Work in progress

Realisierung von lock()/unlock()

Ute Bormann, TI2

2023-10-13

Inhalt

1. Erste Implementierungsversuche
2. Locking-Algorithmen

Teil 1:
Erste Implementierungsversuche

16 Realisierung von lock()/unlock(), Ute Bormann

Kritische Abschnitte

- Erkennen: Was muss überhaupt geschützt werden?
  - Verändern von gemeinsamen „Daten“
  - Zugriff auf exklusiv zu nutzende Geräte ...

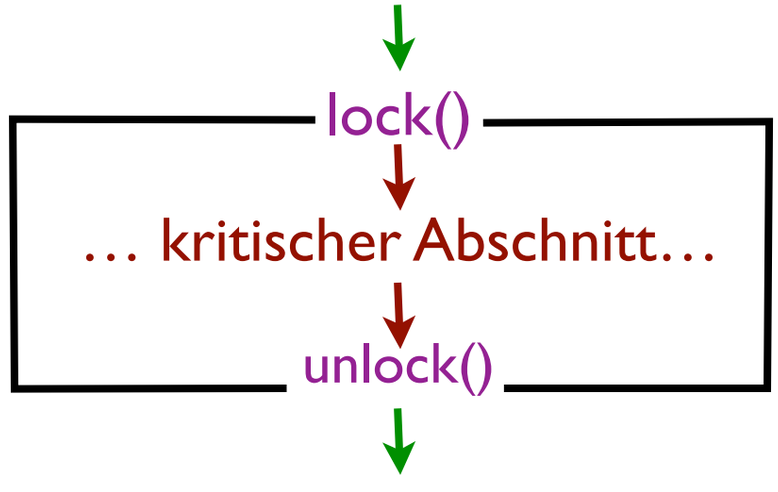
Beispielprogramm:

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

void *drucken(char*);

int said = 0;


main() {
  pthread_t english, deutsch;

  pthread_create(&english, NULL, &drucken, "Hello World");
  pthread_create(&deutsch, NULL, &drucken, "Hallo Welt");
  ...
  pthread_join(english, NULL);
  pthread_join(deutsch, NULL);
}
```

```
void *drucken(char *param) {
  for (;;) {

    if (said < 10) {
    said++;

      cout << param << endl;
    } else {

      break;
    }
  }
}
```

Kritische Abschnitte

- Erkennen: Was muss überhaupt geschützt werden?
  - Verändern von gemeinsamen „Daten“
  - Zugriff auf exklusiv zu nutzende Geräte ...
- Gewährleisten des gegenseitigen Ausschlusses:
  - Sichern bei Betreten des kritischen Abschnitts
    ⇒ Eintrittsprotokoll („Tür hinter sich zuschließen“)
  - Freigeben bei Verlassen
    ⇒ Austrittsprotokoll („Tür wieder öffnen“)



Beispielprogramm:

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

void *drucken(char*);

int said = 0;
pthread_mutex_t *said_mutex;

main() {
  pthread_t english, deutsch;

  pthread_create(&english, NULL, &drucken, "Hello World");
  pthread_create(&deutsch, NULL, &drucken, "Hallo Welt");
  ...
  pthread_join(english, NULL);
  pthread_join(deutsch, NULL);
}
```

```
void *drucken(char *param) {
  for (;;) {
    pthread_mutex_lock(said_mutex);
    if (said < 10) {
      said++;
      pthread_mutex_unlock(said_mutex);
      cout << param << endl;
    } else {
      pthread_mutex_unlock(said_mutex);
      break;
    }
  }
}
```

a) Genereller Ausschluss von Nebenläufigkeit

1) Unterbrechungen ausschalten

```
disable_interrupts();     (≙ lock())
... kritischer Abschnitt ...
enable_interrupts();      (≙ unlock())
```

⇒ keine Unterbrechungen ⇒ keine „Prozesswechsel“

⇒ nur ein Prozess im kritischen Abschnitt

a) Genereller Ausschluss von Nebenläufigkeit

1) Unterbrechungen ausschalten

```
disable_interrupts();     (≙ lock())
... kritischer Abschnitt ...
enable_interrupts();      (≙ unlock())
```

⇒ keine Unterbrechungen ⇒ keine „Prozesswechsel“

⇒ nur ein Prozess im kritischen Abschnitt

Genauer: Prozess oder Thread

a) Genereller Ausschluss von Nebenläufigkeit

1) Unterbrechungen ausschalten

```
disable_interrupts();     (≙ lock())
... kritischer Abschnitt ...
enable_interrupts();      (≙ unlock())
```

⇒ keine Unterbrechungen ⇒ keine „Prozesswechsel“

⇒ nur ein Prozess im kritischen Abschnitt

- (In Unix: spl() Set Processor Level)
- Aber:
  - Interrupts sind zeitkritisch
  - Funktioniert nur bei Einprozessorsystem
  - Im User-Mode nicht zulässig

2) Bei Mehrprozessorsystem auch andere Prozessoren blockieren

```
disable_interrupts();
broadcast_stop();
... kritischer Abschnitt ...
broadcast_continue();
enable_interrupts();
```

(zusätzlich zu Unterbrechungsausschluss)

⇒ keine nebenläufigen (parallelen/quasi-parallelen) Aktivitäten

- Aber: nicht unbedingt realisierbar

2) Bei Mehrprozessorsystem auch andere Prozessoren blockieren

```
disable_interrupts();
broadcast_stop();
... kritischer Abschnitt ...
broadcast_continue();
enable_interrupts();
```

(zusätzlich zu Unterbrechungsausschluss)

⇒ keine nebenläufigen (parallelen/quasi-parallelen) Aktivitäten

- Aber: nicht unbedingt realisierbar

- Außerdem in beiden Fällen: i.d.R. zuviel gesperrt
  (viele Aktivitäten wollen kritischen Abschnitt gar nicht betreten)

