Universität Bremen

Betriebssysteme

Kritische Abschnitte

Dr. Bernhard J. Berger
AG Sichere Systeme

Universität Bremen

Themenübersicht

- Motivation
- Race Condition
- Transaktion
- Kritische Abschnitte
- Race Condition II
- Race Condition III

Grundlagen

- Aktives Warten
  - Striktes Abwechseln
  - Petersons Algorithmus
  - Test-and-Set Variable
  - Bewertung
- Ringbuffer
- Nicht-blockierendes Schreiben

Lösungen

- Nachweise
- POSIX-Bibliothek
- Umsetzung

Implementierung

Motivation – Therac-25

- Elektronenlinearbeschleuniger
- Strahlentherapie von Krebstumoren
- Todesfälle in den 1980er auf Grund überhöhter Strahlendosis

The technician recalled changing the command 'x' to 'e' that day. It was found that doing it quickly enough resulted in radiation overdose in almost 100% of cases.

Quelle: https://pvs-studio.com/en/blog/posts/0438/

Erinnerung – Race Condition

- **Szenario:** Zwei Personen versuchen gleichzeitig Geld vom gleichen Konto abzuheben.
- **Randbedingungen:** Der Kontostand darf nicht unter null sinken.

```
ATM(P1):

int a   = inputAccountNumber();
float x = inputAmount();
float b = getBalance(a);

if ( x <= b ) {
  b = b - x;
  setBalance(b);
  outputCash(x);
}
```

||

```
ATM(P2):

int a   = inputAccountNumber();
float x = inputAmount();
float b = getBalance(a);

if ( x <= b ) {
  b = b - x;
  setBalance(b);
  outputCash(x);
}
```

Parallele Ausführung

Erinnerung – Race Condition

- **Szenario:** Zwei Personen versuchen gleichzeitig Geld vom gleichen Konto abzuheben.
- **Randbedingungen:** Der Kontostand darf nicht unter null sinken.

```
ATM(P1):

int a   = inputAccountNumber();
float x = inputAmount();  // x = 20
float b = getBalance(a);  // b = 100

if ( x <= b ) {           // 20 <= 100
  b = b - x;              // b = 80
  setBalance(b);          // Setzt auf 80
  outputCash(x);
}
```

```
ATM(P2):

int a   = inputAccountNumber();
float x = inputAmount(); // x = 50
float b = getBalance(a); // b = 100

if ( x <= b ) {          // 50 <= 100
  b = b - x;             // b = 50
  setBalance(b);         // Setzt auf 50
  outputCash(x);
}
```

Transaktion

- Eine Transaktion ist eine logisch kohärente Sequenz von Programmanweisungen
- Stammt aus Datenbanken aber wurde in den Bereich der Betriebssysteme übernommen
- Für eine isolierte Transaktion gibt es üblicherweise die folgenden Korrektheitsanforderungen:
  - **Atomizität** Eine Transaktion wird gar nicht oder vollständig ausgeführt
  - **Konsistenz** Die Auswirkungen einer Transaktion muss mit konsistent mit den Daten-Constraints sein
  - **Isolation** Die Auswirkung einer Transaktion darf erst nach Abschluss sichtbar sein. Zwischenzustände dürfen nicht sichtbar sein.
  - **Dauerhaftigkeit** Der Effekt einer Transaktion muss persistent sein (auch im Falle eines Hardware-Ausfalls)

Gleichzeitige Transaktionen

Gegeben eine Sammlung von Transaktionen $T_1, T_2, \dots, T_n$, dann ist eine gleichzeitige Ausführung $(T_1 \parallel T_2 \parallel \cdots \parallel T_n)$ serialisierbar, wenn es eine Permutation $\pi: \{1, 2, \dots, n\} \rightarrow \{1, 2, \dots n\}$ gibt, so dass der Effekt der gleichzeitigen Ausführung die gleiche Auswirkung wie eine sequentielle Ausführung haben $T_\pi(1); T_\pi(2); \dots; T_\pi(n)$.

