Inhalt

- Nebenläufigkeit
- POSIX Multithreading mit C
- Thread-Synchronisation mit joins und barriers

Nebenläufigkeit

- Betriebssysteme/Programme können mehrere Aufgaben gleichzeitig oder „quasi gleichzeitig“ ausführen („Parallele Programmierung“, „concurrency“).
- Nebenläufigkeit kann die Programmierung vereinfachen und die Performance steigern.
- Um separate Kontrollflüsse zu realisieren, werden folgende Konstrukte verwendet:
  - **Prozesse**
  - **Threads** (thread: Faden)
  - **Fibers** (*lightweight threads*, nur Windows)
- Diesen „Verwaltungseinheiten“ kann z.B.:
  - eine CPU zugeteilt werden oder
  - eine Ablaufstrategie zugewiesen werden

Begriffe

Programm

- eine Menge von Anweisungen
- Zustand z.B.: fehlerhaft, qualifiziert, ausverkauft, auf Datenträger, … aber nicht laufend!
  → „toter“ Code (Quellcode, Objectcode, Binärcode, …), der im Moment nicht ausgeführt wir

Prozess oder Task

- ein in Ausführung befindliches Programm
  → Instanz eines Programms
- Zustand: bereit, blockiert, laufend, …
- Prozesse können nebenläufig/parallel ausgeführt werden.
- Taskmanager (Windows) listet alle aktuellen Tasks
- System Monitor (Systemüberwach., Ubuntu) zeigt Prozessinfos
- ps-Kommando (process status, UNIX) zeigt diverse Merkmale der Prozesse und threads (z.B. `ps -eLf`).

Aus einem Programm können mehrere Instanzen gebildet werden.

Thread

- Ein Prozess kann in mehrere Kontrollflüsse (threads) unterteilt werden.
- Auch threads können parallel auf mehreren CPUs/Kernen ausgeführt werden.
- Varianten:
  native threads:
  - werden vom BS verwaltet, programmierbar über system-calls
  - Beispiele: Windows threads, Linux threads (deprecated), native POSIX threads library (NPTL)

  threads eines Laufzeitsystems:
  - innerhalb eines Prozesses, für das BS nicht sichtbar
  - programmierbar über entspr. Hochsprachenkonstrukte
  - Beispiele: Modula, die klassische JVM, Ada threading, …

  Laufzeitsystemthreads als native threads umgesetzt:
  - Beispiele: .NET threading, aktuelle JVMs, …
    ??? C++11 g++

Unterschiede/Merkmale von Prozessen und Threads

Prozess

- hat eigenen Adressraum
- Prozesse sind streng gegeneinander geschützt
- nur per system-calls programmierbar
- Kommunikation aufwendig über BS (file I/O, shared mem.)
- start, stop und switch aufwendig („context-switch“)

