

Übungen zu Systemprogrammierung 2 (SP2)

Ü 7 – Ringpuffer

Andreas Ziegler, Jürgen Kleinöder

Lehrstuhl für Informatik 4
Verteilte Systeme und Betriebssysteme

Friedrich-Alexander-Universität
Erlangen-Nürnberg

WS 2016 – 16. bis 20. Januar 2017

http://www4.cs.fau.de/Lehre/WS16/V_SP2



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



**PROGRAMMIERWETTBEWERB
AN DER FAU
28.01.2017 - 10H BIS 15H**

**RAUM 02.151-113
(CIP-POOL 2)
INFORMATIK-HOCHHAUS**

**10+ AUFGABEN
5 STUNDEN
3 STUDENTEN
1 COMPUTER**

INFOS UND ANMELDUNG: <https://icpc.cs.fau.de/>



Agenda

7.1 Werbeblock: ICPC

7.2 Synchronisation des Ringpuffers

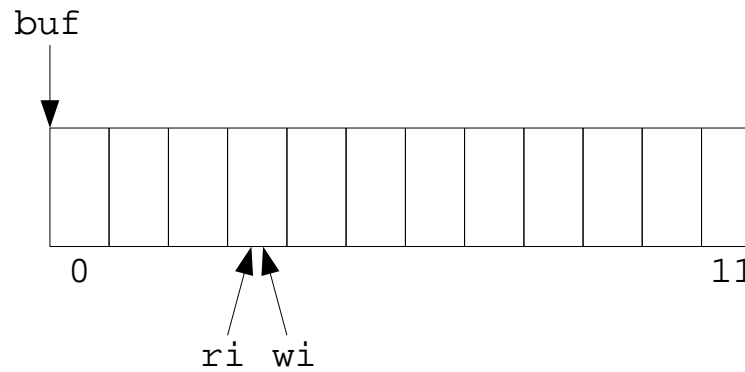
7.3 ABA-Problem bei der Verwendung von CAS

7.4 Vorteile nicht-blockierender Synchronisation



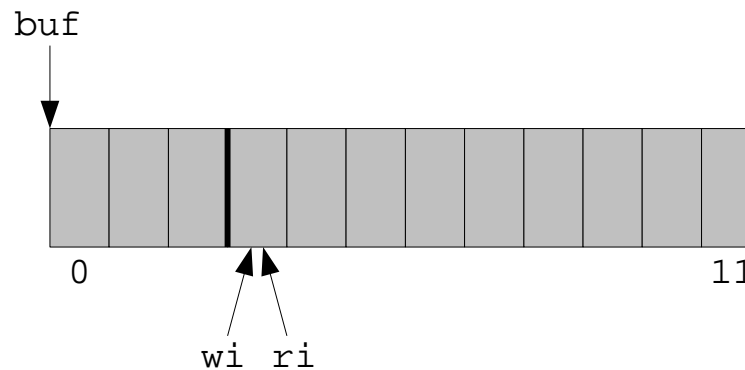
Über-/Unterlaufsituationen

■ 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

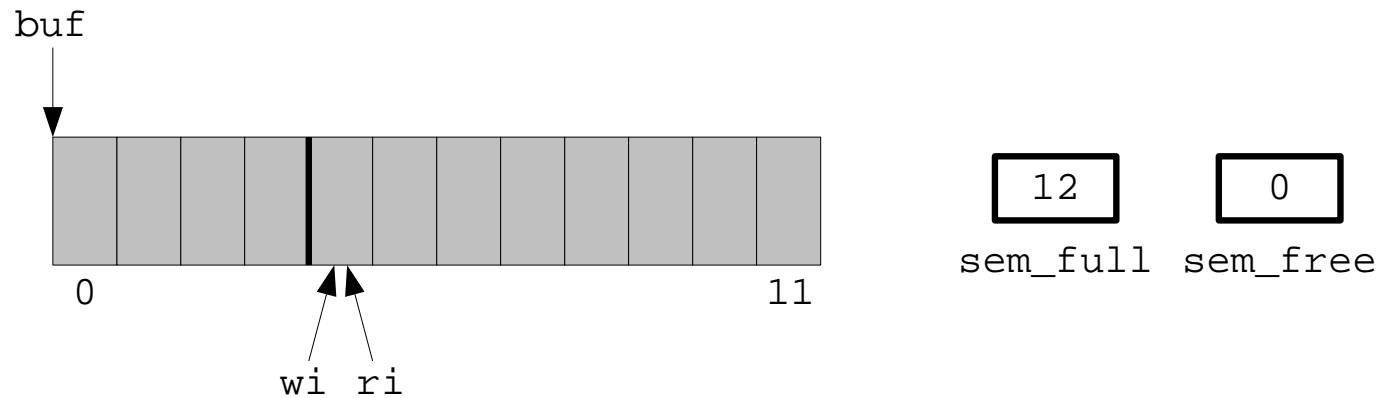


Wettlauf der Leser

- Auslesen des Slots und Inkrementieren des Leseindex r_i 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)



