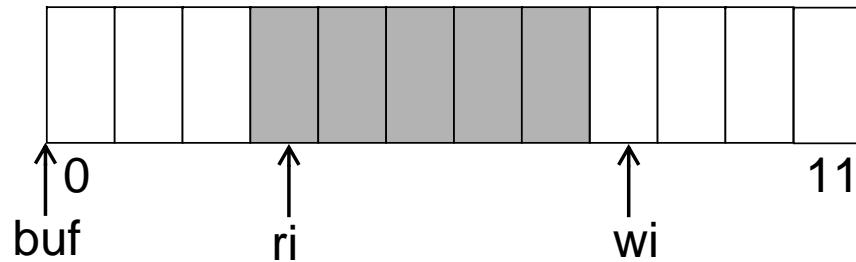


U6 6. Übung

- Besprechung der Aufgabe 4: jbuffer
- Synchronisation eines Ringpuffers
 - ◆ ABA-Problem
- Threads und Signale
- Threads und Prozesse
- Aufgabe 5: mother

U6-1 Synchronisation eines Ringpuffers



■ Parameter und Zustand

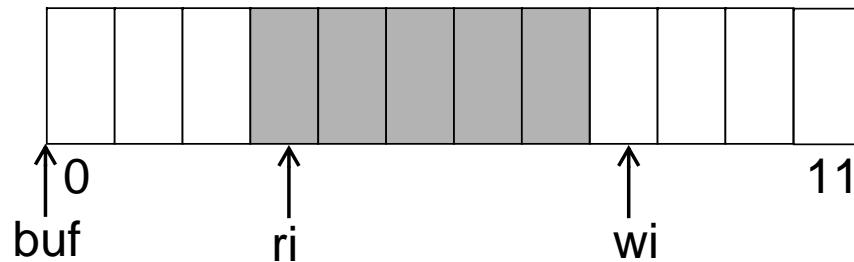
- ◆ Anzahl der Slots (hier: 12)
- ◆ Leserposition = Index des nächsten zu lesenden Slots (hier: 3)
- ◆ Schreiberposition = Index des nächsten zu schreibenden Slots (hier: 8)

■ Slots als konsumierbare Betriebsmittel

- ◆ Schreiber konsumiert freie Slots, produziert belegte Slots
- ◆ Leser konsumieren belegte Slots, produzieren freie Slots

SP - U

1 Ringpuffer: Basisoperationen



■ Basisoperationen:

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

```
int get(void) {
    int fd, pos;

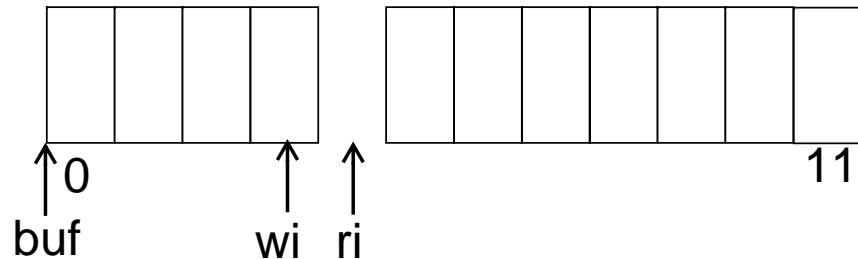
    pos = ri;
    ri = (pos + 1) % 12;

    fd = buf[pos];

    return fd;
}
```

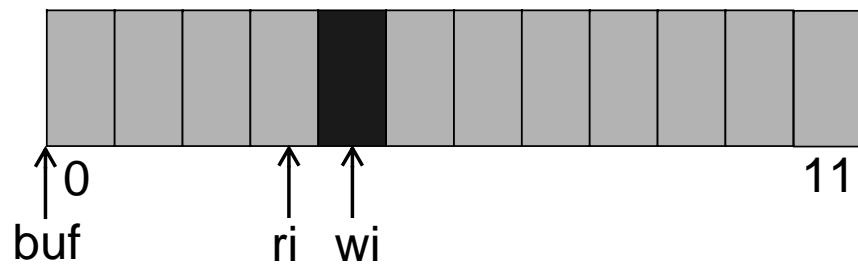
2 Über-/Unterlaufsituationen

- Unterlauf: Alle vollen Slots wurden von Lesern konsumiert



◆ Leser hängen nun vom Fortschritt des Schreibers ab - weiteres Lesen problematisch