⇒
- Unnötige Verzögerung von zeitkritischen Vorgängen
- Verschwendung von CPU-Kapazität (im zweiten Fall)

b) Selektiver Ausschluss von Nebenläufigkeit

- Prozesswechsel und parallele Prozesse weiterhin zulassen
- Aber bei Betreten eines kritischen Abschnitts andere Prozesse genau daran hindern

  ⇒ kritische Abschnitte so spezifisch wie möglich wählen

  ⇒ maximale Nebenläufigkeit

b) Selektiver Ausschluss von Nebenläufigkeit

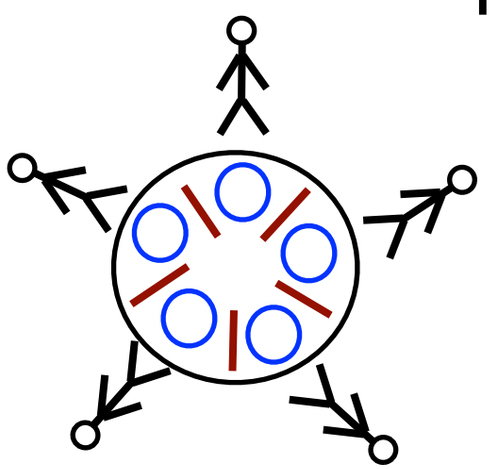
- Prozesswechsel und parallele Prozesse weiterhin zulassen
- Aber bei Betreten eines kritischen Abschnitts andere Prozesse genau daran hindern

  ⇒ kritische Abschnitte so spezifisch wie möglich wählen

  ⇒ maximale Nebenläufigkeit

Achtung:

- Geschachtelte kritische Abschnitte können zu Verklemmungen führen (s. Speisende Philosophen)



- Also ggf. Verklemmungsbehandlung vorsehen

- Folge spezifischer kritischer Abschnitte:
  Jeder benötigt eigenen „Schließmechanismus“

  ⇒ Spezifische Schlossvariable (key) dafür vorsehen

- Im Prinzip:

  `key==false`: kritischer Abschnitt ist frei

  `key==true`: kritischer Abschnitt ist besetzt

- Also:

  `lock()` ⇒ Sofern noch frei: `key = true;`

  sonst warten bis frei, dann `key = true;`

  `unlock()` ⇒ `key = false;`

Lock-Algorithmen

1. Versuch:

```
bool key = false;  //Abschnitt frei

lock() {                        unlock() {
  while (key)                     key = false;
      ;                         }
  key = true;
}
```

Lock-Algorithmen

1. Versuch:

```
bool key = false;  //Abschnitt frei

lock() {                        unlock() {
  while (key)                     key = false;
      ;                         }
  key = true;
}
```

Funktioniert das?

key	*Prozess/Thread 1*	*Prozess/Thread 2*
false		
	if key? Nein	
		if key? Nein
true	key = true	
		key = true
	kritischer Abschnitt	
		kritischer Abschnitt

Lock-Algorithmen

1. Versuch:

```
bool key = false;  //Abschnitt frei

lock() {                        unlock() {
  while (key)                     key = false;
      ;                         }
  key = true;
}
```

Funktioniert das?

key	*Prozess/Thread 1*	*Prozess/Thread 2*
false		
	if key? Nein	
		if key? Nein
true	key = true	
		key = true
	kritischer Abschnitt	
		kritischer Abschnitt

⇒ Problem: Zugriff auf key ist selbst kritischer Abschnitt

2. Versuch

- Erst eigene Absicht anmelden, dann überprüfen, ob anderer auch will

⇒ Jeder hat eigene „Schlossvariable“

```
bool key1, key2 = false;

lock1() {                    lock2() {
  key1 = true;                 key2 = true;
  while (key2)                 while (key1)
    ;                            ;
}                            }

unlock1() {                  unlock2() {
  key1 = false;                key2 = false;
}                            }
```

2. Versuch

- Erst eigene Absicht anmelden, dann überprüfen, ob anderer auch will

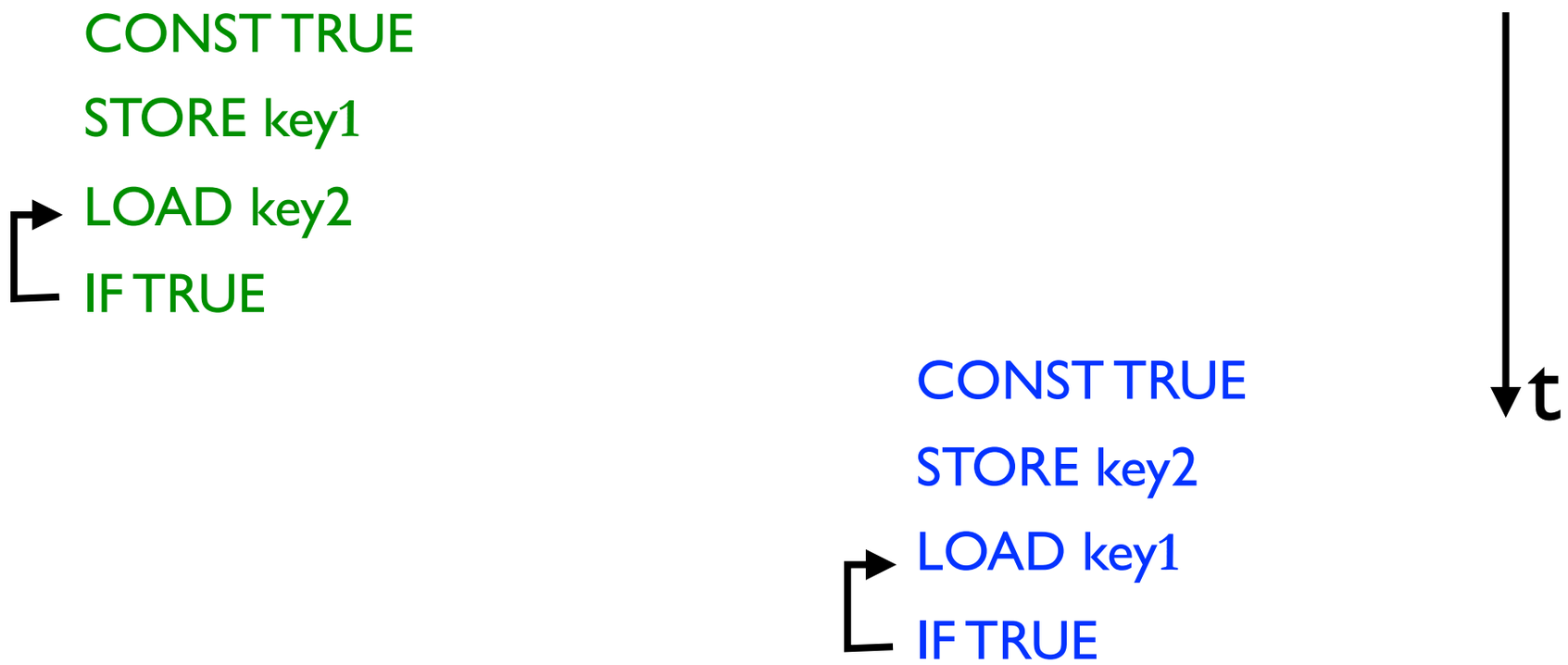
  ⇒ Jeder hat eigene „Schlossvariable“

```
bool key1, key2 = false;

lock1() {                    lock2() {
  key1 = true;                 key2 = true;
  while (key2)                 while (key1)
    ;                            ;
}                            }

unlock1() {                  unlock2() {
  key1 = false;                key2 = false;
}                            }
```