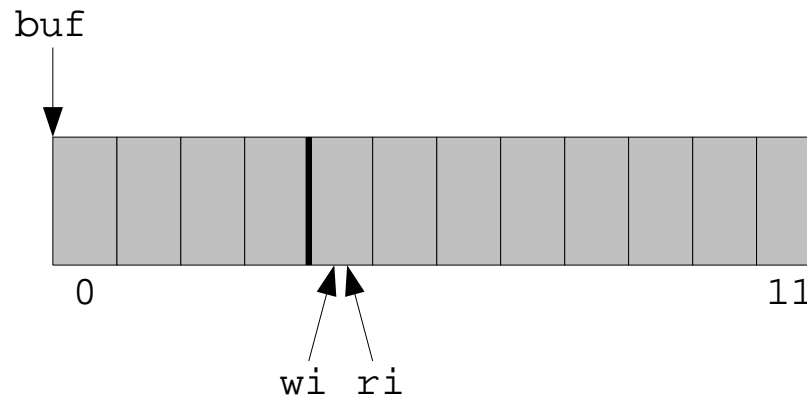
Wettlauf der Leser



- 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);
    return fd;
}
```





12 0
sem_full sem_free

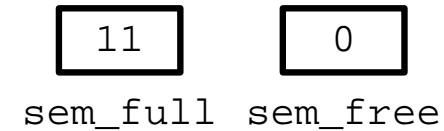
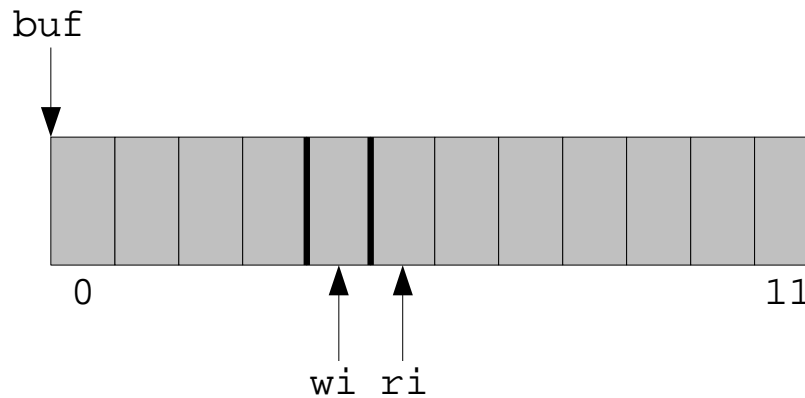
- Überlaufsituation: Schreiber blockiert, weil keine Slots frei

```
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;  
}
```

W

```
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

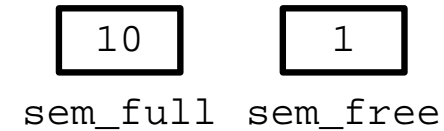
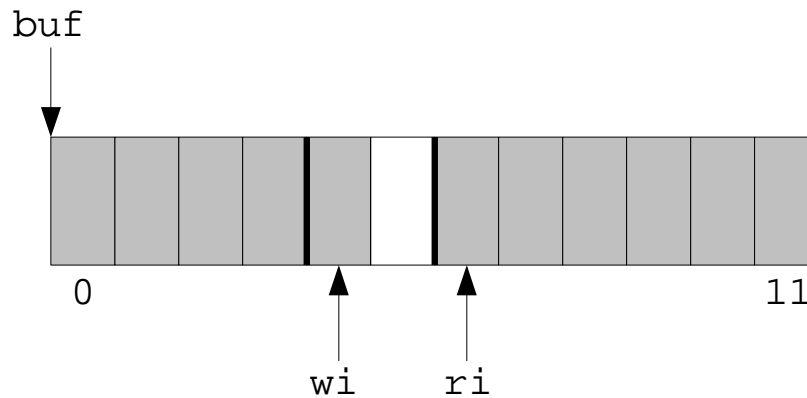
```
int get(void) {  
    int fd, pos, npos;  
    P(sem_full);  
    do {  
        pos = ri;  
        npos = (pos + 1) % 12;  
    } while(!cas(&ri, pos, npos));  
    fd = buf[pos]; pos: 4  
    V(sem_free);  
    return fd;  
}
```

