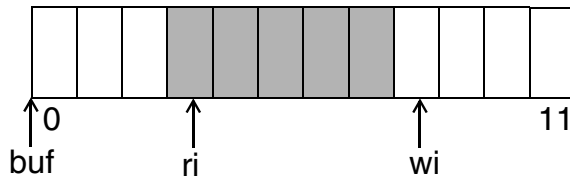
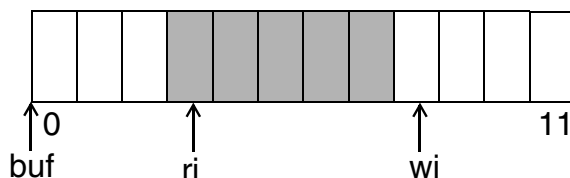


U12 Ringpuffer



- 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

U12-1 Ringpuffer: Basisoperationen



■ Basisoperationen:

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

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

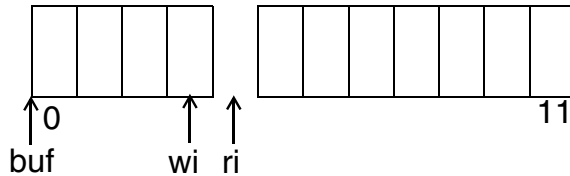
    pos = ri;
    ri = (pos + 1) % 12;

    fd = buf[pos];

    return fd;
}
```

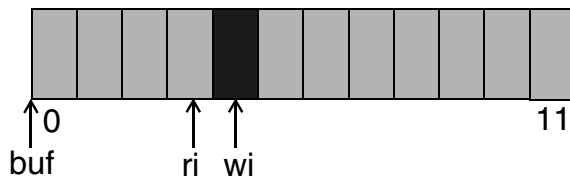
U12-2 Über-/Unterlaufsituationen

- Unterlauf: Alle vollen Slots wurden von Lesern konsumiert



- ◆ Leser hängen nun vom Fortschritt des Schreibers ab

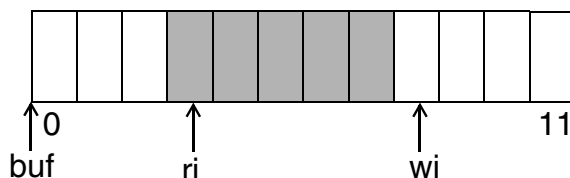
- Überlauf: Alle freien Slots wurden vom Schreiber konsumiert



- ◆ Schreiber hängt nun vom Fortschritt der Leser ab

- ☞ Verwaltung des Betriebsmittelbestands mit zählenden Semaphoren

U12-2 Über-/Unterlaufsituationen: Synchronisation



5 7
sem_full sem_free

- Basisoperationen:

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

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

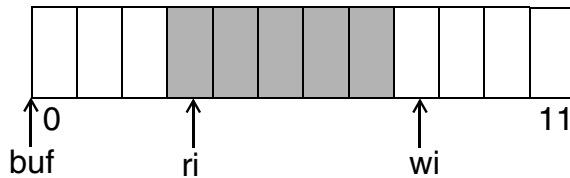
    V(sem_full);
}
```

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

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

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

U12-3 Wettlauf der Leser



5 7
sem_full sem_free

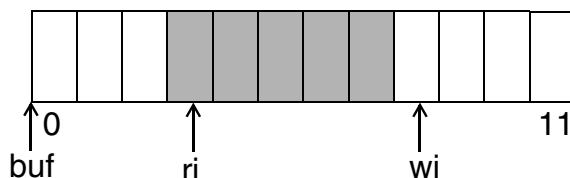
- Mehrere Leser können sich gleichzeitig in `get()` befinden