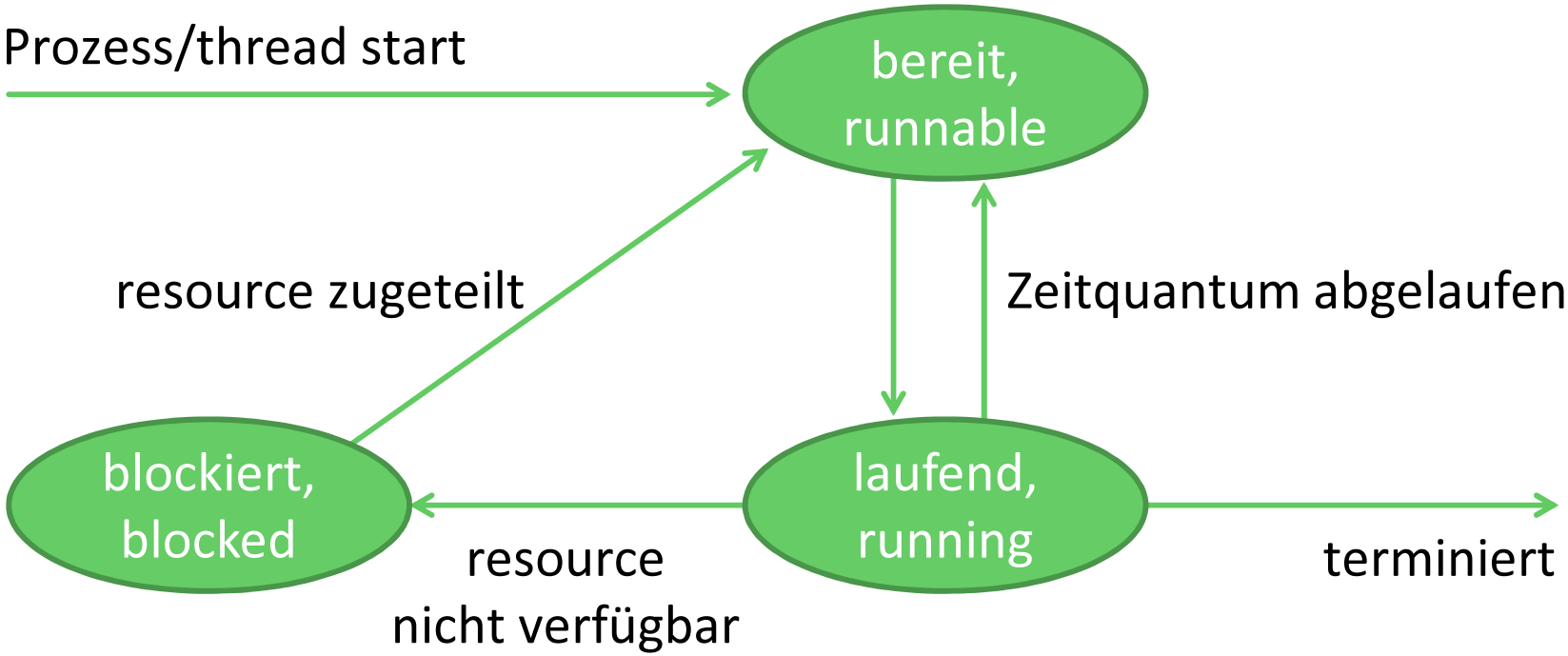
Thread

- ein Adressraum für alle threads eines Prozesses
- eigener stack (auto Var.)
- Kommunikation einfach und schnell über gemeinsame Variablen (globale Daten)
- start, stop, switch sehr schnell
- z.T. programmierbar mit Hochsprachenkonstrukten

→ „lightweight process“

Zustandsdiagramm

- Prozesse und threads werden anhand eines Zustandsdiagramms vom BS verwaltet.
- Diese Diagramme sind stark systemabhängig. Generell sehen sie etwa so aus:



- Die Zustände bereit/blockiert werden als Kollektionen von Prozessen/threads realisiert (z.B. Warteschlangen).

Dispatching

Der „dispatcher“ führt die Prozesswechsel aus:

- Registerinhalte in process control block (PCB) retten
- aus PCB des neues Prozesses Register laden …

Scheduling

schedule: Fahrplan

Der „scheduler“ implementiert eine Strategie für die Prozess- oder Threadwechsel.

– diverse Strategien möglich:

a) Round-Robin („im Kreis herum“), Zeitquantum pro Prozess
b) First-In-First-Out
c) RR oder FIFO mit festen oder dynamischen Prioritäten
d) Varianten c) mit oder ohne Preemption (hoch priorisierte Prozesse verdrängen niedrig priorisierte Prozesse)
e) …

Ubuntu NPTL (Native POSIX Threads Library) threads

- sog. 1:1 Implementation, d.h. threads und Prozesse bilden für das BS jeweils eine *scheduling entity*.
  ➔ bzgl. scheduling kein Unterschied zu Prozessen
- scheduling-Strategien (immer preemptive):
  - **SCHED_OTHER (default time sharing)**
  - **SCHED_BATCH**
  - **SCHED_IDLE**

  queue mit lowest prio. plus dyn. prio. innerhalb der queue (s. nächste Seite)

