

Übungen zu Systemprogrammierung 2 (SP2)

Ü 7 – Ringpuffer

Christoph Erhardt, Jens Schedel, Jürgen Kleinöder

Lehrstuhl für Informatik 4
Verteilte Systeme und Betriebssysteme

Friedrich-Alexander-Universität
Erlangen-Nürnberg

WS 2012/13 – 14. bis 18. Januar 2013

http://www4.cs.fau.de/Lehre/WS12/V_SP2

Agenda

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



7.1 Synchronisation des Ringpuffers

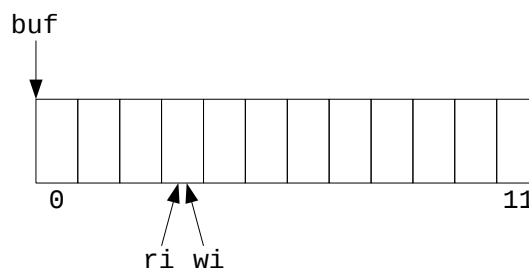
7.2 ABA-Problem bei der Verwendung von CAS

7.3 Vorteile nicht-blockierender Synchronisation



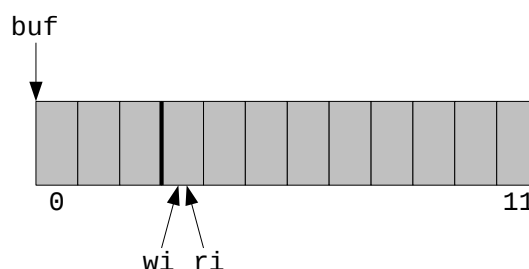
Über-/Unterlaufsituationen

■ Leerer Ringpuffer:



- Weiteres Lesen würde noch nicht gefüllten Slot liefern → Unterlauf!

■ Voller Ringpuffer:



- Weiteres Schreiben würde vollen Slot überschreiben → Überlauf!

■ Synchronisation mit Hilfe zweier Semaphoren



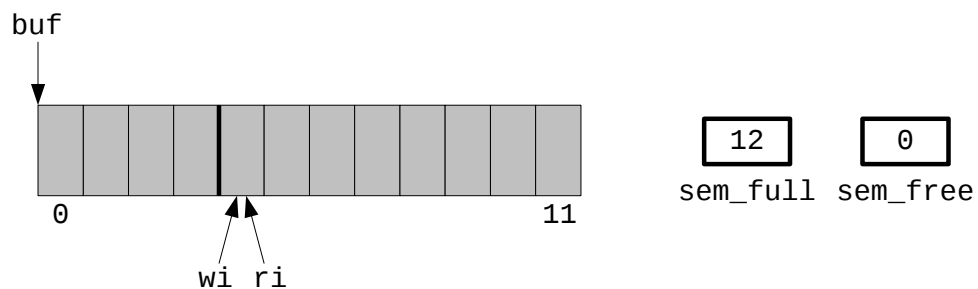
Wettlauf der Leser

- 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)

27-ABA_handout



Wettlauf der Leser

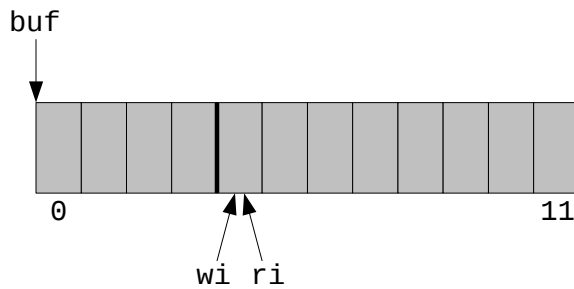


- Erhöhen des Leseindex mittels CAS – vollständig korrekt?

```
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;
}
```