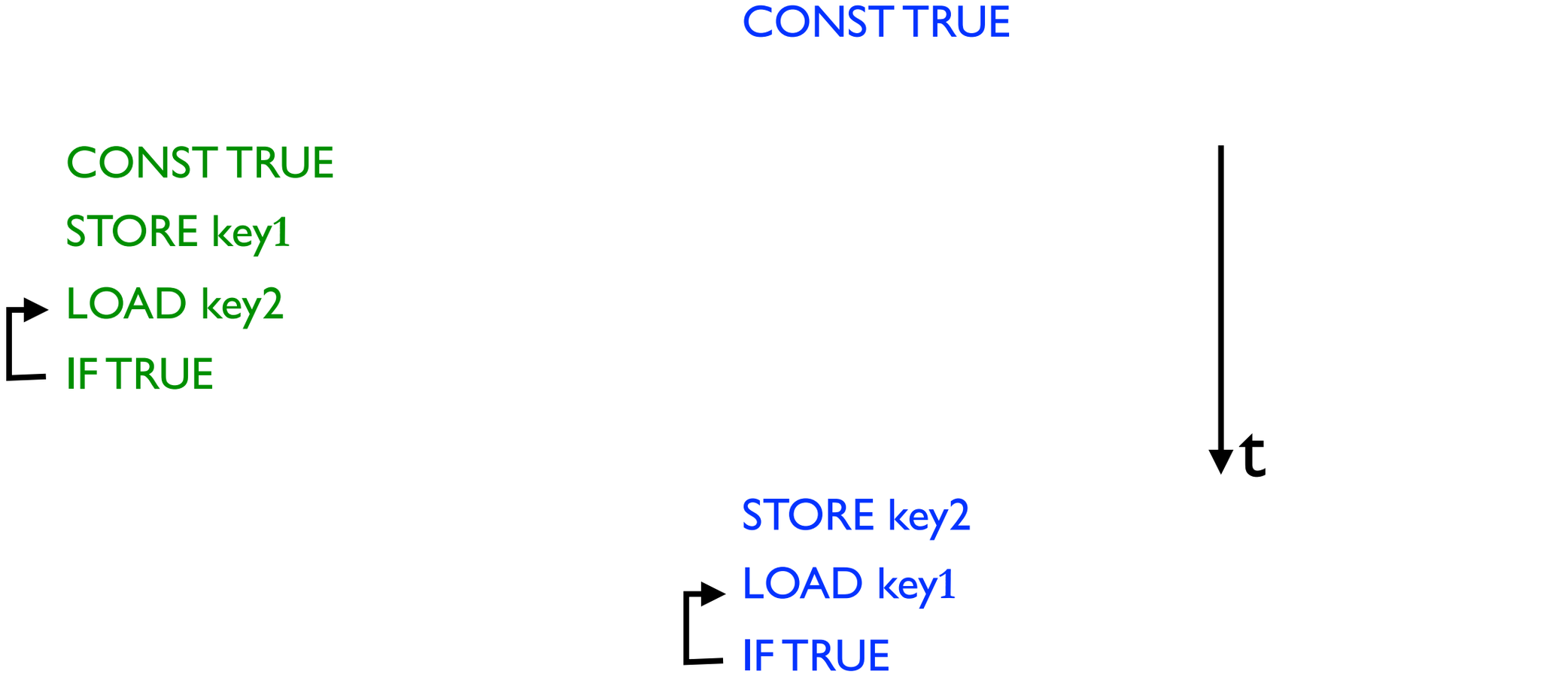
- Besser? Sobald Absicht des anderen erklärt, kein Eintritt mehr möglich