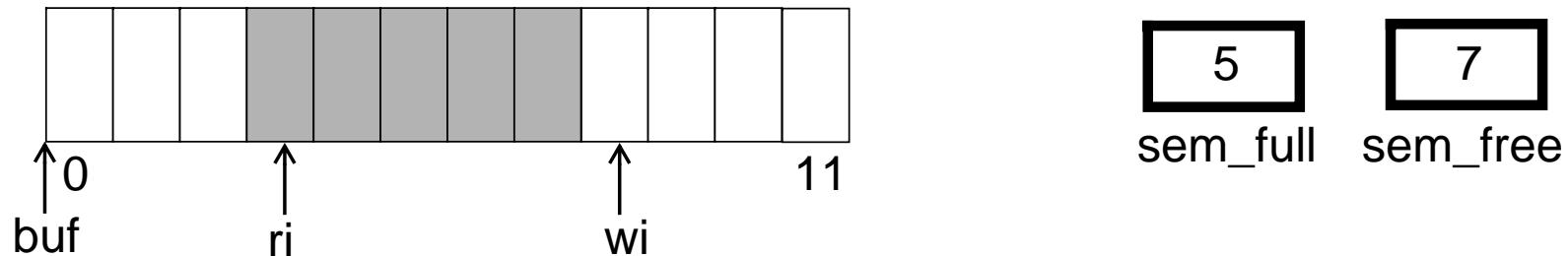
- Überlauf: Alle freien Slots wurden vom Schreiber konsumiert



◆ Schreiber hängt nun vom Fortschritt der Leser ab - weiteres Schreiben problematisch

☞ Verwaltung des Betriebsmittelbestands mit zählenden Semaphoren

3 Über-/Unterlaufsituationen: Synchronisation



■ Basisoperationen:

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

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

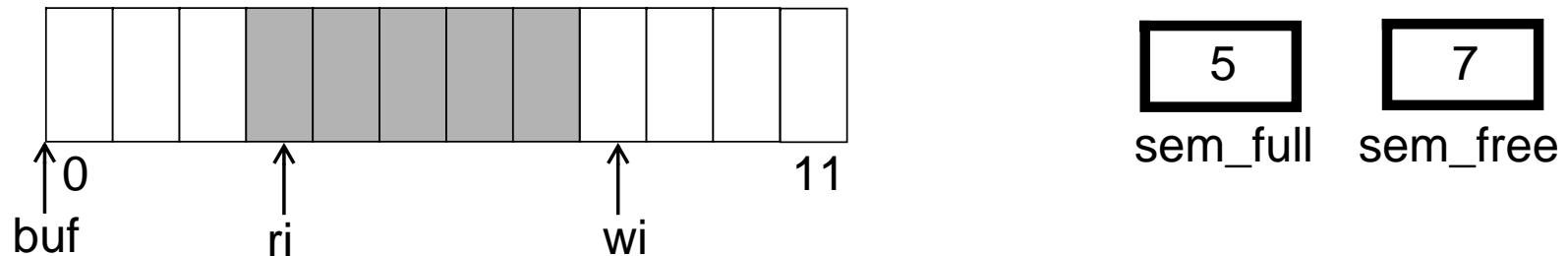
    V(sem_full);
}
```

```
int get(void) {
    int fd, pos;
    P(sem_full);

    pos = ri;
    ri = (pos + 1) % 12;