27-ABA_handout





12	0
sem_full	sem_free

■ Überlaufsituation: Schreiber blockiert, weil keine Slots frei

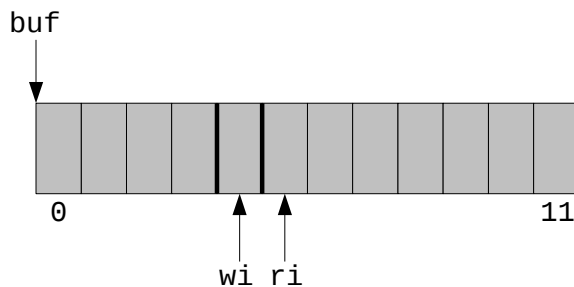
```
int get(void) {
    int fd, pos, npos;
    P(sem_full);
    do {
        pos = ri;
        npos = (pos + 1) % 12;
    } while(!cas(&ri, pos, npos));
    fd = buf[pos];
    V(sem_free);
    return fd;
}
```

W

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

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

    V(sem_full);
}
```



11	0
sem_full	sem_free

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

R1

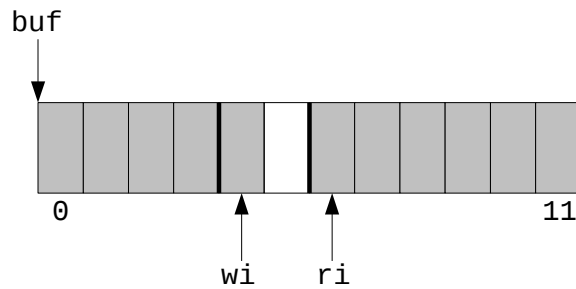
```
int get(void) {
    int fd, pos, npos;
    P(sem_full);
    do {
        pos = ri;
        npos = (pos + 1) % 12;
    } while(!cas(&ri, pos, npos));
    fd = buf[pos]; pos: 4
    V(sem_free);
    return fd;
}
```

W

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

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

    V(sem_full);
}
```



10	1
sem_full	sem_free

- 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 (blue wavy line) and **R2** (grey wavy line) are shown. R2 is at `pos: 5`. R1 is at `pos: 4`.

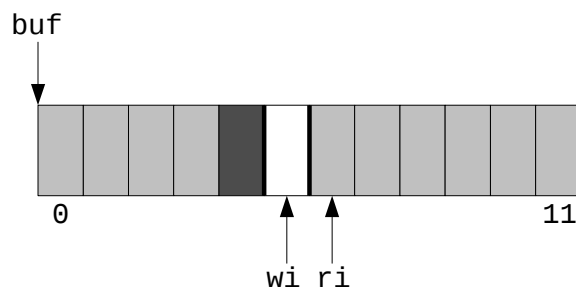
```

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

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

    V(sem_full);
}
  
```

W (black wavy line) is shown.



11	0
sem_full	sem_free

- 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 (blue wavy line) and **R2** (grey wavy line) are shown. R2 is at `pos: 5`. R1 is at `pos: 4`.

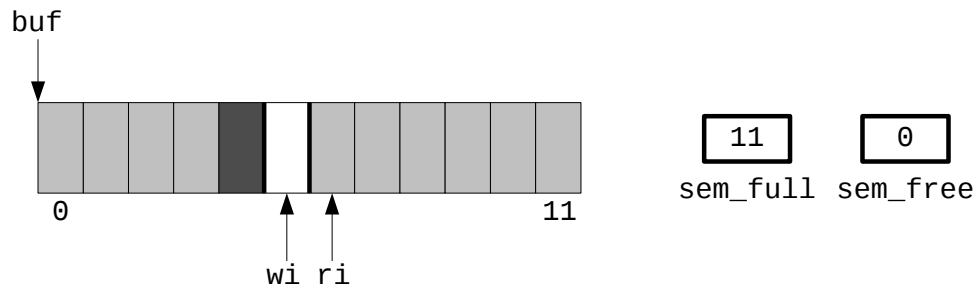
```

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

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

    V(sem_full);
}
  
```

W (black wavy line) is shown.



