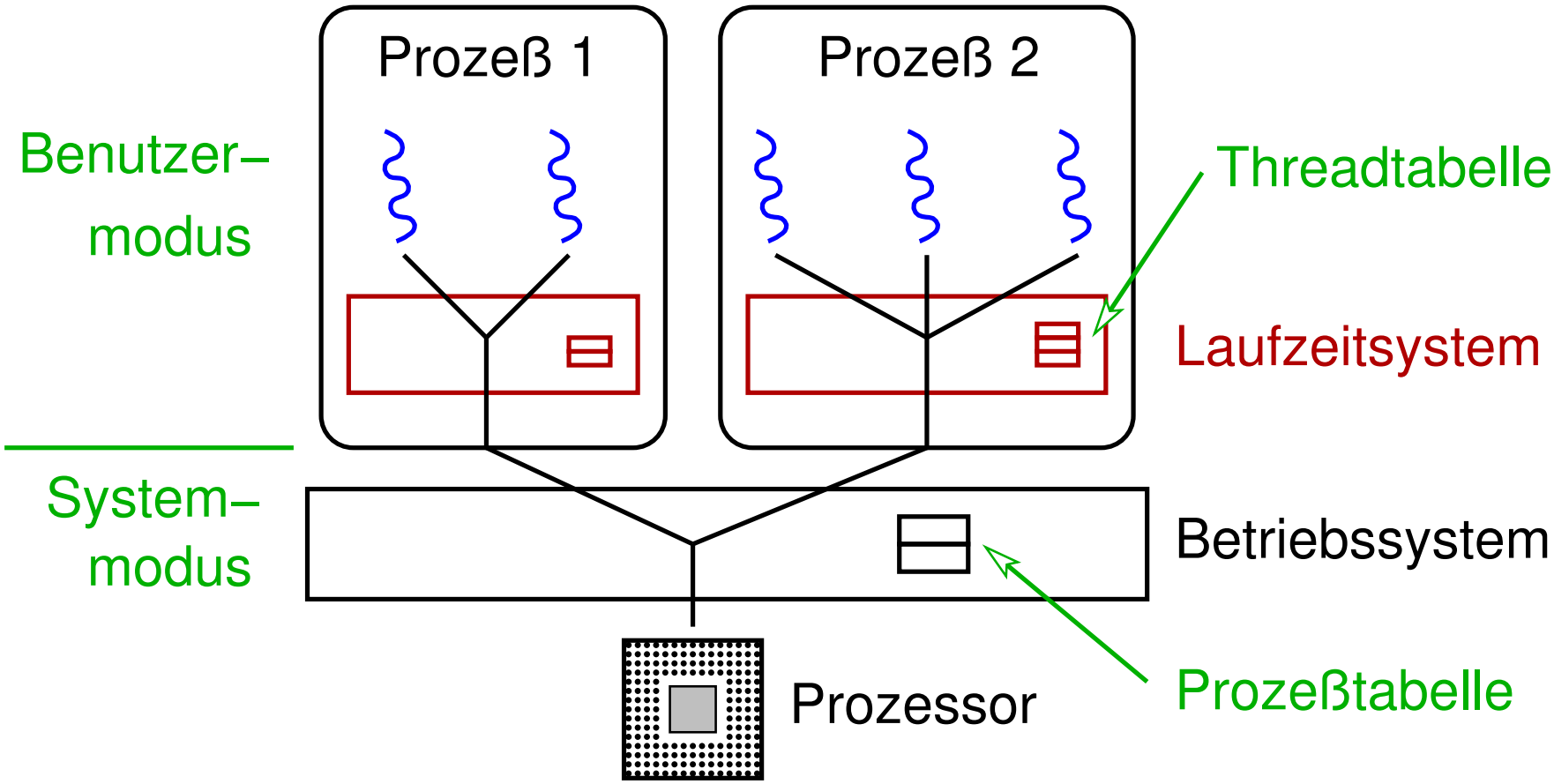


➥ Heute gängigste Realsierungsvariante

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

Realisierung im Benutzeradreßraum



➥ Genutzt in frühen Thread-Implementierungen

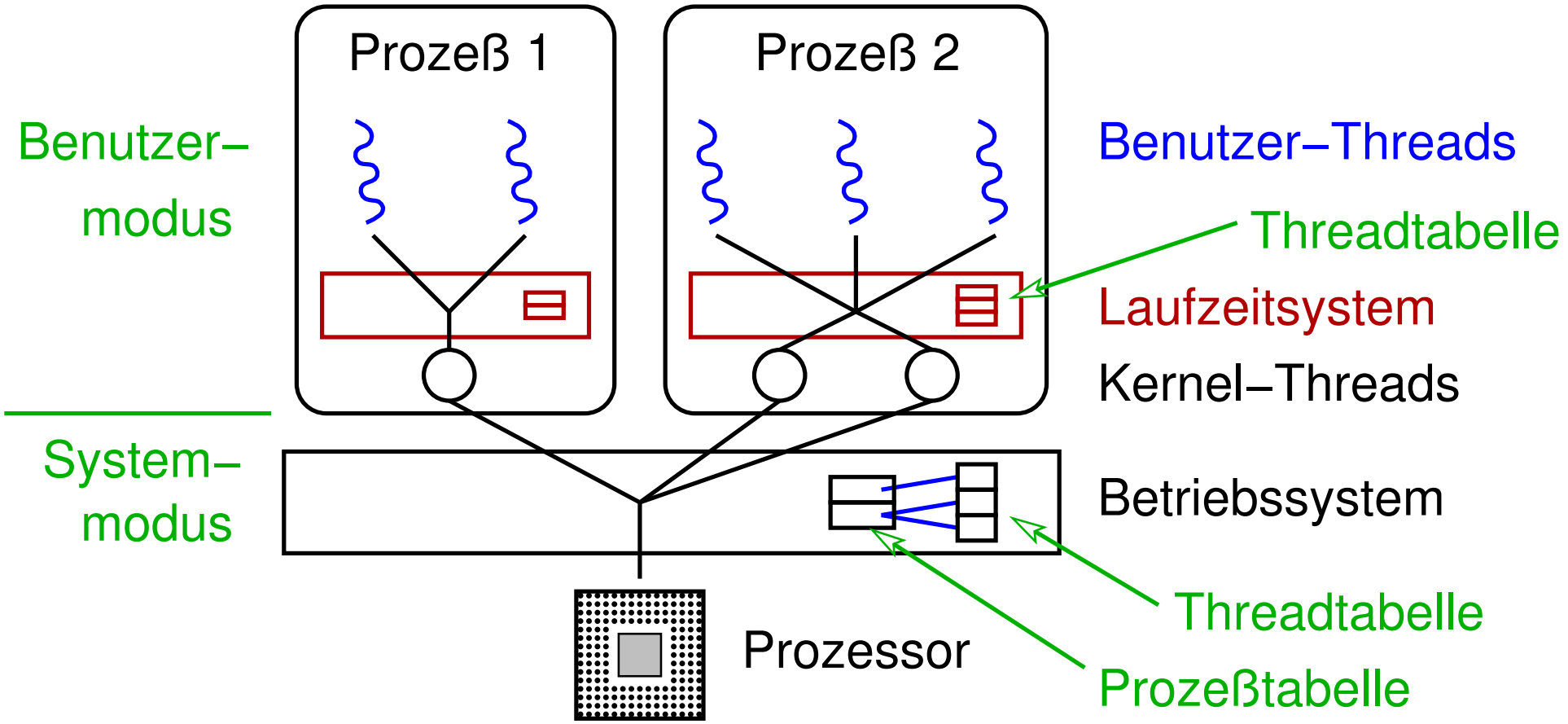
Realisierung im Benutzeradreßraum: Diskussion

- Vorteile:
  - keine Unterstützung durch BS notwendig
  - schnelle Threaderzeugung und Threadwechsel
    - z.B. Zeit für Erzeugung (Linux 4.19, Intel i7, 3.4 GHz)

Benutzer-Thread	Kernel-Thread	Prozess
0,03 μs	20 μs	200 μs

  - 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

Hybride Realisierung



➥ Sinnvoll für Programme mit sehr vielen nebenläufigen Aktivitäten

2.7 Schnittstellen zur Nutzung von Threads

- Alle heute gängigen BSe unterstützen Threads
- Anwendungen nutzen jedoch i.d.R. nicht die Systemaufrufe, sondern höhere Programmierschnittstellen
- Beispiele:
  - POSIX Threads (zusätzliche Programmierbibliothek für C)
  - C++-Threads (Teil der C++-Standardbibliothek)
  - Java Threads (in Sprache und Klassenbibliothek integriert)

2.7.1 Threads in Java

- Java unterstützt in der Sprache die Programmierung mit Threads
  - unterstützt durch die Java Klassen-Bibliothek
  - unabhängig vom Betriebssystem
- Programmiermodell:
  - bei Start des Programms: genau ein (Master-)Thread
  - Master-Thread erzeugt bei Bedarf weitere Threads
  - Prozess terminiert erst, wenn alle Thread beendet sind