Das Geldabheben aus dem Beispiel verstößt gegen die Serialisierbarkeitseigenschaft, da es keine Ausführungsreihenfolge gibt, die das gleiche Ergebnis liefert:

Angenommen Prozess 1 möchte 80€ abheben und Prozess 2 möchte 50€ abheben. In beiden Möglichkeiten wird nur eine Transaktion durchgeführt. Die andere wird abgelehnt.

Kritischer Abschnitt

Ein Code-Abschnitt einer Transaktion, dessen parallele Ausführung die Serialisierbarkeitseigenschaft verletzt, nennt man einen kritischen Abschnitt.

Hinweis: Der kritische Abschnitt in nebenläufigen Transaktionen kann unterschiedliche Code-Abschnitte enthalten.

Beispiel: Eine Transaktion führt eine Abhebung, während eine andere Transaktion eine Geldeinzahlung durchführt.

Kritischer Abschnitt – Beispiel

```
ATM(P1):

int a   = inputAccountNumber();
float x = inputAmount();
float b = getBalance(a);

if ( x <= b ) {
  b = b - x;
  setBalance(b);
  outputCash(x);
}
```

```
ATM(P2);

int a = inputAccountNumber()
float c = inputCash();
float b = getBalance(a);

b = b + c;
setBalance(b);
```

Parallele Ausführung

■ *Kritischer Abschnitt*

Race Conditions – Beispiel

- **Szenario:** Nicht-atomarer schreibender Zugriff auf den Speicher.
- **Randbedingungen:** Auf einer 32-Bit-Architektur braucht das Schreiben eines 64-Bit Integer zwei Taktzyklen.

```
long long z;
```

P1:

```
long long foo = 1ll << 32;
```

```
z = foo;
```

Parallele Ausführung

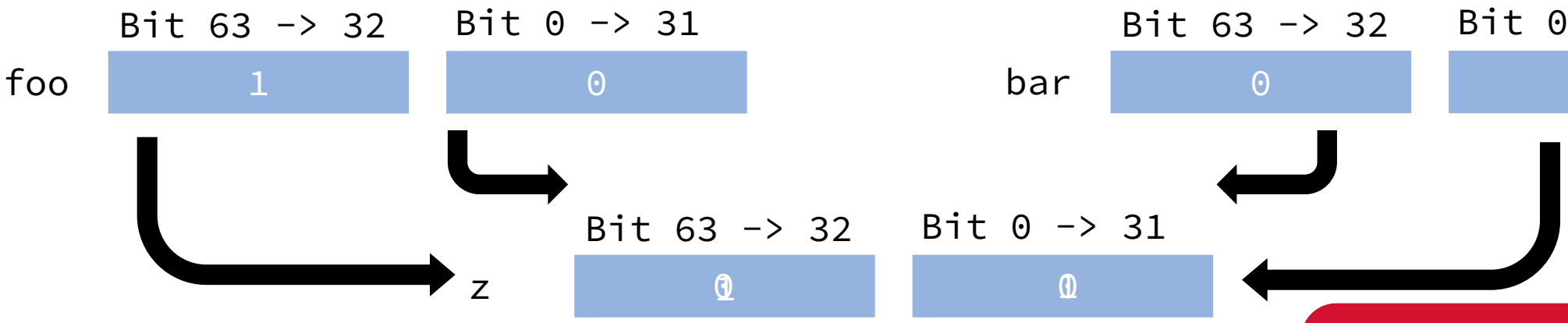
P2:

```
long long bar = 1ll;
```

```
z = bar;
```

Kritischer Abschnitt

Race Conditions – Beispiel



Diese nebenläufige Transaktionen sind nicht serialisierbar, da die serielle Ausführung in $z = [1, 0]$ oder $z = [0, 1]$ endet

```
long long z;

P1:
long long foo = 1ll << 32;

z = foo;
```