    fd = buf[pos];
    V(sem_free);
    return fd;
}
```

4 Wettlauf der Leser



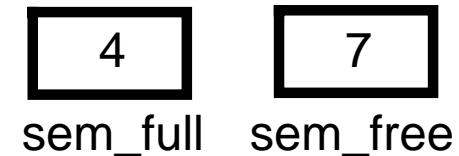
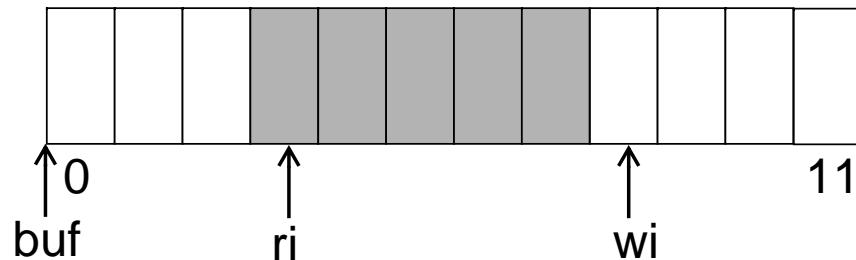
- Mehrere Leser können sich gleichzeitig in `get()` befinden

```
int get(void) {
    int fd, pos;
    P(sem_full);

    pos = ri;
    ri = (pos + 1) % 12;

    fd = buf[pos];
    V(sem_free);
    return fd;
}
```

4 Wettlauf der Leser



- R1 wird nach dem Laden von `ri` verdrängt

```

int get(void) {
    int fd, pos;
    P(sem_full);

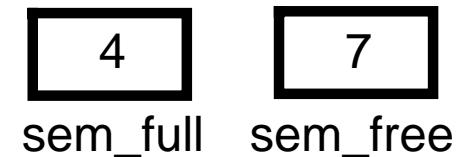
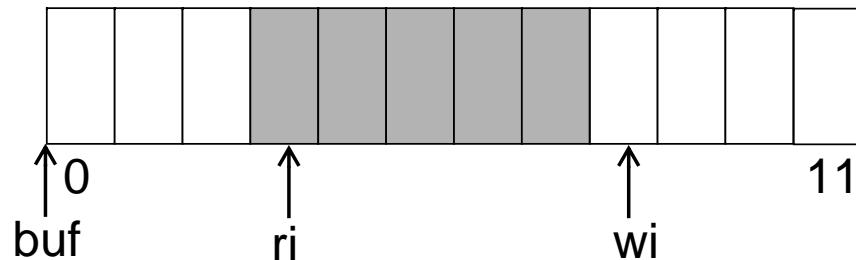
    pos = ri;
    ri = (pos + 1) % 12;

    fd = buf[pos];
    V(sem_free);
    return fd;
}

```

R1
pos: 3

4 Wettlauf der Leser



- Ein zweiter Leser R2 betritt get()

```

int get(void) {
    int fd, pos;
    P(sem_full);

    pos = ri;
    ri = (pos + 1) % 12;

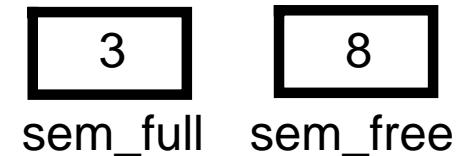
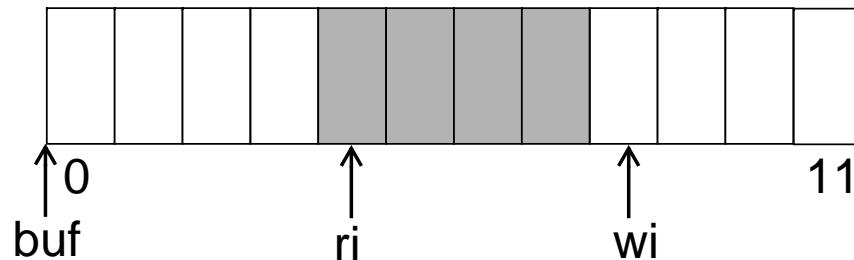
    fd = buf[pos];
    V(sem_free);
    return fd;
}

```

R1

pos: 3

4 Wettlauf der Leser



- R2 entnimmt Slot 3, ri wird auf 4 erhöht

```
int get(void) {
    int fd, pos;
    P(sem_full);