  ⇒ gegenseitiger Ausschluss

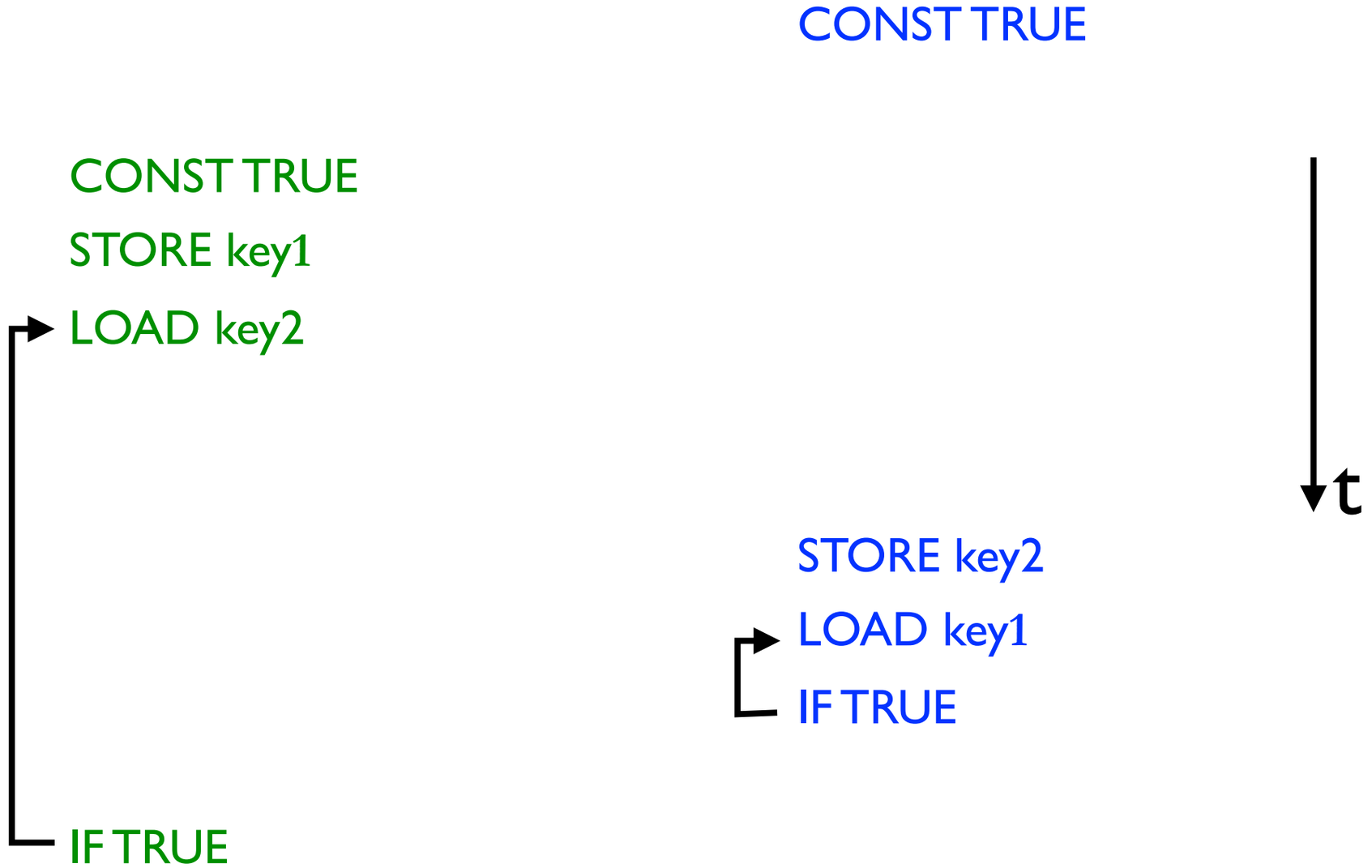
- Plausibilisierung: Runterbrechen auf „unteilbare“ Operationen und Durchspielen der Möglichkeiten



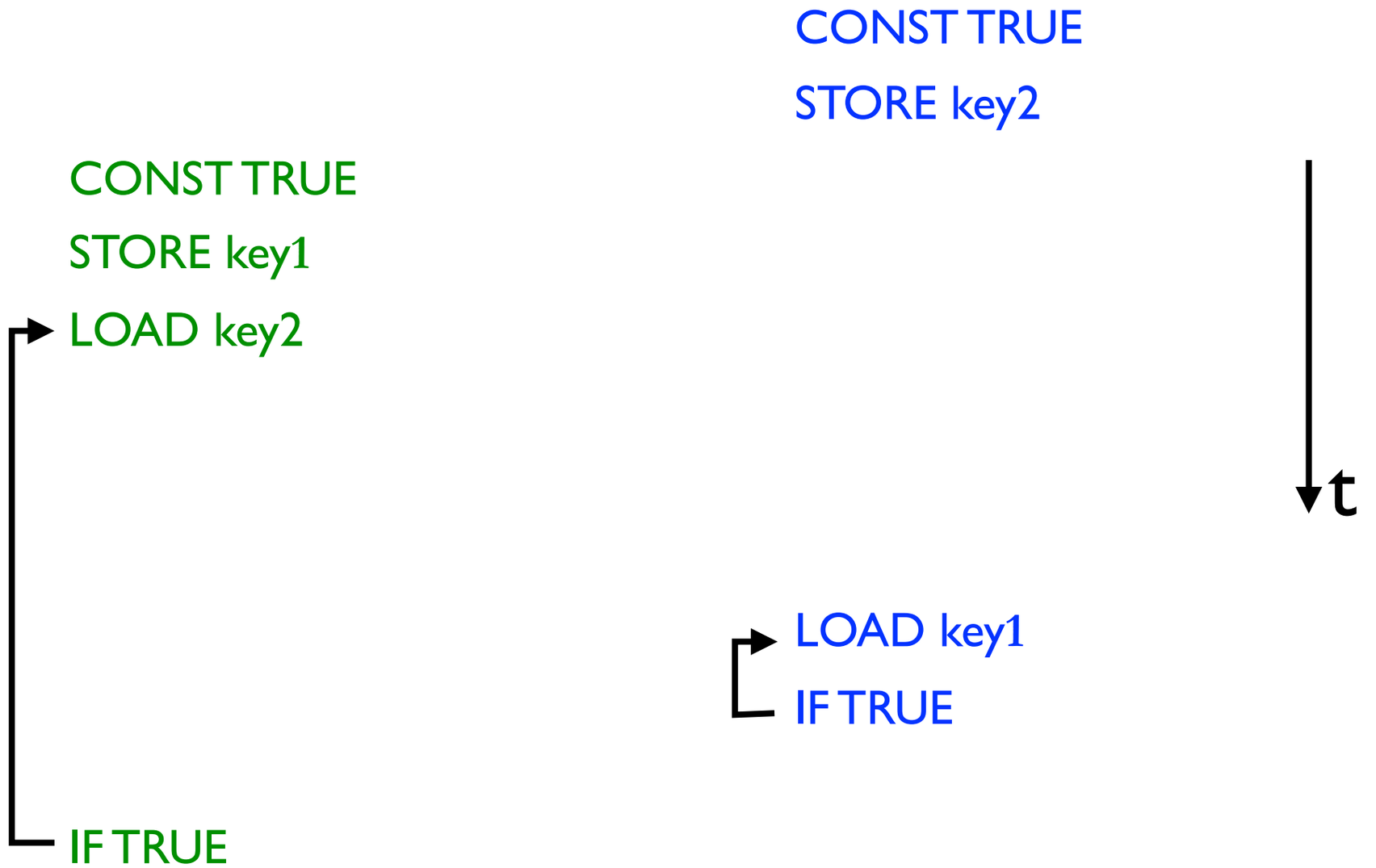
- Plausibilisierung: Runterbrechen auf „unteilbare“ Operationen und Durchspielen der Möglichkeiten