Parallele Ausführung

```
P2:
long long bar = 1ll;

z = bar;
```

■ *Kritischer Abschnitt*

Kritischer Abschnitt – Beispiel

```
int myStack[MAX];
size_t s = 0;

while (1) {

  myStack[s] = produce();
  s = s + 1;

}
```

```
while (1) {
  if ( s > 0 ) {
    s = s - 1;
    int x = myStack[s];
    consume(x);
  }
}
```

Parallele Ausführung

■ *Kritischer Abschnitt*

Universität Bremen

Themenübersicht

- Motivation
- Race Condition
- Transaktion
- Kritische Abschnitte
- Race Condition II
- Race Condition III

Grundlagen

- Aktives Warten
  - Striktes Abwechseln
  - Petersons Algorithmus
  - Test-and-Set Variable
  - Bewertung
- Ringbuffer
- Nicht-blockierendes Schreiben

Lösungen

- Nachweise
- POSIX-Bibliothek
- Umsetzung

Implementierung

Aktives Warten

```
void enter_critical(int pid) {
  while ( <condition for entering CS not fulfilled> );
  // now CS can be entered
}

void leave_critical(int pid) {
  <reset condition variables to indicate that CS has been left>;
}
```

Aktives Warten führt eine Schleife aus, bis eine Bedingung zum Betreten des kritischen Abschnitts erfüllt ist. Beim Betreten wird sichergestellt, dass die Bedingung nicht mehr hält und beim Verlassen muss die Bedingung wieder hergestellt werden.

```
enter_critical(pid);
// kritischer Abschnitt
leave_critical(pid);
```

Striktes Abwechseln

- Prozesse wechseln sich ab
- Reihenfolge strikt vorgegeben
- Prozesseanzahl festgelegt

```
const int process_count = ...;
int turn = 0;

void enter_critical(int pid) {
  while(turn != pid);
}

void leave_critical(int pid) {
  turn = (turn + 1) % process_count;
}
```

- Verfahren ist eher akademischer Natur
- Alle Prozesse müssen in den Abschnitt eintreten
- Wenn Prozess k nicht eintreten möchte, dann werden die Prozesse $k + 1$ ... nie aktiv

Algorithmus von Peterson

- Prozesse wechseln sich ab
- Reihenfolge nach Bedarf

```
const int process_count = 2;

volatile int turn; // Aktiver Prozess
volatile int interested[process_count] = {false}; // Initialisierung mit false

void enter_critical(int pid) {
  int other = 1 - pid;
  interested[pid] = true; // Interesse bekunden
  turn = other; // Anderer Prozess hat Vorrang

  while (interested[other] && turn == other);
}

void leave_critical(int pid) {
  // Signalisiert, dass wir den Bereich verlassen haben
  interested[pid] = FALSE;
}
```

Test-and-Set Variable

- Prozessorgestützte Umsetzung
- Assembler `CMPXCHG`, `CMPXCHG8B`, `CMPXCHG16B`
- Assembler-Befehl ist atomar
- `CMPXCHG` Vergleichswert, Neuer Wert
  - `eax` enthält Sollwert
  - Vergleichswert ist die Sperrvariable
  - Prüft `eax == Vergleichswert`
  - Setzt neuen Wert bei Gleichheit
  - Anderenfalls wird Vergleichswert zurück gegeben
- Beispiel:
  - Wert 0 bedeutet, dass Abschnitt frei ist
  - Wert ungleich 0 bedeutet, dass der Abschnitt belegt ist
  - Jeder Prozess prüft auf 0 oder setzt seine ID

