Kapitel 6: Threads

6: Threads

Nebenläufigkeit

- Das übergeordnete Thema dieses Vortrags ist Nebenläufigkeit.
- Nebenläufigkeit bedeutet das zwei Handlungen *parallel* ausgeführt werden.
- Das Gegenteil von nebenläufig ist *sequenziell*.
- Zwei Handlungen sind *sequenziell* zueinander, wenn die eine erst nach der anderen ausgeführt werden kann.

Beispiel

- Kaffee kochen, Kaffee trinken ist sequenziell.
- Kaffee trinken und Kuchen essen ist nebenläufig.

Nebenläufigkeit? Warum?

Wozu benötigt man Nebenläufigkeit in der IT?
Nebenläufigkeit ermöglicht es mehreren Prozesses/Threads parallel auf eine geteilte Ressourcen zuzugreifen.
(CPU, Bildschirm, Speicher, …)

Beispiel: Ressource Wi-Fi

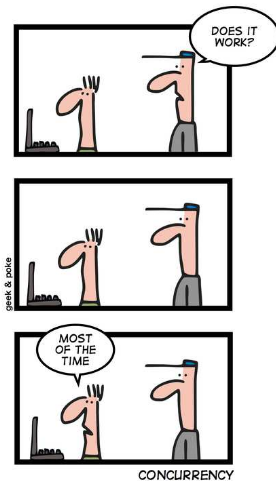
- Accesspoint an dem mehrere Benutzer angemeldet sind
- Gleichzeitig mehrere Dateien herunterladen
- Parallel mit verschiedenen Freunden chatten
- Tabbed Browsing
- …

Nebenläufigkeit ist schwierig

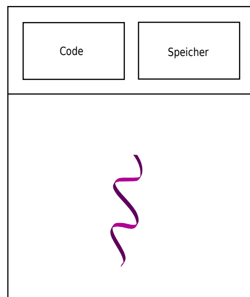
Warum ist Nebenläufigkeit so schwierig?

- Übliche Programmiersprachen haben eine sequenzielle Semantik.
- Das Denken in parallelen Handlungssträngen fällt Menschen schwer.
- Intuitiv erwartet man, dass nebenläufige Aktionen in einer bestimmten (scheinbar “richtigen”) Reihenfolge ausgeführt werden. Folge: “Race Conditions”.

Simply Explained: Nebenläufigkeit



Prozess und Thread

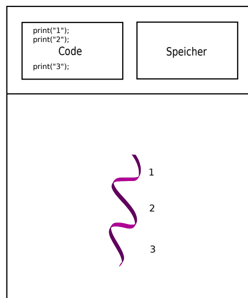


(Singlethreaded) Prozess

Thread (Faden)

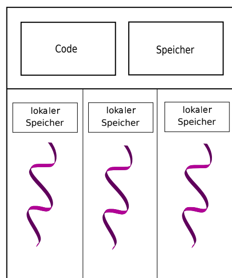
Sequenzieller Ausführungsstrang eines Prozesses.

Beispiel: 1 2 3




```
main():
    print("1");
    print("2");
    print("3");
```

Multithreaded Prozess



- Ein Prozess kann mehrere Threads haben (Multithreading)
- Threads ermöglichen Nebenläufigkeit innerhalb eines Prozesses
- Wichtig: Anweisungen der einzelnen Threads sind im Regelfall voneinander unabhängig

Exkursion: Scheduler

- Threads konkurrieren um Ressourcen (z.B. CPU-Zeit)
- Das Betriebssystem sorgt für eine (gerechte) Verteilung der Ressourcen.

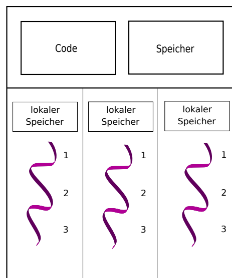
Beispiel: Thread-Scheduler

- Jeder Thread bekommt eine bestimmte Zeitspanne an CPU-Zeit zugeteilt.
- Jeder Thread wird am Ende der Zeitspannne unterbrochen.

Hinweise

- Die Zeitspannen sind so gewählt, dass der Benutzer glaubt, dass alle Threads parallel laufen.
- Mehrkernprozessoren ermöglichen parallele Ausführung.