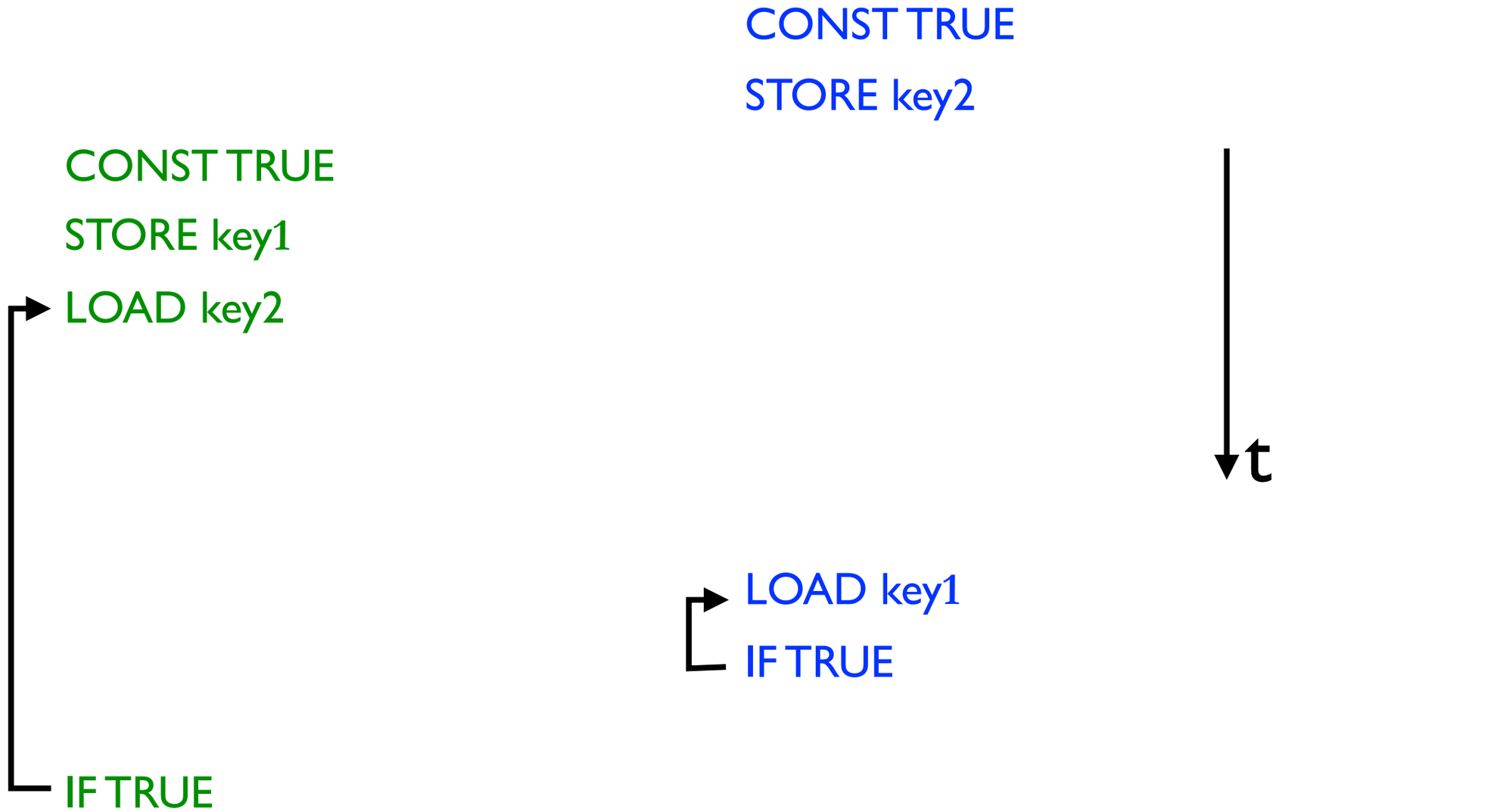
- Plausibilisierung: Runterbrechen auf „unteilbare“ Operationen und Durchspielen der Möglichkeiten



- Plausibilisierung: Runterbrechen auf „unteilbare“ Operationen und Durchspielen der Möglichkeiten



- Plausibilisierung: Runterbrechen auf „unteilbare“ Operationen und Durchspielen der Möglichkeiten



- Verklemmung, wenn beide nebenläufig Absicht erklären
  ⇒ beide können für immer in anschließender Schleife hängenbleiben