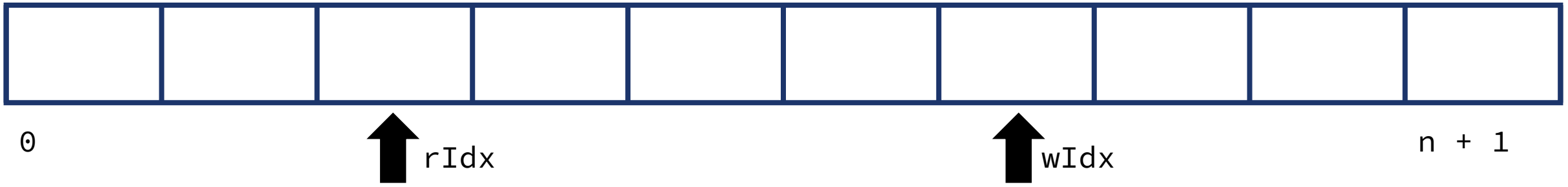
Prioritäteninversion

- Aktives Warten nutzt offensichtlich CPU-Zeit
- Bei $|Aktive\ Prozesse| > |CPUs|$ können Probleme entstehen
- Angenommen $|Aktive\ Prozesse| = 2$ und $|CPUs| = 1$
  - Angenommen P_1hat hohe Priorität und wartet
  - Angenommen P_2 hat niedrige Priorität und ist in kritischem Abschnitt
  - Scheduler priorisiert P_1und dieser belegt CPU mit aktivem Warten
  - P_2 benötigt länger, weil P_1 die CPU blockiert
  - In schlechten Situationen wird P_2 nicht mehr auf die CPU gelassen

Ringbuffer – Kommunikation

- Zwei Prozesse wollen Daten austauschen
- FIFO-Prinzip soll eingehalten werden
- Buffer hat beschränkte Größe
- Schreiber wartet aktiv, wenn Buffer voll ist
- Leser wartet aktiv, wenn Buffer leer ist

- `type buffer[n+1];`
- Buffer leer, wenn `rIdx == wIdx`
- Schreibender Prozess
  - Schreib immer `buffer[wIdx]`
  - Setzt anschließend `wIdx = (wIdx+1) % (n+1)`
  - Schreibt nur, wenn danach Buffer nicht leer ist
- Lesender Prozess
  - Liest immer `buffer[rIdx]`
  - Setzt anschließend `rIdx = (rIdx+1) % (n+1)`
  - Liest nur, wenn Buffer nicht leer ist



Non-blocking Write Protocol

- Zwei Prozesse wollen Daten austauschen
  - Prozess 1 schreibt nur Daten
  - Prozess 2 liest nur Daten
  - Nicht alle Daten müssen zugestellt werden
- Schreibender Prozess
  - Kann immer schreiben
  - Überschreibt *alte* Daten
- Lesender Prozess
  - Liest nur, wenn der Schreiber nicht schreibt
  - Prüft nach Abschluss, ob es eine Kollision gab
  - Verwirft Daten bei Kollision

Non-blocking Write Protocol

```
uintn_t CCF = 0; // Concurrency Control Flag
myType_t a;  // Buffer (e.g. structure or array)
             // updated by Writer/read by Reader

void writer() {
  while (1) {
    do_other_things();

    CCF++;
    write(&a);
    CCF++;
  }
}
```

```
void reader() {
  int c0, c1;
  myType_t b;

  while (1) {
    do {
      do { c0 = CCF; }
      while ( c0 & 1 ); // Wait until c0 is even

      b = copy(a);
      c1 = CCF;
    } while ( c1 != c0 );

    process(b);
  }
}
```

Universität Bremen

Themenübersicht

- Motivation
- Race Condition
- Transaktion
- Kritische Abschnitte
- Race Condition II
- Race Condition III

Grundlagen

- Aktives Warten
  - Striktes Abwechseln
  - Petersons Algorithmus
  - Test-and-Set Variable
  - Bewertung
- Ringbuffer
- Nicht-blockierendes Schreiben

