

# Übungen zu Systemprogrammierung 2 (SP2)

## Ü 7 – Ringpuffer

Christian Eichler, Jürgen Kleinöder

Lehrstuhl für Informatik 4  
Verteilte Systeme und Betriebssysteme

Friedrich-Alexander-Universität  
Erlangen-Nürnberg

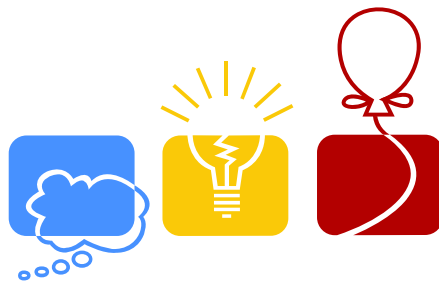
WS 2017 – 15. bis 19. Januar 2018

[http://www4.cs.fau.de/Lehre/WS17/V\\_SP2](http://www4.cs.fau.de/Lehre/WS17/V_SP2)



## ICPC Wintercontest an der FAU

### ICPC – Programmierwettbewerb, auch an der FAU



Think – Create – Solve



## Agenda

- 7.1 Werbeblock: ICPC
- 7.2 Synchronisation des Ringpuffers
- 7.3 ABA-Problem bei der Verwendung von CAS
- 7.4 Vorteile nicht-blockierender Synchronisation



## ICPC Wintercontest an der FAU

### ICPC – Was ist das?

- International Collegiate Programming Contest – veranstaltet von der ACM
- dabei sollen Teams aus drei Studenten innerhalb von fünf Stunden neun bis elf knifflige und originelle Programmieraufgaben lösen
- Problem: nur ein Computer steht zur Verfügung, aber kein Internet ☹
- dreistufiger Wettbewerb mit *Local Contest* in Erlangen, *Regional Contest* (irgendwo in Nordwesteuropa) und *World Finals* (irgendwo in der Welt)



Local Contest  
(Erlangen)



Regional Contest  
(Delft)



World Finals  
(Orlando)



## ICPC an der FAU

- am **Samstag, 27. Januar 2018** findet wieder ein FAU Local Contest statt
- von **11 bis 16 Uhr** im Informatikhochhaus
- teilnehmen darf jede/r Student/in der FAU, **Fachrichtung egal!**
- es wird jeweils **zu dritt** programmiert (**Einzelanmeldung** möglich)
  
- es wird außerdem eine **Practice Session** für alle Neulinge stattfinden, bei der (einfache) typische Probleme gezeigt und erklärt werden
  - Ort und Zeit werden noch bekannt gegeben
  - mehr **Infos/Anmeldung**: <https://icpc.cs.fau.de>

### Wichtig: Anmeldung

Zur Teilnahme am Wettbewerb ist eine Anmeldung unter <https://icpc.cs.fau.de> unbedingt erforderlich. Deadline: **22.01.2018**.



## Was bringt mir das Ganze?

- Spaß und Pizzabrötchen ☺
- **Programmiererfahrung** und Vertiefung gelernter Algorithmen
- jeder Teilnehmer erhält eine **Urkunde**
- beim Local Contest im Sommer werden die Teams bestimmt, die zum **NWERC** fahren dürfen, um unsere Uni zu vertreten
- die ganz Guten dürfen zu den **World Finals** (z.B. 2017 Rapid City (South Dakota/USA)) fahren, um unsere Uni zu vertreten



## FAU-Teams können mithalten!

- FAU eine der **erfolgreichsten** deutschen Unis der letzten Jahre
  - 2010-2012 deutschlandweiten *Subregional* gewonnen (2013+2014: 3. Platz)
  - beim **Regional Contest** 2007-2014 insgesamt 9 Medaillen gewonnen (davon **6 Gold**)
  - FAU durch 2. Platz beim NWERC 2014 als einziges deutsches Team zu den World Finals 2015 qualifiziert.
- 2003 (Beverly Hills), 2010 (Harbin), 2011 (Orlando), 2015 (Marrakesh) war ein Team der FAU zu den World Finals qualifiziert
- bisher größter Erfolg bei den World Finals: 2011: Team deFAUit, 7. Platz von 105 Teams (bzw. ca. 10 000 Teams in den Vorausscheiden)



## Wo kann ich trainieren?

- Training an der FAU:
  - FAU Online Judge: <https://icpc.cs.fau.de/oj>
  - einmaliges Freischalten mittels EST-Account notwendig
  - mit Problemen z.B. vom Winter 2012
  - Hilfestellung über das Online-Judge-Frontend bzw. IRC-Channel #hallowelt im IRCnet (z.B. [irc.uni-erlangen.de](irc://irc.uni-erlangen.de))
- Online-Plattformen zum Trainieren von Programmieraufgaben:
  - Codeforces: <http://codeforces.com>
  - SPOJ: <http://spoj.pl>
  - UVA: <http://uva.onlinejudge.org>



## Beispielproblem: „All in All“

You have devised a new encryption technique which encodes a message by inserting between its characters randomly generated strings in a clever way. To validate your method, however, it is necessary to write a program that checks if the message is really encoded in the final string.