Die Klasse `Thread`

- Die Methode `void run()` dieser Klasse kann nebenläufig in einem neuen Thread ausgeführt werden
- Typisch: eigene Klasse erbt von `Thread` und überschreibt die Methode `run()`:

```
public class MyThread extends Thread {
  public void run() {
    ... // nebenläufig auszuführender Code
  }
}
```

- Achtung: ein Objekt dieser Klasse ist (noch) **kein** BS-Thread
- Erzeugung eines BS-Threads:
  - Aufruf der Methode `void start()`
  - nur einmal pro Java-Objekt erlaubt

Weitere Methoden der Klasse `Thread` (unvollständig)

- `void join()`
  - wartet bis der BS-Thread seine Ausführung beendet hat
- `void setDaemon()`
  - markiert den Thread als Hintergrund-Thread
  - am Programmende wird der BS-Thread terminiert, ohne zu warten

Beispiel: Hello World!

```
public class MyThread extends Thread
{
  public void run() // Methode wird durch den neuen Thread abgearbeitet
  {
    System.out.println("Hello World!");
  }
  public static void main(String[] args)
         throws InterruptedException
  {
    MyThread t = new MyThread();   // Erzeuge Java–Objekt
    t.start();   // erzeuge neuen BS–Thread
    t.join();    // warte auf Beendigung des BS–Threads
  }
}
```

Besonderheiten

- Methode `run()` hat weder Argumente noch Rückgabewert
  - Datenaustausch mit dem Thread muß über Attribute erfolgen:
    - erzeuge Java-Objekt
      - initialisere Eingabe-Attribute im Konstruktor
    - starte BS-Thread (`start()`)
    - warte auf Beendigung des BS-Threads (`join()`)
    - lies Ergebnisse aus Attributen des Java-Objekts
- Erben von `Thread` nicht immer möglich (keine Mehrfachvererbung in Java)
  - daher auch Delegation möglich:
    - Konstruktor akzeptiert Objekt mit Schnittstelle `Runnable`
    - `run()` wird an dieses Objekt delegiert

Beispiel: Hello World 2!

```
public class Greeter implements Runnable
{
  private String whom;
  private int result;
  public Greeter(String whom) {
    this.whom = whom;
  }
  public int getResult() {
    return result;
  }
  public void run() // Methode wird durch den neuen Thread abgearbeitet
  {
    System.out.println("Hello " + whom + "!");
    result = 1;
  }
```

Beispiel: Hello World 2! ...

```
public static void main(String[] args)
{
  Greeter greet = new Greeter("World");
  Thread t = new Thread(greet); // Erzeuge Java–Objekt
  t.start();                    // erzeuge neuen BS–Thread
  try {
    t.join();                   // warte auf Beendigung des BS–Threads
  }
  catch (InterruptedException e) {
  }
  System.out.println("Result: " + t.getResult());
}
}
```

2.7.2 Threads in C++

- Der C++ Sprachstandard enthält seit 2011 (C++-11) eine Threadschnittstelle
  - implementiert durch C++ Standard-Bibliothek
  - unabhängig vom Betriebssystem
- Programmiermodell:
  - bei Start des Programms: genau ein (Master-)Thread
  - Master-Thread erzeugt bei Bedarf weitere Threads und sollte auf deren Beendigung warten
  - Prozess terminiert, wenn Master-Thread terminiert
    - falls noch andere Threads laufen, wird eine Exception geworfen

Erzeugung von Threads

- Klasse `std::thread`
  - repräsentiert einen laufenden Thread
- Erzeugung eines neuen Threads:

  `std::thread myThread(`*`function, args ...`*`);`
  - erzeugt ein neues C++-Objekt und einen neuen BS-Thread
  - BS-Thread wird terminiert (mit Exception), wenn C++-Objekt deallokiert wird
    - bei obiger Deklaration am Ende des Gültigkeitsbereichs von `myThread`
  - *`function`*: Funktion, die der BS-Thread ausführen soll
    - kein Rückgabewert, aber Ergebnisparameter möglich
  - `args ...`: Argumente, die an *`function`* übergeben werden

Methoden der Klasse `thread` (unvollständig)

- `void join()`
  - wartet bis der Thread seine Ausführung beendet hat
  - nach Rückkehr dieser Methode kann das C++-Objekt gelöscht werden
- `void detach()`
  - koppelt den BS-Thread von dem C++-Objekt ab
  - d.h. der BS-Thread läuft auch dann noch weiter, wenn das C++-Objekt gelöscht wird
  - die Methode `void join()` kann nicht mehr aufgerufen werden
- Die Klasse überschreibt u.a. den Zuweisungsoperator
  - stellt sicher, daß beim Kopieren des C++-Objekts *kein* neuer BS-Tread erzeugt wird