3. Versuch: Nicht auf eigener Absicht beharren

```
bool key1, key2 = false;

lock1() {                    lock2() {
  key1 = true;                 key2 = true;
  while (key2) {               while (key1) {
    key1 = false;                key2 = false;
    ... Warten ...               ... Warten ...
    key1 = true;                 key2 = true;
  };                           };
}                            }

unlock1() {                  unlock2() {
  key1 = false;                key2 = false;
}                            }
```

3. Versuch: Nicht auf eigener Absicht beharren

```
bool key1, key2 = false;

lock1() {                    lock2() {
  key1 = true;                 key2 = true;
  while (key2) {               while (key1) {
    key1 = false;                key2 = false;
    ... Warten ...               ... Warten ...
    key1 = true;                 key2 = true;
  };                           };
}                            }

unlock1() {                  unlock2() {
  key1 = false;                key2 = false;
}                            }
```

- Keine Verklemmung möglich, da eigene Absicht im Konfliktfall zurückgenommen wird

  ⇒ dann erneut probieren

  ⇒ vielleicht klappt's diesmal (abhängig von Unterbrechungen)

- Allerdings: Konfliktfall kann immer wieder reproduziert werden, z.B.

```
key1 = true                          key2 = true

if key2?  JA                         if key1?  JA

key1 = false                         key2 = false

... Warten ...                       ... Warten ...
```

```
key1 = true                          key2 = true

if key2?  JA                         if key1?  JA

key1 = false                         key2 = false

    ...                                  ...
```

⇒ „After-you-after-you“-Problem

⇒ Lösen durch Priorisierung (einer „gewinnt“)

4. Versuch: Versuch 3 mit Priorität versehen ⇒ Prozess1 beharrt wieder

```
bool key1, key2 = false;

lock1() {                    lock2() {
  key1 = true;                 key2 = true;
  while (key2)                 while (key1) {
    ;                            key2 = false;
}                                ... Warten ...
                                 key2 = true;
                               };
                             }

unlock1() {                  unlock2() {
  key1 = false;                key2 = false;
}                            }
```

4. Versuch: Versuch 3 mit Priorität versehen ⇒ Prozess1 beharrt wieder

```
bool key1, key2 = false;

lock1() {                    lock2() {
  key1 = true;                 key2 = true;
  while (key2)                 while (key1) {
    ;                            key2 = false;
}                                ... Warten ...
                                 key2 = true;
                               };
                             }

unlock1() {                  unlock2() {
  key1 = false;                key2 = false;
}                            }
```

- Wenn nur ein Prozess in den kritischen Abschnitt will, kommt er rein
- Wenn beide wollen, gewinnt Prozess1 ⇒ kein After-you-after-you mehr

4. Versuch: Versuch 3 mit Priorität versehen ⇒ Prozess1 beharrt wieder

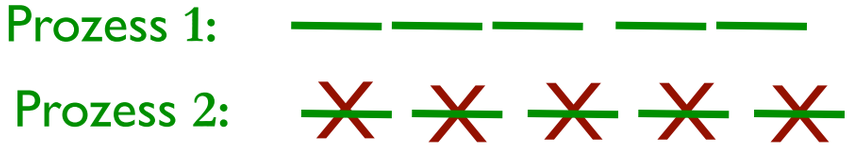
```
bool key1, key2 = false;

lock1() {                    lock2() {
  key1 = true;                 key2 = true;
  while (key2)                 while (key1) {
    ;                            key2 = false;
}                                ... Warten ...
                                 key2 = true;
                               };
                             }
```

```
unlock1() {                  unlock2() {
  key1 = false;                key2 = false;
}                            }
```

- Wenn nur ein Prozess in den kritischen Abschnitt will, kommt er rein
- Wenn beide wollen, gewinnt Prozess1 ⇒ kein After-you-after-you mehr
- Aber: Prozess1 hat Vorteil gegenüber Prozess2

  ⇒ keine Fairness

  ⇒ „Verhungern“ von Prozess2 möglich (Starvation)



5. Versuch: Fairness erzielen, indem Prozesse abwechselnd zugelassen werden?

```
int favorit = 1;

lock1() {                          lock2() {
  while (favorit != 1)               while (favorit != 2)
    ;                                  ;
}                                   }

unlock1() {                         unlock2() {
  favorit = 2;                        favorit = 1;
}                                   }
```

⇒ gegenseitiger Ausschluss gewährleistet

(Kein Konflikt: favorit zwar gemeinsame Variable, aber nur in unlock() verändert)

5. Versuch: Fairness erzielen, indem Prozesse abwechselnd zugelassen werden?

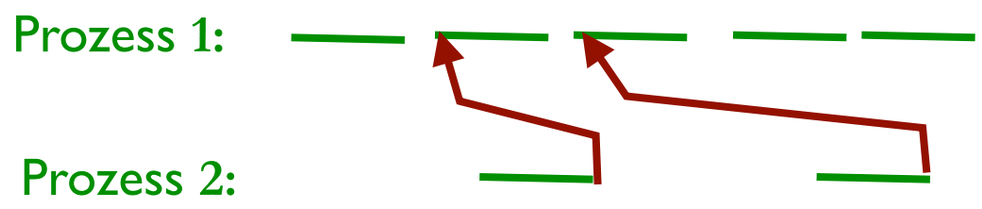
```
int favorit = 1;

lock1() {                          lock2() {
  while (favorit != 1)               while (favorit != 2)
    ;                                  ;
}                                   }

unlock1() {                         unlock2() {
  favorit = 2;                        favorit = 1;
}                                   }
```

⇒ gegenseitiger Ausschluss gewährleistet

(Kein Konflikt: favorit zwar gemeinsame Variable, aber nur in unlock() verändert)

- Aber: Wenn Prozess2 seltener oder gar nicht in kritischen Abschnitt will, wird Prozess1 unnötig/vollständig blockiert

⇒ Auch Verhungern möglich



Implementierungen von lock()/unlock()

- Einige nicht funktionierende Algorithmen kennengelernt
- Aufgetretene Probleme:
  - fehlender gegenseitiger Ausschluss
  - Verklemmungen
  - After-you-after-you
  - Verhungern

⇒ Gibt es denn auch funktionierenden Algorithmus?