### Input Specification:

The input contains several test cases. Each is specified by two strings  $s$ ,  $t$  of alphanumeric ASCII characters.

### Output Specification:

For each test case output, if  $s$  is a subsequence of  $t$ , i.e. if you can remove characters from  $t$  such that the concatenation of the remaining characters is  $s$ .

### Sample Input:

```
sequence subsequence
person compression
VERDI vivaVittorioEmanueleReDiItalia
caseDoesMatter CaseDoesMatter
```

### Sample Output:

```
Yes
No
Yes
No
```



7.1 Werbeblock: ICPC

7.2 Synchronisation des Ringpuffers

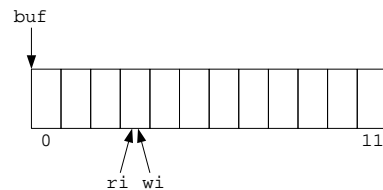
7.3 ABA-Problem bei der Verwendung von CAS

7.4 Vorteile nicht-blockierender Synchronisation



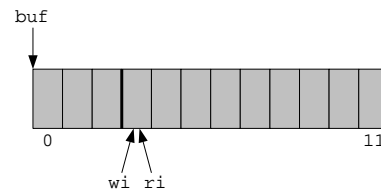
## Unter-/Überlaufsituationen

### Leerer Ringpuffer:



Weiteres Lesen würde noch nicht gefüllten Slot liefern  
→ Unterlauf!

### Voller Ringpuffer:



Weiteres Schreiben würde vollen Slot überschreiben  
→ Überlauf!

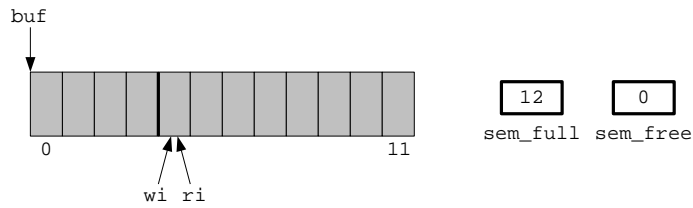
☞ Synchronisation mit Hilfe zweier Semaphore



## Wettlauf der Leser

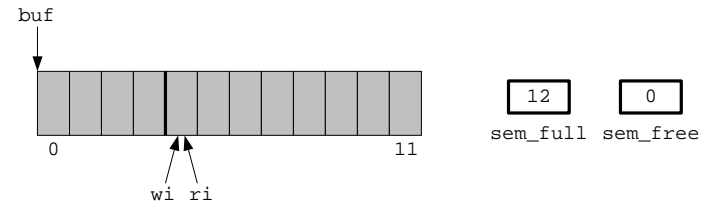
- Auslesen des Slots und Inkrementieren des Leseindex  $ri$  geschieht nicht atomar
  - Mehrere Threads könnten nebenläufig den selben Slot auslesen
- Es existiert keine Abhängigkeit der Leser untereinander  
→ Nicht-blockierende Synchronisation möglich
- Synchronisation mittels *Compare and Swap* (CAS)





- Erhöhen des Leseindex mittels CAS – vollständig korrekt?

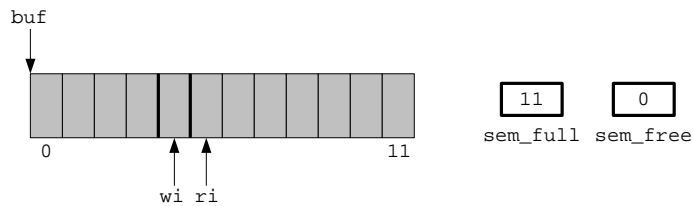
```
int get(void) {
    int fd, pos, npos;
    P(sem_full);
    do { // Wiederhole...
        pos = ri; // Lokale Kopie des Werts ziehen
        npos = (pos + 1) % 12; // Folgewert lokal berechnen
    } while(!cas(&ri, pos, npos)); // ... bis CAS erfolgreich
    fd = buf[pos];
    V(sem_free);
}
```



- Überlaufsituation: Schreiber blockiert, weil keine Slots frei

```
int get(void) {
    int fd, pos, npos;
    P(sem_full);
    do {
        pos = ri;
        npos = (pos + 1) % 12;
    } while(!cas(&ri, pos, npos));
    fd = buf[pos];
    V(sem_free);
    return fd;
}

void add(int val) {
    P(sem_free);
    buf[wi] = val;
    wi = (wi + 1) % 12;
    V(sem_full);
}
```



