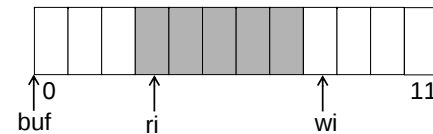


U5 Ringpuffer und Threads

- Synchronisation eines Ringpuffers
- Threads und Signale
- Threads und Prozesse

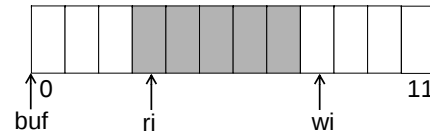
U5-1 Synchronisation eines Ringpuffer

U5-1 Synchronisation eines 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

1 Ringpuffer: Basisoperationen



■ Basisoperationen:

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

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

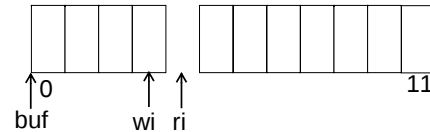
    pos = ri;
    ri = (pos + 1) % 12;

    fd = buf[pos];

    return fd;
}
```

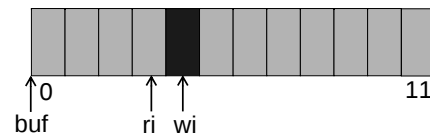
2 Über-/Unterlaufsituationen

■ Unterlauf: Alle vollen Slots wurden von Lesern konsumiert



◆ Leser hängen nun vom Fortschritt des Schreibers ab

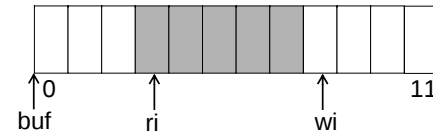
■ Überlauf: Alle freien Slots wurden vom Schreiber konsumiert



◆ Schreiber hängt nun vom Fortschritt der Leser ab

☞ Verwaltung des Betriebsmittelbestands mit zählenden Semaphoren

3 Über-/Unterlaufsituationen: Synchronisation



5 7
sem_full sem_free

■ Basisoperationen:

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

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

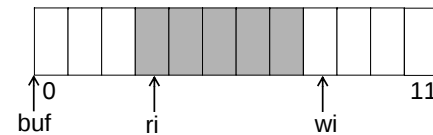
    V(sem_full);
}
```

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

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

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

4 Wettlauf der Leser



5 7
sem_full sem_free

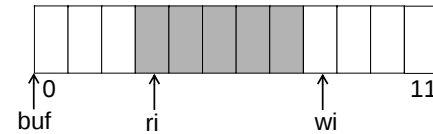
■ Mehrere Leser können sich gleichzeitig in get() befinden

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

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

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

4 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);

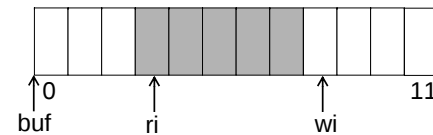
    pos = ri;
    ri = (pos + 1) % 12;

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

R1
↓
pos: 3

SP - Ü

4 Wettlauf der Leser



4 7
sem_full sem_free

■ Ein zweiter Leser R2 betritt `get()`

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

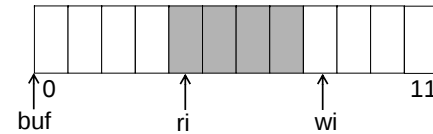
    pos = ri;
    ri = (pos + 1) % 12;