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

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

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

U12-3 Wettlauf der Leser



4 7
sem_full sem_free

- R1 wird nach dem Laden von `ri` verdrängt

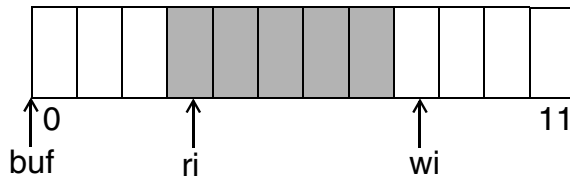
```
int get() {
    int fd, pos;
    P(sem_full);

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

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

R1
↓
pos: 3

U12-3 Wettlauf der Leser



4

sem_full

7

sem_free

■ Ein zweiter Leser R2 betritt get()

```

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

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

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

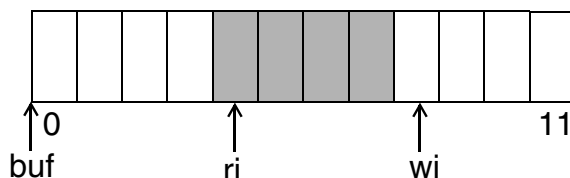
```

R1

pos: 3

R2

U12-3 Wettlauf der Leser



3

sem_full

8

sem_free

■ R2 entnimmt Slot 3, ri wird auf 4 erhöht

```

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

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

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

```

R1

pos: 3

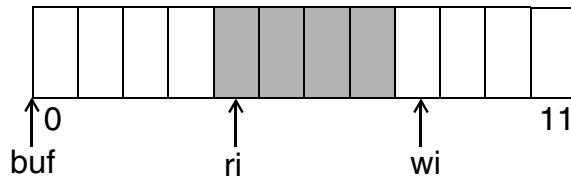
R2

pos: 3

ri := 4

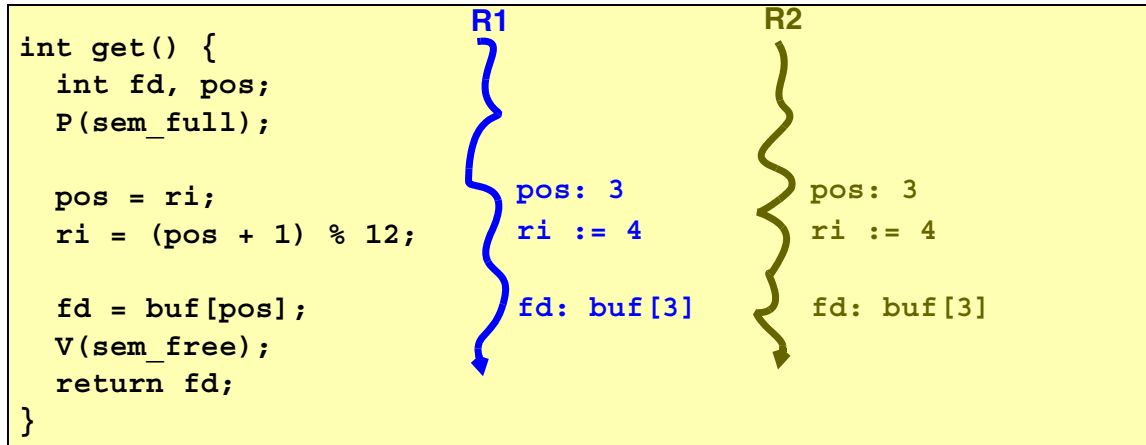
fd: buf[3]

U12-3 Wettlauf der Leser



3 9
 sem_full sem_free

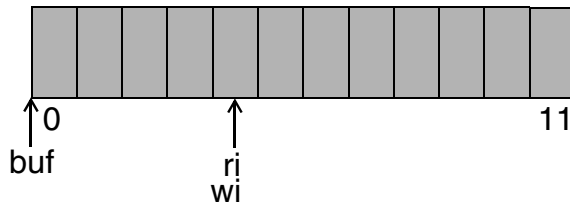
- R1 komplettiert `get()` ebenfalls mit Slot 3



U12-3 Wettlauf der Leser

- Inkrementieren des Leseindex `ri` nicht atomar
- Es existiert keine Abhängigkeit zwischen den Lesern
 - ☞ nicht-blockierende Synchronisation möglich
hier mittels Compare-And-Swap (CAS)

U12-3 Wettlauf der Leser

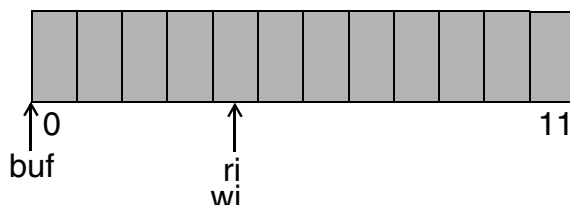


12	0
sem_full	sem_free

■ Erhöhung des Leseindex mittels CAS