Beispiel: Multithreaded Prozess




```
run():
    print("1");
    print("2");
    print("3");
```

```
main():
    t1 = new Thread();
    t2 = new Thread();
    t3 = new Thread();
    t1.run(); t2.run(); t3.run();
```

Was ist die Ausgabe des Programms? Die Ausgabe ist **undefiniert**!

Kurze Überlegung

- Wir haben 3 Threads und nur 1 Ausgabe.
- Die Threads laufen parallel.
- Das Schreiben auf die Ausgabe ist sequenziell.
  (Es gibt keine Überlagerung von Zeichen)
- $\Longrightarrow$ Parallel ankommende Schreibanweisungen werden sequenziell abgearbeitet.
- Analogie: Warteschlangen bei Kassen.

Mögliche Ausgaben

Geteilte Ressource: Ausgabe

Drei mögliche Ausgaben des Beispielprogramms

1. Ausgabe: `1 2 3 1 2 3 1 2 3`

2. Ausgabe: `1 2 1 2 1 2 3 3 3`

3. Ausgabe: `1 1 2 2 1 2 3 3 3`

6.1: Race Conditions

Race Condition (Wettlaufsituation)

- Mehrere Threads laufen um die Wette.
- Das Ergebnis einer Operation hängt vom *Gewinner* ab.

Anmerkungen

- Unbeabsichtigte Race Conditions sind oft Grund für schwer auffindbare Fehler.
- Denn oftmals verschwinden die Symptome im Debug-Modus.

Beispiel: Berechtigungen

```
void readFile(string filename):
   if  permission_denied(filename, READONLY)  then
      return  error("Permission denied");
   end if
   fd = open(filename, READONLY);
   { Lese den Inhalt der Datei }
```

Frage: Wo ist hier das Problem, d.h. was kann hier passieren?

Race Condition Beispiel: Bank-Transaktion

Protagonisten

- Geteilte Ressource: Kontostand
- Thread 1: Einzahlung von 200 EUR
- Thread 2: Abbuchung von 400 EUR

Thread 1	Thread 2	Kontostand
X = Kontostand;		1000
X = X + 200;	Y = Kontostand;	1000
Kontostand = X	Y = Y - 400;	1200
	Kontostand = Y	600

Lösung: Synchronisation

Synchronisation

Ein Verfahren welches den gemeinsamen Zugriff auf geteilte Ressourcen regelt.

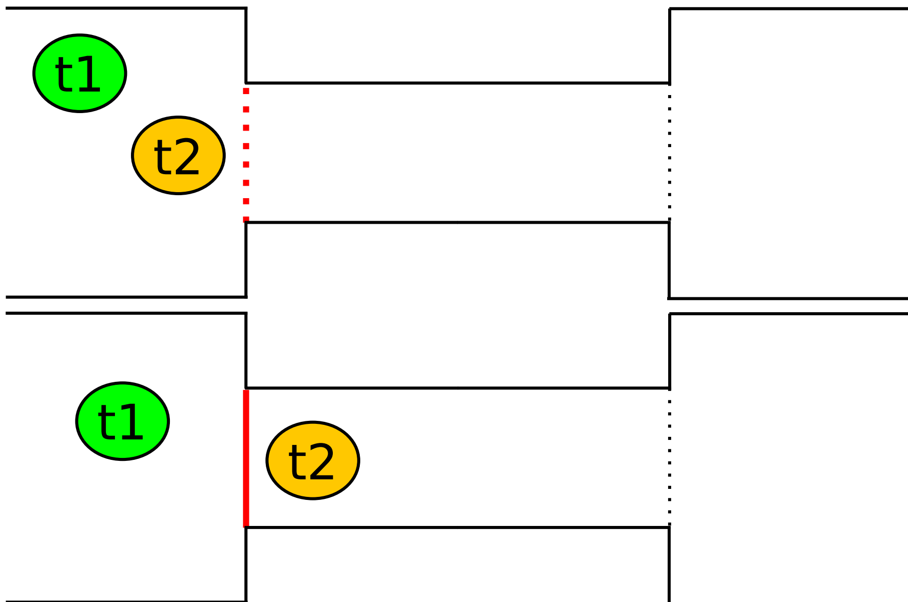
Realisierungsansatz von Dijkstra

1. Identifizierung von **kritischen Abschnitten**.
   (Üblicherweise Codeabschnitte bei denen eine gemeinsame Ressource *modifiziert* wird.)
2. P-Operation beim Betreten eines kritischen Abschnitts.
3. V-Operation beim Verlassen eines kritischen Abschnitts.