R1

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

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;
}
    
```

R1
R2

```

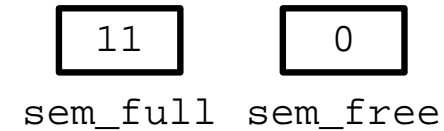
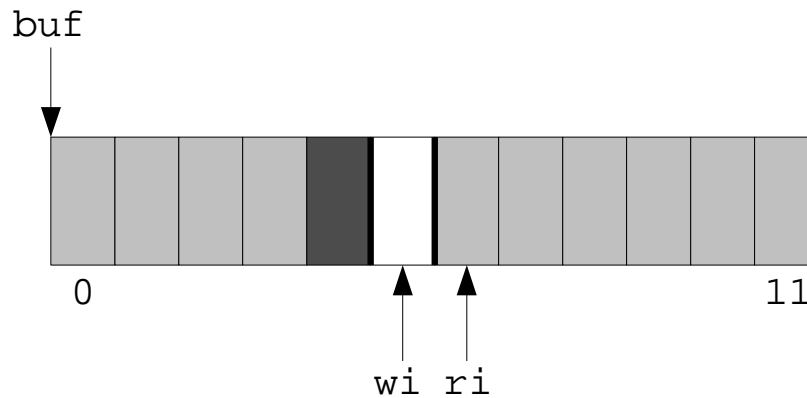
void add(int val) {
    P(sem_free);

    buf[wi] = val;
    wi = (wi + 1) % 12;

    V(sem_full);
}
    
```

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;
}
    
```

R1 R2

```

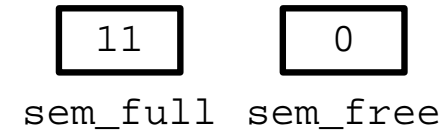
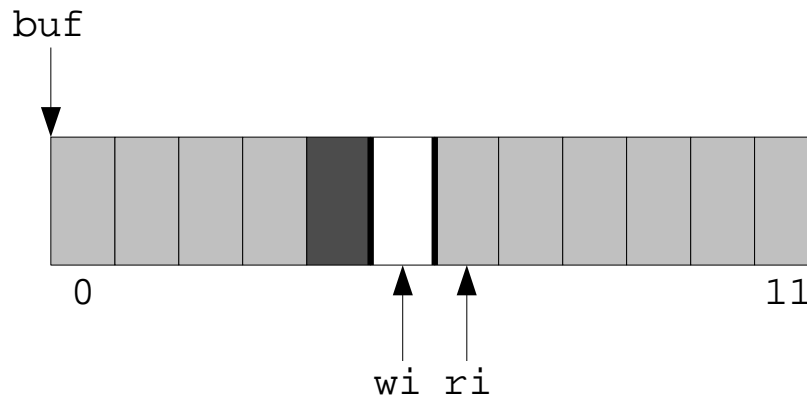
void add(int val) {
    P(sem_free);

    buf[wi] = val;
    wi = (wi + 1) % 12;

    V(sem_full);
}
    
```

W





- 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;
}
    
```

R1 R2

```

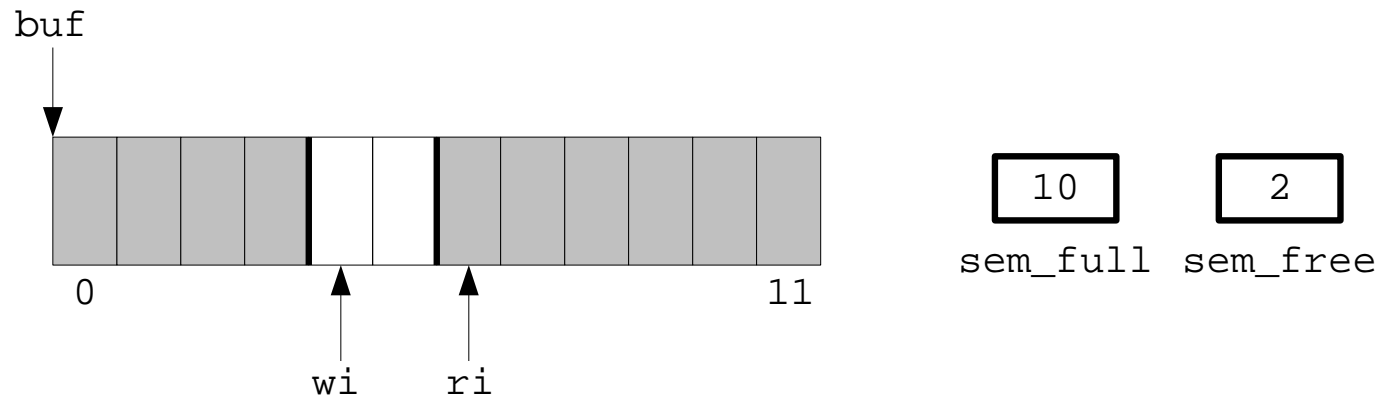
void add(int val) {
    P(sem_free);

