UNIVERSITÄT BERN

2405 Betriebssysteme
III. Prozesse und Threads

Thomas Staub, Markus Anwander
Universität Bern

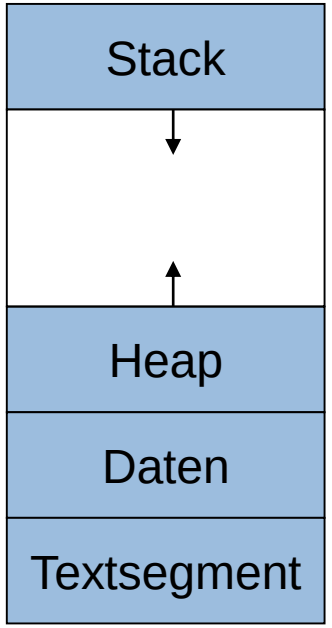
Inhalt

1.1.1 Definition Prozess

- informell: Prozess = Programm in Ausführung
- besteht aus
  - Programmcode (Textsegment)
  - Aktivität: Befehlszähler + Prozessorregisterinhalt
  - Stack (Laufzeitkeller) mit temporären Daten
  - globale Daten
  - Heap für dynamisch zugeteilten Speicher
- Jeder Prozess besitzt eine virtuelle CPU.
- Prozesse stellen selbst Betriebsmittel dar.

zeigt, wo im code wir sind

lokale variabeln auf stack

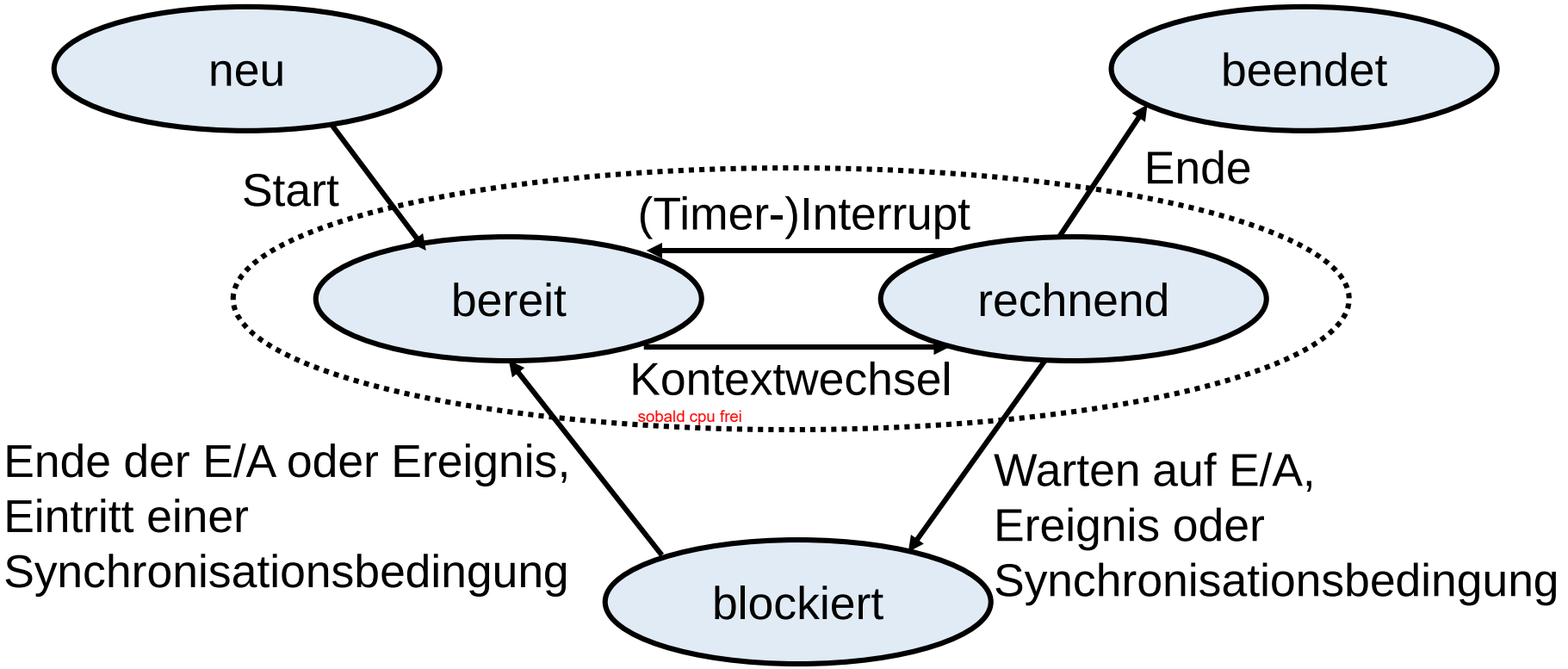


globale variablen sind hier, programmcode auch

1.1.2 Prozesszustände

- neu
  - Prozess wurde erzeugt.
- rechnend
  - Instruktionen des Prozesses werden auf der CPU ausgeführt.
- blockiert
  - Prozess wartet auf ein Ereignis sollten möglichst von cpu entfernt werden, um effizienz
    (z.B. Ende einer E/A-Operation, Signal, Eintritt einer Synchronisationsbedingung).
- bereit
  - Prozess wartet auf CPU-Zuteilung.
- beendet
  - Prozess hat die Ausführung abgeschlossen.