Fragen – Teil 1

- Wie kann man den gegenseitigen Ausschluss gewährleisten? Warum ist ein Unterbrechungsausschluss dabei nicht immer das geeignete Mittel?
- Nach welchen Kriterien wird die Korrektheit bzw. Güte von Locking-Algorithmen bewertet?
- Warum sollte man die Bewertung von Locking-Algorithmen auf der Grundlage von unteilbaren Operationen durchführen?

Teil 2:
Locking-Algorithmen

16 Realisierung von lock()/unlock(), Ute Bormann

Implementierungen von lock()/unlock()

- Einige nicht funktionierende Algorithmen kennengelernt
- Aufgetretene Probleme:
  - fehlender gegenseitiger Ausschluss
  - Verklemmungen
  - After-you-after-you
  - Verhungern

  ⇒ Gibt es denn auch funktionierenden Algorithmus?

- Verfeinern von Versuch 3 durch Kombination mit Versuch 5

  ⇒ After-you-after-you durch geschickte Prioritätenregelung vermeiden

Wdh.: Versuch 3

```
bool key1, key2 = false;

lock1() {
  key1 = true;
  while (key2) {

      key1 = false;
      … warten …
      key1 = true;

  };
}

unlock1() {

  key1 = false;
}
```

```
lock2() {
   key2 = true;
   while (key1) {

      key2 = false;
      … warten …
      key2 = true;

   };
}

unlock2() {

  key2 = false;
}
```

```
bool key1, key2 = false;
int favorit = 1;

lock1() {
  key1 = true;
  while (key2) {
    if (favorit != 1) {
      key1 = false;
      while (favorit != 1)
        ;
      key1 = true;
    };
  };
}

unlock1() {
  favorit = 2;
  key1 = false;
}
```

```
lock2() {
  key2 = true;
  while (key1) {
    if (favorit != 2) {
      key2 = false;
      while (favorit != 2)
        ;
      key2 = true;
    };
  };
}

unlock2() {
  favorit = 1;
  key2 = false;
}
```

Ist der Algorithmus korrekt?

⇒ Ausschließen der möglichen Probleme

- Gegenseitiger Ausschluss
  - Gleicher Mechanismus wie bei Versuch 2 und 3
  - Prozess kann kritischen Abschnitt nur betreten, wenn anderer nicht will

Ist der Algorithmus korrekt?

⇒ Ausschließen der möglichen Probleme

- Gegenseitiger Ausschluss
  - Gleicher Mechanismus wie bei Versuch 2 und 3
  - Prozess kann kritischen Abschnitt nur betreten, wenn anderer nicht will

- Verklemmungsfreiheit
  - Ähnlicher Mechanismus wie bei Versuch 3
  - Es können nicht beide Prozesse auf Eintrittsabsicht beharren
  - Nicht-Favorit nimmt Absicht zurück

Ist der Algorithmus korrekt?

⇒ Ausschließen der möglichen Probleme

- Gegenseitiger Ausschluss
  - Gleicher Mechanismus wie bei Versuch 2 und 3
  - Prozess kann kritischen Abschnitt nur betreten, wenn anderer nicht will

- Verklemmungsfreiheit
  - Ähnlicher Mechanismus wie bei Versuch 3
  - Es können nicht beide Prozesse auf Eintrittsabsicht beharren
  - Nicht-Favorit nimmt Absicht zurück

- Starvation-Freiheit
  - Wer gerade im kritischen Abschnitt war, ist nicht Favorit für's nächste Mal

    ⇒ Fairness gewährleistet
  - Wenn einer gar nicht will, kommt der andere auch wiederholt rein
  - Auch kein „After-you-after-you“, da klare Favorit-Regelung

Lock/Unlock und unteilbare Operationen

- Bisherige Lock-Algorithmen setzen gewisse Menge von „unteilbaren“ Maschineninstruktionen voraus

  ⇒ sequentielles Durchlaufen hat intendiertes Ergebnis

```
Prozess/Thread1      Prozess/Thread2
    CONST TRUE
    STORE key1
                          LOAD key1
                            ↓
                          TRUE
```

Einprozessorsystem

Lock/Unlock und unteilbare Operationen

- Bisherige Lock-Algorithmen setzen gewisse Menge von „unteilbaren“ Maschineninstruktionen voraus

  ⇒ sequentielles Durchlaufen hat intendiertes Ergebnis

```
Prozess/Thread1      Prozess/Thread2
    CONST TRUE
    STORE key1
                          LOAD key1
                             ↓
                             ?
```

Mehrprozessorsystem

- Muss in modernen Rechnerarchitekturen nicht so sein.
- Beispiel: Mehrprozessorsystem mit jeweils eigenen CPU-Caches, die nur bei Bedarf nach weiterer Cache-Line zurückgeschrieben werden