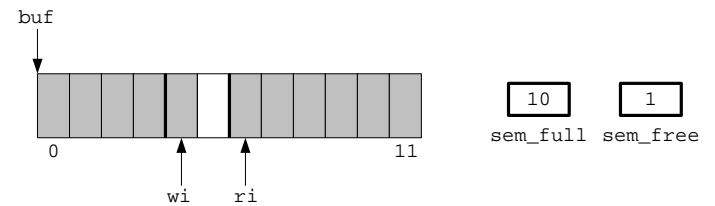
- R1 sichert sich Leseindex 4, wird nach erfolgreichem CAS verdrängt

```
int get(void) {
    int fd, pos, npos;
    P(sem_full);
    do {
        pos = ri;
        npos = (pos + 1) % 12;
    } while(!cas(&ri, pos, npos));
    fd = buf[pos];
    V(sem_free);
    return fd;
}

void add(int val) {
    P(sem_free);
    buf[wi] = val;
    wi = (wi + 1) % 12;
    V(sem_full);
}
```

R1  
pos: 4

W ↓



- R2 durchläuft get() komplett, entnimmt Datum in Slot 5

```
int get(void) {
    int fd, pos, npos;
    P(sem_full);
    do {
        pos = ri;
        npos = (pos + 1) % 12;
    } while(!cas(&ri, pos, npos));
    fd = buf[pos];
    V(sem_free);
    return fd;
}

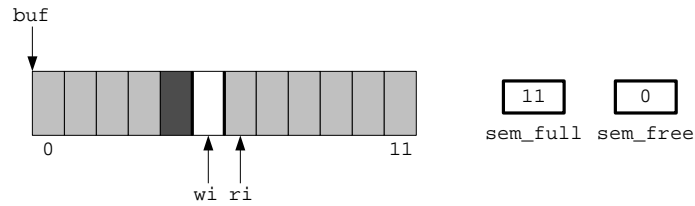
void add(int val) {
    P(sem_free);
    buf[wi] = val;
    wi = (wi + 1) % 12;
    V(sem_full);
}
```

R1  
pos: 4

R2  
pos: 5

W ↓





- W wird deblockiert, komplettiert `add()` und überschreibt Slot 4

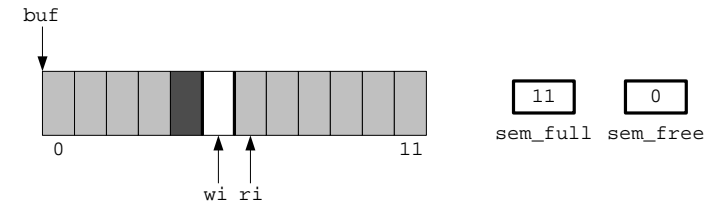
```

int get(void) {
  int fd, pos, npos;
  P(sem_full);
  do {
    pos = ri;
    npos = (pos + 1) % 12;
  } while(!cas(&ri, pos, npos));
  fd = buf[pos];
  V(sem_free);
  return fd;
}

void add(int val) {
  P(sem_free);
  buf[wi] = val;
  wi = (wi + 1) % 12;
  V(sem_full);
}

```

R1 (red) and R2 (blue) are shown as vertical lines. R1 is at pos: 4 and R2 is at pos: 5. A black arrow labeled 'W' points to the write operation in the code.



- Ursache: FIFO-Entnahmeeigenschaft des Puffers nicht sichergestellt

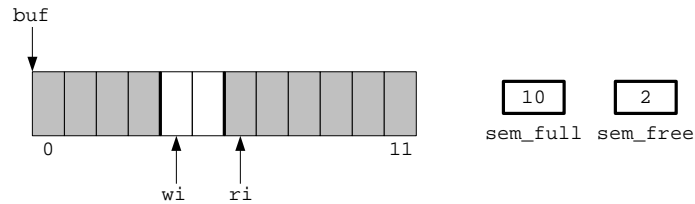
```

int get(void) {
  int fd, pos, npos;
  P(sem_full);
  do {
    pos = ri;
    npos = (pos + 1) % 12;
  } while(!cas(&ri, pos, npos));
  fd = buf[pos];
  V(sem_free);
  return fd;
}

void add(int val) {
  P(sem_free);
  buf[wi] = val;
  wi = (wi + 1) % 12;
  V(sem_full);
}

```

R1 (red) and R2 (blue) are shown as vertical lines. R1 is at pos: 4 and R2 is at pos: 5. A black arrow labeled 'W' points to the write operation in the code.