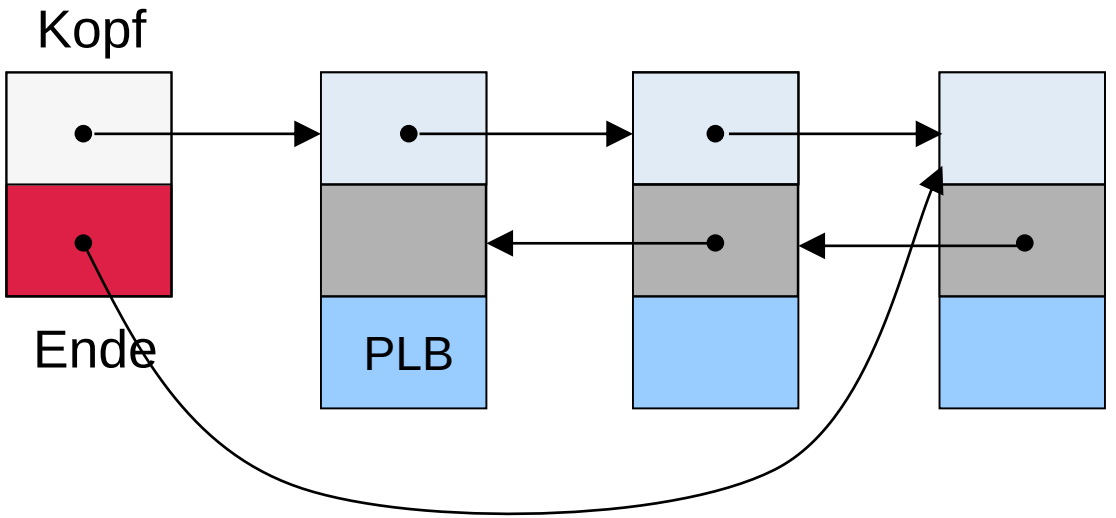
1.1.3 Prozesszustandsdiagramm



k rechnende Prozesse zu einem Zeitpunkt (k: Anzahl der Prozessoren)

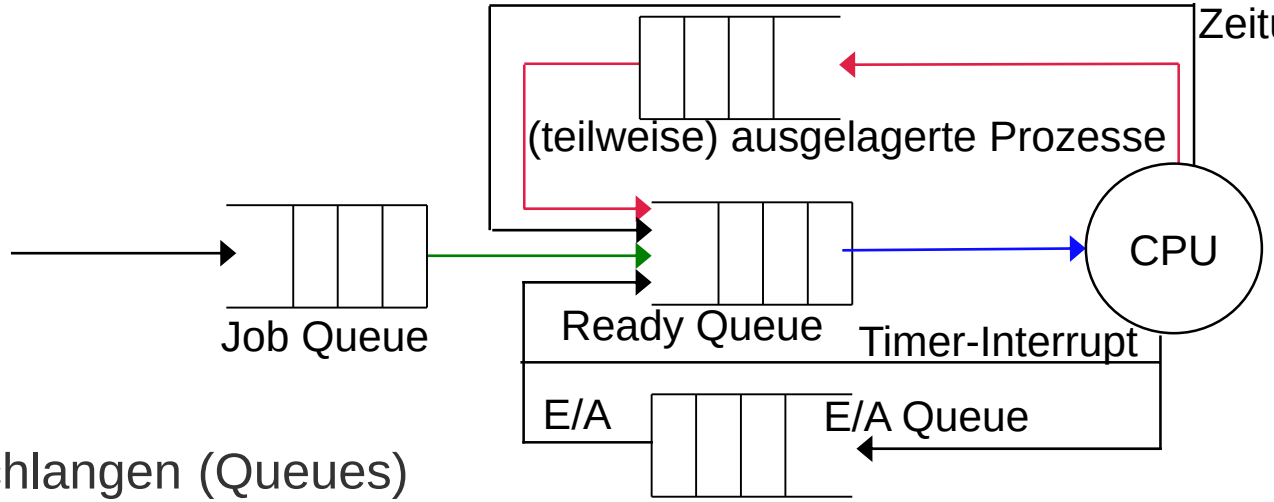
1.1.4 Prozessleitblock

- Beschreibung des Prozesszustands,
  z.B. auch der zur Ausführung benötigten Betriebsmittel
- Prozessleitblock (PLB) repräsentiert einen Prozess im Betriebssystem.
- Verkettung von Prozessleitblöcken in den Warteschlangen



Zeiger
Prozesszustand
Prozessnummer
Befehlszähler
Register
Speicherverwaltungs-Info.
CPU-Scheduling-Info.
Accounting-Info.
E/A-Status-Info. ...

1.2.1 Scheduling



- Warteschlangen (Queues)
  - Job Queue: abgeschickte, auf Massenspeicher abgelegte Prozesse
  - Ready Queue: Menge aller bereiten Prozesse im Hauptspeicher
  - E/A(-Geräte)-Queue: Menge aller auf E/A-Gerät wartenden Prozesse
  - Queue (teilweise) ausgelagerter Prozesse
- Prozesse migrieren zwischen einzelnen Warteschlangen.
- Arten von Scheduling
  - Kurzfristiges Scheduling: zur Prozessorzuteilung an bereite Prozesse
  - Mittelfristiges Scheduling: temporäres Aus- und Einlagern von Prozessen (Swapping)
  - Langfristiges Scheduling:
    weniger häufige, aber komplexere Entscheidungen, z.B. Mix von CPU- und E/A-gebundenen Prozessen