Lösungen

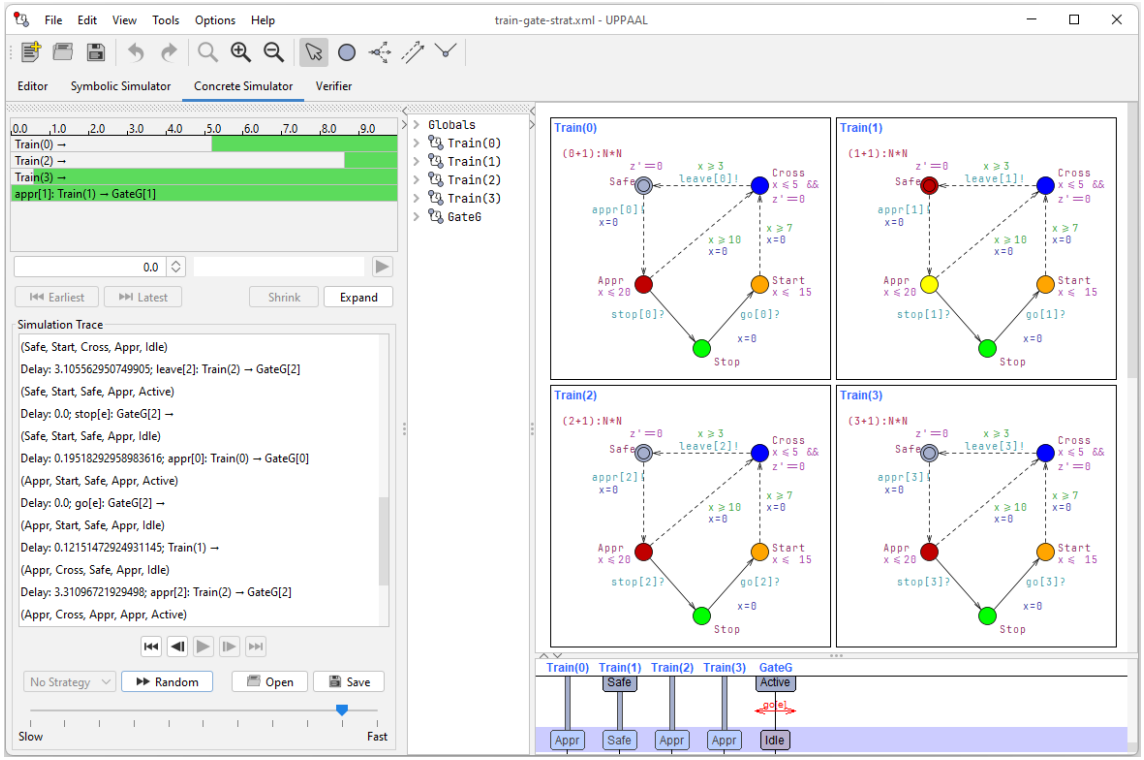
- Nachweise
- POSIX-Bibliothek
- Umsetzung

Implementierung

Verifikation von Protokollen

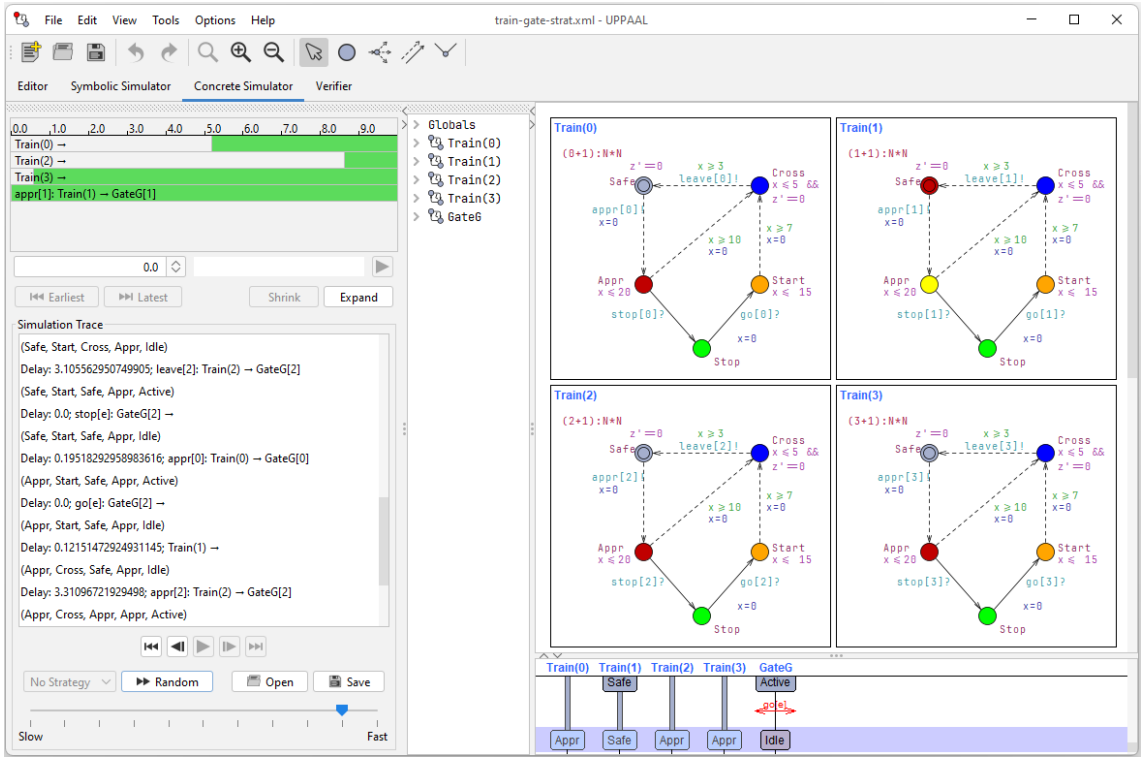
- Korrektheit bisher über Intuition
- Intuition häufig nicht ausreichend