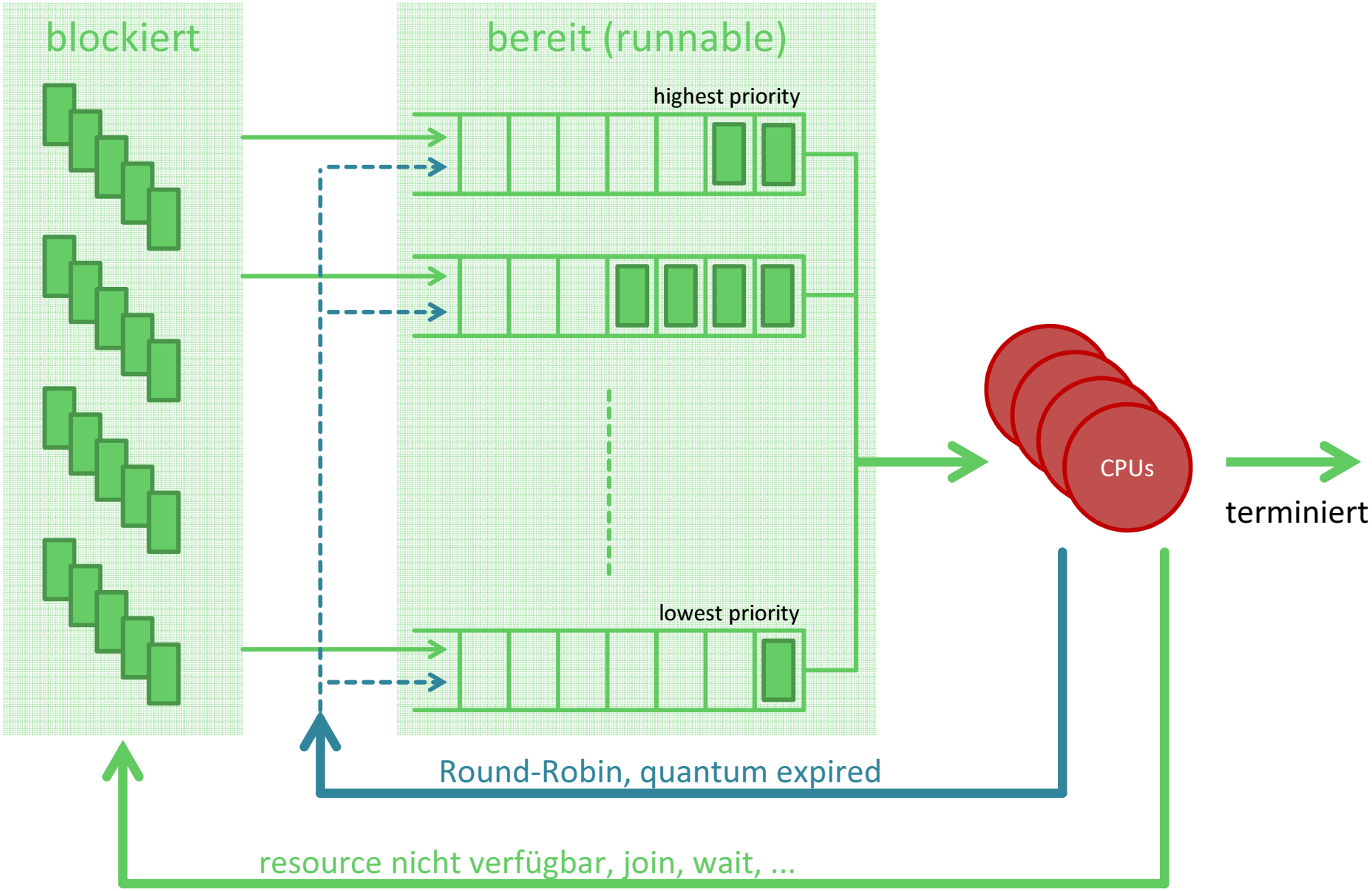
  - **SCHED_RR (Round-Robin)**
  - **SCHED_FIFO**

  feste Prio., alle queues mit prio=**0** bis prio=**99**,

  ➔ ***realtime scheduling***
- einstellbarer *contention scope*:
  - threads konkurrieren mit allen threads (systemweit)
  - threads konkurrieren mit threads des eigenen Prozesses
- einstellbare *Prozessor-Affinität*:
  - threads können CPUs bzw. Kernen zugewiesen werden