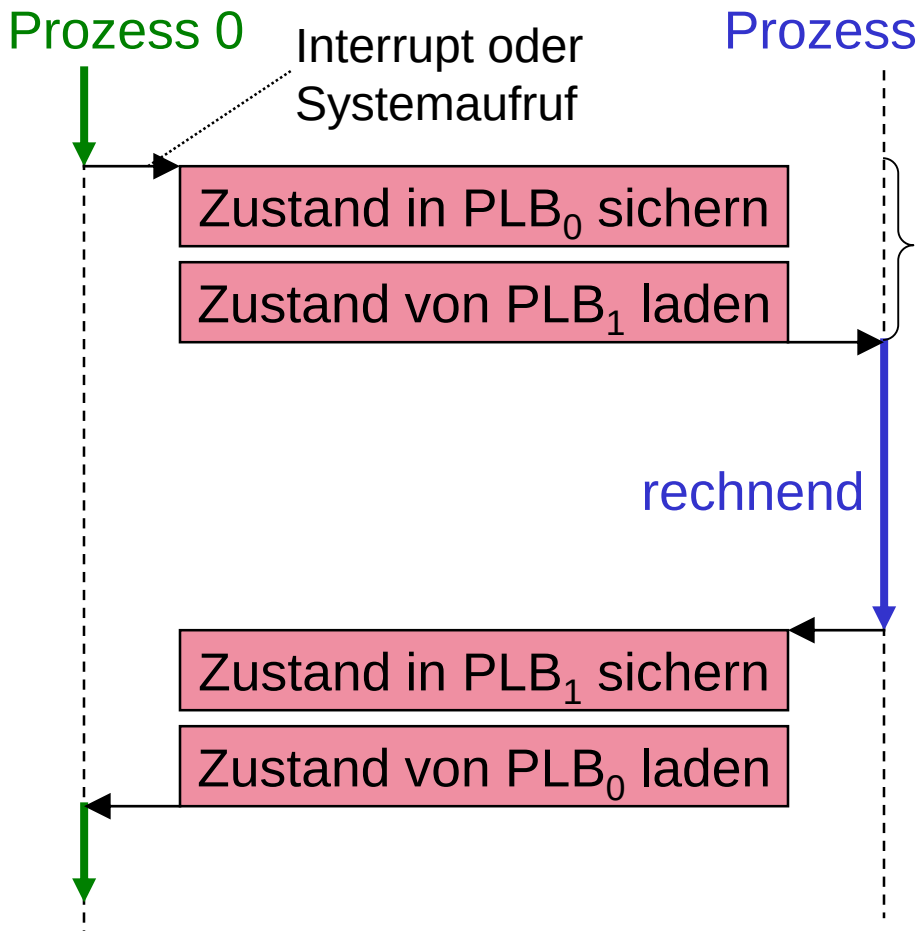
1.2.2 Dispatcher

Aufgabe: gibt dem vom Scheduler ausgewählten Prozess die Kontrolle über die CPU

- Kontextwechsel
- Wechsel in den Benutzermodus
- Sprung auf korrekte Stelle im Anwendungsprogramm und Fortsetzung der Ausführung

zustan von aktuellem prozess auf cpu sichern und zustand des neuen prozesses herstellen/wiederherstellen

1.2.3 Prozessumschaltung



> Kontextwechsel durch Dispatcher
  - Sicherung des alten Prozesszustands in Prozessleitblock (PLB)
  - Laden des neuen Prozesszustands aus PLB

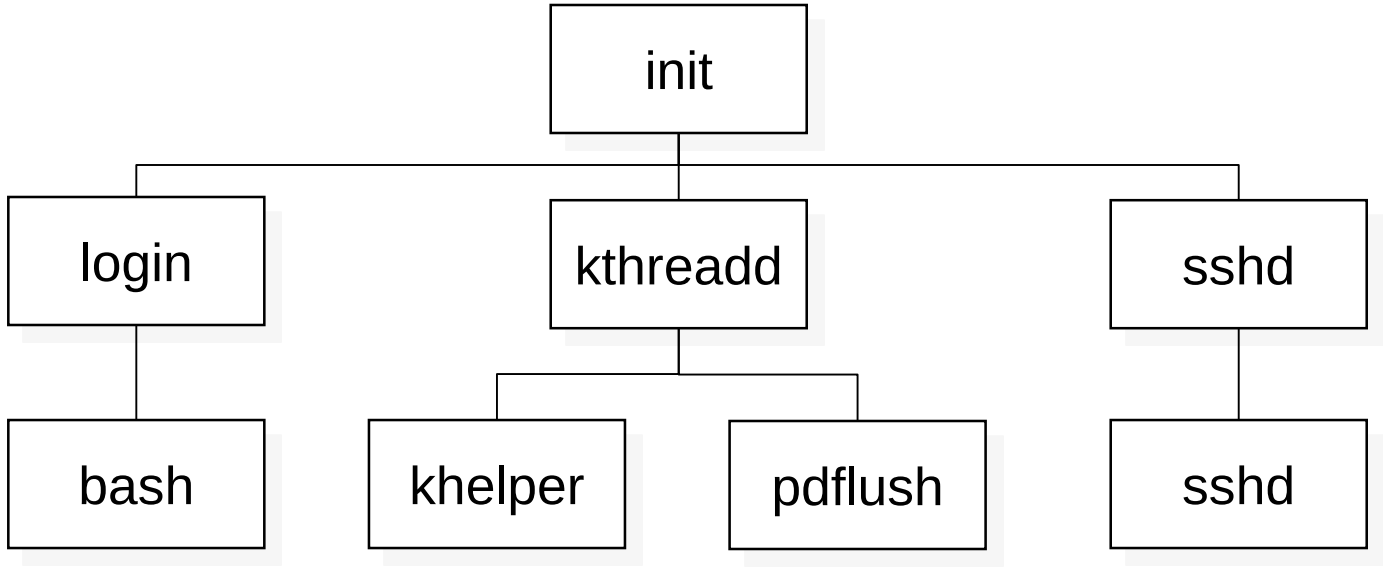
> relativ hoher Overhead (1-1000 μs)