  Beispiel: **Safety-Critical-Systems**
- Verifikationsansätze helfen beim Nachweis



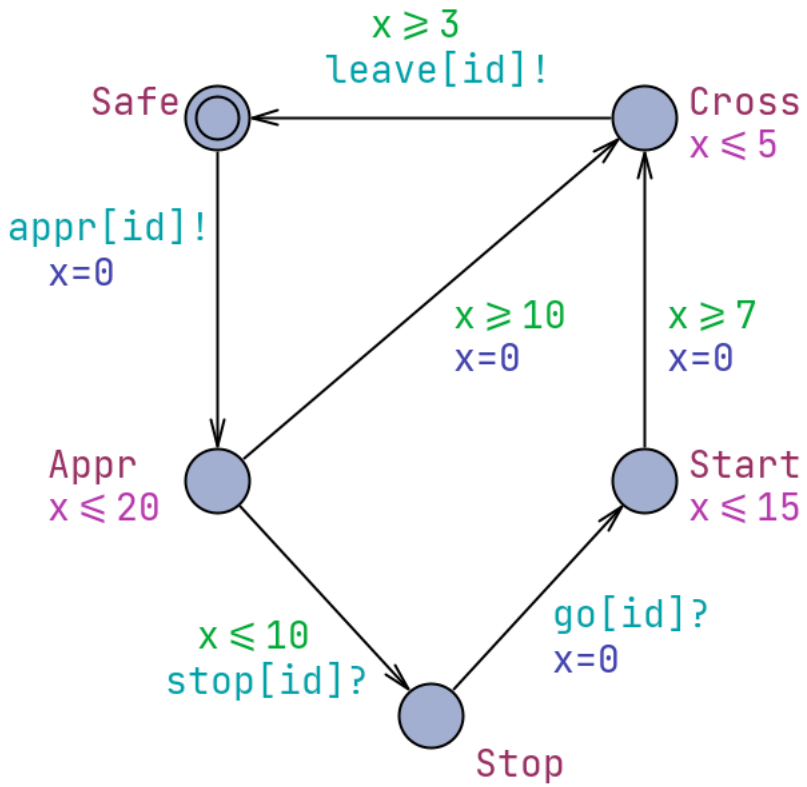
UPPAAL

- UPPAAL erlaubt
  - Modellierung
  - Simulation
  - Verifikation
- Prozesse werden mit Timed-Automata modelliert
- Verifikation mit temporaler Logik
  - CTL (Computation Tree Logic)
  - TCTL (Timed Computation Tree Logic)
  - Beschreibung von erwünschtem Programmverhalten