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

R1 R2
↓
pos: 3

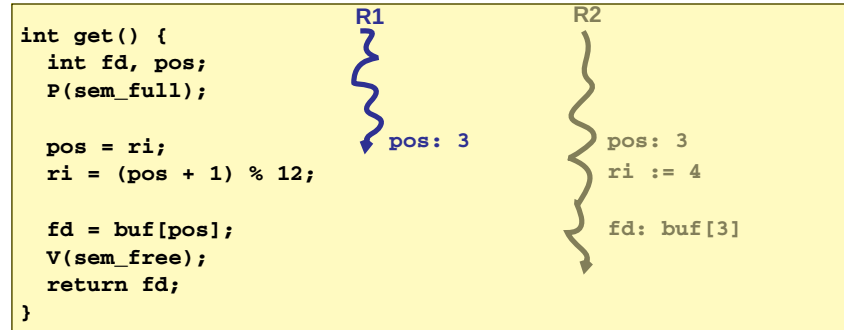
SP - Ü

4 Wettlauf der Leser



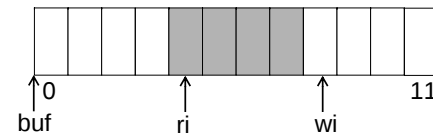
3 8
sem_full sem_free

■ R2 entnimmt Slot 3, `ri` wird auf 4 erhöht



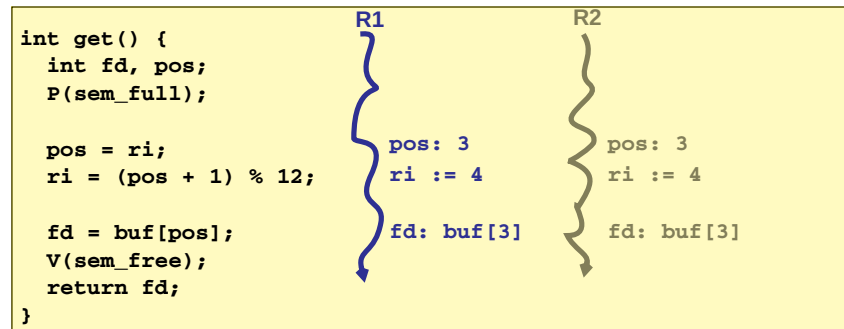
SP - Ü

4 Wettlauf der Leser



3 9
sem_full sem_free

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

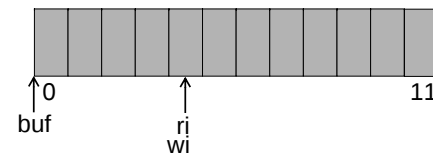


SP - Ü

4 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)

4 Wettlauf der Leser

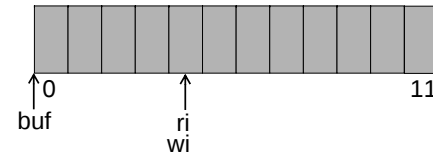


12	0
sem_full	sem_free

- Erhöhung des Leseindex mittels CAS

```
int get() {
    int fd, pos, npos;
    P(sem_full);
    do { /* Wiederhole... */
        pos = ri;
        npos = (pos + 1) % 12; /* Lokale Kopie des Werts ziehen */
        /* 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

■ 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

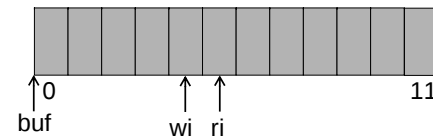
```
void add(int val) {
    P(sem_free);

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

    V(sem_full);
}
```

SP - Ü

4 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

W

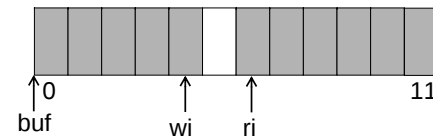
```
void add(int val) {
    P(sem_free);

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

    V(sem_full);
}
```

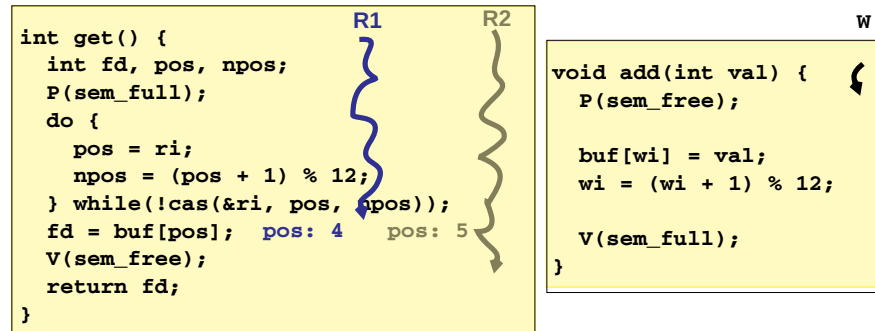
SP - Ü

4 Wettlauf der Leser



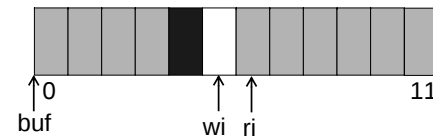
10 1
sem_full sem_free

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



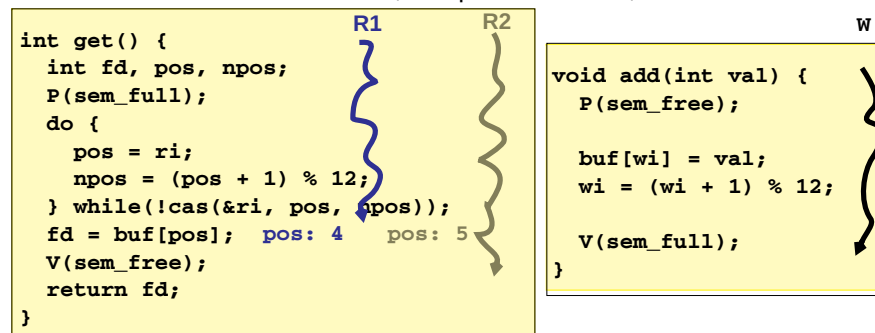
SP - Ü

4 Wettlauf der Leser



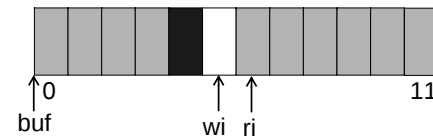
11 0
sem_full sem_free

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



SP - Ü

4 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;
}

```

```

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

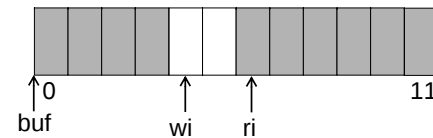
    buf[wi] = val;
    wi = (wi + 1) % 12;

    V(sem_full);
}

```

R1 R2 W

4 Wettlauf der Leser



10 2
 sem_full sem_free

■ 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;
}

```

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

U5-2 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

U5-3 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