1.3.1 Prozesserzeugung

- Prozesserzeugung bei Systeminitialisierung, durch Benutzer oder laufende Prozesse
- Elternprozess erzeugt Kindprozess (z.B. durch Systemaufruf **`Create_process`**), Kindprozesse erzeugen weitere Kindprozesse.
- Optionen
  - Teilen von Ressourcen
    - Eltern und Kinder teilen alle Ressourcen.
    - Kinder teilen nur Untermenge der Ressourcen der Eltern.
    - Eltern und Kinder teilen keine Ressourcen.
  - Ausführung
    - Nebenläufiges Ausführen von Eltern- und Kindprozessen
    - Eltern warten bis Kindprozesse beendet sind.
  - Adressraum
    - Kindprozess als Duplikat des Elternprozess
    - Kindprozess lädt eigenes Programm.

1.3.2 Beispiel: Prozesserzeugung in UNIX

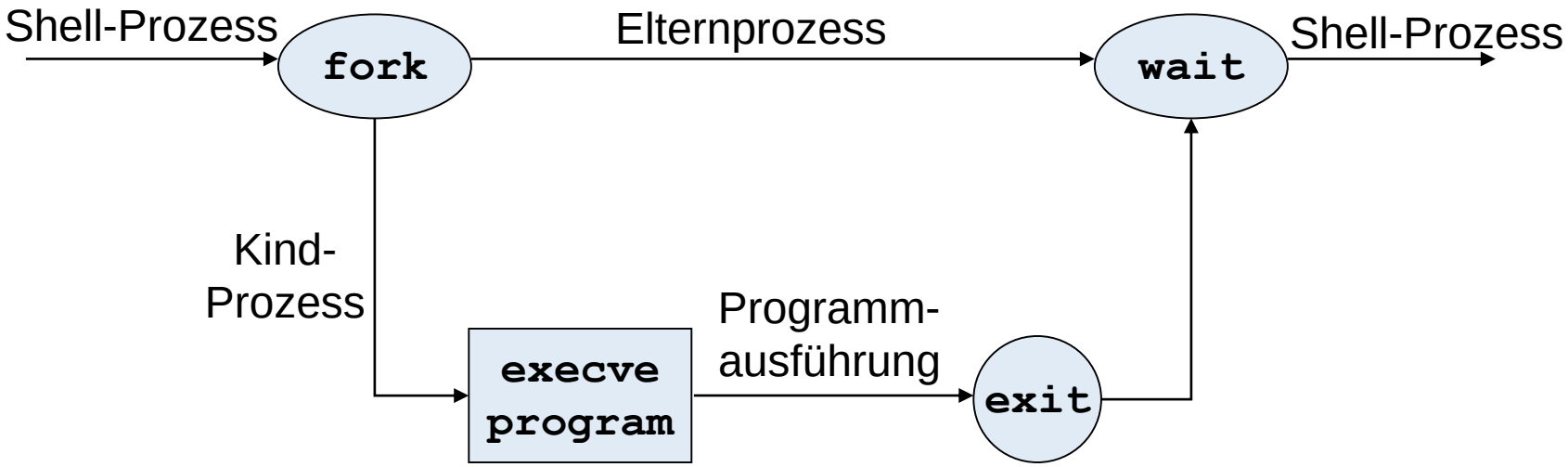
- **`fork`** erzeugt Prozesskopie mit neuer Prozessnummer (ID)
  - return-code = 0: Kindprozess
  - return-code ≠ 0: Elternprozess erhält ID des Kindprozess (zum späteren Löschen).
- **`execve`** ersetzt nach **`fork`** Speicherbereich mit neuem Programm.
- Beispiel: Linux



1.3.3 Beenden von Prozessen

- Prozess führt letzte Anweisung aus und beauftragt Betriebssystem mit der Löschung (**`exit`**).
- Elternprozess kann auf Beendigung eines Kindprozess inklusive Datenrückgabe warten (**`wait`**).
- Elternprozess beendet Kindprozess (**`abort`**), mögliche Gründe:
  - Zugeteilte Ressourcen wurden verbraucht.
  - Kindprozess wird nicht länger benötigt.
  - Elternprozess terminiert.

1.3.4 Beispiel: Prozesssteuerung in UNIX



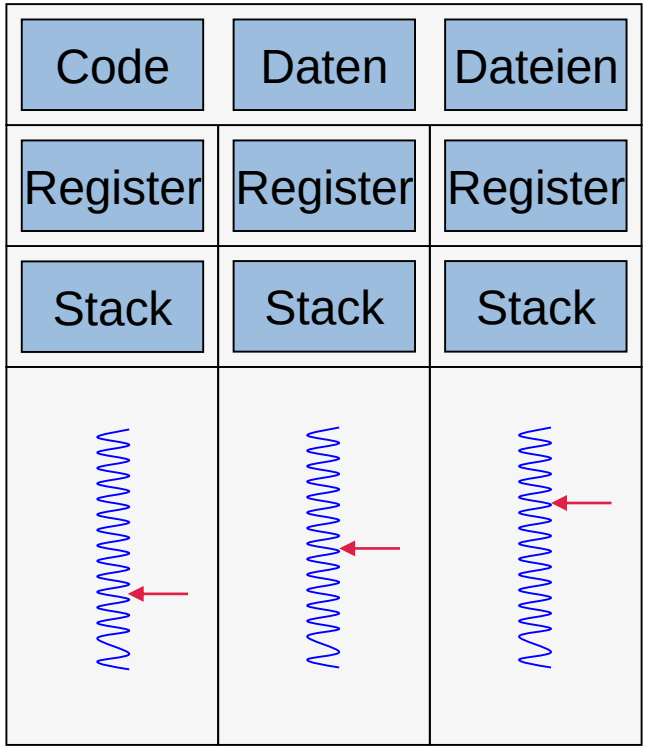

```
pid = fork();
if (pid < 0){
        /* fork failed */
} else if (pid > 0) {
        /* Eltern-Code */
} else {
        /* Kind-Code */
}
```