Timed Automata

- Prozess entspricht Timed Automata
- Bestandteile
  - Knoten
  - Kanten
- Knoten
  - Invariante
  - Initial ⇒ Startzustand
  - Urgent ⇒ Zeit verstreicht nicht
  - Committed ⇒ Zeit verstreicht nicht und Nachfolgezustand wird sofort betreten
- Kanten
  - Select ⇒ Zufällige Auswahl von Variablenwerten
  - Guard ⇒ Bedingung für Transistion
  - Sync ⇒ Synchronisation zwischen Automaten
  - Update ⇒ Aktualisierung von globalen Variablen



POSIX – Mutex

- Mutex für gegenseitigen Ausschluss
- `pthread_mutex_init` immer verwenden
- Mutex sperren
  - `pthread_mutex_lock`
  - `pthred_mutex_trylock`
- Mutex freigeben
  - `pthread_unlock`
- Mutex löschen
  - `pthread_mutex_destroy`
- Rekursive Mutexe möglich
  - Mehrfaches sperren in einem Thread möglich
  - `pthread_mutexattr_t attr;`
  - `pthread_mutexattr_init(&attr);`
  - `pthread_mutexattr_settype(&attr, PTHREAD_MUTEX_RECURSIV`
  - `pthread_mutex_init(&mutex, &attr);`

```c
#include <pthread.h>

// Initialisierung statischer Variablen
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

int main() {
    // Initialisierung von dynamischen Variablen
    pthread_mutex_init(&mutex, NULL);

    pthread_mutex_lock(&mutex);
    // kritischer Abschnitt
    pthread_mutex_unlock(&mutex);

    if(pthread_mutex_trylock(&mutex) == 0) {
        // kritischer Abschnitt
    } else {
        // sonstiger Code
    }

    pthread_mutex_destroy(&mutex);
}
```

POSIX – Read/Write Lock

- Producer/Consumer Szenario
  - `pthread_rwlock_init`
  - `pthread_rwlock_destroy`
- Leser können parallel lesen
  - `pthread_rwlock_rdlock`
  - `pthread_rwlock_unlock`
- Schreiber brauchen exklusiven Zugriff
  - `pthread_rwlock_wrlock`
  - `pthread_rwlock_unlock`

```c
#include <pthread.h>

// Initialisierung statischer Variablen
pthread_rwlock_t rwlock = PTHREAD_RWLOCK_INITIALIZER;

int main() {
    // Initialisierung von dynamischen Variablen
    pthread_rwlock_init(&rwlock, NULL);


    pthread_rwlock_rdlock(&rwlock);
    // Nur, wenn kein Write Lock aktiv
    // Weitere Read Locks werden zugelassen
    // Write Locks blockieren
    // Alle Read Locks müssen freigegeben werden
    pthread_rwlock_unlock(&rwlock);

    pthread_rwlock_wrlock(&rwlock);
    // Nur erfolgreich, wenn kein Reader oder Schreiber
    // Reads und Write Locks blockieren
    pthread_rwlock_unlock(&rwlock);

    pthread_rwlock_destroy(&rwlock);
}
```

POSIX – Semaphoren

- Semaphoren-Implementierung
- Bekannt aus Technische Informatik 2
- Beschränkt parallele Prozesse in einem Abschnitt

```c
#include <semaphore.h>

int main() {
    sem_t sem;

    // Initialisierung: Non-Shared und Wert 5
    sem_init(&sem, 0, 5);

    // Semaphoren erniedrigen
    sem_wait(&sem);

    // Nicht blockierende Alternative
    if(sem_trywait(&sem) == 0) {
    }

    // Semaphore inkrementieren
    sem_post(&sem);

    // Semaphore zerstören
    sem_destroy(&sem);
}
```