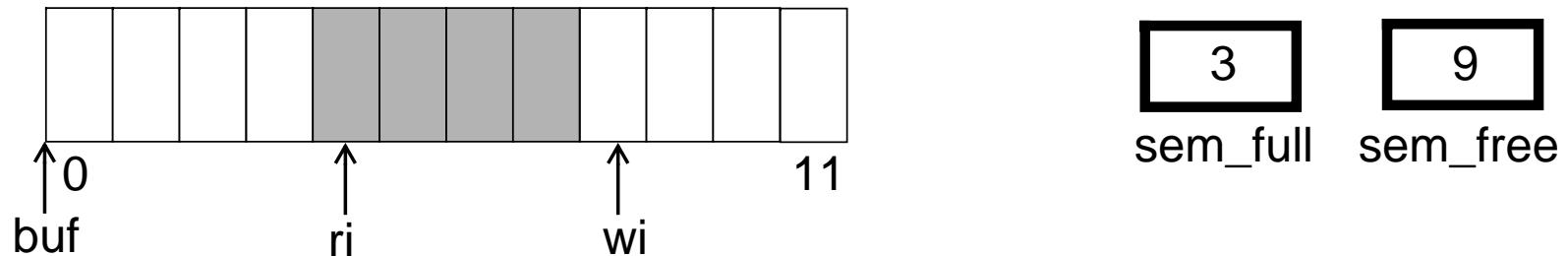
    pos = ri;
    ri = (pos + 1) % 12;

    fd = buf[pos];
    V(sem_free);
    return fd;
}
```

R1
pos: 3

R2
pos: 3
ri := 4
fd: buf[3]

4 Wettlauf der Leser



3	9
sem_full	sem_free

- R1 komplettiert get() ebenfalls mit Slot 3

```
int get(void) {
    int fd, pos;
    P(sem_full);

    pos = ri;
    ri = (pos + 1) % 12;

    fd = buf[pos];
    V(sem_free);
    return fd;
}
```

R1

pos: 3
ri := 4
fd: buf[3]

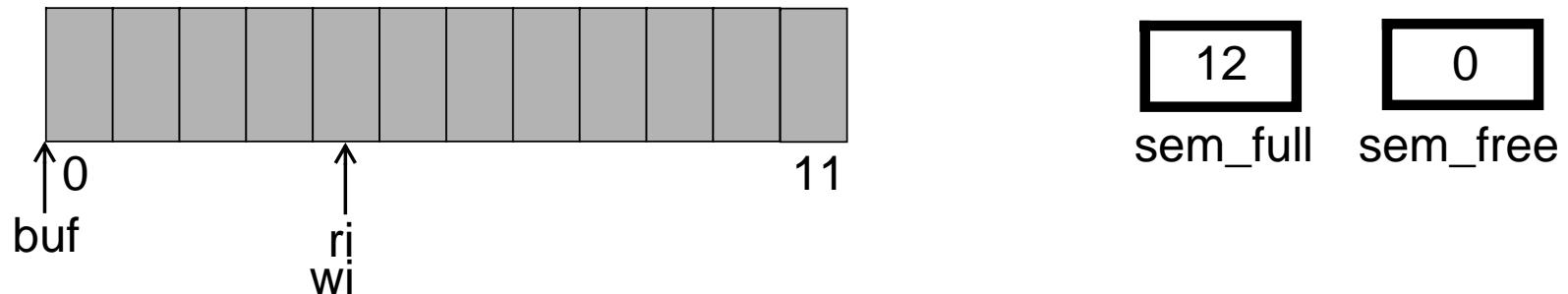
R2

pos: 3
ri := 4
fd: buf[3]

4 Wettlauf der Leser

- Inkrementieren des Leseindex ri nicht atomar
- Es existiert keine Abhangigkeit zwischen den Lesern
 - ☞ nicht-blockierende Synchronisation moglich hier mittels Compare-And-Swap (CAS)

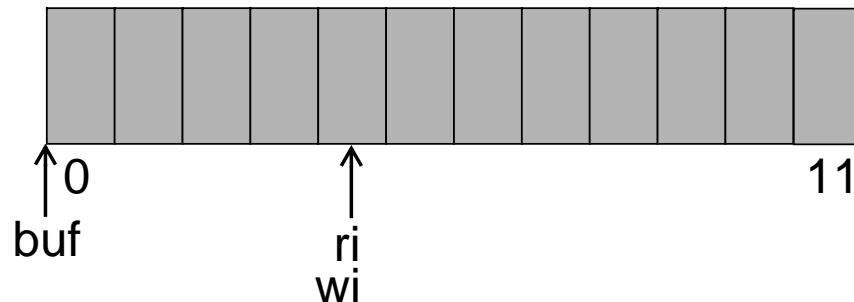
4 Wettlauf der Leser



■ Erhöhung des Leseindex mittels CAS