2.1.1 Threads und Tasks

- Thread = leichtgewichtiger Prozess (Englisch: Faden)
  - besteht aus
    - Thread ID
    - Befehlszähler
    - Register
    - Stack
  - teilt mit anderen Threads
    - Programmcode
    - globale Daten
    - Betriebssystemressourcen (offene Dateien, Signale)
- Task = Ansammlung von Threads
- traditioneller, schwergewichtiger Prozess = Task + 1 Thread

2.1.2 Threads

- Ein Thread einer Task kann arbeiten während andere blockiert sind.
  → Parallelität, Beispiel: File-Server
- Erzeugen von Kind-Threads
- Koordination und Steuerung erforderlich
- Gemeinsamer Speicher vereinfacht Interprozesskooperation.
- keine Schutzmechanismen
- effiziente Kontextwechsel (gemeinsame Ressourcen)
- Multi-Threading: mehrere Threads in einem Prozess (Task)
- Thread-Zustände wie bei Prozessen



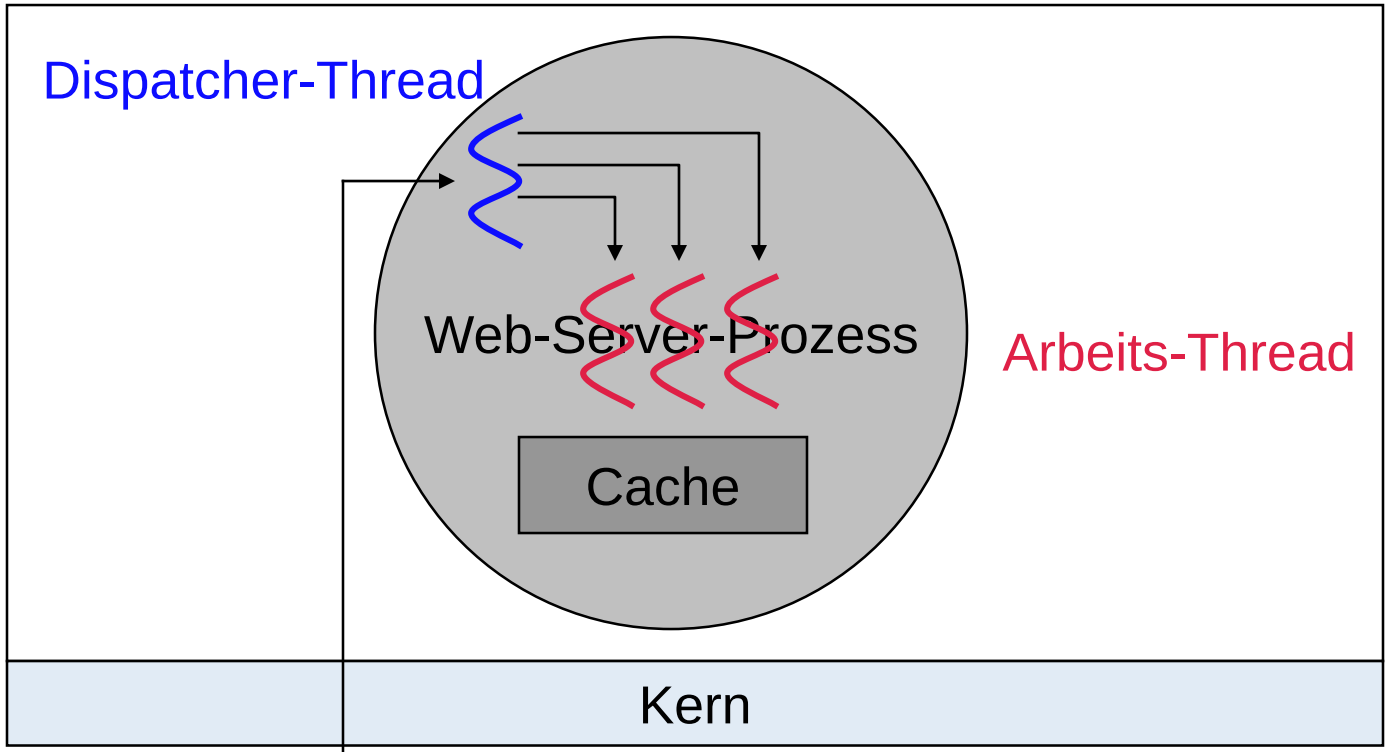
2.1.3 Nutzung und Vorteile von Threads

- In vielen Applikationen laufen mehrere Aktivitäten gleichzeitig ab, wobei einzelne davon blockieren. Quasiparallele Aktivitäten können mit Threads einfacher programmiert werden.
- Mit Threads sind keine Betriebsmittel verbunden. Threads können daher einfacher und schneller erzeugt werden (Faktor 100 im Vergleich zu schwergewichtigen Prozessen).
- Vorteile
  - Leistungsvorteile bei Mischung von CPU- und E/A-gebundenen Threads
  - Reaktionsfreudigkeit
  - Threads können auf verschiedene Prozessoren abgebildet werden.
  - Teilen von Ressourcen, z.B. Speicher
  - Geringerer Ressourcenbedarf im Vergleich zu schwergewichtigen Prozessen
  - Skalierbarkeit in Multiprozessorumgebungen