- Lösung: Entnahme des Datums **innerhalb** der CAS-Schleife

```

int get(void) {
  int fd, pos, npos;
  P(sem_full);
  do {
    pos = ri;
    npos = (pos + 1) % 12;
    fd = buf[pos]; // Datum bereits vorsorglich entnehmen
  } while(!cas(&ri, pos, npos));
  V(sem_free);
  return fd;
}

```

## Schreibindex

- Szenario: nur ein Produzenten-Thread
  - Kein nebenläufiger Zugriff auf den Schreibindex
  - `volatile` nicht erforderlich

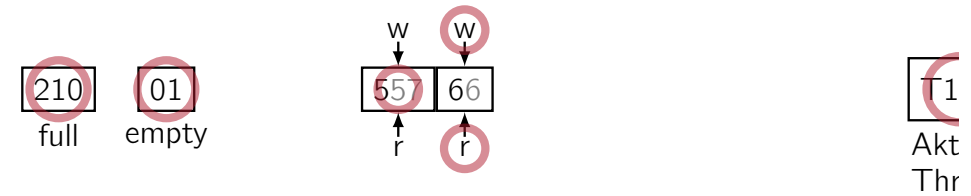
## Leseindex

- Szenario: mehrere Konsumenten-Threads möglich
  - Nebenläufiger Zugriff auf den Leseindex möglich
  - GCC-Doku: `[_sync_bool_compare_and_swap()]` is considered a full barrier. That is, no memory operand will be moved across the operation, either forward or backward. Further, instructions will be issued as necessary to prevent the processor from speculating loads across the operation and from queuing stores after the operation.
  - `volatile` also nicht falsch, aber nicht zwangsläufig erforderlich

# Agenda

- 7.1 Werbeblock: ICPC
- 7.2 Synchronisation des Ringpuffers
- 7.3 ABA-Problem bei der Verwendung von CAS
- 7.4 Vorteile nicht-blockierender Synchronisation

# ABA-Problem bei der Verwendung von CAS



```
bbGet();  
bbGet() {  
    ...  
    int retVal = 0;  
    P(full);  
    do {  
        ...  
        retVal = 5;  
    } while(!cas(&r, 0, 1));  
    ...  
    V(empty);  
}
```

/\*5\*/ bbGe  
bbPu  
/\*6\*/ bbGe

# ABA-Problem bei der Verwendung von CAS

- bbGet() liefert 5 statt 7 zurück
  - CAS schlägt nicht fehl, weil r nach dem Wiedereinlasten des Threads den selben Wert hat wie vor dessen Verdrängung
  - Zwischenzeitliche Wertänderung von r wird nicht erkannt
- Grundsätzliches Problem von inhaltsbasierten Elementaroperationen wie CAS
- Erhöhte Auftrittswahrscheinlichkeit, je kleiner der Puffer und je höher die Systemlast
- Gegenmaßnahmen siehe Vorlesung C | X-4 S. 24ff.

# ABA-Problem in den Griff bekommen

- Einführen eines Generationszählers, der bei jeder erfolgreichen Operation inkrementiert wird
- ABA-Situation: Leseindex hat nach Umlaufen des Ringpuffers wieder den alten Wert – aber Generationszähler hat anderen Wert → CAS schlägt fehl
- **Möglichkeit 1:** separate Zählvariable
  - Erfordert *Double-Word-CAS*
- **Möglichkeit 2:** eingebetteter Generationszähler
  - Nutzung der oberen Bits des Leseindex
- Keine hundertprozentige Sicherheit möglich:
  - Generationszähler hat begrenzten Wertebereich und kann überlaufen
  - Je nach Größe des Zählers und konkretem Szenario (hoffentlich) ausreichend unwahrscheinlich

# Agenda

7.1 Werbeblock: ICPC

7.2 Synchronisation des Ringpuffers

7.3 ABA-Problem bei der Verwendung von CAS

7.4 Vorteile nicht-blockierender Synchronisation



# Vorteile nicht-blockierender Synchronisation

- Vorteile gegenüber sperrenden oder blockierenden Verfahren (Auswahl):
  - Rein auf Anwendungsebene, keine teuren Systemaufrufe
  - Geringere Mehrkosten als bei Locking, wenn die CAS-Operation auf Anhieb funktioniert
  - Konkurrierende Fäden werden vom Scheduler nach dessen Kriterien eingeplant
  - Durch Locks wird eine Abhängigkeit vom Halter des Locks geschaffen:
    - Halter des Locks wird möglicherweise im kritischen Abschnitt verdrängt
    - Der „Zweite“, „Dritte“ usw. werden durch den „Ersten“ verzögert
- In unserem konkreten Anwendungsbeispiel kommen diese Vorteile nicht wirklich zum Tragen
  - Übungsbeispiel zum Begreifen des Konzepts

