

Übungen zu Systemprogrammierung 2

Ü 7 – Ringpuffer

Sommersemester 2018

Christian Eichler, Jürgen Kleinöder

Lehrstuhl für Informatik 4
Friedrich-Alexander-Universität Erlangen-Nürnberg



Lehrstuhl für Verteilte Systeme
und Betriebssysteme



FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG

TECHNISCHE FAKULTÄT



7.1 Synchronisation des Ringpuffers

7.2 ABA-Problem bei der Verwendung von CAS

7.3 Vorteile nicht-blockierender Synchronisation

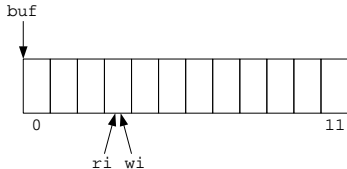


7.1 Synchronisation des Ringpuffers

7.2 ABA-Problem bei der Verwendung von CAS

7.3 Vorteile nicht-blockierender Synchronisation

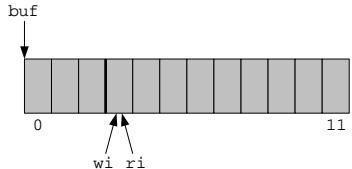
Leerer Ringpuffer:



Weiteres Lesen würde noch nicht
gefüllten Slot liefern

→ Unterlauf!

Voller Ringpuffer:



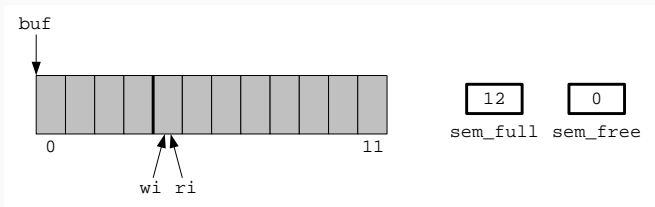
Weiteres Schreiben würde vollen
Slot überschreiben

→ Überlauf!

👉 Synchronisation mit Hilfe zweier Semaphore

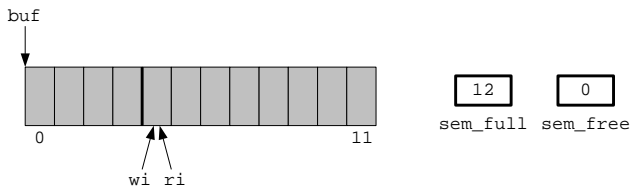


- Auslesen des Slots und Inkrementieren des Leseindex r_i geschieht nicht atomar
 - Mehrere Threads könnten nebenläufig den selben Slot auslesen
- Es existiert keine Abhängigkeit der Leser untereinander
→ Nicht-blockierende Synchronisation möglich
- Synchronisation mittels *Compare and Swap* (CAS)



■ 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);  
}
```

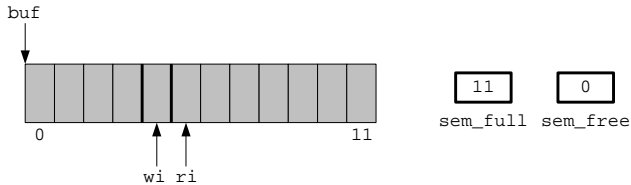


■ Überlaufsituation: Schreiber blockiert, weil keine Slots frei

```
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);  
}
```

W
↓



- 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;
}
    
```

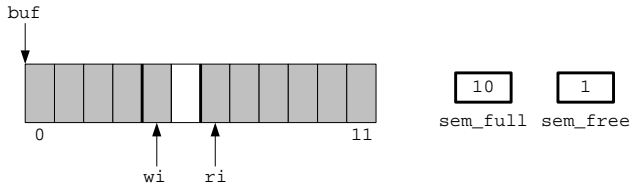
R1

pos: 4

```

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

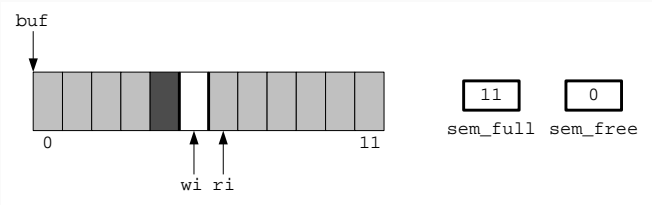
```