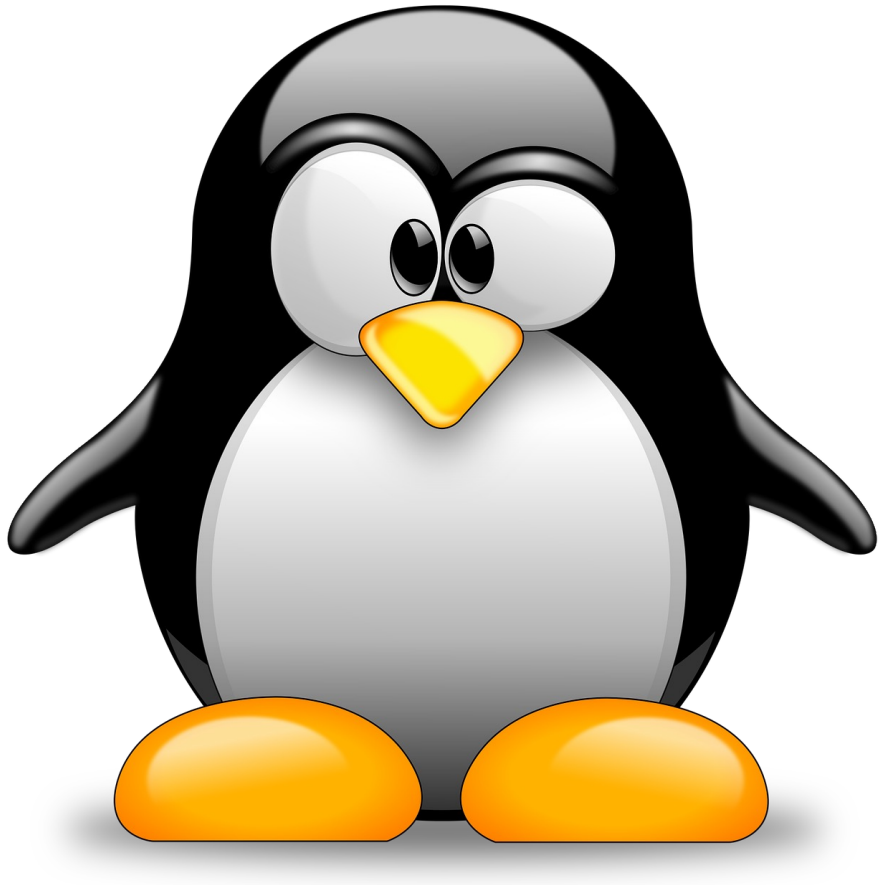
POSIX – Spinlock

- Spinlock (busy wait) für kurze kritische Abschnitte

```c
#include <pthread.h>

int main() {
    pthread_spinlock_t spinlock;

    // Initialisierung
    pthread_spin_init(&spinlock, PTHREAD_PROCESS_PRIVATE);


    pthread_spin_lock(&spinlock);
    // Kritischer Abschnitt
    pthread_spin_unlock(&spinlock);

    // Spinlock freigeben
    pthread_spin_destroy(&spinlock);
}
```

Futex – Unterstützung durch den Kernel

- Futex steht für fast user-space mutex
  - Zunächst Sperre über cmpxchange/test-and-set (User-Space)
  - Andernfalls warten über futex-Systemaufruf
- Effizienter kernel-basierter Locking-Mechanismus
- Können über Prozessgrenzen genutzt werden
- Mächtiges Konzept (aber kompliziert)
- Kernel verwaltet
  - Warteschlange pro Mutex (`futex.c`)
  - Hashtabelle zur Zuordnung Variable zu Warteschlange
- Warteschlange ist prioritätenbasiert
- Thread wird nur geweckt, wenn er aktiv werden kann



Universität Bremen

Themenübersicht

- Motivation
- Race Condition
- Transaktion
- Kritische Abschnitte
- Race Condition II
- Race Condition III

Grundlagen

- Aktives Warten
  - Striktes Abwechseln
  - Petersons Algorithmus
  - Test-and-Set Variable
  - Bewertung
- Ringbuffer
- Nicht-blockierendes Schreiben

Lösungen

- Nachweise
- POSIX-Bibliothek
- Umsetzung

Implementierung