2.1.4 Threads: Beispiele

- Textverarbeitungsprogramm mit Threads für
  - Eingabe
  - Formatierung
  - automatischer Speicherung
- Browser mit mehreren Fenstern
- Datei- oder Web-Server mit Thread pro Request

2.1.5 Beispiel: Multi-Threaded Web-Server




```
while true {
    get_next_request (&buf);
    handoff_work (&buf);}
```

```
while (TRUE) {
    wait_for_work(&buf);
    look_for_page_in_cache(&buf,&page);
    if (page_not_in_cache(&page))
        read_page_from_disk(&buf,&page);}
```

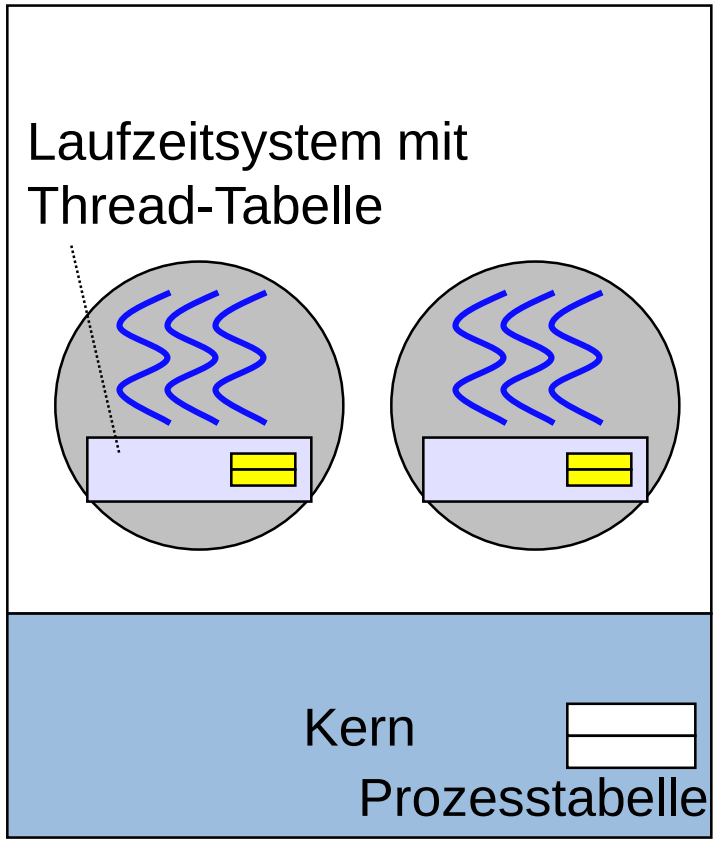
2.2 Thread-Typen

- User Threads
  - Beispiel: Java
- Kernel Threads
  - Beispiel: Mach
- Hybride Threads
  - Multiplexen von User Threads auf Kernel Threads
  - Beispiele: Linux, Windows, Solaris

2.2.1 User Threads

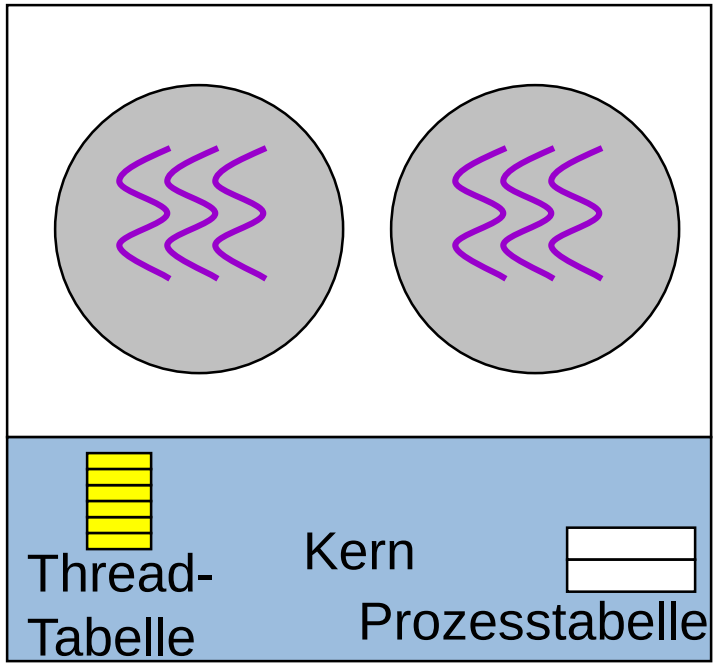
- Manipulation durch Thread-Library
- unabhängig vom / transparent für Betriebssystem
- kein Wechsel des Adressraums bei Thread-Wechsel
- Prozesse mit eigener Thread-Tabelle
- Anwendungsspezifisches Scheduling von Threads
- Threads sollten CPU freiwillig aufgeben.
- Blockieren des gesamten Prozess bei blockierendem Systemaufruf

  → ggf. Abfangen blockierender Systemaufrufe, z.B. durch **`select`** vor **`read`**
- Fairnessproblem bei unterschiedlicher Thread-Anzahl in den einzelnen Prozessen