```
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;
}
```

4 Wettlauf der Leser



12	0
sem_full	sem_free

- Überlaufs situation: Schreiber blockiert, weil keine freien Slots verfügbar

```
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;
}
```

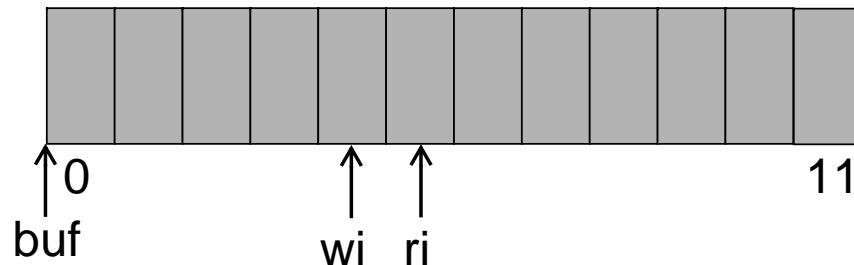
W

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

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

    V(sem_full);
}
```

4 Wettlauf der Leser



11	0
sem_full	sem_free

- R1 sichert sich Leseposition 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

```

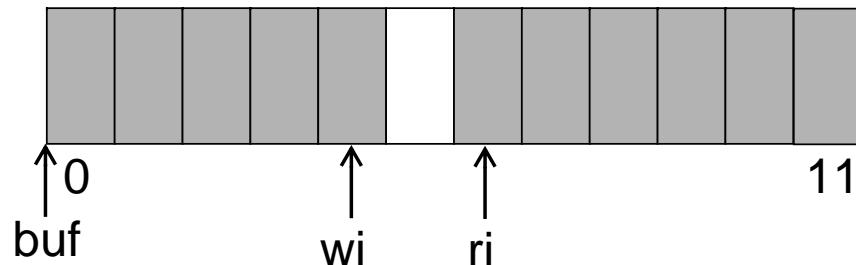
void add(int val) {
    P(sem_free);

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

    V(sem_full);
}
  
```

W

4 Wettlauf der Leser



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];  pos: 4    pos: 5
    V(sem_free);
    return fd;
}
  
```

R1 R2

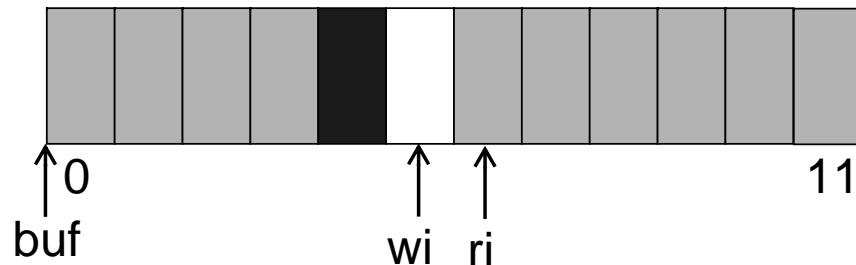
A blue wavy arrow points from the 'pos' variable in the first iteration of the loop to the value '5'. A grey wavy arrow points from the 'pos' variable in the assignment 'fd = buf[pos];' to the value '5'.