Beispiel: Hello World!

```
#include <iostream>
#include <thread>

void greet(std::string whom)
{
    std::cout << "Hello " << whom << "!\n";
}

int main(int argc, char **argv)
{
    std::thread t(greet, "World"); // Erzeuge einen neuen Thread
    t.join();                      // Warte auf Beendigung
    return 0;
}
```

2.7.3 Threads in C

- PThreads (IEEE 1003.1c): Standard-Schnittstelle zur Programmierung mit Threads
  - implementiert als System-Bibliothek
  - (weitgehend) unabhängig vom Betriebssystem
- Programmiermodell
  - bei Start des Programms: genau ein (Master-)Thread
  - Master-Thread erzeugt bei Bedarf weitere Threads und sollte auf deren Beendigung warten
  - Prozess terminiert, wenn Master-Thread terminiert

Erzeugung von Threads

- `int pthread_create(pthread_t *thread,`
  `pthread_attr_t *attr,`
  `void *(*function)(void *),`
  `void *arg)`
  - Eingabeparameter:
    - `attr`: Thread-Attribute
      - z.B. für Scheduling (i.a. BS-abhängig)
    - `function`: Funktion, die der Thread ausführen soll
    - `arg`: Argument, das an `function` übergeben wird
  - Ergebnisse:
    - Rückgabewert: Status (erfolgreich bzw. Fehler)
    - `*thread`: Zeiger auf (opake) Thread-Struktur

Thread Management (unvollständig)

- `void pthread_exit(void *retval)`
  - aufrufender Thread wird beendet (mit Rückgabewert `retval`)
- `int pthread_join(pthread_t thread, void **retval)`
  - wartet bis der gegebene Thread terminiert
  - gibt den Rückgabewert in `*retval` zurück
- `int pthread_cancel(pthread_t thread)`
  - sendet Terminierungsanfrage an den Thread
  - vor Terminierung: Aufruf eines *Cleanup-Handlers*
  - Thread kann Terminierungsanfragen ignorieren

Beispiel: Hello World!

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

void *greet(void *arg) {
  std::cout << "Hello " << (char*)arg << "!\n";
  return NULL;
}
int main(int argc, char **argv) {
  pthread_t t;
  // Erzeuge einen neuen Thread
  if (pthread_create(&t, NULL, greet, (void*)"World")!=0) {
    /* Fehlerbehandlung! */
  }
  pthread_join(t, NULL);   // Warte auf Beendigung
  return 0;
}
```

2.7.4 Ablauf einer Threaderzeugung (Linux)

1. Speicher für den (privaten) Keller des Threads anfordern
   - ➥ für Rückkehradressen, Argumente, lokale Variable, ...
   - ➥ typische Größe: 8 MiB*
   - ➥ Systemaufruf: `mmap` (legt Bereich im virtuellen Adreßraum an)
2. Kellerende durch schreib-/lesegeschützten Bereich sichern
   - ➥ bei Kellerüberlauf: Ausnahme
   - ➥ Systemaufruf: `mprotect` (Zugriffsrechte für Speicher setzen)
3. Neuen Thread erzeugen
   - ➥ Systemaufruf: `clone` (Prozess klonen)
   - ➥ hier: neuer „Prozess“ teilt sich alle Ressourcen mit dem alten, d.h. ist ein Thread
   - ➥ Parameter: Keller, aufzurufende Funktion und Argument

* Nach NIST: 1 KiB = 1024 Byte, 1 MiB = 1024 KiB, ...

2.8 Zusammenfassung / Wiederholung

- Zwei Aspekte:
  - Prozeß: Einheit der Ressourcenverwaltung, Schutzeinheit
  - Thread: Einheit der Prozessorzuteilung
    - pro Prozeß mehrere Threads möglich
- Thread-Schnittstellen:
  - Java Threads
- Threadmodell
  - Zustände „rechnend“, „bereit“, „blockiert“ + andere
  - Warteschlangen
- Zum Prozeß gehören u.a.:
  - Adreßraum, geöffnete Dateien, Signale, Privilegien, ...

2.8 Zusammenfassung / Wiederholung ...

- Zum Thread gehören u.a.:
  - Befehlszähler, CPU-Register, Keller(zeiger), Scheduling-Zustand, ...
- Threadwechsel:
  - Umladen des Prozessorkontexts
  - bei Prozeßwechsel auch Wechsel der Speicherabbildung
  - kann bei Systemaufruf, Ausnahme und Interrupt erfolgen