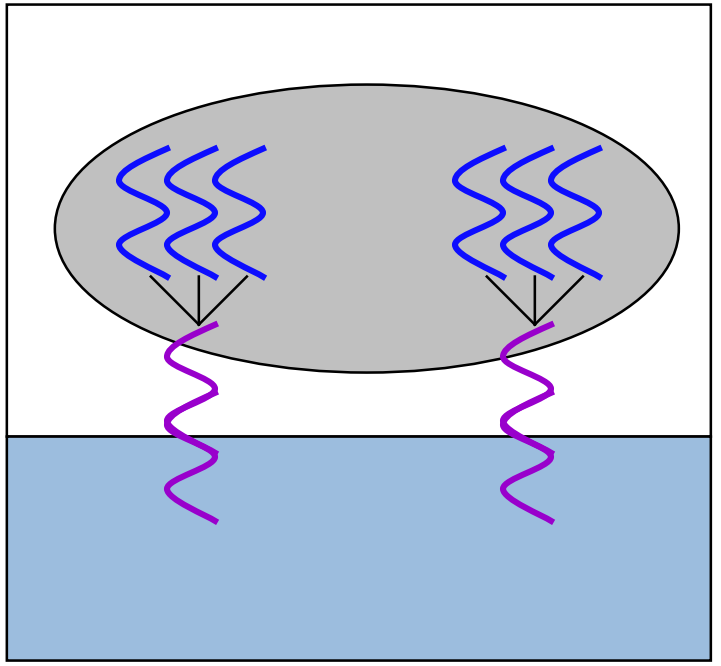
2.2.2 Kernel Threads

- Thread-Tabelle speichert Registerinhalte und andere Zustandsinformationen.
- Systemaufrufe zum Erzeugen oder Löschen von Threads
- Implementierung von möglicherweise blockierenden Aufrufen als Systemaufrufe.
- Bei einem blockierendem Thread ist ein Wechsel zu einem Thread des selben oder eines anderen Adressraums möglich.
- Thread Recycling
  - Wiederbenutzung von Kern-Datenstrukturen für neuen Thread
- Höherer Overhead / Kosten



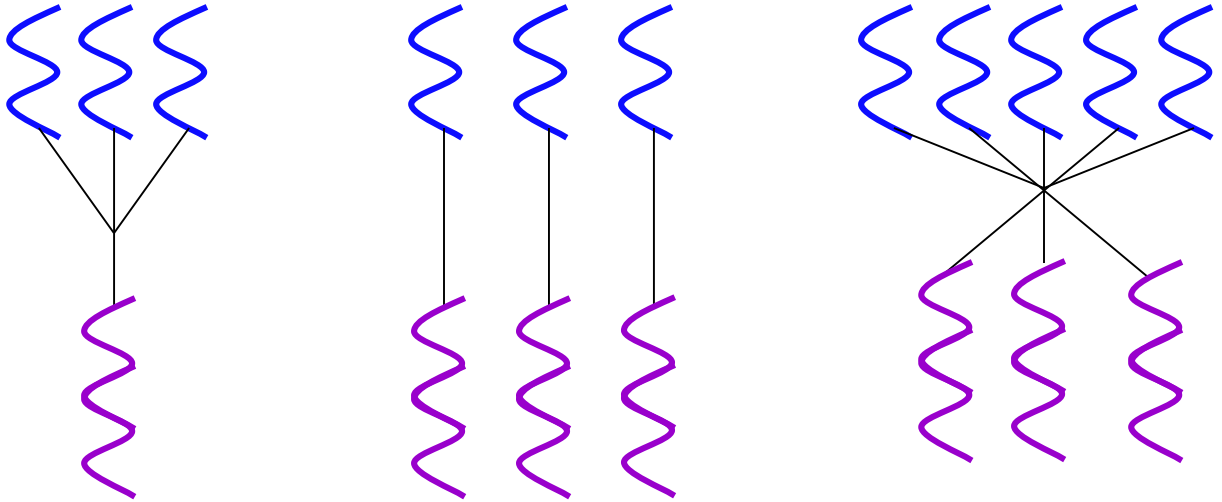
2.2.3.1 Hybride Threads

- Kombinieren der Vorteile von User Threads und Kernel Threads
- Multiplexen von User Threads auf Kernel Threads
- Kern führt Scheduling nur auf Kernel Threads durch.
- User Threads werden wie üblich verwaltet.



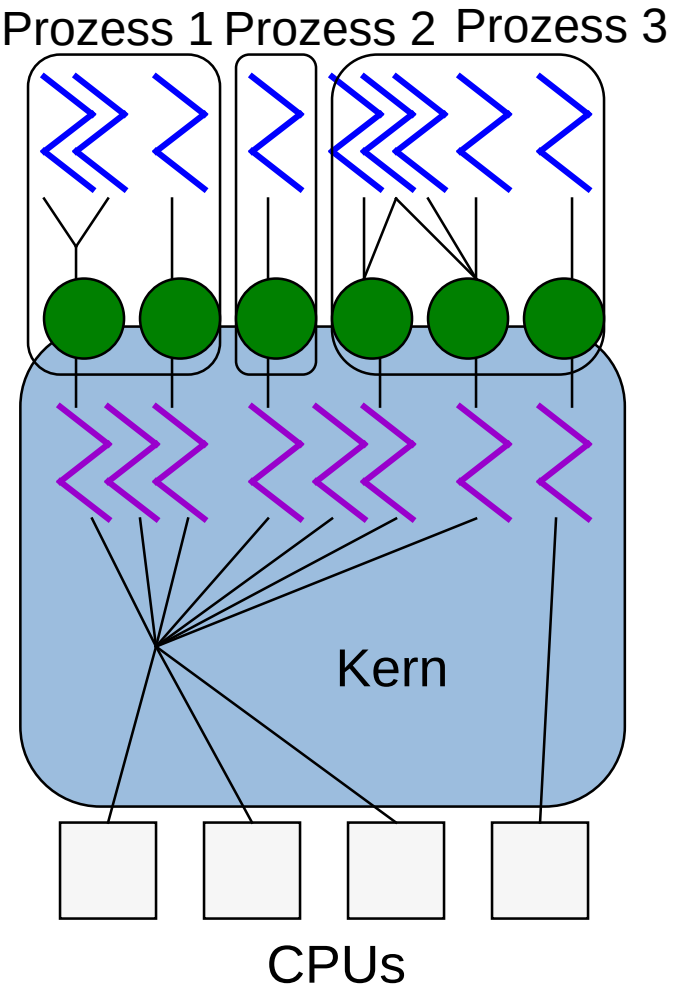
2.2.3.2 Multiplexen von User Threads auf Kernel Threads