  ⇒ key1=TRUE im Cache, noch nicht im Hauptspeicher

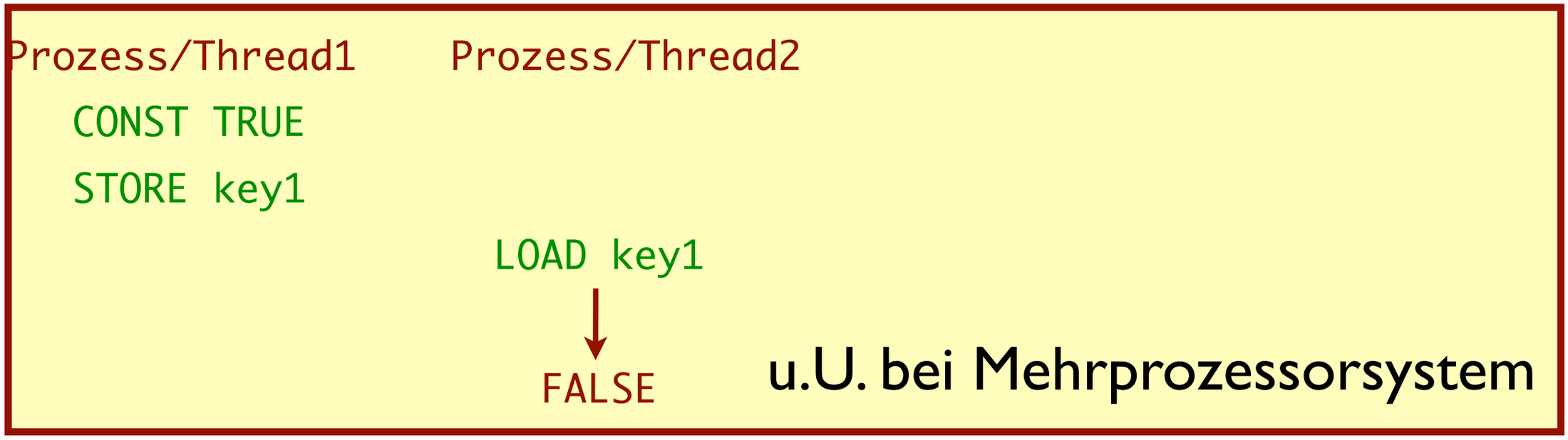
  ⇒ anderer Prozess/Thread liest falschen Wert

  ⇒ Algorithmus geht schief

Lock/Unlock und unteilbare Operationen

- Bisherige Lock-Algorithmen setzen gewisse Menge von „unteilbaren“ Maschineninstruktionen voraus

  ⇒ sequentielles Durchlaufen hat intendiertes Ergebnis



- Muss in modernen Rechnerarchitekturen nicht so sein.
- Beispiel: Mehrprozessorsystem mit jeweils eigenen CPU-Caches, die nur bei Bedarf nach weiterer Cache-Line zurückgeschrieben werden

  ⇒ key1=TRUE im Cache, noch nicht im Hauptspeicher

  ⇒ anderer Prozess/Thread liest falschen Wert

  ⇒ Algorithmus geht schief

- Vielzahl von unangenehmen Auswirkungen, ganz besonders für Schutz von kritischen Abschnitten
- Aber enorme Performance-Steigerung möglich

- Vielzahl von unangenehmen Auswirkungen, ganz besonders für Schutz von kritischen Abschnitten
- Aber enorme Performance-Steigerung möglich

- Daher: „Normales“ Arbeiten möglichst schnell

  \+ spezielle Unterstützung für kritische Abschnitte (langsamer)

  ⇒ „Unteilbare“ Spezialoperationen vorsehen

  - Bewirken z.B. sofortiges Rausschreiben des Caches

- Vielzahl von unangenehmen Auswirkungen, ganz besonders für Schutz von kritischen Abschnitten
- Aber enorme Performance-Steigerung möglich

- Daher: „Normales“ Arbeiten möglichst schnell

  \+ spezielle Unterstützung für kritische Abschnitte (langsamer)

  ⇒ „Unteilbare“ Spezialoperationen vorsehen

  - Bewirken z.B. sofortiges Rausschreiben des Caches
  - Vereinfachen Lock-Algorithmen

    ⇒ mehrere Schritte zu einem zusammenfassen

    z.B. Wert laden
    \+ Wert ändern
    \+ Wert abspeichern
    } in einem Schritt

  - Währenddessen ist Zugriff anderer Prozessoren auf dieses Speicherwort blockiert

Beispiel 1: Test_and_set

```
bool key = false;
lock() {
  while                    (key) ;
  key = true;
}
unlock() {
  key = false;
}
```

Wdh.: Versuch 1

Beispiel 1: Test_and_set

```
bool key = false;
lock() {
  while (test_and_set (key))
    ;
}
unlock() {
  key = false;
}
```

- Semantik von test_and_set (key):
  1. Abfragen des Werts von key
     ⇒ liefert true/false ⇒ nur bei false weiter
  2. Setzen auf true
     ⇒ true ➔ true: „noop“
     false ➔ true: kritischer Abschnitt reserviert
- Beides in einem Schritt
  ⇒ kein anderer Prozess kann dazwischenfunken

Beispiel 2: Load_locked / Store_conditional

- Test_and_set ist eigentlich zu mächtig
- „Laden“ und „speichern“ können getrennte Operationen sein, sofern nebenläufiger Zugriff anderer Prozesse bemerkt wird

```
bool key = false;
lock() {
  do {
    k = load_locked(&key);
  } while (k==true || !store_conditional(&key,true));
}

unlock() {
  key = false;
}
```

- Store_conditional liefert false, wenn anderer Prozess seit load_locked auf key zugegriffen hat (+ key bleibt false) ⇒ erneut probieren
- Algorithmus hat potentielles „After-you-after-you“-Problem

Fragen – Teil 2

- Skizziere einen einfachen Locking-Algorithmus.

Zusammenfassung

- Unterbrechungsausschluss
- aktives Warten mit Schlossvariablen
  ⇒ verschiedene Algorithmen betrachtet
- Potentielle Probleme:
  - kein gegenseitiger Ausschluss
  - Verklemmungen
  - Verhungern
  - After-you-after-you
- Spezielle unteilbare Operationen (z.B. Test_and_set)
- Beispielimplementierungen

Realisierung von lock()/unlock() – Fragen

1. Wie kann man den gegenseitigen Ausschluss gewährleisten? Warum ist ein Unterbrechungsausschluss dabei nicht immer das geeignete Mittel?
2. Nach welchen Kriterien wird die Korrektheit bzw. Güte von Locking-Algorithmen bewertet?
3. Warum sollte man die Bewertung von Locking-Algorithmen auf der Grundlage von unteilbaren Operationen durchführen?
4. Skizziere einen einfachen Locking-Algorithmus.