■ 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

```

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

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

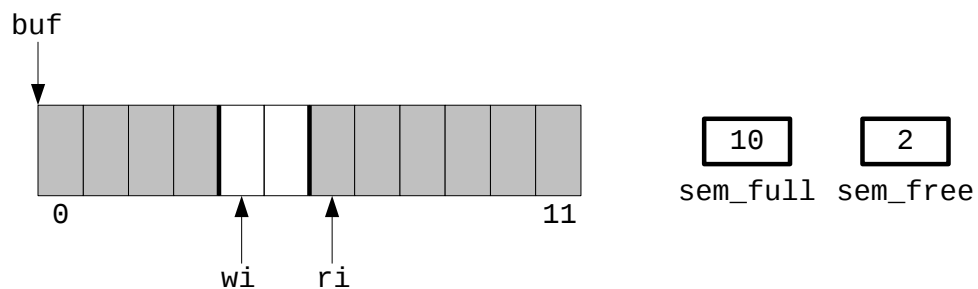
    V(sem_full);
}

```

W

Wettlauf der Leser

Vorsicht bei CAS!



■ 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

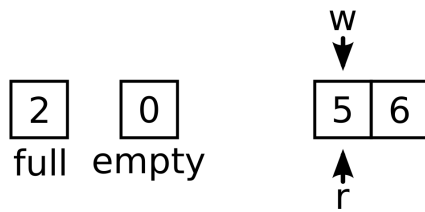
7.1 Synchronisation des Ringpuffers

7.2 ABA-Problem bei der Verwendung von CAS

7.3 Vorteile nicht-blockierender Synchronisation



ABA-Problem bei der Verwendung von CAS



Aktiver Thread

T1

```
bbGet();
```

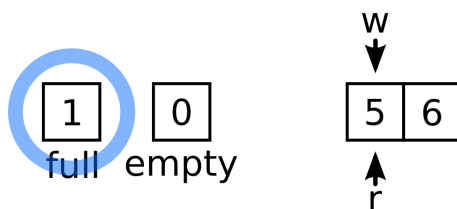
T2

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

27-ABA_handout



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);  
    }  
}
```

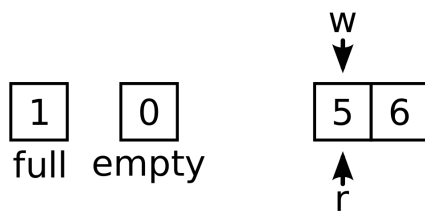
T2

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

27-ABA_handout



ABA-Problem bei der Verwendung von CAS



T2
Aktiver
Thread

T1

bbGet();

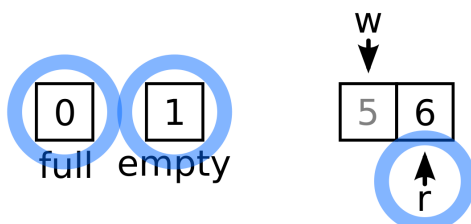
```
bbGet() {  
    ...  
    int retVal=0;  
    do {  
        ...  
        retVal = 5;  
    } while (!cas(&r,0,1));  
    ...  
    V(empty);  
}
```

T2

→ **bbGet();**
bbPut(7);
bbGet();



ABA-Problem bei der Verwendung von CAS



T2
Aktiver
Thread

T1

bbGet();

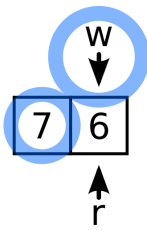
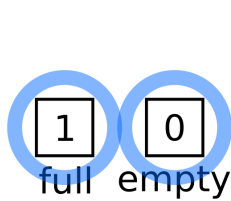
```
bbGet() {  
    ...  
    int retVal=0;  
    do {  
        ...  
        retVal = 5;  
    } while (!cas(&r,0,1));  
    ...  