Mutex

- Mutex ist die Abkürzung von Mutual Exclusion (dt. “gegenseitiger/wechselseitiger Ausschluss”)
- Ein Mutex ist so etwas wie eine globale Variable
- Mittels eines Mutex lässt sie der Zugriff auf eine gemeinsame Ressource sperren
- Funktionsweise:
  1. Thread sperrt einen Codeabschnitt (P-Operation)
  2. Thread arbeitet den gesperrten Codeabschnitt ab
  3. Thread gibt den gesperrten Codeabschnitt wieder frei (V-Operation)
- Mutex = digitale Sicherheitschleuse für kritische Codeabschnitte

Mutex: Illustration



Beispiel: Bankkonto mit Mutex

Geteilte Ressource: Kontostand

Mutex: M

Thread 1: Einzahlung von 200 EUR

Thread 2: Abbuchung von 400 EUR

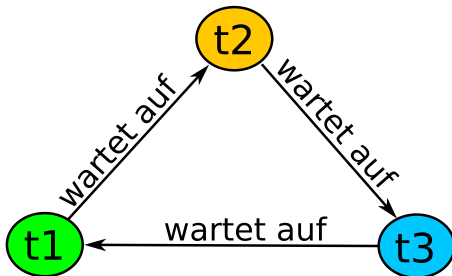
Thread 1 (t1)	Thread 2 (t2)	Kontostand
lock(M);		1000
X = Kontostand	lock(M) → block(t2);	1000
X = X + 200;	*(blocked)*	1000
Kontostand = X	*(blocked)*	1200
unlock(M) → unblock(t2)	*(wartend)*	1200
	Y = Kontostand	1200
	Y = Y - 400;	1200
	Kontostand = Y	800
	unlock(M)	800

Probleme

Frage: Haben wir mit Mutexen unser Synchronistationsproblem gelöst?

Antwort: Ja, aber wir haben uns ein neues Problem eingefangen, nämlich die sogenannten Deadlocks (Verklemmungen).

6.2: Deadlocks



Deadlock (Verklemmung)

Zyklischer Ressourcenkonflikt zwischen mehreren Threads, bei der jeder beteiligte Thread auf die Freigabe (Unlock) von gemeinsamen Ressourcen wartet, die ein anderer beteiligter Thread bereits exklusiv gesperrt hat.

Deadlock-Beispiel aus der realen Welt

Quelle: Vorlesung Betriebssysteme: Dr. Christian Braun (Hochschule Mannheim WS12/13)

Deadlock-Beispiel: Überweisungen

- Thread 1 möchte 100 EUR von Konto A auf B überweisen.
- Thread 2 möchte 200 EUR von Konto B auf A überweisen.

Vorgehensweise bei Umbuchung

1. Sperrung des ersten Kontos
2. Sperrung des zweiten Kontos
3. Überweisung des Betrags durchführen
4. Konten entsperren

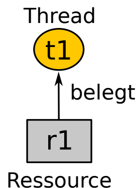
Thread 1 (t1)	**Thread 2 (t2)**
Mutex.lock(A);	Mutex.lock(B);
Mutex.lock(B) $\rightarrow$ t1.wait();	Mutex.lock(A) $\rightarrow$ t2.wait();

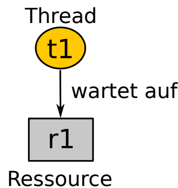
Wir haben einen Deadlock, denn

- Thread 1 wartet auf Thread 2 und
- Thread 2 wartet auf Thread 1.