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;
}

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

Execution flow is indicated by vertical lines and arrows:

- R1 (Red):** Starts at the `while` loop, reaches `pos: 4`, and then proceeds to `fd = buf[pos];`.
- R2 (Blue):** Starts at the `while` loop, reaches `pos: 5`, and then proceeds to `fd = buf[pos];`.
- W (Black):** Starts at the `add` function, reaches `buf[wi] = val;`, and then proceeds to `wi = (wi + 1) % 12;`.



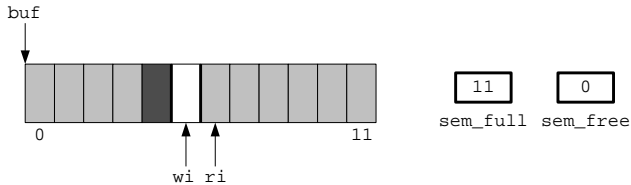
- W wird deblockiert, komplettiert add() und **überschreibt Slot 4**

```

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);
}
    
```

Execution flow is indicated by wavy arrows: **R1** (red) points to the `while` loop in `get`, with a label `pos: 4` below it. **R2** (blue) points to the `while` loop in `get`, with a label `pos: 5` below it. **W** (black) points to the `add` function, with a label `pos: 5` below it.



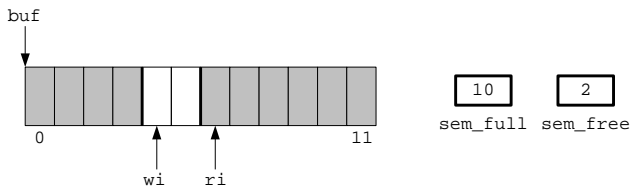
■ Ursache: FIFO-Entnahmeeigenschaft des Puffers nicht sichergestellt

```

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);
}
    
```

Diagram illustrating the race condition between the reader (R1) and the writer (R2) in the `get` function. The reader (R1) is shown with a red wavy arrow and the label `pos: 4`. The writer (R2) is shown with a blue wavy arrow and the label `pos: 5`. A third wavy arrow labeled `W` is shown on the right side of the `add` function.



■ 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



7.1 Synchronisation des Ringpuffers

7.2 ABA-Problem bei der Verwendung von CAS

7.3 Vorteile nicht-blockierender Synchronisation



2 0
full empty



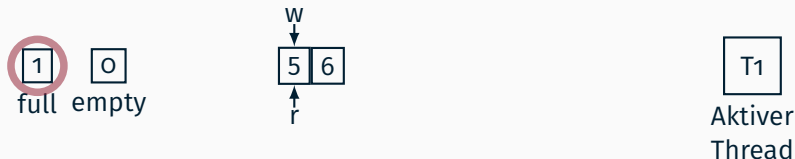
Aktiver
Thread

T1

bbGet();