    V(empty);  
}
```

T2

→ **bbGet();**
bbPut(7);
bbGet();



ABA-Problem bei der Verwendung von CAS



T2

Aktiver Thread

T1

bbGet();

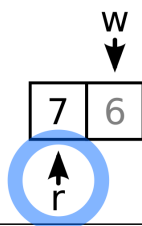
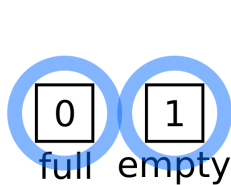
```
bbGet() {  
    ...  
    int retVal=0;  
    do {  
        ...  
        retVal = 5;  
    } while (!cas(&r,0,1));  
    ...  
    V(empty);  
}
```

T2

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



ABA-Problem bei der Verwendung von CAS



T2

Aktiver Thread

T1

bbGet();

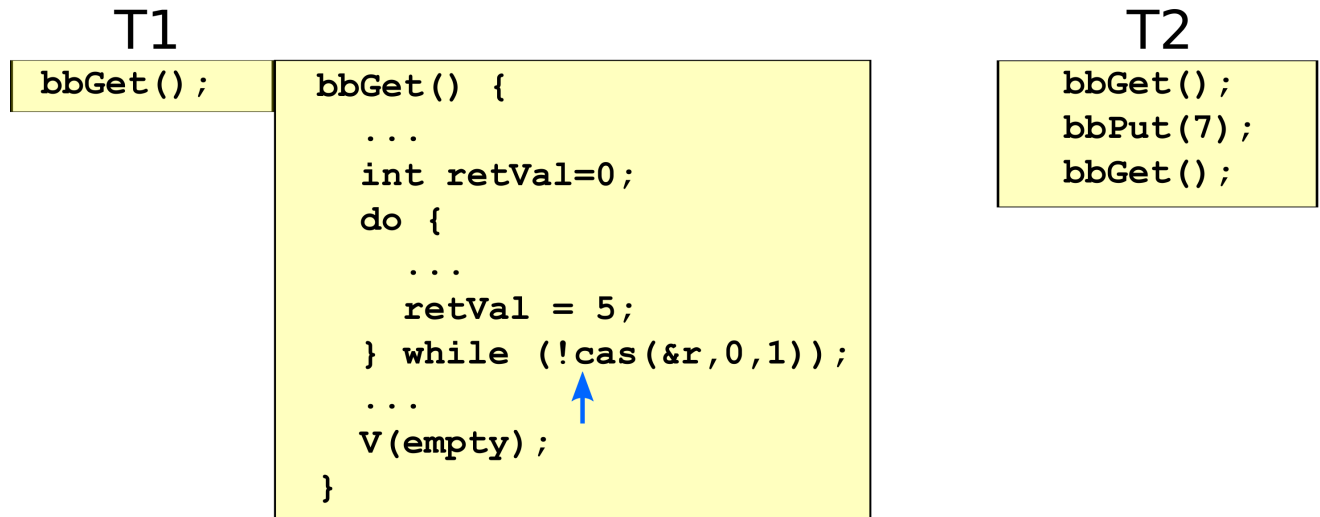
```
bbGet() {  
    ...  
    int retVal=0;  
    do {  
        ...  
        retVal = 5;  
    } while (!cas(&r,0,1));  
    ...  
    V(empty);  
}
```

T2

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



ABA-Problem bei der Verwendung von CAS



ABA-Problem bei der Verwendung von CAS

- `bbGet()` liefert 5 statt 7 zurück
 - CAS schlägt nicht fehl, weil `r` nach dem Wiedereinladen 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 Synchronisation des Ringpuffers
- 7.2 ABA-Problem bei der Verwendung von CAS
- 7.3 Vorteile nicht-blockierender Synchronisation



- Vorteile gegenüber sperrenden oder blockierenden Verfahren (Auswahl):
 - Rein auf Anwendungsebene, keine teuren Systemaufrufe
 - 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
- Relevant vor allem in massiv parallelen Systemen
- In unserem konkreten Anwendungsbeispiel kommen diese Vorteile nicht wirklich zum Tragen
 - Übungsbeispiel zum Begreifen des Konzepts