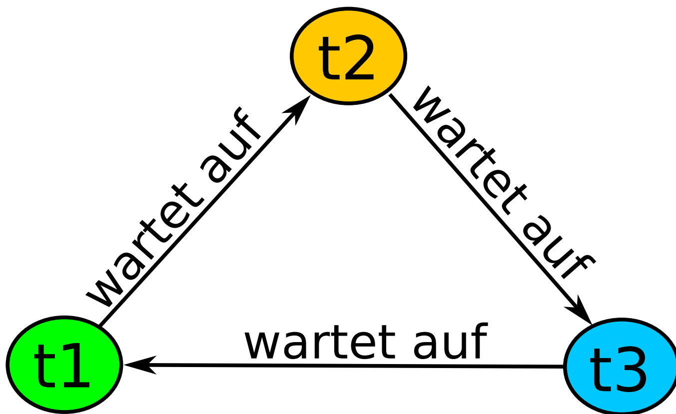
Ressourcen-Belegungsgraph

- Mit gerichteten Graphen können die Beziehungen von Threads und Ressourcen dargestellt werden
- Modellierung von Deadlocks
- Threads (t) werden als kreisförmige Knoten dargestellt
- Ressourcen (r) werden als rechteckige Knoten dargestellt
- Kante $r \rightarrow t$: Der Thread t belegt die Ressource r
- Kante $t \rightarrow r$: Der Thread t wartet auf die Ressource r





Ressourcen-Belegungsgraph: Beispiel



Ein Zyklus entspricht einem Deadlock

Ressourcen-Belegungsgraph: Deadlocks

Thread 1	**Thread 2**	**Thread 3**
Anforderung $r1$	Anforderung $r2$	Anforderung $r3$
Anforderung $r2$	Anforderung $r3$	Anforderung $r1$
Freigabe $r1$	Freigabe $r2$	Freigabe $r3$
Freigabe $r2$	Freigabe $r3$	Freigabe $r1$

Frage

(a) Was kann passieren, falls die Threads der Reihe nach abgearbeitet werden?

(b) Was kann passieren, falls die Threads parallel verarbeitet werden?

Miniexkurs: Gerichteter Graph

- Ein gerichteter Graph $G(V, E)$ (kurzschreibweise G) besteht aus zwei Mengen.
  - V: Menge aller Knoten (engl. *vertex/node*)
  - E: Menge gerichteter Kanten E
- Eine gerichtete Kante $e = \{v_1, v_2\}$ ist eine geordnete Menge von zwei Knoten $v_1, v_2 \in V$.
- Wir bezeichnen v_1 als Startknoten und v_2 als Endknoten.
- Wir schreiben $v \in G$ für $v \in V$ und $e \in G$ für $e \in E$.
- Eine Liste von Knoten $P = \{v_1, \ldots, v_n\}$ ist ein Pfad in G, falls für alle Paare $z_i = \{v_i, v_{i+1}\}$ mit $i = 1, \ldots .n - 1$ gilt: $z_i \in G$
- Ein Pfad P enthält einen Zyklus falls zwei Knoten $v_i = v_j$ mit $i \neq j$ und $v_i, v_j \in P$ existieren. ($\rightarrow$ Tafel)

Ressourcen-Belegungsgraph: Deadlocks erkennen

Mittels Tiefensuche (engl. *Depth-first search*, DFS) lassen sich Zyklen in einem Graphen G erkennen.

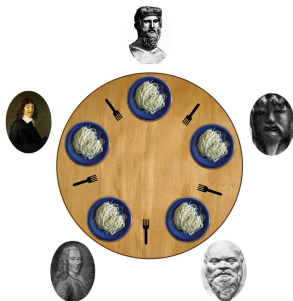
Zyklus(G):

```
1: for all v ∈ G do
2:     v.marked = white;
3: end for
4: for all v ∈ G do
5:     if DFS(G, v) then
6:         return true;
7:     end if
8: end for
9: return false;
```

DFS(G, v):

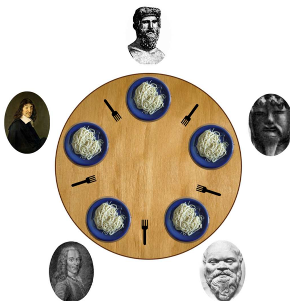
```
 1: switch(v.marked)
 2:     case black: return false;
 3:     case gray: return true;
 4:     case white: v.marked = gray;
 5: end switch
 6: for all w ∈ successorsOf(v) do
 7:     if DFS(G,w) then
 8:         return true
 9:     end if
10: end for
11: v.marked = black;
12: return false;
```

Beispiel: Philosophenproblem (Aufbau)



- Fünf Philosophen sitzen an einem runden Tisch.
- Jeder hat einen Teller mit Spaghetti vor sich.
- Zwischen zwei Philosophen befindet sich jeweils eine Gabel.
- Zum Essen von Spaghetti benötigt jeder Philosoph zwei Gabeln.
- Philosophen können entweder Essen oder Denken.