POSIX threads auch für Windows: *winpthreads* library

POSIX RR und FIFO scheduling Prinzip

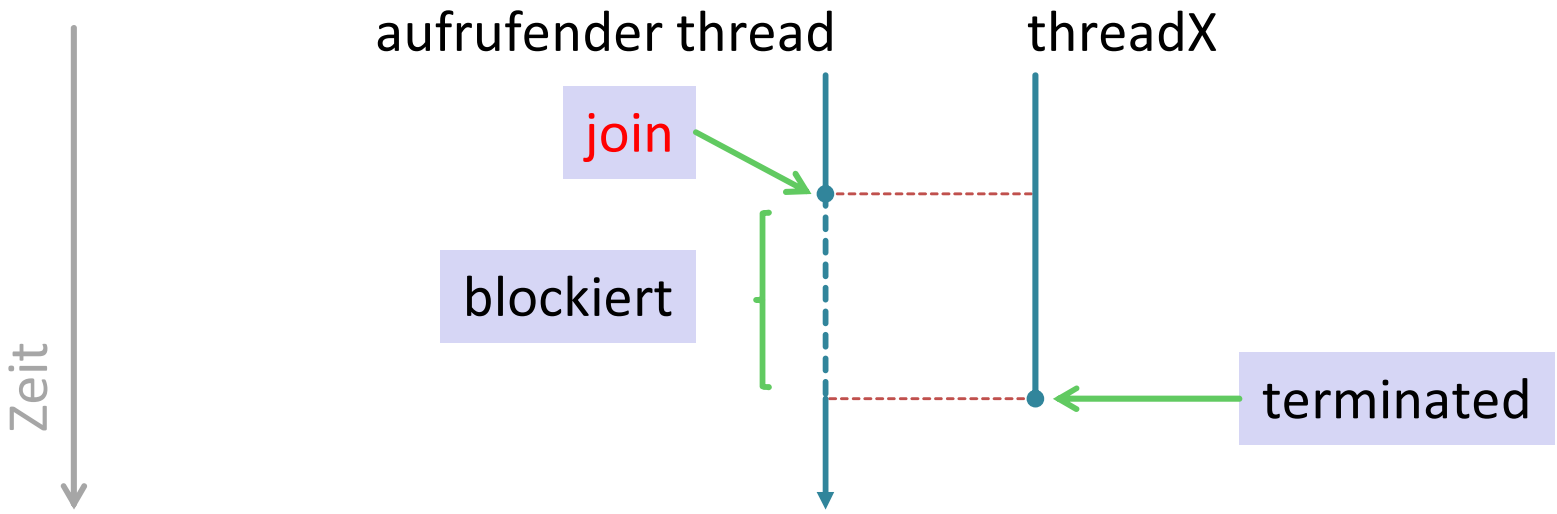


– Synchronisationsmechanismen für **pthreads**:

- **join**, auf Terminierung eines anderen threads warten
- **barrier**, mehrere threads warten an einem Synch.-Punkt
- **mutex**, exklusiver Zugriff für kritische Bereiche
- **condition variables**, bedingte Benachrichtigung, z.B.: *aufwecken, wenn...*
- **Semaphore**, „*mutex mit Zähler*“, Zugriff auf kritische Bereiche für eine definierte Anzahl von threads. Varianten:
  - unnamed semaphor: nur innerhalb des Prozesses bekannt
  - named semaphor: systemweit bekannt
- **spinlock**, locking mit aktiver Warteschleife