```

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

W

4 Wettlauf der Leser



11	0
sem_full	sem_free

- Schreiber W wird deblockiert, komplettiert add(), 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];  pos: 4      pos: 5
    V(sem_free);
    return fd;
}
  
```

R1 R2

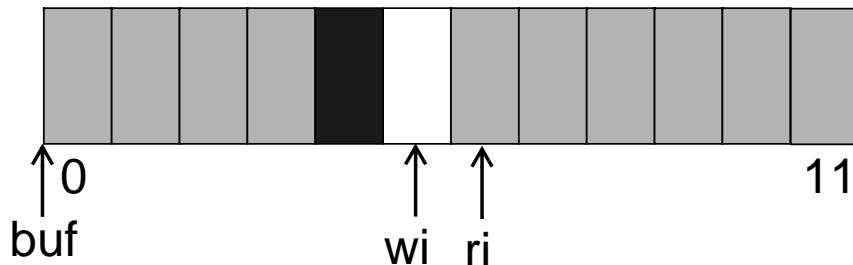
A blue curly brace labeled "R1" covers the code from "P(sem_full)" to "while(!cas(&ri, pos, npos))". A brown curly brace labeled "R2" covers the code from "buf[pos]" to "V(sem_free)". A blue arrow points from the "cas" line to the "buf[pos]" line, indicating the update of slot 4.

```

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

A black curly brace labeled "W" covers the entire "add" function code.

4 Wettlauf der Leser



11	0
sem_full	sem_free

- Problem: FIFO-Entnahmeeigenschaft 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];  pos: 4   pos: 5
    V(sem_free);
    return fd;
}
  
```

R1 R2

W

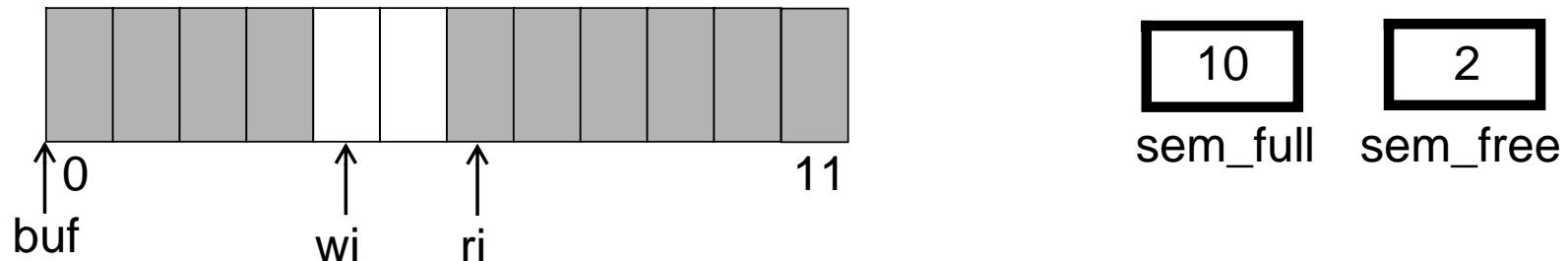
```

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

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

    V(sem_full);
}
  
```

4 Wettlauf der Leser



- Lösung: Entnahme des Datums vor Durchführung von CAS

```
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;
}
```

5 Vorteile nicht-blockierender Synchronisation

- Vorteile gegenüber sperrenden oder blockierenden Verfahren (Auswahl):
 - ◆ konkurrierende Fäden werden vom Scheduler nach dessen Kriterien eingeplant
 - ◆ rein auf Anwendungsebene: keine teuren Systemaufrufe
 - ◆ 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
- im konkreten Anwendungsbeispiel kommen diese Vorteile nicht wirklich zum Tragen
 - ☞ Übungsbeispiel zum Begreifen des Konzepts

6 ABA-Problem



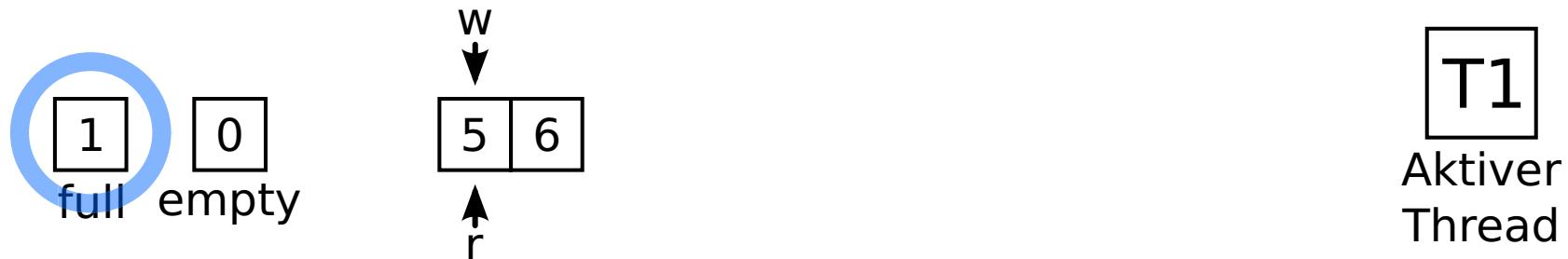
T1