Beispiel: Philosophenproblem (Vorgehen)



- Wenn ein Philosoph hungrig wird, dann
  - nimmt er zuerst die Gabel links von seinem Teller,
  - dann die auf der rechten Seite und beginnt zu essen.
- Solange nur einzelne Philosophen hungrig sind ist alles OK.
- **Frage:** Was passiert, wenn alle Philosophen gleichzeitig hungrig werden?

Auflösung

Preisfrage: Wie löst man einen Deadlock?

Hinweis: Wie löst man einen Mexican Standoff?

Antwort: Terminierung (mind.) eines Threads.

Beseitigung durch Rollback

Deadlock-Beseitigung durch Rollback

- Der Zustand eines Threads wird regelmäßig gesichert.
- Bei einem Deadlock wird der Thread auf den zuletzt gesicherten Zustand zurückgesetzt und für kurze Zeit suspendiert.

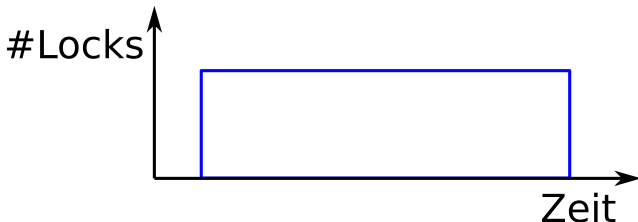
Anmerkung:

- Diese Methode funktioniert nicht immer.
  Negativbeispiel: Brennen von CDs/DVDs.
- Vorgang ist komplizierter als die Beendigung eines Threads.

Vermeidung von Deadlocks: Nummerierung

- Durchnummerierung von Ressourcen $r_1, r_2, \ldots$
- Ein Thread darf nur Ressourcen in aufsteigender Reihenfolge belegen
- Mit dieser Regel bleibt der Ressourcen-Belegungsgraph immer zyklenfrei
- Angenommen, $t1$ belegt r_1 und $t2$ belegt r_2
- $t1$ kann jetzt r_2 anfordern, aber $t2$ darf nicht r_1 anfordern, da er ja schon r_2 belegt. Daher kann es jetzt keinen Deadlock mehr geben.
- Die Überlegung ist auf n Threads mit $n > 2$ übertragbar

Vermeidung von Deadlocks: Preclaiming



- Thread versucht alle Ressourcen die er für eine *Transaktion* benötigt zu belegen. Gelingt es ihm nicht so gibt er alle bisher belegten Ressourcen wieder frei und wartet kurz.
- Nach Ausführung der *Transaktion* gibt der Thread alle Ressourcen die er belegt wieder frei.
- Problem: Ressouren werden oft länger belegt als nötig.

6.3: Epilog

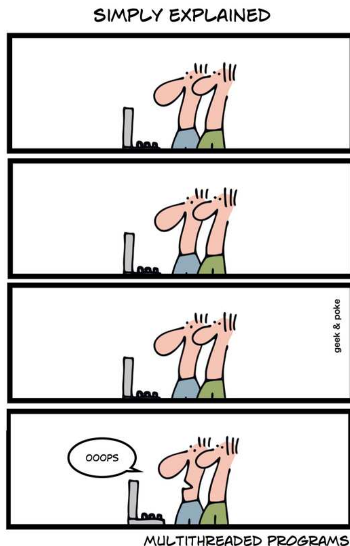
Thread-Support

- Moderne Programmiersprachen unterstützen mehr oder weniger nativ Threads und Mutexe (Threads in Java, C# oder C++11, Tasks in Ada, ...)
- Ansonsten kann man auf geeignete Bibliotheken zugreifen (z.B. `pthreads`)
- Das nächste Kapitel widmet sich der `pthread`-Bibliothek

Programmierer und Nebenläufigkeit

> “Wir schätzen, dass weniger als die Hälfte der Programmierer bei Herstellern und Anwendern wirklich in der Lage ist, nebenläufige Software zu schreiben.”
>
> — David Cearly, Computer-Zeitung vom 19. Mai 2008.

Simply Explained: Multithreaded Program



Zusammenfassung

Nach diesem Kapitel sollten Sie …

- …wissen was ein Thread ist.
- …wissen was unter Synchronisation verstanden wird.
- …den Umgang mit einem Mutex beherrschen.
- …wissen wie Deadlocks erkannt und vermieden werden können.
- …wissen wie Deadlocks aufgelöst werden können.