T2

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

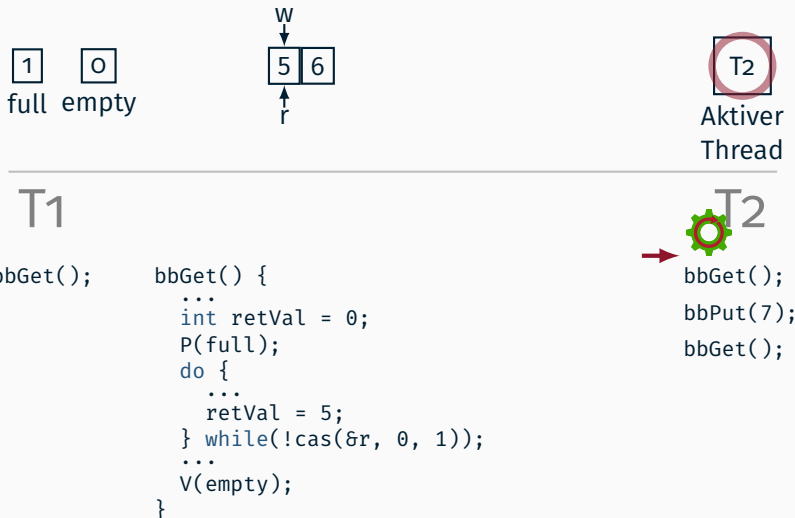


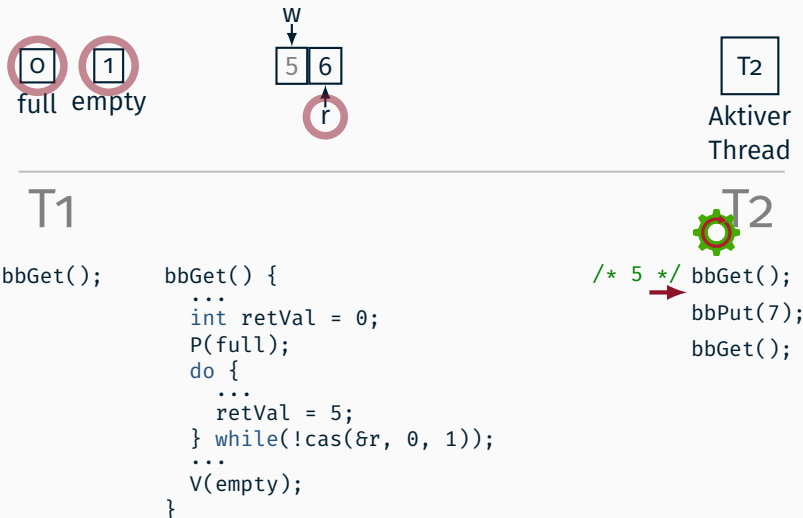
bbGet();

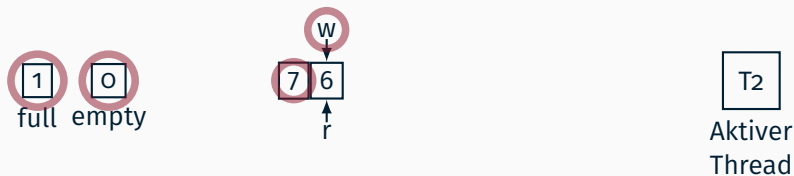
```
bbGet() {  
    ...  
    int retVal = 0;  
    P(full);  
    do {  
        ...  
        retVal = 5;  
        → } while(!cas(&r, 0, 1));  
        ...  
        V(empty);  
    }  
}
```

T2

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



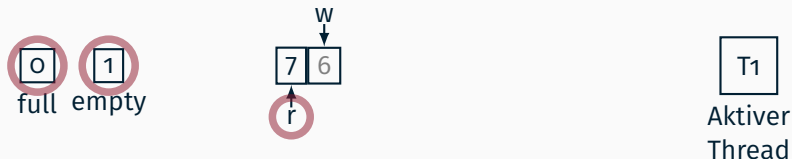


T1

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

T2

```
/* 5 */ bbGet();  
→ bbPut(7);  
bbGet();
```



bbGet();

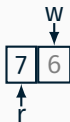
```
bbGet() {  
    ...  
    int retVal = 0;  
    P(full);  
    do {  
        ...  
        retVal = 5;  
    } while(!cas(&r, 0, 1));  
    ...  
    V(empty);  
}
```

T2

```
/* 5 */ bbGet();  
        bbPut(7);  
/* 6 */ bbGet();
```



0 1
full empty



T1
Aktiver
Thread



T1
bbGet();

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

T2

```
/* 5 */ bbGet();  
        bbPut(7);  
/* 6 */ bbGet();
```



- `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.



- 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



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
 - 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