```
int get() {
    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;
}
```

U12-3 Wettlauf der Leser



12	0
sem_full	sem_free

■ Überlaufsituation: Schreiber blockiert, weil keine freien Slots verfügbar

```
int get() {
    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;
}
```

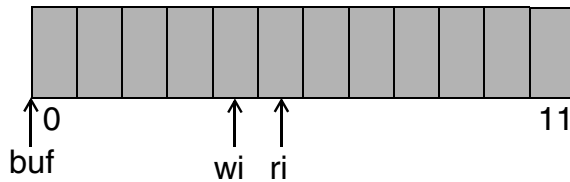
W

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

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

    V(sem_full);
}
```

U12-3 Wettlauf der Leser



11	0
sem_full	sem_free

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

```

int get() {
    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);

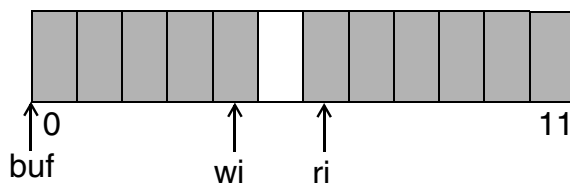
    buf[wi] = val;
    wi = (wi + 1) % 12;

    V(sem_full);
}

```

W

U12-3 Wettlauf der Leser



10	1
sem_full	sem_free

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

```

int get() {
    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

```

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

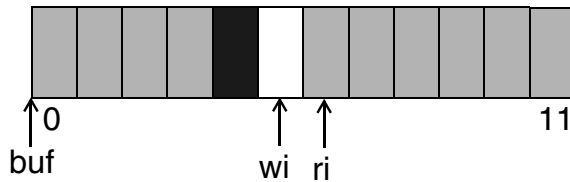
    buf[wi] = val;
    wi = (wi + 1) % 12;

    V(sem_full);
}

```

W

U12-3 Wettlauf der Leser



11 0
 sem_full sem_free

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

```

int get() {
    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;

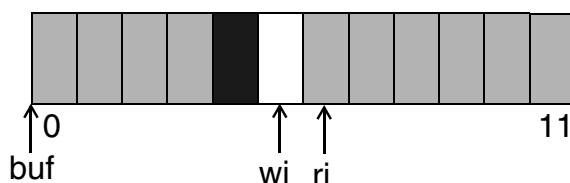
    V(sem_full);
}

```

W

pos: 4 pos: 5

U12-3 Wettlauf der Leser



11 0
 sem_full sem_free

- Problem: FIFO-Entnahmeeigenschaft nicht vorhanden

```

int get() {
    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;

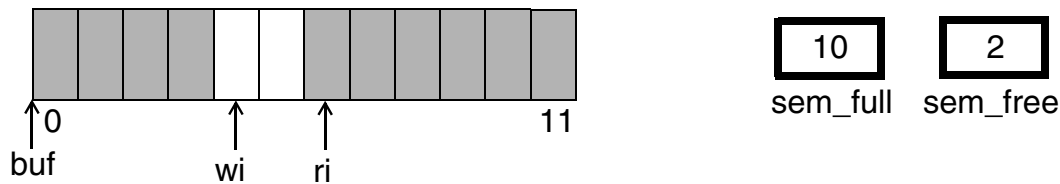
    V(sem_full);
}

```

W

pos: 4 pos: 5

U12-3 Wettlauf der Leser



■ Lösung: Entnahme des Datums vor Durchführung von CAS

```
int get() {
    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;
}
```

U12-3 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