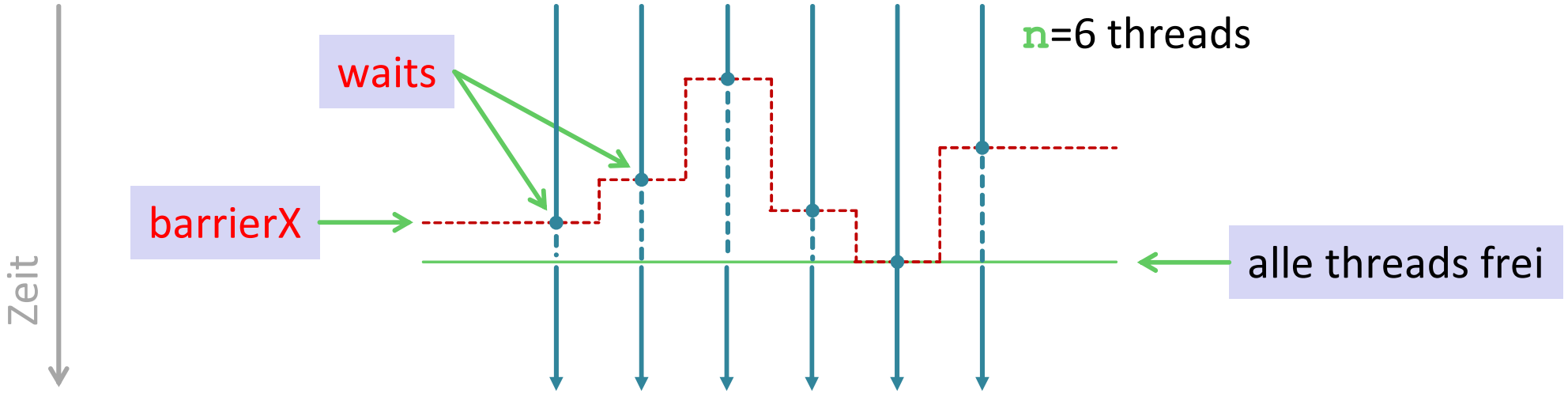
join

- Mit **`pthread_join(threadX, ...)`** wird auf die Terminierung eines anderen threads **`threadX`** gewartet.
- Der aufrufende thread suspensiert sich selbst
  ➔ Zustand *blockiert*
- Die Terminierung des threadX hebt die Blockade auf.



barrier

- Eine Variable vom Typ `pthread_barrier_t` wird mit `pthread_barrier_init(…, …, n)` initialisiert.
  ➔ maximal warten `n` threads an dieser Barriere.
- Mit `pthread_barrier_wait(barrierX)` warten rufende threads an der Barriere (blockiert).
- Der `n`-te wait-Aufruf gibt alle `n` threads wieder frei.



Beispiel zu POSIX barrier

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

pthread_barrier_t barrier42;

void* sachWatt()
{
     puts("thread an Barriere");
     pthread_barrier_wait(&barrier42);
     puts("Barriere aufgeloest");
}

main()
{
     pthread_t t1, t2;
     int barrierThreadCount = 3;   // 2 + main-thread
     pthread_barrier_init(&barrier42, NULL, barrierThreadCount);

     pthread_create(&t1, NULL, sachWatt, NULL);
     pthread_create(&t2, NULL, sachWatt, NULL);

     sleep(5);

     pthread_barrier_wait(&barrier42);
     puts("main-thread hat nach 5 Sekunden die Barriere geloest");
     sleep(1);  // threads Zeit zur Ausgabe lassen
}
```

Ausgabe:

```
thread an Barriere
thread an Barriere
main-thread hat nach 5 Sekunden die Barriere geloest
Barriere aufgeloest
Barriere aufgeloest
```

thread-safety

Thread sichere Funktionen können von mehreren threads gleichzeitig aufgerufen werden.
Sie müssen:

a) **wiedereintrittsfähig sein** (reentrant)
   Für jede Aufrufinstanz müssen separate Daten angelegt werden (ein eigener Stack).

b) **ihre kritischen Abschnitte schützen** (crit. sections)
   Die Operationen auf thread-globale Daten und andere gemeinsame resources müssen ohne Unterbrechung ausgeführt werden (atomisch).

! Thread-Sicherheit ist erforderlich für die thread-Funktionen selbst und die von ihnen aufgerufenen Funktionen!