- N:1
  - 1 Thread kann zu einem Zeitpunkt auf Kern zugreifen.
- 1:1
  - Nebenläufigkeit, aber Erzeugung vieler Kernel Threads
- N:M
  - Vermeiden der o.g. Nachteile



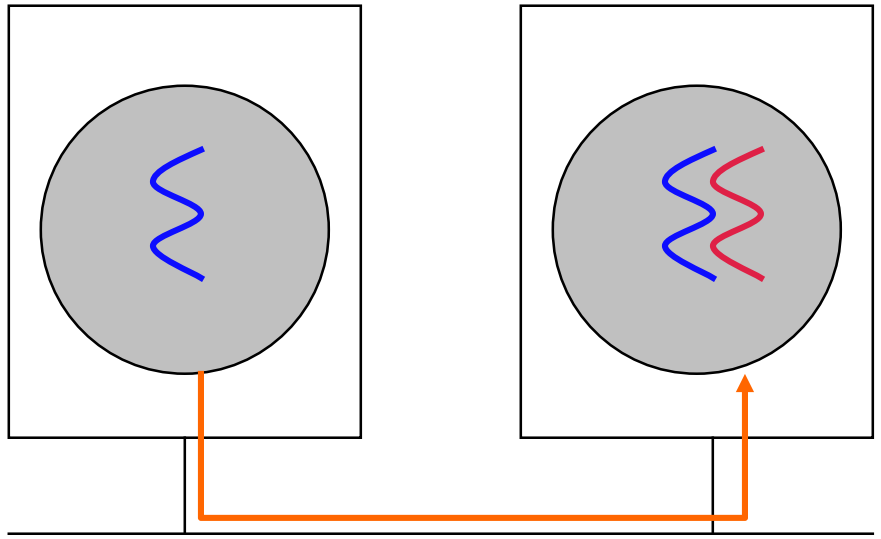
2.2.3.3 Beispiel: Leichtgewichtsprozesse

- Leichtgewichtsprozess (Lightweight Process, LWP)
  = virtueller Prozessor,
  Datenstruktur zwischen User Threads und Kernel Threads.
- Jeder LWP ist mit einem Kernel Thread verbunden.
- Kernel Threads als Scheduling-Objekte für Betriebssystem
- Falls Kernel Thread blockiert:
  LWP und User Threads werden auch blockiert.
- Mehrere LWPs sind für einen E/A-intensiven Prozess nützlich.
- Beispiel: Solaris 2



2.3.1 Pop-Up Threads

- Traditioneller Ansatz für ankommende Nachrichten:
  - Prozess oder Thread ist an **`receive`** Systemaufruf blockiert und wartet auf ankommende Nachricht.
- Pop-Up Threads
  - Ankommende Nachricht veranlasst das System einen neuen Thread (im Kern), welcher die Nachricht verarbeitet, zu erzeugen.
  - schnelle Thread-Erzeugung „frischer“ Threads (keine Wiederherstellung, z.B. vom Stack)



2.3.2 Thread Pools

- Thread-Erzeugung verursacht gewissen Aufwand.
- Lösung: Thread Pool
  - Erzeugen von Threads bei Task-Start und Ablegen in einem Pool
  - Bei einkommenden Anfragen werden verfügbare Threads aus Pool entnommen um Anfrage zu bearbeiten und am Ende wieder in Pool zurückgelegt.
  - Ist kein Thread verfügbar, muss gewartet werden, bis ein Thread im Pool verfügbar wird.

2.3.3 Thread-Steuerung

- Bibliotheksprozeduren zur Steuerung von Threads
- POSIX API für User/Kernel-Threads
  - **`pthread_create`**
    - Erzeugen eines neuen Threads im Adressraum des Aufrufers
  - **`pthread_exit`**
    - selbständiges Beenden eines Threads
  - **`pthread_join`**
    - Thread wartet auf das Ende eines anderen Threads.
- Linux
  - Systemaufruf **`clone`**
    - erzeugt neuen Prozess, der den gleichen Adressraum mit dem aufrufenden Prozess teilt, vgl. **`fork`**
- oft auch Aufrufe zum freiwilligen Aufgeben der CPU:
  wichtig, weil User Threads für Betriebssystem unsichtbar

2.3.4 Multi-Threaded Code

Probleme beim Erzeugen von Multi-Threaded Code aus Single-Threaded Code

- Gefahr des Überschreibens globaler Variablen
  → private globale Variable für jeden Thread
- Nicht reentrante Bibliotheksfunktionen
  → Markieren der Bibliothek als belegt
- Signale
  - Schwierigkeit der Zuordnung von Signalen durch Kern zu User Threads bzw. Zuordnung von nicht-Thread-spezifischen Signalen, z.B. Tastatur
- Stack-Management
  - Automatisches Wachsen von User-Thread-Stacks (Unterstützung durch Kern)