    buf[wi] = val;
    wi = (wi + 1) % 12;

    V(sem_full);
}
    
```

W





- 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;
}
```



Brauchen wir das `volatile`-Schlüsselwort?

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



T1

```
bbGet();
```

T2

```
bbGet();  
bbPut(7);  
bbGet();
```



ABA-Problem bei der Verwendung von CAS



T1

```
bbGet();
```

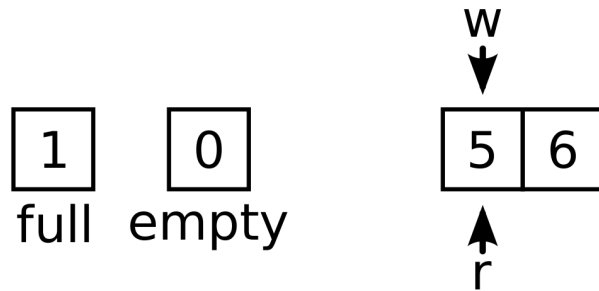
```
bbGet() {  
    ...  
    int retVal=0;  
    do {  
        ...  
        retVal = 5;  
        → } while (!cas(&r,0,1));  
        ...  
        v(empty);  
    }  
}
```

T2

```
bbGet();  
bbPut(7);  
bbGet();
```



ABA-Problem bei der Verwendung von CAS



T1

bbGet();

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

T2

→ bbGet();
bbPut(7);
bbGet();



ABA-Problem bei der Verwendung von CAS



T1

```
bbGet();
```

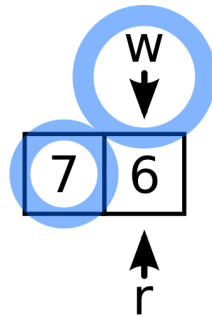
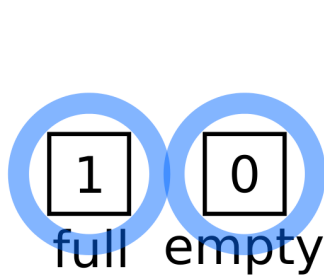
```
bbGet() {  
    ...  
    int retVal=0;  
    do {  
        ...  
        retVal = 5;  
    } while (!cas(&r,0,1));  
    ...  
    v(empty);  
}
```

T2

```
→ bbGet();  
   bbPut(7);  
   bbGet();
```



ABA-Problem bei der Verwendung von CAS



T2

Aktiver
Thread

T1

```
bbGet();
```

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

T2

```
bbGet();  
bbPut(7);  
bbGet();
```



ABA-Problem bei der Verwendung von CAS



T1

```
bbGet();
```

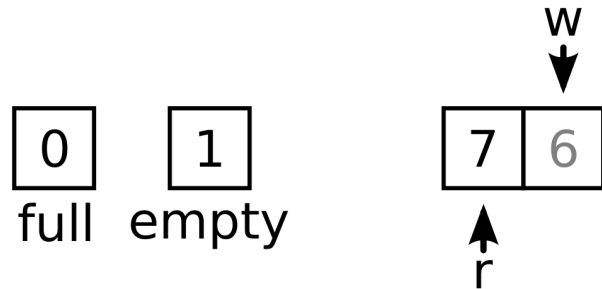
```
bbGet() {  
    ...  
    int retVal=0;  
    do {  
        ...  
        retVal = 5;  
    } while (!cas(&r,0,1));  
    ...  
    V(empty);  
}
```

T2

```
bbGet();  
bbPut(7);  
bbGet();
```



ABA-Problem bei der Verwendung von CAS



T1

Aktiver Thread

T1

```
bbGet();
```

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

T2

```
bbGet();  
bbPut(7);  
bbGet();
```



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