```
bb_get();
```

T2

```
bb_get();
bb_put(7);
bb_get();
```

6 ABA-Problem



T1

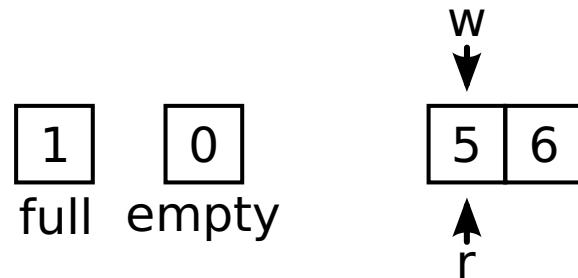
```
bb_get();
```

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

T2

```
bb_get();
bb_put(7);
bb_get();
```

6 ABA-Problem



Aktiver Thread

T1

```
bb_get();
```

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

T2

```
→ bb_get();
bb_put(7);
bb_get();
```

6 ABA-Problem



T1

```
bb_get();
```

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

T2

→

```
bb_get();
bb_put(7);
bb_get();
```

6 ABA-Problem



T1

```
bb_get();
```

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

T2

```
bb_get();
```

→

```
bb_put(7);
```

```
bb_get();
```

6 ABA-Problem



T1

```
bb_get();
```

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

T2

```
bb_get();
```

```
bb_put(7);
```

```
bb_get();
```

6 ABA-Problem



T1

```
bb_get();
```

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

T2

```
bb_get();
bb_put(7);
bb_get();
```

6 ABA-Problem

- `bb_get()` liefert 5 statt 7 zurück
 - ◆ die zwischenzeitliche Wert-Änderung des Leseindex `r` wird von CAS nicht erkannt
- Grundsätzliches Problem von inhaltsbasierten Elementaroperationen
- Gegenmaßnahmen siehe Vorlesung C | X-4 S. 24ff.

U6-2 Threads und Signale

■ Signale können...

- ◆ an einen Thread gerichtet sein
 - Synchrone Signale (z. B. SIGSEGV, SIGPIPE)
 - Signale, die mit `pthread_kill(3)` geschickt wurden
- ◆ an einen Prozess gerichtet sein
 - alle anderen Signale (z. B. mit `kill(2)` erzeugte Signale)

■ Signalbehandlung gilt prozessweit

- ◆ an Thread gerichtete Signale werden von diesem bearbeitet
- ◆ an Prozess gerichtete Signale werden von beliebigem Thread bearbeitet

■ Signalmaske ist Thread-lokal

- ◆ von einem Thread blockierte Signale, die...
 - an diesen gerichtet sind, werden verzögert
 - an dessen Prozess gerichtet sind, werden von anderem Thread bearbeitet

U6-3 Threads und Prozesse

- Verwendung von `fork(2)` in mehrfädigen Prozessen grundsätzlich problematisch
 - ◆ Bei `fork(2)` wird der aufrufende Thread geklont; alle anderen Threads sind im Kind nicht mehr vorhanden
 - ◆ Gelockte Mutexe bleiben gelockt und können nicht freigegeben oder zerstört werden
 - ◆ Kind kann inkonsistenten Zustand kopieren
- Unproblematisch, wenn geforkt wird, um `exec(2)` auszuführen
 - ◆ beim Aufruf von `exec(2)`
 - werden alle Mutexe und Bedingungsvariablen zerstört
 - verschwinden alle Threads - bis auf den aufrufenden